smx®

User's Guide

Version 4.4.0

October 2017

by Ralph Moore

μd Micro Digital

Revisions

date	version	comments
9/88	1.x	preliminary
12/88	1.0	first release
8/89	1.1	bring into full agreement with Reference Manual and Protosystem; index thoroughly revised
1/90	1.2	minor corrections
6/90	1.3	revision updates
2/91	2.0	major revision including addition of new chapters
11/95	3.2	complete rewrite including addition of new chapters and thoroughly revised index
7/01	3.5	major revision to update to v3.5 and separation of x86-specific information into a new section
5/04	3.6	update to v3.6; moved x86 chapters to SMX Target Guide; moved code examples from SMX Quick Start to this manual
5/05	3.7	update to v3.7
3/07	3.7	C-only API, one-shot tasks discussion, and minor corrections
10/10	4.0	update to v4.0
10/11	4.0	updates for new scheduler and other changes
10/12	4.1	update to v4.1
1/14	4.2	update to v4.2 and substantial rewrite
5/14	4.2.1	update to v4.2.1 and fixes to examples
5/15	4.3	update to v4.3
2/16	4.3.1	heap
5/16	4.3.2	heap
11/16	4.3.2	misc
10/17	4.4	update to v4.4

Table of Contents

FOREWORD

This manual is a tutorial on the theory and use of the smx multitasking kernel. It is written both for programmers who are new to multitasking kernels and for programmers who have used smx or other kernels, in the past. It is intended not only to present smx features in an organized way, but also to help you understand how to apply them. A real-time multitasking kernel is a complex, multi-dimensional system. It simply is not possible to present one in a nicely linear, sequential manner. Hence, it is necessary to have either repetitious text or forward reference. We have chosen the latter in order to reduce your reading and to allow you to study subjects in the order you prefer.

Much has changed since smx was first released 25 years ago. Processors are much more powerful and memory is much cheaper. A commercial RTOS must bridge the gap from “small” processors (still very powerful, by historical standards) with limited memory on low-cost SoCs to very powerful processors with huge memories, caches, and complex architectures. Malware, unheard of then, is becoming an increasing problem. The complexity of embedded systems has increased at least an order of magnitude in the past 25 years.

Such large changes have ramifications on RTOS architecture and have resulted in major changes to smx in this v4.2 release. The biggest shift has been from wringing out maximum performance to increased functionality, ease of use, and safety. Some features of smx, which have been judged too complex, have been eliminated in favor of an easier-to-use API. Also emphasis has shifted to safer implementation of both the RTOS and the application, more error checking, more error recovery features, and fewer error-prone services. We judge these to be a good use of increased processor speeds and less-costly memories.

Due to modern high speed processors, kernel calls can be treated almost like C statements, without regard for their execution times. Hence, heavily using a strong, well-constructed kernel can lead to reduced coding and debug times and stronger systems. v4.2 is a major milestone on the path to a new smx, which supports this goal. I hope you enjoy working with it.

Ralph Moore
smx Architect

SECTION I: INTRODUCTION

This section provides an introduction to basic smx features, a discussion of objects, and an introduction to tasks.

references

(1) You should read the smx Reference Manual, in parallel with this manual, because it supplies details not covered here and provides additional examples, which may be helpful. The glossary at the end covers all smx terminology.

(2) The SMX Quick Start manual provides information about getting started using SMX. It covers installation, how to build and run the Protosystem, and how to begin development of an application. It is highly recommended to experiment with smx in an evaluation kit while studying this manual.

(3) See the SMX Target Guide for details about your CPU architecture and using your tools.

(4) The smxBase User's Guide provides information about low-level code development using smxBase and smxBSP.

(5) The smx++ Developers Guide is for C++ developers.

(6) The smxAware User's Guide provides directions for using this valuable debug tool.

(7) White papers at www.smxrtos.com/articles provide more technical discussion of some topics.

Chapter 1 Under the Hood

This chapter briefly introduces important features of smx in order to make the following chapters easier to understand. Detailed discussions of these features are provided in later chapters.

three-level structure

Interrupt service routines (ISRs) do the most time-critical operations, such as inputting or outputting data, re-enabling the interrupt mechanism in the device, and re-enabling interrupts.

Link service routines (LSRs) do the next most time-critical operations such as range-testing, comparing, scaling, and making messages. They provide a mechanism for deferred interrupt processing, timers, and making smx calls.

Tasks do the least time-critical operations, such as data and event processing, control, error code checking, protocol operations, etc.

smx services

smx services are implemented with system service routines (SSRs). SSRs are task-safe, which means that they cannot be preempted by tasks nor LSRs. ISRs are permitted to use only the smx_LSR_INVOKE() macro, certain low-level pipe functions, and smxBase functions. Thus ISRs do not access smx variables. As a consequence, interrupts are seldom disabled by smx, and then only briefly, producing low interrupt latency.

dynamic control blocks

All RTOSs use control blocks to store information about their system objects. Kernel services use the information in control blocks to perform their functions. smx uses dynamic control blocks, which are allocated at run time, rather than static control blocks, which are allocated at link time. This has certain advantages. See the next chapter for more discussion.

dynamically allocated memory regions

The fundamental unit of data management is the dynamically allocated region (DAR). smx comes with two pre-defined DARs: System DAR (SDAR) and Application DAR (ADAR). All dynamically allocated smx objects are placed in SDAR and dynamically allocated application objects such as the heap, stacks, block pools, etc. are placed in ADAR. This helps to protect smx objects from accesses to application objects. Also, since SDAR is small, it can be located in on-chip SRAM to improve performance.

messaging

smx provides conventional pipe messaging, which is generally known as *queue messaging*; it also provides *exchange* messaging. The latter is more powerful and safer than pipe messaging and it is the preferred method. It provides a no-copy method to anonymously send messages between tasks and LSRs. Exchange messaging fosters client/server designs. It adds new capabilities such as passing priorities, broadcasts, multicasts, and distributed message assembly.

accurate vs. precise time

In subsequent discussions, *accurate* means tick resolution, whereas *precise* means tick counter clock resolution. On a relatively slow 50MHz processor with a 100Hz tick, one tick time is equivalent to 500,000 instructions. Obviously a tick resolution is too coarse for time recording and measurements. smx timeouts and timing functions have tick resolution. Profiling, event timestamps, and time measurements have precise resolution. Precise resolution varies from one instruction to about 25 instructions, depending upon the processor.

performance factors

For ARM, the first four parameters are passed via registers, which is much more efficient than passing them via the stack. Hence most smx services are restricted to four or fewer parameters. This has the additional beneficial result of making smx calls simpler to use. Power comes from combining simple services effectively.

The large discrepancy between the speeds of modern processors and external memories, favors adding processor cycles to save memory accesses. For this reason, various techniques such as limiting size fields and counter fields and saving control block indices instead of handles are used to reduce the sizes of smx control blocks — especially when possible to restrict control blocks to multiples of cache line sizes. Control block pools are cache-line aligned, if external memory is being used.

error handling

smx services return FALSE or NULL upon failure. Peeking at the task error field:

```
err = SMX_ERR;
```

reveals the cause of the failure for the task. If err == SMXE_TMO (= 1), a timeout occurred; otherwise err is the error number (see xdef.h). This permits local error checking and recovery. For simplicity, local error checking is omitted in most examples in this manual and in the smx Reference Manual. The degree to which you implement local error checking in your code is your decision. However, it should be noted that smx provides a central error manager, smx_EM(), which records errors and does catastrophic error recovery. Also, smx services will not accept invalid handles. See the Error Management chapter for more information.

safety

Safety is a big concern for smx v4.2. In this release, much effort has been put into eliminating unnecessary complexity, which can cause confusion. Also there has been a focus on making features less susceptible to erroneous usage. For example of the following:

```
BlockRel(blk, pool);
BlockRel(blk);
```

the first is more error-prone because it requires getting both the blk handle and the pool handle correct. Releasing the block to the wrong pool is not possible in the second example. Generally speaking, more automatic functions are less likely to cause trouble. Also effort has been put into showing good coding practices in the examples. See Chapter 28 for more information.

abbreviated names

Abbreviations such as INF (infinite timeout) and ct (current task) or self are used in this manual to simplify both text and examples. In most cases the abbreviations are obvious. If you are not sure of a definition, see main.h. You may prefer to use the abbreviations in your code, unless you wish to keep them out of your name space. In that case, use the full names which define them (e.g. SMX_TMO_INF and smx_cf) in main.h.

configuration constants

Application configuration constants such as NUM_TASKS are defined in acfg.h. They are passed to the smx library via the configuration structure, smx_cf. Hence it is not necessary to recompile the smx library to add more tasks or make other changes. This can be done by changing NUM_TASKS in acfg.h and rebuilding the application. Run-time configuration is beneficial for working with pre-compiled smx libraries such as when using an smx evaluation kit. It also saves recompiling smx during debug sessions.

Fundamental configuration constants such as enabling profiling, stack scanning, and event buffer recording are present in xcf.h and require the smx library to be recompiled. In general, these configuration constants enable or disable corresponding code to be included in the smx library and thus affect its size.

programming languages

smx is written in C and thus can be compiled with C-only compilers. It can also be compiled with C++ compilers for interoperability with C++ code. In addition, we offer smx++, which provides a C++ API for smx.

For simplicity, assembly files are rarely mentioned herein. However, when necessary, application code can be written in assembly language and some assembly macros are provided by smx. For example, smx_ISR_ENTER() and smx_ISR_EXIT() are provided in the processor architecture include files. Also, discussions concerning header files apply equally to include files, if assembly code is present.

smxBase and smxBSP

smx is built upon smxBase, which is also the foundation for SMX middleware. smxBase, in turn, rests upon the smxBSP for the target processor. Services which are prefixed with sb_ come from smxBase or smxBSP. Please note that **these are not thread-safe**. See the smxBase User’s Guide for sb_function descriptions and see the smx Target Guide for detailed discussion of processor architectures and tools as they relate to smx, smxBase, and smxBSP.

Protosystem

The Protosystem is provided in every smx evaluation kit and smx delivery, in the APP directory. It is intended to allow you to get started more quickly, and it includes the necessary project file for your processor and tools. It should make and run immediately on the intended evaluation board. We recommend that you do these immediately and use the debugger to step through Protosystem code. This is a good way to get familiar with your processor, tools, and smx. Also work with smxAware to get familiar with it.

As you read the chapters which follow, you can add experimental code of your own to better understand explanations of how smx works. See the SMX Quick Start manual for more help with getting started.

esmx

esmx is a library of examples for smx. Be sure to download the latest version of esmx from www.smxrtos.com/eval, since it is steadily being improved. The ESMX directory consists of suites of examples for sections of smx. For example, etask.c has examples for tasks. Also included are project files and DebugTour.pdf. See the latter for instructions on how to build the esmx library and to link it to the Protosystem. When this has been done, running the Protosystem, causes the esmx example suites to run immediately after initialization.

Most examples are taken from the smx manuals and completed so that they will compile and run. The idea is to allow you to step through examples as you read the accompanying text in the smx manual. The way to do this is to find the example, then put a breakpoint at its start, and start the Protosystem. Be sure to take the esmx Debug Tour first, in order to learn how to get the maximum benefit from stepping through esmx examples.

It is also recommended to simply start from the beginning of a test suite of interest or esmx, itself, and step through all of the examples. This may save a lot of reading and give you some good ideas to solve your problems — i.e. read only when you do not understand an example.

esmx is also the name of the base example task. It and related code are contained in esmx.c. esmx is created and started by esmx_Init(). It runs after Idle has completed initialization and restarted itself with 0 (MIN) priority. esmx has priority 1. Common smx objects are also defined in esmx.c. See esmx.h for definitions of the short names used in esmx and the manuals.

Hopefully you will find examples that are close to what you need, and you will be able to copy and paste them into your code, just changing the names. In examples requiring additional tasks, or tasks which can be suspended or stopped, the example code, running under the esmx task, will create and start t2a, etc. These tasks are higher priority than esmx, so they preempt and run. When done, esmx resumes and deletes the tasks and other objects created for the example. This "cleanup" code is necessary so that examples do not interfere with each other. In your case, it is probably not desirable to do this, and that code should be left behind.

Chapter 2 Introduction to Objects

Kernels deal with objects. The kinds of objects supported, and how they are defined, determine how a kernel operates. For convenience, objects can be grouped into three categories:

(1) application objects

(2) system objects

(3) smx objects

In the first category are objects such as arrays, structures, and functions which you create via normal programming. These are not relevant to smx and are not discussed herein. The second category includes objects such as tasks, messages, and exchanges which you create via smx services. These produce the multitasking environment for the application. The third category are objects created and used by smx to do its job. These are mostly control blocks and pointers. Generally, smx objects are not of concern to you. However, they are discussed in this manual to help you to better understand how smx operates.

system objects

smx supports the following system objects:

(1) tasks

(2) block pools

(3) blocks

(4) messages

(5) exchanges

(6) semaphores

(7) mutexes

(8) timers

(9) event queues

(10) event groups

(11) pipes

(12) heap blocks

(13) task stacks

(14) interrupt service routines (ISRs)

(15) link service routines (LSRs)

control blocks

Control blocks provide the necessary information to control the above system objects. There is one type of control block for each of the system objects listed above, except for task stacks, ISRs, and LSRs, which have no control blocks. They are, in order:

(1)	**TCB**	task control block
(2)	**PCB**	pool control block
(3)	**BCB**	block control block
(4)	**MCB**	message control block
(5)	**XCB**	exchange control block
(6)	**SCB**	semaphore control block
(7)	**MUCB**	mutex control block
(8)	**TMCB**	timer control block
(9)	**EQCB**	event queue control block
(10)	**EGCB**	event group control block
(11)	**PICB**	pipe control block
(12)	**CCB**	chunk control block in heap
(13)	**CDCB**	chunk debug control block in heap
(14)	**HCB**	heap control block

A control block is created by the call which creates the corresponding system object. Except for heap chunk control blocks, which are spread throughout the heap, control blocks of the same type are grouped together into *control block pools*. For example, all TCBs are grouped into the TCB pool, which is called smx_tcbs. An exception to this is that XCBs, SCBs, ECBs, and EGCBs are grouped into one pool called the QCB pool. The number of each system object, and hence the size of each control block pool, is user-defined in acfg.h by constants such as NUM_TASKS.

Control blocks store information about their corresponding system objects. Kernel services use the information in control blocks to perform their functions. Most kernels use statically-defined control blocks, which are defined at link time. For them, an object is defined as follows:

```
TCB  taskA;
```

and the object is referenced by its address:

```
TaskStart(&taskA);
```

smx uses dynamic control blocks:

```
TCB_PTR  taskA;
```

and the task is referenced by its handle, taskA:

```
TaskStart(taskA);
```

taskA is actually a pointer, but it is called a handle and it is treated like a variable. It is necessary for the user only to define handles. Control blocks are handled automatically. On first use of a control block type, its pool is automatically allocated from SDAR. Hence all control blocks of the same type or related types are grouped together and are contiguous. This has efficiency, size, safety, and flexibility advantages over statically-allocated control blocks, which may be scattered throughout memory.

Control block pools are base pools, which are controlled by base pool control blocks (PCBs). The PCBs are statically defined in xglob.c — e.g. smx_tcbs. Each has fields to specify the first and last block pointer, a free list pointer, and the number and size of blocks in the pool. See the smxBase User's Guide for discussion of base pools.

If a particular system object is not used in an application, a pool for it will never be created. To save even more RAM, the pool control block defined in xglob.c can be deleted. Control blocks are cleared when released. This makes it easy to tell if a control block is in use, when debugging.

Other smx objects are principally counters, pointers, constants, and special queues. You will seldom, if ever, need direct access to these.

handles

System objects (tasks, messages, etc.) are identified by their *handles*. A handle is a pointer to the object's control block. For example:

```
MCB_PTR  amsg;
MSGA *mbp;
amsg = smx_MsgReceive(xchg, &mbp, NO_WAIT);
```

Here, amsg is declared to be a pointer to an MCB. amsg can be either a global or a local variable. smx_MsgReceive() returns the handle of the message. This value is stored in amsg. From this point on, amsg identifies the message and is to be used in subsequent smx calls.

```
smx_MsgSendPR(amsg, xchg ,PR0, NO_REPLY);
```

It is important to notice that amsg points to the message control block, not to the message itself. mbp points to the message.

Using handles is natural and does not entail any special actions on the part of the user. Debuggers show control block structures for handles as if they were static control blocks. A significant operational difference is that for static control blocks, kernel "create" services actually only initialize their fields. smx create services allocate control blocks, then initialize their fields. For static control blocks "delete" services actually just clear their fields. smx delete services clear the control block, release it back to its pool, then clear its handle so that it cannot be used again, by mistake.

smxBase uses static control blocks, as is appropriate for low-level static code using few control blocks.

naming system objects

It is good practice to give the best names to system objects. For example,

```
TCB_PTR  send_data;
send_data = smx_TaskCreate (send_data_main, 3, 0, NO_FLAGS, "send_data");
smx_TaskStart (send_data);
```

The function name, send_data_main, may never be used again (except for its prototype and definition), but the task name, send_data, is likely to be used frequently in both code and documentation.

Chapter 3   Introduction to Tasks

Tasks are workers

```
TCB_PTR      smx_TaskCreate(FUN_PTR code, u8 pri, u32 stksz, u32 flags, const char *name)
BOOLEAN      smx_TaskStart(TCB_PTR task)
```

what is a task?

A task is usually thought of as some amount of work which is started and finished within a short period of time. It is normally part of a larger job. For example, if the job is to assemble an engine, then mounting the water pump is a task of that job. If this were being done by a robot, someone would have written a program to control the robot to mount the water pump. As each engine passed by, the robot would perform this same task over and over, always using the same program. This illustrates the difference between a task and a program: a *task* is a useful bit of work; a *program* is a set of instructions for doing that work.

A *software task* is usually defined to be one pass through a particular section of code, which accomplishes part of a job. However, in modern multitasking systems, this definition is too narrow. Tasks frequently wait for more work when they finish and, in fact, they may never end. We can still think of a software task as being a portion of the work, but it is not confined to one repetition of the work. A task is same as a *thread of execution,* popularly known as a *thread.* Some people prefer this name; we prefer task.

An *active* task is one that has been created. Every active task has a task control block (TCB) associated with it. The TCB contains its main function pointer, its state, and other information necessary to control operation of the task. An active task also has a timeout and may or may not have a stack.

task types

smx supports two task types: *normal* and *one-shot*. Normal tasks have permanent stacks, whereas one-shot tasks borrow temporary stacks from a stack pool. Normal tasks, which are the type supported by other kernels, are somewhat easier to use and are the subject of the rest of this chapter. One-shot tasks offer the advantage of sharing stacks and are discussed in the One-Shot Tasks chapter.

scheduling tasks

There are three basic algorithms for controlling which task runs at a given time:

(1) round-robin

(2) time slicing

(3) preemption

smx supports all of these.

round-robin

The first algorithm, *round-robin*, is a *cooperative* algorithm. This means that each task voluntarily gives up the processor at one or more points in its code. This is easily done as follows:

```
smx_TaskBump (smx_ct, NO_CHG);
```

which moves smx_ct (the currently running task) to the end of its priority level and causes the next task at that level to begin running. If there is no other task at the same level, then smx_ct will continue running. When using round-robin scheduling, all tasks are normally at the same priority level, hence each runs in turn, until it gives up the processor. Note that tasks do not necessarily run in a fixed order, because a task may be waiting for a resource (e.g. at an exchange), and thus it will be skipped over.

Round-robin scheduling can be used in systems where tasks have different priorities, provided that higher-priority tasks give up the processor regularly so that lower-priority tasks can run. This would be useful, for example, if one or more short tasks were of urgent priority (e.g. controlling stepper motors) and the remaining tasks were of average priority.

Round-robin scheduling is typically used only in very simple systems. It may be a good intermediate step when upgrading from a *superloop* implementation, because it maintains a similar structure, yet introduces improved control.

time slicing

Time slicing is usually implemented to achieve “fairness” among tasks of the same priority. smx implements time slicing among tasks at priority 0. It is enabled by setting SMX_CFG_TIMESLICE in xcfg.h to the number of ticks per time slice. If set to 0, time slicing is disabled. Time slicing is performed by bumping the current task, if it has priority 0, to the end of the 0 level. This is not an accurate mechanism and it should not be assumed that each task will run exactly one time slice. It is intended for background tasks that need to share available processor time equally.

Time slicing is not allowed above priority 0 because doing so can result in starving lower-priority tasks for processor time. Hence, time slicing is safe only at priority 0, where no tasks can be blocked. If it is necessary to limit run times of higher priority tasks, see the run-time limiting section in the Precise Profiling chapter.

preemption

The third algorithm, *preemption*, is the most natural for embedded systems and produces the best results, especially for *hard real-time* requirements. Basically, the algorithm is that the *top task* runs at all times. The top task is the highest-priority, longest-waiting task that is in the *ready queue*. The only time that this algorithm does not apply is if the current task is *locked*. A locked task runs until it unlocks, suspends, or stops itself, at which time the top task will take over.

Task switching occurs whenever a new task becomes the top task and ct is not locked. This can happen due to:

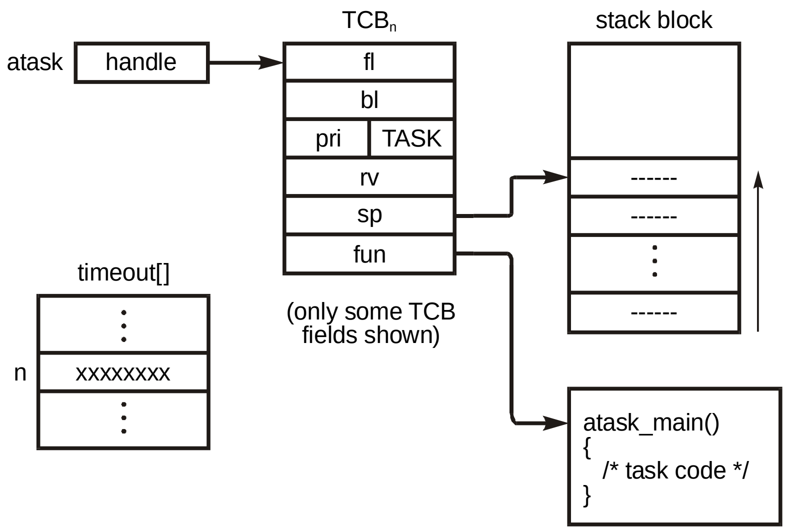
(1) an SSR called from a task

(2) an SSR called from an LSR, due to an interrupt

A preemptive system is compatible with an interrupt-driven system. It shares the advantages of responsiveness and directness, but it also shares the disadvantages of resource conflicts and the need for reentrant code. Sometimes a combination of round-robin or time slicing plus preemption for a few high-priority tasks may be the best solution.

how does smx handle tasks?

Tasks are system objects which are regulated by task control blocks (TCBs). A task can be visualized as follows:



The objects shown above comprise a task. Central to these is the TCB, of which only a few fields are shown. The first two TCB fields, the forward link (fl) and the backward link (bl), are used to link the task into wait queues. When the task is not in a queue, the forward link field is NULL (0) and the backward link is undefined. The next two fields shown are the priority field and the control block type field, which identifies this as a TCB. The rv field stores the return value from the last SSR call (such as a block handle).

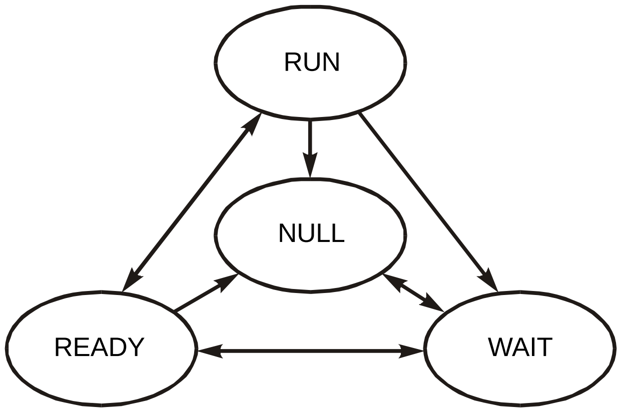
If the task has been assigned a stack, stp and sbp will point to the top and bottom of the stack, respectively. (For simplicity, these have been omitted in the above diagram.) The TCB stores the processor stack pointer in the sp field when a task is suspended. If the task has not yet run or has been stopped, sp is NULL. There also is a register save area pointer, pointing to where registers are saved when a task is suspended. The fun field points to the code that will run when the task is started. To see all TCB fields, see the TCB structure definition in xtypes.h.

For each task, there is a 31-bit timeout stored in the smx_timeout array. Timeouts prevent unbounded waits. See the Timing Chapter for more on timeouts.

Note that the task's TCB is pointed to by the handle stored in atask. As previously described in the Objects Chapter, this handle identifies the task.

task states

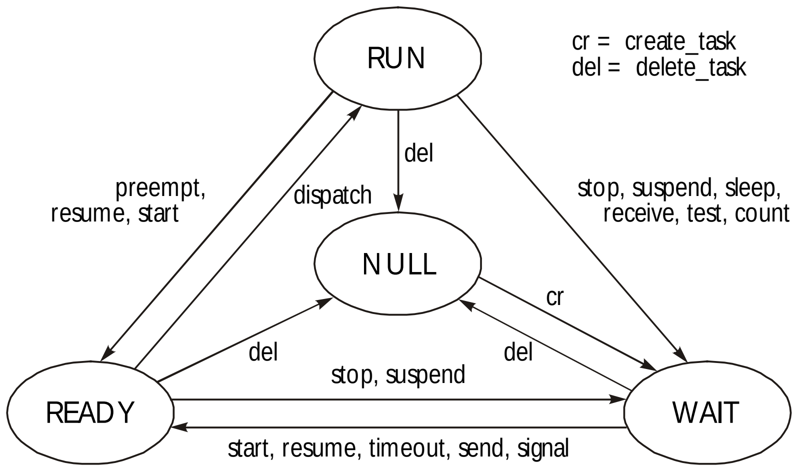
A task may be in one of four states:



A task is in the NULL state prior to being created or after being deleted; it is in the WAIT state when it is waiting for an event such as receiving a message, a signal, or a timeout; it is in the READY state when it is ready to run; and it is in the RUN state when it is actually running. Only one task, at a time, may be in the RUN state, but any number of tasks may be in the other states. A task's state can be determined from the state field in its TCB.

state transitions

Transitions from one state to another are caused by SSRs and smx scheduler actions. The following diagram shows state transitions for various operations:



Except for preempt and dispatch, which are scheduler operations, and except for timeout, which is an LSR operation, each name represents a system service implemented by a system service routine (SSR). To interpret this diagram, it is helpful to realize that only the task in the run state (i.e. the current task) can call smx services. When the task is in another state, it cannot call system services. However, LSRs invoked by ISRs can also call most smx services and smx_TimeoutLSR() resumes tasks due to timeouts.

The NULL state has the simplest transitions: it can be entered from any state by smx_TaskDelete(); it can be exited only to the wait state, by smx_TaskCreate(). The RUN state can be entered only due to a scheduler dispatch from the READY state. It is exited to the READY state due to preemption by a higher priority task or due to its calling certain task services. It is exited to the WAIT by its calling system services that suspend or stop it. The transition from WAIT to READY is caused by system services called by the current task that resume or start the task.

Task state transitions can seem complicated, at first. However, understanding them is crucial to understanding how smx works.

task main function

The function which runs when a task is started is known as the task's main function. It is comparable to main() of a C program. Its address is stored by smx_TaskCreate() in tcb.fun. Normal task main functions are defined like this:

```
void  t2a_main(void);
```

t2a_main() need not be the only function associated with t2a. It can, of course, call other functions, and it can even be replaced:

```
void  t2a_main(void)
{
    other_function();
    t2a->fun = t2a_run;     /* change t2a main function */
}
```

The next time t2a is started, t2a_run() will run instead of t2a_main():

```
void  t2a_run(void)
{
    //...
}
```

Note: The preferred way to restart a task with a new main function is as follows:

```
smx_TaskStartNew(t2a, 0, 2, t2a_run);
```

creating tasks

Creating a task is done as follows:

```
TCB_PTR   t2a;
void  appl_init(void)
{
    t2a = smx_TaskCreate(t2a_main, PR2, 200, NO_FLAGS, "t2a");
}
```

t2a is the name of the task. It means "task a at priority 2". (If you cannot come up with better names, this system is pretty good and used in many places in this manual.) As discussed, previously, t2a is defined as a TCB pointer, and it stores the task handle returned by smx_TaskCreate(). In this case, t2a is being created by the application initialization function, but tasks can be created in any function, task, or LSR.

Note that t2a is defined as a global variable, so that functions in other files can use it. As a general rule, task handles should be defined globally so that they are accessible by all tasks and LSRs.

In the above example, t2a is created with t2a_main() as its code, PR2 as its priority, a stack of 200 bytes is allocated from the heap, no task flags are set, and the task is named "t2a".

starting tasks

When a task is first created, it is in a *dormant* state (i.e. timeout inactive and not in any queue). It will stay in this state, potentially forever, until another task or LSR starts it with:

```
TCB_PTR t2a;

smx_TaskStart (t2a);
```

This puts t2a into the ready queue, but it does not actually run the task. That is done by the scheduler.

alternate main function

smx also accepts a main function of the form:

```
void  t2a_main(u32 par);
```

To create a task with this type of main function:

```
t2a = smx_TaskCreate((FUN_PTR)t2a_main, PR2, 200, NO_FLAGS, “t2a”);
```

The only difference is the FUN_PTR typecast. To start the task:

```
smx_TaskStartPar(t2a, par);
```

This main function allows you to pass in a parameter to the task initialization. See example 2, below, for a case where this is useful.

task example 1

The following example shows how to create and start three tasks.

```
TCB_PTR t2a, t2b, t3a;

void  appl_init(void)   /* non-preemptible */
{
    t2a = smx_TaskCreate(t2a_main, P2, 100, NO_FLAGS, "t2a");
    t2b = smx_TaskCreate(t2b_main, P2, 100, NO_FLAGS, "t2b");
    t3a = smx_TaskCreate(t3a_main, P3, 100, NO_FLAGS, "t3a");
    smx_TaskStart(t2b);
    smx_TaskStart(t2a);
}

void  t2a_main(void)
{
    smx_TaskStart(t3a);
    fun2a();
}

void  t2b_main(void)
{
    /* do task initialization here */
    while (1)    /* endless loop */
    {
        fun2b();
        smx_TaskSuspend(self, 2);  /* wait 2 ticks */
    }
}
```

```
void  t3a_main(void)
{
    fun3a();
}
```

This very simple example does nothing useful. The appl_init() function is assumed to be non-preemptible. It creates t2a, t2b, and t3a and it starts t2b and t2a. Both are put into priority 2 level in rq and t2b is first. Note that t3a is dormant, because it has not been started. Hence, t2b will run first, initialize itself, then go into an infinite loop. In the loop, it calls fun2b(), then waits 2 ticks. This allows task t2a to run, which starts task t3a. t3a preempts t2a immediately and calls fun3a(). Then task t3a *autostops*, which causes it to become dormant again. Now task2a resumes running (since t2b is still waiting) and it calls fun2a(), then it also autostops and becomes dormant.

Although very simple, this example illustrates some important points about multitasking:

(1) Although t3a was created with the other tasks, it remains dormant until it is started.

(2) A task can be created in one place and started in another.

(3) t2b runs ahead of t2a because it was started first. The longest waiting task at a priority level is called the *top* task and it runs first at that level.

(4) Tasks can autostop and become dormant. If fun2a() were to start t3a again, it would start from its beginning and run again.

task example 2

The following example illustrates the three different scheduling methods, using arrays of tasks:

```
#define SMX_CFG_TIMESLICE       1       /* in xcfg.h */

TCB_PTR  tts[T],  trr[R],  t2a;

void  appl_init(void)    /* non-preemptible */
{
    int n;

    for (n = 0;  n < T; n++)
    {
        tts[n] = smx_TaskCreate((FUN_PTR)tts_main, P0, 100, NO_FLAGS, NULL);
        smx_TaskStartPar(tts[n], n);
    }

    for (n = 0;  n < R; n++)
    {
        trr[n] = smx_TaskCreate((FUN_PTR)trr_main, P1, 100, NO_FLAGS, NULL);
        smx_TaskStartPar(trr[n], n);
    }

    t2a = smx_TaskCreate(t2a_main, P2, 100, NO_FLAGS, NULL);
    smx_TaskStart(t2a);
}

void  tts_main(u32 n)       /* time-sliced tasks */
{
    /* do task n initialization here */
    while (1)
    {
        /* do task n operations here */
    }
}

void  trr_main(u32 n)        /* round robin tasks */
{
    /* do task n initialization here */
    while (1)
    {
        /* do task n operations here */
        smx_TaskBump(self, NO_PRI_CHG);  /* bump self to end of rq level */
    }
}

void  t2a_main(void)   /* preemptive task */
{
    /* do task initialization here */
    while (1)
    {
        fun2a();
        smx_TaskSuspend(self, 5); /* wait 5 ticks */
    }
}
```

The time slice period is set to 1 tick in xcfg.h; it normally is 0 (off). The initialization code creates an array of T time slice tasks, tts[n], all of which use the tts_main() function, and starts each one. These tasks are all at priority 0, as they must be for time slicing. Initialization then creates a similar array of R round-robin tasks, trr[n] and starts each one. These are at priority 1. Finally initialization creates and starts t2a at priority 2.

Since t2a has highest priority, it runs first, then waits 5 ticks. From this point on, it will preempt every 5 ticks. Next, the round robin tasks run, in order — trr[0], trr[1], etc. After running, each bumps itself to the end of rq level 1. (Note: smx_TaskBump() is discussed in More on Tasks Chapter.) This creates round-robin scheduling. It is assumed in this example that the round robin tasks wait for events and do not run all the time. This allows the time slice tasks to run. Each runs for one tick, and then smx moves the task to the end of rq level 0.

Although very simple, this example illustrates some additional important points about multitasking:

(1) Tasks can share identical code. smx_TaskStartPar() allows passing in a parameter which identifies to the code which task is running. This concept takes some thought. What is happening, is that the task is providing the context — stack, register contents, etc. — for the code to run. The idea of tasks sharing code is not a far-out idea; it is commonly used. Note that the parameter passed in could be a pointer to a task-specific structure containing more information, rather than just an integer.

(2) It is practical to create arrays of tasks. However, it is not necessary to do so for round-robin or time slicing — the tasks could have different names and different main functions.

(3) The timesharing tasks will *starve* (i.e. not run) if the round-robin tasks never stop running.

summary

In this chapter we have covered the basics of what tasks are, how smx manages them, and how to create and start them. Before delving deeper into tasks, we will cover memory management and intertask communication. See the chapters which follow them for more information on tasks:

(1) Tasks

(2) One-Shot Tasks

SECTION II: SERVICES

This section covers smx services. Each chapter deals with a different functional area of smx. Basic services are presented first, followed by advanced services. If you are new to multitasking, we recommend skipping the advanced services on the first reading. The basic services should be enough to get started and enough for simple applications. This section is intended to give you a good overview of smx before proceeding to the development section, which will get you going on your project.

If you are an experienced user, you may prefer to skim the basic services in order to pick up smx differences, and then study the advanced services. Advanced smx services and techniques are for dealing with problems, such as reducing memory usage, improving performance, improving security, safety, and reliability, and dealing with difficult design problems. smx provides novel ways of dealing with complex embedded system issues. Effective use of these features can make a big difference in project success.

Chapter 4   Memory Management

introduction

Tasks need memory to work in. Dynamic memory management provides the capability to obtain memory, when needed, and to release it when no longer needed. This flexibility makes systems more adaptable and easier to program and maintain. It usually reduces the memory requirement, since memory can often be released when not needed by one task and re-used by another.

smx supports four basic forms of memory management:

(1) dynamically allocated regions (DARs)

(2) block pools

(3) heap

(4) stacks

DARs provide the basic dynamic memory structure. They are used for one-time allocation of blocks, block pools, control block pools, stacks, heap, etc.

Block pools provide simple, fast memory management. Since the blocks are all the same size, block pools are not subject to fragmentation, as is a heap. Also they provide fully deterministic memory allocation, which is not possible with a heap. Uses include: task work spaces, task status and state information, data storage, messages, I/O buffers, and the stack pool.

Generally speaking, block pools are best for predictable numbers of specific-size blocks, whereas the heap is best for unpredictable numbers of random size blocks.

DARs, base block pools, and smx block pools are discussed in this chapter. The smx heap and stacks are discussed in the chapters that follow.

DARs

```
u8 *        sb_DARAlloc(SB_DCB_PTR darp, u32 sz, u32 align)
BOOLEAN     sb_DARFreeLast(SB_DCB_PTR darp)
BOOLEAN     sb_DARInit(SB_DCB_PTR darp, u8 *pi, u32 sz, BOOLEAN fill, u32 fillval)
```

As indicated by the “sb_” prefix, these functions are part of smxBase. See the smxBase User’s Guide for full descriptions of these functions and how to use them.

Briefly, each DAR has a DAR control block (DCB) that is statically allocated. Space for the DAR is also statically allocated — usually by the linker command file. **sb_DARInit()** is used to initialize a DAR as follows: darp is the address of the DCB, pi points to the memory area, sz is its size, and fill specifies whether to fill it with fillval. sb_DARInit() initializes the DCB and does the filling, if any. **sb_DARAlloc()** gets a block from the specified DAR of sz bytes and aligned on an align-byte boundary. Using align helps to optimize performance. It is normally at least 4 for ARM, and it is SB_CACHE_LINE size if a data cache is being used. **sb_DARFreeLast()** allows returning a block if the rest of an operation fails after allocating a block.

DAR usage by smx

DARs allow smx objects to be separated from application objects. smx objects are put into System DAR, SDAR, and application objects are put into Application DAR, ADAR. This has these advantages:

(1) It helps to protect smx objects from application bugs. This is beneficial during debug and it improves reliability in the field.

(2) smx performance can be improved by locating the relatively small SDAR in on-chip SRAM.

SDAR is used by smx for all smx control blocks and other system objects, such as the LSR queue and error buffer; ADAR is used for the task stack pool, the smx heap, block pools, and other application objects. SDAR should not be used for application objects. Also, it is recommended SDAR and ADAR be located in different areas of memory, if possible.

SDAR and ADAR are defined as arrays in mem.c and are put into separate named sections, .smx_sdar and .smx_adar, so they can be independently located by the linker command file. It is also possible to use fixed addresses in the sb_SDAR_ and sb_ADAR_ macros.

Additional DARs can be defined for special purposes, such as dealing with different kinds of memory. For example, ADARF (F = fast) might be defined to locate task stacks in fast memory in order to improve performance.

DAR services are not SSRs. Hence they are not task-safe and must be used with care from tasks (e.g. lock the task or disable interrupts).

base block pools

```
BOOLEAN     sb_BlockPoolCreate(u8 *dp, PCB_PTR pool, u16 num, u16 sz)
BOOLEAN     sb_BlockPoolCreateDAR(SB_DCB *dar, PCB_PTR pool, u8 num, u16 sz, u16 align)
u8 *        sb_BlockPoolDelete(PCB_PTR pool)
u8 *        sb_BlockGet(PCB_PTR pool, u16 clrsz)
BOOLEAN     sb_BlockRel(PCB_PTR pool, u8 *dp, u16 clrsz)
```

Base block pools are part of smxBase. They following is a brief overview of how they are used with smx. For detailed information on base block pools, see the smxBase User's Guide.

base block pool usage by smx

Base block pools are used by smx primarily for its control blocks and for the stack pool. They are also used by smx++ to overload the *new* and *delete* operators and can be used by applications, when speed is important. The above services are not SSRs. Hence they are not task-safe and must be used with care from within a task (e.g. lock the task or disable interrupts). For applications, they are best used in ISRs and other low-level code.

Note: The examples below show how smx uses base block pools. These are not operations that the application ever needs to do, but they serve as examples of usage while also showing a little about the internals of smx.

Each base pool is controlled by a statically-defined pool control block (PCB) as follows:

```
PCB    poolA;
```

sb_BlockPoolCreate() creates a block pool from a pointer to a free memory area and from the address of the PCB to be used to control it. The block pool will contain num blocks of sz bytes. This service can be handy for creating a pool from a static area or from a block allocated from the heap:

```
u8 dp[2000];
  -OR-
u8 *dp = (u8*)smx_HeapMalloc(2000);

sb_BlockPoolCreate(dp, &poolA, 100, 20, "poolA")
```

creates a pool of 100 20-byte blocks, starting at dp. The blocks are singly-linked into a free list starting at poolA.pn, and the first word of the each block points to the next free block.

sb_BlockPoolCreateDAR() automatically allocates an aligned block pool from the specified DAR. This service provides more automatic and safer operation. It is used by smx to create control block pools:

```
PCB    smx_tcbs;
sb_BlockPoolCreateDAR(sb_sdar, &smx_tcbs, NUM_TASKS, sizeof(TCB), SB_CACHE_LINE,  "smx_tcbs")
```

In the above example, a pool of task control blocks (TCBs) is created in SDAR; it is controlled by smx_tcbs, which is a static PCB. NUM_TASKS (see acfg.h) specifies the number of TCBs in the pool, and the pool is aligned on a cache-line. smx control block pools and other objects are never deleted, so sb_BlockPoolDelete() is not used by smx.

sb_BlockGet() is used to get a block, such as a TCB, from a pool:

```
TCB_PTR  taskA;
taskA = (TCB_PTR)sb_BlockGet(&smx_tcbs, 4);
```

In this case, a TCB is removed from the TCBs pool and its first 4 bytes are cleared (to get rid of the link address). The address of the block is loaded into the task handle, taskA.

sb_BlockRel() can be used to release a block, such as a TCB, back to its pool, given its handle.

```
sb_BlockRel(&smx_tcbs, taskA, sizeof(TCB));
```

This returns the TCB pointed to by task and clears it, except for the free-pool link in its first 4 bytes.

Blocks typically are not returned in the reverse order of that in which they were obtained. Hence, over time the free list becomes scrambled and bears no relationship to block order by memory address. (This can be disconcerting when tracing a block free list via a debugger.)

smx block pools

```
BCB_PTR     smx_BlockGet(PCB_PTR pool, u8 **bpp, u32 clrsz)
BCB_PTR     smx_BlockMake(PCB_PTR pool, u8 *bp)
u32         smx_BlockPeek(BCB_PTR blk, SMX_PK_PARM par)
PCB_PTR     smx_BlockPoolCreate(u8 *p, u8 num, u16 size, const char *name)
PCB_PTR     smx_BlockPoolCreateDAR(SB_DCB_PTR dar, u8 num, u16 size, u16 align, const  char *name)
u8 *        smx_BlockPoolDelete(PCB_PTR *pool)
u32         smx_BlockPoolPeek(PCB_PTR pool, SMX_PK_PARM par)
BOOLEAN     smx_BlockRel(BCB_PTR blk, u16 clrsz)
u32         smx_BlockRelAll(TCB_PTR task)
u8 *        smx_BlockUnmake(PCB_PTR *pool, BCB_PTR blk)
```

Except for low-level code such as ISRs it is recommended that application code use smx block pools, for the following reasons:

(1) Block pool services are preemption-safe.

(2) A block is automatically freed to its correct pool.

(3) A block is automatically freed if its owner task is deleted.

(4) Pool information can be obtained via a block’s handle.

(5) They are more easily created and deleted than base block pools.

As can be seen from the above API, except for a few additional services, the smx block pool API is similar to the base pool API. The main difference is that smx blocks have both a handle and a data pointer. Manipulating blocks by their handles provides safe, automatic operation.

smx block pool services are implemented as SSRs. Thus they are task-safe and LSR-safe. They are effectively ISR-safe since ISRs are not allowed to call SSRs and do not access internal smx variables. Preemption safety is important when operating in a multitasking environment. It is easy to forget that a task, unless locked, can be preempted at any time, and that, even if it is locked, can be interrupted at any time. Unless proper precautions are taken, Murphy's law guarantees that a task will be preempted at the worst possible time.

creating and deleting smx block pools

The block pools created by smxBase and by smx are identical, except that pools from smxBase have statically defined PCBs and pools from smx have dynamically allocated PCBs from the PCB pool, smx_pcbs. The difference in usage is that for smxBase, as shown previously, a PCB must first be defined, then its address is used. This is appropriate for simple, low-level code and is consistent with other smxBase objects. For data block pools created by smx, a PCB is automatically allocated from the PCB pool when the data block pool is created, then its handle (PCB_PTR) is used. This pool is automatically created the first time a PCB is needed. Dynamic PCBs facilitate creating smx block pools, when needed, and deleting them, when not needed. This is consistent with other smx objects, such as tasks, semaphores, etc.

An *smx block* consists of two parts: (1) its data block, which contains the block's data, and (2) its block control block (BCB), which contains information used by smx to manage the block. The data block comes from a data block pool and the BCB comes from a BCB pool. The two are joined by smx_BlockGet() or smx_BlockMake() and they are separated by smx_BlockRel(), smx_BlockRelAll(), or smx_BlockUnmake(). For these operations, it does not matter how the data block pool was created. This is an important feature, which creates flexibility for block usage.

The BCB pool is a base block pool, which contains free BCBs. It uses a static base PCB called smx_bcbs and it is created when the first smx block pool is created:

```
PCB  smx_bcbs;
sb_BlockPoolCreate(p, &smx_bcbs, NUM_BLOCKS, sizeof(BCB));
```

where NUM_BLOCKS is defined in acfg.h, for the application. The above operation is performed automatically by smx.

A block pool can be created using smx, as follows:

```
#define NUM 100;
#define SIZE 20;
PCB_PTR poolA;
u8 p[2000]   - -OR--   u8 *p = (u8*)smx_HeapMalloc(2000);
poolA = smx_BlockPoolCreate(p, NUM, SIZE, "poolA");
```

This creates a pool of 100 20-byte blocks starting at p, with poolA as the handle (PCB_PTR) and named "poolA". Naming is a convenience for debugging and for smxAware. NULL can be used, instead, if no name is desired. p, num, and size must not be 0, and size must be a multiple of SB_DATA_ALIGN, which is 4 for ARM. This call can fail for other reasons, such as inability to create the PCB pool or the BCB pool or to get a PCB. See the Reference Manual for details.

The above pool create can be a handy way to create a block pool, if you wish to allocate space for it statically or if you want to be able to release the pool back to the heap when done with it. These are at opposite ends of the spectrum. As with base block pools, the user is responsible to make sure that the block is aligned, as desired, and that there is sufficient free memory for the pool.

A third way to create a block pool using smx is:

```
PCB_PTR poolA;
poolA = smx_BlockPoolCreateDAR(&adar, NUM, SIZE, SB_CACHE_LINE, "poolA");
```

In this example, space for the pool is automatically allocated from ADAR and aligned on an SB_CACHE_LINE-byte boundary. The pool is the same size and name as in the previous example. **smx_BlockPoolCreateDAR()** can fail for the same reasons as above, as well as insufficient DAR space for the pool

Note that there are similarities, as well as subtle differences, between base pool creates and smx pool creates. In fact, the latter call the former to create the actual pools.

A pool created by either of the above functions may be deleted by **smx_BlockPoolDelete()**, which returns a pointer to the pool block, releases its PCB back to the PCB pool, and clears its handle. This pointer can be used to free the block back to the heap or to re-purpose it, if it is a static block or a DAR block:

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

Delete fails if the pool handle is invalid. This could occur if the pool is actually a base pool. There is not much point in releasing a pool block that did not originate from the heap — releasing only the PCB does not gain much memory. However, it might be useful to give a static or DAR pool a different name or to reuse the memory space.

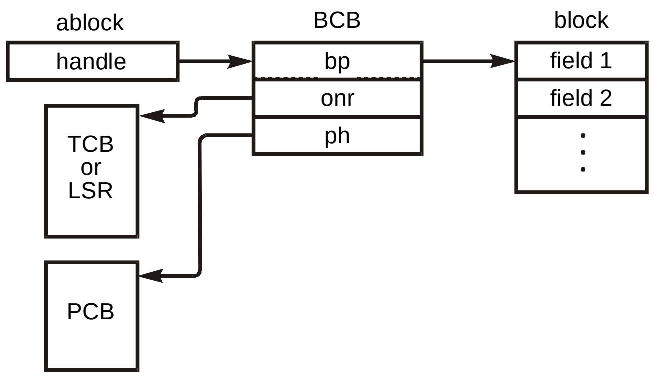
getting and releasing smx blocks

smx_BlockGet() is use to get a block from a block pool:

```
BCB_PTR blk;
u8*     bp;

blk = smx_BlockGet(poolA, &bp, 4);
/* fill block using bp */
```

In this case a data block is obtained from poolA, a BCB is obtained from the BCB pool, and the BCB is linked to the data block. The BCB handle is returned in blk. This handle will be used by all subsequent smx services to handle the block. In addition, the first 4 bytes of the data block are cleared and its address is loaded into bp. bp is a work pointer defined by the user which is used to load the block with data.

The structure of an smx block is as follows:



Note that the BCB is small — only 12 bytes. Hence, its overhead on a typical data block of 100 bytes size is not great. This facilitates having hundreds of smx blocks in a system. However, if the BCB overhead is too great, then base blocks can be used instead (i.e. sb_BlockGet()) as long as their limitations are understood and allowed for.

smx_BlockGet() can be called from a task or an LSR. The handle of the requesting task (smx_ct) or the address of the LSR (smx_clsr) is stored in the onr field of the BCB. This indicates that the block is in use. (Note that whether in use or free, a BCB is still physically in the BCB pool, and bp either points to a data block or to the next BCB in the free list. Hence, the only way to distinguish a block in use from a free block is via the onr field.) The block pointer, bp, and the pool handle, ph, are also stored in the BCB. ph is the pool handle (i.e. it points to the PCB for the pool.) Additional information is stored in the PCB, such as NUM and SIZE.

smx_BlockGet() is aborted, with a NULL return, if the pool is invalid or either the data block pool or the BCB pool is empty. In the latter cases, the task cannot wait at the pool for a block. Instead, the task should wait at a resource semaphore, as shown below:

```
PCB_PTR poolA;
SCB_PTR sr;
TCB_PTR t2a;

poolA = smx_BlockPoolCreateDAR(&adar, NUM, SIZE, SB_CACHE_LINE, "poolA");
sr = smx_SemCreate(RSRC, NUM, "sr");

void t2a_main(void)
{
    BCB_PTR blk;
    u8 *bp;

    smx_SemTest(sr, INF);
    blk = smx_BlockGet(poolA, &bp, 4);
    /* use bp to access data block here */
    smx_BlockRel(blk, SIZE);
    smx_SemSignal(sr);
}
```

In the above example, the resource semaphore, sr, is initialized to a count equal to the number of blocks, NUM, in poolA. When a task, such as t2a, needs a block, it tests sr. If a block is available, sr’s internal count will be > 0, the test will pass, and the internal count will be decremented. If no block is available, t2a will be suspended on sr for a block to become available. Any number of tasks may wait at sr in priority/arrival order. When t2a is done with blk, it releases it back to poolA and signals sr. This allows the current top waiting task to get the block from poolA.

The reason that a task cannot directly wait at a pool is because PCBs do not have forward and backward links. To do so would create too much difference between base pools and smx pools. If it is not desired to use a resource semaphore, then the other alternative is to fail and possibly retry later:

```
if ((blk = smx_BlockGet(poolA, &bp, 4)) != NULL)
{
    /* use bp to access the data block here */
    smx_BlockRel(blk, SIZE);
}
else
    /* try again later */
```

Note that in the first example, bp is defined as a local variable for the task. This is a good practice because it helps to keep task data areas separate from each other.

A useful trick is to declare bp as a pointer to a structure:

```
struct  {
  u32*  time;
  u32   d[N];
}  *bp;
```

Then the data block can be more easily accessed:

```
bp->time = smx_SysEtimeGet();
while (i = 0; i < N; i++)
    bp->d[i] = data_in(portA);
```

smx_BlockRel() is used to release a block, given its handle, blk:

```
smx_BlockRel(blk, SIZE);
```

blk is released to its pool, if it has one, and its BCB is released to the BCB pool. BlockRel() will fail and return FALSE if blk is invalid (i.e. not a BCB handle). smx_BlockGet() and smx_BlockRel() are interrupt safe with respect to sb_BlockGet() and sb_BlockRel(). This means that these smx SSRs can be used from tasks at the same time that the base functions are being used from ISRs on the same pool. So, for example, ISR1 could get a block from poolA at the same time that t2a was returning a block to poolA.

smx_BlockRelAll() releases all blocks owned by a task and returns the number released. To do this it searches the BCB pool for a BCB whose owner is the task, then calls smx_BlockRel() to release that block. This process is repeated until all BCBs have been checked:

```
u32  num;
num = smx_BlockRelAll(taskA);
```

Blocks are not cleared because blocks owned by a task may be of various sizes. smx_BlockRelAll() will fail if the task handle is invalid. This service is used when a task is deleted by smx_TaskDelete(). It may also be useful when a task is stopped in order to release blocks that may not be needed for a long time, and it may be useful in recovery situations. smx_BlockRelAll() is interrupt-safe.

making and unmaking smx blocks

smx_BlockMake() converts a *bare* block (i.e. one with no BCB) to an smx block. The bare block can be from a base pool, a DAR, the heap, or can be a static block. For example:

```
u8 *bp;
BCB_PTR blk:

bp = sb_BlockGet(poolA, 4);
...
blk = smx_BlockMake(poolA, bp);
```

In this example, a base block is obtained from base block poolA, then made into an smx block. The result is no different from:

```
blk = smx_BlockGet(poolA, &bp, 4);
```

However, smx_BlockGet() cannot be used from an ISR because it is an SSR. Hence, Get() + Make() is used by an ISR to get a base block, fill it, and pass it on to an LSR, which makes it into an smx block. The smx block can then be passed on to a task for processing — possibly by passing its handle via a pipe. (See the Pipes chapter for an example of this.)

Whichever way blk is obtained, it can be released by a task, as follows:

```
smx_BlockRel(blk, SIZE);
```

produces the same result — i.e. it will be released to its pool if it has one. It is important to note that the block was obtained by an ISR, in the first case, and released by a task. The former used an smxBase function and the latter used an smx service (SSR). Hence an ISR and a task are able to easily share a block pool. This facilitates no-copy input.

In the case of a block that is not in a pool, Make() is used as follows:

```
u8  bp[NUM];
blk = smx_BlockMake(NULL, bp);
```

NULL is loaded into the pool handle of the BCB to indicate that the block has no pool. In this case,

```
smx_BlockRel(blk, SIZE);
```

releases the BCB and clears the block, but it does not attempt to release it to a pool. Note that a static block could be located in ROM as well as in RAM. In that case, it would be a read-only block and SIZE should be 0 when releasing it. A ROM block may not seem to be of much use, but such a block could be a table passed from task to task. A receiving task could get the table pointer from:

```
typedef struct  {
    /* table fields */
} T1 *tp;

tp = (T1*)smx_BlockPeek(blk, SMX_PK_BP);
```

and then use tp->field operations to access information telling it how to process data blocks that it is receiving from other sources.

smx_BlockUnmake() converts an smx block to a bare block. It does so by releasing its BCB back to the BCB pool. Unmake() is used as follows:

```
BCB_PTR blk;
PCB_PTR poolA;
u8 *bp;

blk = smx_BlockGet(poolA, &bp, 4);
...
u8  *bp1;
bp1 = smx_BlockUnmake(&poolA, blk);
```

In this case, blk might be obtained and loaded by a task, then unmade into a bare block by an LSR, and passed to an ISR. The block unmake loads a global pointer, bp1, for use by the ISR. This is because bp, used to load the block, is probably a local task pointer. When the ISR has completed sending the data in the block it can release it using:

```
sb_BlockRel(poolA, bp1, SIZE);
```

This will release the data block back to its smx pool. It is important to note that the block was obtained by a task and released by an ISR. The former used an smx service (SSR) and the latter used an smxBase function. Hence a task and an ISR are able to easily share a block pool. This facilitates no-copy output.

smx_BlockMake() and smx_BlockUnmake() are complementary — one reverses the other's actions. smx_BlockGet() and smx_BlockRel() are also complementary and they are consistent with smx_BlockMake() and smx_BlockUnmake(). Blocks may originate from block pools created by either smx or smxBase and they are automatically returned to their correct pools. This eases the interchange of blocks of data between foreground code and background tasks. Blocks can also be from other sources, such as the heap, a DAR, or can be statically defined (e.g. as an array) and still work smoothly with these functions. This permits maximum flexibility.

peeking

The **smx_BlockPeek()** and **smx_BlockPoolPeek()** functions allow obtaining information concerning a block and its pool. Although it is possible to read BCB and PCB fields directly, this is discouraged because future versions of smx are expected to utilize a software interrupt (SWI) API in order to run smx in privileged mode and application code in user mode. In that case, smx objects would no longer be accessible from application code.

Available block peek parameters are as follows:

SMX_PK_BP	data block pointer
SMX_PK_ONR	owner
SMX_PK_POOL	pool
SMX_PK_NEXT	next block in the free list (only works if this block is free)

This function can be used only on blocks that are in use — otherwise, there is no BCB. Zero is returned for POOL if there is no pool.

Available block pool peek parameters are as follows:

SMX_PK_NUM	number of blocks in pool
SMX_PK_FREE	number of blocks in free list
SMX_PK_FIRST	first block in free list
SMX_PK_MIN	first block in pool
SMX_PK_MAX	last block in pool
SMX_PK_NAME	name of pool
SMX_PK_SIZE	block size

Example usage:

```
blks_used = smx_BlockPoolPeek(poolA, SMX_PK_NUM) - smx_BlockPoolPeek(poolA, SMX_PK_FREE);
```

The second operation counts the number of blocks in the free list of poolA.

To trace the free list of poolA:

```
u8 *b, *bn;
for (b = smx_BlockPoolPeek(poolA, SMX_PK_FIRST); b !=NULL; b = bn)

{
    bn = smx_BlockPeek(poolA, SMX_PK_NEXT);
    /* use bn to access block */
}
```

When the end of the free list is reached or if it is empty, b == NULL.

To find blocks in use:

```
BCB_PTR blk, max;
u8 *bp;

max = (BCB_PTR)smx_BlockPoolPeek(&smx_bcbs, SMX_PK_MAX);
for (blk = (BCB_PTR)smx_BlockPoolPeek(&smx_bcbs, SMX_PK_MIN); blk <= max; blk++)
{
    if (smx_BlockPeek(blk, SMX_PK_ONR))
    {
        bp = smx_BlockPeek(blk, SMX_PK_BP);
        /* use bp to access block */
    }
}
```

Blocks in use have BCBs, so in the above example, the BCB pool is searched for BCBs with owners. When such a block is found its data pointer is obtained, which can then be used to access its data block. (BCBs are cleared when returned to the BCB pool. Hence, a zero owner field indicates a free BCB.)

It is possible to use smx_BlockPoolPeek, instead of sb_BlockPoolPeek(), in this example, even though smx_bcbs is actually a base pool, because both use PCBs. The smx version is preferable here because it is an SSR and hence task-safe. (In this particular case, peeking at smx_bcbs.pi and smx_bcbs.px outside of an SSR would probably be safe, but it is not a good practice.)

message block pools

An smx message is actually the same as an smx block, except that a message control block (MCB) is linked to the data block instead of a block control block (BCB). The data block pool is identical for each, and, in fact, the smx block pool create and delete functions are also used for message block pools. Furthermore, messages and blocks can share the same data block pool. Messages are discussed in the Exchange Messaging Chapter.

summary

This chapter has presented three basic methods for memory management:

(1) Dynamically allocated regions (DARs).

(2) Base block pools.

(3) smx block pools.

The first two are part of smxBase, and their APIs are described in the smxBase User's Guide. They can be used by smx applications, but care is advised because they are not task-safe. smx comes with two DARs: SDAR for system objects and ADAR for application objects. Additional DARs can be defined, if desired. smx uses base block pools for control blocks and other smx objects. Base blocks are also good for ISRs and low-level (non-task) application code.

smx block pool services should be used from tasks because they are task-safe and safer due to being more automatic. The smx BlockMake() and BlockUnmake() functions, which are new with v4.2, allow making bare blocks into smx blocks and unmaking smx blocks into bare blocks. Bare blocks include base blocks, static blocks, heap blocks, and DAR blocks. They are useful for I/O.

smx provides two additional memory management techniques: (1) the smx heap and (2) task and system stacks. These are discussed in separate chapters, which follow.

Chapter 5 Heap

BOOLEAN	smx_HeapBinSort(u32 binno, u32 num)
void*	smx_HeapCalloc(u32 num, u32 sz)
BOOLEAN	smx_HeapExtend(u32 xsz, u8* xp)
BOOLEAN	smx_HeapFree(void *bp)
BOOLEAN	smx_HeapInit(u32 sz, u8* hp)
void*	smx_HeapMalloc(u32 sz)
void*	smx_HeapRealloc(void *cbp, u32 sz)
BOOLEAN	smx_HeapRecover(u32 sz, u32 num)

acfg.h:

HEAP_ADDRESS	Starting address of heap (NULL if in ADAR)
HEAP_SPACE	Size of heap, in bytes
HEAP_DC_SIZE	Size of donor chunk, in bytes

xcfg.h:

SMX_HEAP_DATA_FILL	Fill pattern for data block in inuse chunks
SMX_HEAP_FENCE_FILL	Fill pattern for fences in debug chunks
SMX_HEAP_FREE_FILL	Fill pattern for free chunks
SMX_HEAP_DTC_FILL	Fill pattern for donor and top chunks
SMX_HEAP_SAFE	Enable safety checks.

introduction

A heap provides an alternative memory management facility to those discussed in the previous chapter. It is a region of memory from which variable-size blocks can be dynamically allocated and to which they can be dynamically returned, when no longer needed.

The smx heap is called *eheap*. It is a high-performance, configurable, self-healing heap, with enhanced safety and debug features. It has some similarities to *dlmalloc*, used in Linux, and to *tcmalloc*, used in Android, in that it is a *bin-type* heap. It differs from them in that its architecture is governed by the following general embedded system characteristics:

- Wide range of RAM sizes from very small to large.
- Good performance and deterministic operation are required.
- High priority tasks must be able to preempt and run quickly.
- Small code size is often necessary.
- Expected to run forever.
- Strong debug support is needed.
- Growing need for ruggedness and self-healing.
- Embedded systems have significant idle time.

Limited RAM: An embedded heap must be efficiently adaptable to heap sizes from tens of kilobytes up to megabytes.

Good Performance: A general-purpose allocator, such as dlmalloc, must be ultra-fast because many applications that use it require tens of thousands of allocations and deallocations per second or even more. This is because they are typically written to use myriad tiny objects with very short lifetimes. Probably many of these applications could be more efficiently written, but instead, reliance is placed upon the allocator to be fast.

The situation is different for embedded systems, which typically are carefully coded to achieve maximum performance within limited constraints. Also, wherever extremely high allocation and deallocation rates are required, embedded applications have the option to use RTOS block pools, instead of the heap. Hence, extreme speed is not usually a primary embedded heap requirement.

Determinism: System determinism is a primary requirement. The heap must not cause mission-critical tasks to miss their deadlines. Too much indeterminism can also cause missed deadlines for those tasks using the heap. Hence an embedded heap must not use extravagant mechanisms, or it must make such mechanisms explicitly controllable by the programmer. In some cases, block pools may be the only solution to meet necessary deadlines.

Small Code is necessary in small embedded systems, but is not likely to be a primary requirement in systems having significant heap usage.

Run Forever: This is typically not a requirement for desktop and enterprise systems because jobs are generally short, memory is plentiful, and the computer can easily be rebooted. However, embedded applications are typically unattended. Hence rebooting is undesirable and may be difficult to do. Therefore, the heap must run trouble-free for long periods.

Strong Debug Support is necessary to find heap-usage bugs before systems are shipped. These include features such as: time-stamping chunks, block overflow fences, block owner identification, block pattern filling to more easily see heap structure in memory, heap-aware debugger plug-in, and heap integrity scanning. If the heap can catch usage errors before other tasks are impacted, it is easier to find their causes and fix them.

Ruggedness and Self-Healing: A heap failure is not usually disastrous in a desktop or enterprise system – the system can simply be rebooted and/or the application re-run. Such is not the case in most embedded systems, which are expected to “keep marching on,” whatever happens. eheap has features to help achieve this.

Idle Time: Embedded systems usually face large load variations, and even under heavy loads they must meet their deadlines. Hence there is significant idle time, on average, which can be used for *heap stoking* to improve performance.

heap vs. block pools

Although eheap has been designed to be fast and deterministic, block pools still offer a faster, more deterministic solution. However, block pools suffer from *internal fragmentation* due to pools having more blocks than usually needed and blocks being larger than usually needed. Block pools do not adjust well to operational changes, such as shifting from sending and receiving data to reading and writing files, which typically require different size data blocks. A heap can make this adjustment, whereas, if using block pools, there must be one pool for each significantly different block size. This can be a big problem if available RAM is too limited.

One useful strategy is to use block pools for mission-critical tasks and the heap for other tasks such as data processing, user interface, and communication. The limitations of block pools may be acceptable for these kinds of tasks. Also, consider using one-shot tasks here, since they get their stacks from the stack pool, rather than from the heap. Under this approach, if the heap fails, then the tasks using it can be stopped, the heap can be reinitialized, and the heap tasks can be restarted. During this upheaval, high-priority, mission-critical tasks will function normally.

eheap vs. compiler heap

smx_HeapMalloc(), smx_HeapCalloc(), smx_HeapRealloc(), and smx_HeapFree() are equivalent to the standard C/C++ run-time library malloc(), calloc(),realloc(), and free(), respectively. eheap services should be used instead of the compiler heap services because the latter are not thread-safe and they lack the advantages of eheap services.

Two methods can be used to replace compiler heap services with eheap services:

(1) For C source code: macros in xapi.h replace compiler heap calls with eheap services. For example, malloc(sz) translates to smx_HeapMalloc(sz).

(2) For precompiled or preassembled code (i.e. .obj or .lib files), heap translation functions in heap.c, in the Protosystem, call equivalent smx heap services. The heap translation functions have the same names as compiler heap functions, so by linking them as a free-standing object file, they replace the compiler heap functions. This file is linked by the Protosystem.

eheap services are SSRs and share a common allocation function, called *malloc()* and a common free function called *free()*. These abbreviated names are used often in the sections below to represent the eheap services.

basic services

The following summarize the basic eheap services. See the smx Reference Manual for detailed descriptions of these services.

smx_HeapMalloc(sz) allocates a block of at least sz bytes from the heap. It is used as follows:

```
void* bp;
u32 sz = 100;

bp = smx_HeapMalloc(sz);
```

If an inuse chunk is allocated, the bp pointer will be 8-byte aligned and the block size will be 16 bytes, minimum. The block returned may be larger than sz, if an exact-fit chunk was not found. For a debug chunk, the block could be 4- or 8-byte aligned, but the minimum block size is still 16 bytes.

smx_HeapCalloc(num, sz) allocates a heap block to contain an array of num blocks of sz bytes from the heap and clears them. It is used as follows:

```
void* bp;
u32 num = 10, sz = 10;

bp = smx_HeapCalloc(num, sz);
```

It is important to note that only one heap block is allocated, and therefore, the blocks in the array cannot be individually freed. The same discussion concerning sz, errors, etc. for HeapMalloc() applies to the heap block allocated by this service.

smx_HeapRealloc(cbp, sz) reallocates an existing block pointed to by cbp to the new size, sz. It can be used to either downsize or upsize the existing block. It is used as follows:

```
void *nbp,*cbp;
u32 sz = 100;

nbp = smx_HeapCalloc(cbp, sz);
```

smx_HeapRealloc() is considerably more complex than the previous two allocation services, but it uses the same heap sub-functions that are used by them. Hence, the same discussions concerning sizes, errors, etc. also apply to it. Two peculiarities, due to the ANSI C/C++ Standard are:

1. if cbp == NULL, a block of sz bytes is allocated from the heap.
2. if sz == 0, cbp is freed to the heap.

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

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

smx_HeapFree(bp) is the smx system service used to free a block back to the heap. bp points to the start of the block. Per the ANSI standard, if bp is NULL, no operation is performed and TRUE is returned. Double frees are potentially a serious problem. smx_HeapFree() will report SMXE_HEAP_ERROR and return FALSE for an attempt to free an already free chunk. See debug section in the Heap Maintenance Chapter for more discussion of this problem.

smx_HeapInit(sz, hp) is used to initialize the heap. It is called automatically by the first malloc(). It can also be directly called, if preferred, or for purposes such as heap recovery.

Heap Theory

physical heap structure

The memory area allocated to a heap is divided into *chunks* of various sizes. Each chunk has a *header*, and its remainder is available for use as a *data block*. Chunks are multiples of 8 bytes, in size, and they are aligned on 8-byte boundaries. There are two main types of chunks: *inuse* and *free*. eheap provides an additional *debug* chunk type, which is discussed in the Heap Maintenance chapter.

An inuse chunk is one that has been allocated to an application via a malloc(). The application uses the data block of the chunk; it does not access the chunk header. inuse chunks have headers of 8 bytes. If the average data block is small, say 24 bytes, the header represents a significant overhead of 25%. If the block could be as small as 8-bytes, the overhead would be 50%. For embedded systems, blocks this small are probably better served from bare block pools, for which overhead is 0.

The first word of the header is a forward link to the next chunk and second word is a back link to the previous chunk. These are used to doubly link chunks into the heap, in physical order, regardless of chunk type. This provides the *physical heap structure*. The heap may be traversed in either direction for purposes such as splitting chunks, merging chunks, scanning chunks, and fixing broken chunks. The physical structure produces a *linear heap*. The smxAware Memory Map Overview shows physical heap structure graphically.

For allocations, linear heaps must be searched from the first free chunk, chunk after chunk until a big-enough free chunk is found. As a heap is used, it tends to become divided into more and more chunks. For a linear heap, this means that allocations take longer and longer, and thus become less and less deterministic. Ultimately, the heap may become so *fragmented* that an allocation fails after searching the entire heap. This kind of heap is not useful for systems with large heaps and high heap activity.

logical heap structure

The *logical heap structure* consists of *heap bins*. Bins hold free chunks of specified sizes. Heap bins are comprised of two arrays: an array of bin sizes, smx_bin_size[], and an array of free chunk links, smx_bin[]. The size of each bin is the smallest chunk handled by that bin. A bin that holds a single size is called a *small bin*. For example, a small bin might handle only 24-byte chunks. A bin that holds a range of sizes is called a *large bin*. For example, a large bin might handle 128, 136, ..., 248-byte chunks (15 total sizes, spaced 8-bytes apart). Note that chunk sizes are multiples of 8-bytes and must be consecutive in a bin — i.e. there can be no missing sizes.

With eheap, the number of bins and the sizes of bins can be selected to best fit the needs of an application. This logical structure adds a *second dimension* to a heap, which allows plucking a big-enough chunk from anywhere in the heap with little or no searching.

Each bin consists of a free forward link (ffl) and a free backward link (fbl) to a doubly-linked list of free chunks. During allocation, a bin is selected by the desired chunk size. If it is a small bin, the first chunk is taken; if it is a large bin, the first big-enough-chunk is taken. Provided that bins are not allowed to become empty, allocations from small bins can be very fast and allocations from large bins can be much faster than from a linear heap and sometimes as fast as a from a small bin. Methods to prevent bins from going dry are presented in the Heap Management Chapter, later in this manual.

Bins are divided into a *lower bin array*, also called the *small bin array* (SBA), and an *upper bin array*. These are discussed, next.

small bin array, SBA

Normally, a heap starts with a small bin array, SBA. SBA bin sizes are 24, 32, 40, etc. up to the top SBA bin, with no missing 8-byte multiples. The SBA can have 0 to 31 bins, but usually has just enough bins to cover the most frequently used *small chunk* sizes. The SBA can be omitted, but that is rare. In small systems, 5 SBA bins might be enough (24, 32, 40, 48, and 56) or 10 SBA bins might be needed (24, 32, 40, 48, 56, 64, 72, 80, 88, and 96). Note that the corresponding block sizes are 8 bytes less – i.e.: 16 to 48 and 16 to 88. Also note that all consecutive bin sizes must be present, even if not used – e.g. bin 48 cannot be omitted.

The advantages of the SBA are that bin selection is very fast (binno = csize/8 – 3), no chunk searching in the bin is necessary, and chunks are the exact sizes needed.

Average chunk sizes used by some applications, can be very small (e.g. 32 bytes), so optimizing small chunk mallocs and frees is important for performance – especially for object-oriented code, such as Java and C++. However, embedded systems that are written in C are likely to have larger average chunk sizes and, in some cases, the upper bins may be more important. eheap allows bin sizes to be adjusted to optimize performance per embedded system. This is discussed in the Heap Maintenance Chapter.

upper bin array, UBA

Above the SBA, are the *upper bins*. These consist of up to 32 bins, of any sizes. The upper bin array may consist of any combination of large bins and small bins. For example, the sizes of upper bins might be defined as 104, 232, 360, 488, and 496. The first large bin is immediately above the 10-bin SBA. It and the next 3 bins are 128 bytes and each has 16 chunk sizes. The fourth bin is a small bin having only the 488-byte chunk size. The last bin, called the *top bin*, handles sizes from 496 bytes and up.

An upper bin is located by doing a binary search on smx_bin_size[] vs. the needed chunk size. So, for example, if there are 15 upper bins, up to 4 searches are required (e.g. 8, 4, 6, 5). This search should be fast since smx_bin_size[] is likely to be located in fast RAM. Once found, the upper bin *free chunk list* is searched for the first big-enough chunk. In the case of a small bin this is the first chunk. In the case of a large bin, it may be necessary to examine several chunks in the bin’s free chunk list. to find a big enough chunk. Bin sorting during idle helps to assure that a best fit is found. It is discussed in a section below.

concepts

Blocks are normally used by software that uses the heap, but the heap, itself, uses chunks. This is likely to cause confusion. For example, bin sizes are determined by chunk sizes, not block sizes. Hence, a 160-byte small bin contains 160-byte chunks and an inuse chunk from this bin will hold a 152-byte data block. However, a debug chunk from this bin might hold only a 140-byte data block, due to its extra overhead. Bins must be thought of in terms of chunk sizes, not block sizes. Also, most pointers used within the heap are chunk pointers, whereas pointers used within the application are block pointers.

Another area of confusion is between physical and logical structure. The positions of chunks in bins have no relationship to their positions in the heap. Hence adjacent chunks in a bin are not likely to be adjacent in the heap and vice versa. This can be confusing, for example, when tracing bin links in a watch window.

chunks

An **inuse chunk** is a chunk that has been allocated and is currently being used by the application. Its metadata consists of a forward link (fl) to the next chunk and a backward link + flags (blf) to the previous chunk. Since all chunks are 8-byte aligned, the 3 low bits of link pointers are available for flags. The flags are: DEBUG (bit 1) and INUSE (bit 0). These are located in the back link, blf. The overhead of an inuse chunk is 8 bytes, so if the average block size is 100 bytes, this results in 8% overhead.

A **free chunk** is a chunk that is available to be allocated and that is currently under control of the heap. Its metadata consists of a Chunk Control Block (CCB) with the following fields:

fl	physical forward link
blf	physical backward link + flags
sz	chunk size, in bytes
ffl	free forward link
fbl	free backward link
binx8	bin number times 8

The first two fields are the same as for inuse chunks and are used for the heap physical structure. The chunk size is used by heap services. The last three fields are used by bins and are discussed below in the bins section. The CCB requires 24 bytes. This establishes the minimum chunk size as 24 bytes. When a free chunk is allocated, the last four fields of the CCB (i.e. last 16 bytes) are overwritten by the data block. This establishes the minimum data block size for an inuse chunk of 16 bytes. Since free chunks are not in use, the CCB does not contribute to overhead (except that 16-byte blocks are the minimum size that can be allocated).

Note: inuse chunk pointers are defined as CCB_PTR. Hence when looking at an inuse chunk via a debugger, it looks like it has the last 4 fields. This is not the case; these are data words.

debug chunk is an inuse chunk with debug information added. Like other chunk types, the first two fields of a debug chunk are fl and blf. The purpose of debug chunks is to help debug heap problems such as block overflows and leaks. Debug chunks behave the same as inuse chunks with respect to most operations. However, because their size differs, they introduce complexities, which detract from the discussion of heap operations. (For example, a debug chunk may appear in a higher bin than an inuse chunk with the same data block size.) Hence, debug chunks are mentioned only as necessary, in discussions, that follow, and treatment of them is deferred until the Heap Management Chapter.

special chunks

In addition to bins, eheap has 4 special chunks: *start chunk (sc), donor chunk (dc), top chunk (tc), and end chunk (ec)*. These special chunks are never put into bins.

Following heap initialization, the heap consists of sc, dc, tc, and ec. sc and ec are permanently-allocated 16-byte, inuse chunks that mark the ends of the heap. The sizes of dc and tc are controlled by user-defined configuration constants, in acfg.h. dc is usually much smaller than tc, but must be at least 24 bytes.

Immediately after initialization, all free heap space is in dc and tc. Until bins begin filling up, due to free() operations, most small chunks will come from dc and most large chunks will come from tc. This serves two purposes: (1) faster chunk allocations, (2) segregation of the physical heap: small chunks will be below dc and large chunks will be above it. This helps to reduce fragmentation caused by small, inuse chunks blocking the merging of larger free chunks. It also helps *localization*, since the dc and tc pointers will tend to be in the data cache. Of course, in time, there will be cross movement of chunks into the other region due to depletion of dc, merging of small chunks, and splitting of large chunks.

In heap theory, localization, is the attempt to assure that chunks allocated close in time are physically close. This tends to increase cache hits.

When dc becomes too small, it can be replenished by putting what is left into a bin, and then allocating a bigger chunk to it. The strategy for doing this is up to the user. Alternatively, using dc can simply be turned off.

allocation algorithm

In the following discussion, *csize* is the minimum chunk size needed for the requested block size. It is determined by adding CHK_OVH to the requested block size. CHK_OVH is 8 bytes for an inuse chunk, but it could be 40 bytes, or more, for a debug chunk.

Allocation order for **small chunks** is as follows: selected SBA bin -> dc -> larger bin -> tc. The correct SBA bin is quickly located, via the simple formula: binno = csize/8 - 3. If occupied, the first chunk in this bin is dequeued and a pointer to its data block is returned. This is the fastest allocation possible with eheap.

If the selected SBA bin is empty, calving a chunk from dc is nearly as fast. The size of dc is set by HEAP_DC_SIZE in acfg.h. It is desirable to make dc large enough to handle the expected maximum number of small chunks. However, there may not be enough RAM for all expected small chunks plus all expected large chunks.

If dc is exhausted, the chunk is taken from the next-occupied larger bin, and if that fails, it is taken from tc. Following startup, most small chunks will come from dc, and be freed to SBA bins. After running for a while, virtually all small chunks will come from the SBA. Allocations from the SBA, dc, and tc are exact fits. Allocations from larger bins may be larger than csize.

Allocation order for **large chunks** is as follows: selected upper bin -> larger bin -> tc. The selected bin is found with a binary search on smx_size[]. If the selected upper bin is occupied, the first big-enough chunk is taken. If the bin is a small bin, this will be the first chunk in it. If the bin is a large bin, malloc() searches through the bin free list until it finds a big-enough chunk. If one is not found, then the first chunk of the next larger occupied bin is taken (this is big enough, by definition). Failing this, the chunk is calved from tc.

Initially the size of tc is determined by HEAP_SPACE – HEAP_DC_SIZE – 24 in acfg.h. A goal of the above algorithms is to keep tc as the source of last resort for big chunks. If tc is not big enough for csize, SMXE_INSUFF_HEAP is reported and malloc() returns 0. Both dc and tc must be larger enough to hold a chunk control block. Methods to deal with heap allocation failure are discussed below and in the Heap Management Chapter.

free algorithm

If chunk merging is enabled (cmerge ON), a freed chunk will be merged first with a lower free chunk, if it is not dc. Then it will be merged with an upper free chunk, including dc and tc. Then the resulting free chunk is put into an appropriate bin, unless it is dc or tc. In the latter case, the smx_dcp or smx_tcp pointer is updated.

For a small bin, a chunk is put at the start of the bin (i.e. its free chunk list). For a large bin, if a chunk is larger than the first chunk in the bin, it is put at the end of the bin. Otherwise it is put at the start of the bin.

finding the next larger occupied bin

As discussed above, it is sometimes necessary to find the next larger occupied bin. eheap uses a 32-bit bin map, *bmap*, to do this. bmap has one bit per bin. When a chunk is loaded into a bin, the bin's bmap bit is set. When a bin becomes empty, its bmap bit is cleared. A simple calculation magically converts bmap into the bin number of the next larger occupied bin. If bmap is 0, there is no next larger occupied bin and the chunk is calved from tc.

eheap has one 32-bit bmap for all bins. Hence, the maximum number of bins is limited to 32. This can be increased to 64, or more, if needed. However, there is a small performance penalty, so bmap is being left at 32, for the time being, which may be enough for embedded systems.

chunk splitting

The chunk found is larger than needed, it is split into an exact-fit chunk and a *remnant,* if the remnant has at least MIN_FRAG (xheap.c) bytes. In the first case, the returned chunk is an exact fit plus any necessary adjustments for 8-byte alignment. In the second case, the whole found chunk is returned. The unused portion is referred to as *internal fragmentation,* which is wasted space.

When there is a remnant, it is put into the correct bin for its size, unless the chunk above it is free and cmerge is ON. In that case, the two chunks are merged and put into the appropriate bin; this seldom occurs. Either way malloc() time is increased and thus splitting is not desirable. Remnants may be too small or odd sizes to be useful for the application or are not needed because there are too many chunks of the same size. Such a chunk may never be used, in which case, it is referred to as *external fragmentation* – i.e. also wasted space. The remnant may sit unused until the allocated chunk is freed and merged back with it. This is better than wasted space, but it also is not good because wasted split and merge operations have occurred.

Avoiding the foregoing problems is helped by maximizing least-big-enough allocations. If there is ample RAM, making MIN_FRAG large can also be used to reduce chunk splitting to times when it is useful. Otherwise, MIN_FRAG must be at least big enough for the smallest chunk size used in the system.

deferred merging

Some heaps immediately merge free chunks in order to minimize fragmentation. eheap permits *deferred merging* of free chunks. Merging is controlled by cmerge[1], which can be turned ON or OFF by smx_HeapSet(). To promote rapid bin filling, cmerge is OFF following heap initialization. Keeping cmerge OFF is recommended to avoid *leaky bins.* For example, suppose a 24-byte chunk is freed to bin 0 and a physically adjacent 48-byte free chunk resides in bin 3. If cmerge is ON, what will happen is that these chunks will be merged into a 72-byte chunk, which will be put into bin 6. Bins 0 and 3 have leaked

[1] cmerge and other heap modes are bit flags in smx_heap.mode.

upward into bin 7. This obviously is not conducive to best performance if the application needs 24 and 48-byte chunks and not 72-byte chunks.

Unfortunately cmerge OFF also leads to leaky bins due to chunk splitting. In this case, a larger-than-needed chunk is taken from a bin, an exact-fit chunk is split off, and the remnant is put into a lower bin. Hence leakage is downward. This, too, can impact performance and is also likely to eventually lead to allocation failure. Setting a large MIN_FRAG value in xheap.c can reduce this problem, but may not solve it.

eheap implements *auto-merge control*. If the amerge mode bit is ON, cmerge will be turned ON when either the space in the top bin (smx_heap_tbsz) plus the space in the top chunk is less than HEAP_TZMIN, which is defined in acfg.h, or neither the largest chunk in the top bin nor the top chunk are large enough for the maximum dynamic chunk size that may be allocated, HEAP_CZMAX. The latter does not include one-time allocations at startup. There is no hysteresis in this algorithm, so as soon as both conditions are met, cmerge is turned OFF. Auto-merge control is implemented in smx_HeapManager(), which is called from idle.

This algorithm has proven to be effective in achieving stable heap operation in a high-heap-usage environment. It is important to define HEAP_CZMAX to help assure that the largest needed chunk can be allocated at all times. If there is heavy use of this chunk size, then it is helpful to define it to be the top bin chunk size, so any chunk in the top bin can be used. Then properly adjusting HEAP_TZMIN helps to assure that many chunks of this size will be available either in the top bin or from the top chunk. If a system tends to create a great deal of heap fragmentation, then HEAP_TZMIN may need to be an appreciable fraction of the over-all heap size. This is especially likely in low-margin heaps (i.e. heap size/heap high water mark).

Other algorithms can be implemented, based upon factors such as number of free chunks, average size of free chunks, total space in bins, etc. Long-run testing of these alternatives on a specific embedded system is necessary to determine which works best and how much heap margin (i.e. unused vs. total space) is necessary to insure that heap failure does not occur. Doubtless, one strategy will fit a specific embedded system better than others.

dynamic merge control

Dynamic merge control can be implemented at the application level. For example, cmerge can be turned OFF by tasks, which are heavy heap users. Tasks written in object-oriented languages such as C++ and Java are likely to be such tasks, since they often do frequent mallocs and frees of small blocks as objects go in and out of scope. Inhibiting merging while these tasks run can improve performance by avoiding leaky bins, particularly in the SBA.

One way to do this is to turn cmerge OFF at the start of the task or of its main loop. cmerge can be turned ON in an exit routine hooked to the task and OFF in an enter routine hooked to the task. This way cmerge is OFF only when the task is actually running and not when it is suspended or preempted. The cmerge mode can be safely set or reset directly by the hooked routines, since they cannot be preempted.

When the task is about to stop running, it can turn cmerge ON so that all of the small chunks it allocated will be merged and become available for larger allocations by other tasks. Of course, this scheme may not avoid heap failure unless heavy heap users are prevented from running simultaneously.

fragmentation

When a heap becomes excessively *fragmented*, small inuse chunks may prevent medium chunks from being merged to form larger chunks that are needed for allocations. When this happens, *heap failure* is the result and SMXE_INSUFF_HEAP is reported. This is not necessarily catastrophic. For example, the task requesting the large block could be rescheduled to run at a later time or a lower priority and try again, or less important tasks could be stopped and their heap blocks freed, with merging enabled. eheap tries to hold reserve space in the top bin and in tc for large chunk allocations. However, over time, both are likely to be eaten up, unless there is ample RAM available.

Deferred merging, as discussed above, is generally viewed as increasing fragmentation, thus making heap failure more likely. If all methods of merge control are resulting in fragmentation leading to heap failures, then it may be necessary to keep cmerge permanently ON. This is more likely to occur in a *tight heap* due to insufficient RAM than in a system where RAM is abundant. The downside is that cmerge continuously ON will reduce performance due to leaky bins.

Another potential problem with deferred merging is *stuck chunks* in large bins. This can happen in a system that allocates random large sizes, some of which are almost never used. When freed, these chunks will be put into large bins. If such a chunk is larger than the chunks predominantly allocated from the bin, it may sit in the bin for a long time until the bin runs out of smaller sizes. Turning cmerge ON will help to remedy this problem.

If merge control fails to stem an occasional heap failure, an eheap recovery service is provided to search the heap to find and merge adjacent free chunks in order to satisfy a failed allocation. If this fails, an eheap extension service can be used to extend the heap into reserve memory, such as a slower RAM area or one reserved for future expansion. Heap recovery and heap extension services are discussed near the end of this chapter.

Unfortunately, reserve memory may not be available in a specific embedded system. If this and other strategies, such as those suggested above, do not work, then in the worst case it may be necessary to reinitialize the heap and restart all tasks using it. Obviously, mission-critical tasks should use block pools instead of the heap. Recovery methods are discussed more later in this chapter.

Heap Special Features

bin seeding

Bin seeding provides an alternative to deferred merging that may be useful in tight heaps. The **smx_HeapBinSeed(num, bsz)** service is used to directly seed num chunks into the bin for the block size, bsz. The bin is not specified, because it depends upon the chunk size, which, in turn, depends upon debug mode. This service allocates a big-enough chunk, splits it into num chunks the right size for blocks of bsz, and frees chunks to their correct bin. While in operation, it temporarily turns cmerge OFF, if it is ON.

Bin seeding, combined with monitoring bin populations, may be the best way to populate bins, in some systems. For example:

```
void FillBins(void)
{
    for (u32 n = 0; n <= smx_sba_top: n++)
        if (smx_HeapBinPeek(n, SMX_PK_COUNT) < 3)
            smx_HeapBinSeed(3, smx_HeapBinPeek(n, SMX_PK_SIZE));
}
```

FillBins() could be called from the idle task to attempt to keep the SBA bins in the range of 2 to 5 chunks, each. Since the new chunks are not separated by inuse chunks they may soon leak out and reunite with larger free chunks if cmerge is ON.

Note that the bin actually populated will depend upon whether debug mode is ON. If it is, chunks may be put into a higher bin than expected. Then, in the release version, when debug mode is OFF, the chunks may be put into a lower bin. This is probably ok, but may cause confusion. Furthermore, the last chunk may be larger than the other chunks due to the way malloc() works and thus it may be put into a higher bin, than the other chunks.

Clearly, keeping bins populated is a non-trivial problem. But if performance is important a solution must be found. Finding the best solution for your system will require experimentation and long runs to verify that correct choices have been made.

Another use for smx_HeapBinSeed() is to populate bins during initialization. This gets the heap off to a faster start and is an alternative to using dc.

bin sorting

As noted previously, operation is improved if large bins are well-sorted. Then malloc() will take the *best fit chunk* in the bin, if there is one. This is generally considered a good way to reduce fragmentation. Also, if a large bin is well-sorted, searches for the smaller chunk sizes in the bin will be faster, on average. Then, choosing bin sizes that are equal or slightly less than the most popular chunk sizes will improve performance. Though unsorted bins may cause suboptimal performance, they do not cause heap failures. Hence, they may be acceptable during periods of peak loading. It is worth noting that lower-priority, heap-using tasks probably will not be able to run during those periods, anyway.

Unlike enterprise applications that run on servers or desk-top computers, embedded applications must have significant idle time to deal with peak loads that are caused by simultaneous asynchronous external events. Also they need spare processing time to handle new requirements, likely to be added in the future, without existing tasks missing their deadlines. eheap provides an efficient bin sorting service, which normally runs from the idle task. Hence, if there is enough idle time, large bins should almost always be well sorted.

The algorithm used is *bubble sort with last turtle insertion* to sort chunks by increasing size in each bin free list. The *last turtle* is the last chunk that may be smaller than a chunk ahead of it. It is called a “turtle” because it moves forward very slowly in a normal bubble sort. A bubble sort requires multiple *passes* through the bin free list. During each pass, the current last turtle is inserted ahead of the first larger chunk found. The chunk that was ahead of the last turtle becomes the new last turtle. A characteristic of the bubble sort is that after k passes, the last k chunks are sorted. Hence the last turtle is at least k chunks from the end of the bin free list and each pass ends with the last turtle. If more than one last turtle was moved during the pass, this chunk will be more than k chunks from the end. Last turtle insertion can result in significant speed up of a bubble sort. If during a pass no chunk is moved, the sort is complete.

When free() puts a large chunk at the end of a sorted bin, last turtle insertion will immediately move it to its most likely position during the first pass. If bin sort is running frequently, this may be all that is required to sort the bin.

Bin sorting is done by calling smx_HeapBinSort(binno, fnum) each time the idle task runs. smx_HeapBinSort() is an SSR, so it cannot be interrupted by another heap service during a run. A *run* consists of testing and possibly moving *fnum*, or more, chunks. Higher priority tasks can preempt between runs and fnum should be chosen so that they meet their deadlines. Of course the smaller fnum is, the longer a bin sort will take, so there is a tradeoff to be made.

A bin sort map, *bsmap*, similar to *bmap* used by malloc() and free(), determines which bins to sort. The bsmap bit of a bin is set if free() puts a chunk at the end of that bin’s free list. This happens whenever the freed chunk is larger than the first chunk in the bin. Otherwise the chunk is put at the start of the bin, in which case, no sort is needed and the bsmap bit is not set. Small bins never need sorting, hence their bsmap bits are never set. Therefore, bsmap shows only those bins that need sorting.

A bin’s bsmap bit is reset when a sort begins for the bin. A global variable, *smx_csbin*, keeps track of the current bin being sorted, and a static variable, *ccp*, keeps track of the next chunk to start the next run. If a preempting free() sets the bin's bsmap bit, due to putting a chunk at the end of the bin, the sort is restarted for the next run. If a preempting malloc() takes a chunk from the bin, it also sets the bin's bsmap bit, causing the sort to start over on the next run. Starting over is not detrimental to a sort, because any sorting already done is preserved. Otherwise each run starts from where the last run left off. Note that one pass may require multiple runs or one run may cover multiple passes, depending upon the seize of fnum vs. the length of the pass, which steadily declines for a sort.

The *binno* parameter in smx_HeapBinSort() specifies the bin to sort. If it is a valid bin number and the bsmap bit is set for the bin, then the binno bin is sorted. If binno is greater than the top bin number, all bins with bsmap bits set are sorted, smallest first. This favors smaller large bins, because they are likely to be more active. smx_HeapBinSort() returns TRUE when a bin has been sorted, or if no bins needed to be sorted. Note that unless bin sorting is frequently preempted, all chunks in the bin free list are likely to be in the data cache and chunk accesses will be fast. Hence large runs (i.e. large fnums) may be possible without causing high-priority tasks to miss their deadlines.

If bins are not being adequately sorted, fnum can be increased or smx_HeapBinSort() can be called from a higher priority task. It might make sense to call it from the priority level of the lowest priority task using the heap. Then it will run ahead of other idle functions. Because it is expected to run frequently, smx_HeapScan() makes no entries in the event buffer, other than those due to reported errors or fixes.

heap recovery

Heap failure, reported by SMXE_INSUFF_HEAP, is likely to be due to a situation where too much free space is allocated to small free chunks and insufficient space is allocated to larger free chunks. This is called fragmentation. A recovery service, **smx_HeapRecover(sz, fnum)**, is provided to deal with this problem. It searches the heap to find and merge adjacent free chunks, in order to create a big-enough free chunk to satisfy the failed allocation.

In order to avoid conflicts with preempting free() operations, cmerge must be OFF when smx_HeapRecover() is running. To avoid conflicts with smx_HeapScan(), smx_HeapRecover() should be used from a higher priority task. In case it preempts smx_HeapScan(), it causes smx_HeapScan() to restart when it is done.

smx_HeapRecover() should be followed by retrying the allocation service that failed, as in this simplified example:

```
while (1)
{
    if (bp = smx_HeapMalloc(size))
        /* use bp */
    else
        if (!smx_HeapRecover(size, 20))
            break;
}
/* try another heap recovery action */
```

In the above, if smx_HeapMalloc() fails, smx_HeapRecover() is called. If it finds a big-enough chunk, smx_HeapMalloc() is called again. If smx_HeapRecover() fails, the while loop is exited and code runs to try another approach. Note that preemption is permitted every 20 chunks. The while loop is needed because a higher priority task could take the chunk just recovered, so the malloc() would fail again and heap recover would be called again.

This example is just for illustration. Generally, it is not practical to implement all calls to smx_HeapMalloc() this way. Furthermore, smx_HeapRecover() is not preemptible, by itself, so it should not be called from multiple tasks, unless a mutex is used to serialize calls to it. See the Reference Manual for a more practical example, where a common recovery task is the only task using smx_HeapRecover(). However, it may not be desirable to add recovery code to every malloc(), as shown in that example. An alternative approach is as follows:

```
TCP_PTR StoppedTask;

void TaskA_main(void)
{
    while (bp = smx_HeapMalloc(size))
    {
        /* use bp and continue normally */
    }
} /* auto stop on a malloc() failure */

void smx_EMHook(SMX_ERRNO errnum, u32 par)   /* par = block size needed */
{
    switch (errnum)
    {
        ...
        case SMXE_INSUFF_HEAP:
            smx_TaskStartPar(RecoveryTask, par);
            StoppedTask = self;
            break;
        ...
    }
}

void RecoveryTaskMain(u32 bsz)
{
    if (smx_HeapRecover(bsz, 20))
        smx_TaskStart(StoppedTask);
    else
        /* try another heap recovery action */
}
```

In this example, TaskA runs normally unless there is an allocation failure, in which case an SMXE_INSUFF_HEAP error is reported and the error manager, smx_EM(), runs. smx_EM() calls smx_EMHook(), which specific error management code. For the insufficient heap error, this code starts the RecoveryTask, which should run at a lower priority so that smx_EM() can return to TaskA and TaskA can autostop. Then TaskA is safely out of action. Other high priority tasks can preempt and run while RecoveryTask runs. This assures that mission-critical tasks will not be blocked.

If a big-enough free chunk is found, taskA is restarted. Otherwise RecoveryTask tak**es** some other action. With this method, the recovery code for heap allocation failures is separate from the application code and transparent to it. Notice that taskA is structured such that it will try to allocate the needed chunk again, when it is restarted. For more sophisticated task structures, see the ideal task structure section in the Tasks chapter.

The is not a completely practical example because other tasks, with failing heap allocations, could restart the recovery task before it finished, resulting in preempting smx_HeapRecover(), which, as noted above, is not preemptible by itself. A better design would be for RecoveryTask to wait at an exchange for error messages sent by smx_EMHook(). If multiple allocation failures occur, heap recovery messages would be queued up at the exchange and processed in priority order.

When it receives a recovery message, the recovery task would call smx_HeapRecover(). If successful, it would restart the stopped task that needed the block and that task would allocate the block and continue. Hence operation is simple and avoids one task tripping over another.

Recovery is possible only if enough free space is found in adjacent free chunks that can be merged. Otherwise, smx_HeapRecover() will fail and some other means must be used to allocate the needed chunk, such as:

(1) Wait and retry.

(2) Free blocks from lower priority tasks with cmerge ON.

(3) Extend the heap with smx_HeapExtend() (discussed below).

(4) Stop all tasks using the heap, reinitialize the heap, and restart the tasks.

(5) Reboot the system.

If insufficient heap failures are due to insufficient RAM for the heap, another approach that might be helpful is to use one-shot tasks. High-level functions that use significant amounts of heap can be put into one-shot tasks of the same priority. Before each task stops, it frees all heap that it allocated, with cmerge ON. Since only one of these tasks can run at a time (due to having the same priority) this forces the tasks to share rather than fight over available heap space.

heap extension

If heap recovery fails, the heap can be extended into reserve memory using **smx_HeapExtend(xsz, xp)**. xsz is the size of the extension and xp is its location. The extension may be adjacent to the present heap or elsewhere in memory. The only requirement is that the extension must be above the current heap. If there is a gap, it is covered by an artificial inuse chunk and the extension becomes the new tc. If there is no gap, the extension is added to the current tc.

A heap extension might be to less desirable memory, such as slower RAM, in which case access to chunks in the extension would be slower, but that is preferable to system failure. The need for heap extension might happen when operating with debug chunks and might not occur in a released system using only inuse chunks. If it did, it is likely that there would be only a few slow chunks, so performance might suffer for only a few unlucky tasks.

Reserve memory could also be memory that is available in some systems and not in others. For example, the system used for debugging might have more memory than production systems. In this case, if no reserve memory is used after long runs it could be assumed that production systems have enough memory.

Unfortunately, reserve memory may not be available in a specific embedded system. Therefore, one of the other solutions suggested in the above heap recovery section must be used.

Chapter 6   Stacks

introduction

Stacks are central to embedded system software and to RTOSs because every running task needs a stack. Stack overflow is probably the leading problem in multitasking embedded systems. For this reason we believe it is essential for you to clearly understand how stacks work in smx and to understand the tools which smx makes available to size stacks correctly and to deal with stack overflows.

In multitasking systems, it is essential that shared subroutines be reentrant. This dictates using auto variables instead of static variables, and most auto variables are stored on the current stack. In addition many compilers pass arguments via the stack. In addition, if code is written with deep subroutine nesting, then task stacks can become large. It there are many tasks, then a large amount of RAM will be required, thus forcing stacks to be located in external RAM rather than internal SRAM. This can hurt performance. Hence, good stack management can be crucial for good performance and for efficient RAM usage.

choice of stacks

smx supports three types of stacks: System Stack (SS), permanent stacks, and shared stacks. The latter are unique to smx. These are discussed below. (Note: For ARM, smx also supports the IRQ Stack which is used only at the very start of interrupt service routines and typically is only 16 bytes.)

system stack (SS)

SS is used for initialization, Interrupt Service Routines (ISRs), Link Service Routines (LSRs), the scheduler, and the smx Error Manager (smx_EM). By handling ISR, LSR, scheduler, and error handling requirements, SS allows task stacks to be much smaller. The total SS requirement would need to be added to each task stack, if it weren't for switching to SS.

During operation, the SS size requirement is relatively small (on the order of 200 bytes). Its size is determined by largest requirement for LSRs, the scheduler, or the error manager plus the amount of stack used by ISRs. This is because the first three are interruptible. Usually the largest LSR usage will dominate since the scheduler and smx_EM use very little stack. If ISR nesting is permitted, then the sum of all ISR stacks must be added to other requirements.

Running smx_EM() under SS saves about 60 bytes of stack for every stack and ensures that a task can be shut down cleanly without causing further errors or missing interrupts.

C++ static initializers, if any, can result in a much bigger stack requirement than for normal operation. If so, special steps may be required to reduce SS after initialization. smx does not provide code for this.

SS location and size is typically controlled by the linker command file. SS is the same as the SVC stack for ARM and Main Stack for ARM-M. It is recommend that SS be located in on-chip SRAM to achieve the best performance of ISRs, LSRs, and the scheduler. Be careful to align SS as required by the processor (at least a word for most processors, and a double word for ARM and ARM-M).

permanent stacks

Permanent, or bound, stacks are the conventional task stacks provided by most RTOSs. A permanent stack is obtained from the heap during task create, and it is permanently assigned (bound) to the task. The stack size is specified as a parameter in smx_TaskCreate() and may be any size desired. Bound stacks are naturally aligned on 8-byte boundaries by the heap.

Typically a bound stack must be large enough for maximum function nesting. Functions use the stack for auto variables, and some compilers save non-volatile registers in the stack during function calls. Some, such as ARM, pass most parameters via registers. Hence, a typical function might use 48 bytes of stack, or more. If code were written such that the functions used in a task could nest up to 10 deep, then 500 bytes would be required for the task's stack. If there were 50 tasks in the system, then clearly stack RAM usage is getting out of hand.

In addition, space for the Register Save Area (RSA) must be added. Non-volatile registers are saved in RSA, when a task is suspended. RSA is below task stack. SMX_RSA_SIZE is defined in the smx processor architecture header file (e.g. xarm.h, xcf.h, etc.).

To make matters worse, some C library functions (e.g. printf()) put large arrays of hundreds of bytes into the current stack. This may work for desktop systems, but obviously, these functions must be avoided in embedded applications, unless RAM is plentiful. Even so, minimizing stack space will pay off if all stacks can be put into on-chip SRAM. There is typically a 10:1 performance difference between on-chip SRAM and off-chip SDRAM. This can only partially be reduced by a data cache, if present — following a task switch there will be many cache misses for stack accesses.

It is beneficial to put some forethought into embedded code structure. The common practice of nesting functions deeply, should be avoided. Many standard C library functions should be avoided, and often, low-stack usage alternatives are available. For some processors, such as ARM, there is a large benefit to limiting the number of parameters since no stack is used to pass up to 4 parameters. Often it is better practice to create more tasks, which are simpler, than a few highly complex tasks with deeply nested functions.

Permanent stacks offer the following advantages:

(1) Stack size can be customized to the task. Some tasks, such as GUI tasks, require very large stacks, and some tasks require only tiny stacks.

(2) They are easier to use since they are always present. They can store auto variables, even when a task is stopped.

(3) Each stack is isolated by heap mechanisms (in addition to padding).

Since each stack is actually a heap block, it can be protected by heap protection mechanisms. For example, if a debug chunk is used, the heap will be surrounded by fences — see discussion in the Heap Chapter. Then smx_HeapScan(), which runs from the idle task will detect stack overflows and report SMXE_HEAP_FENCE_BRKN errors, soon after they occur. Since a chunk debug control block (CDCB) has an owner field, it is possible to determine which task had the stack overflow. Since stacks overflow toward lower memory, the "upper" fences and possibly the heap chunk above the stack will be damaged. The CDCB for the stack chunk is below the stack and thus will not be damaged.

Testing heap fences is not foolproof because stacks often skip over fences without changing them. The same can be said for stack scanning — i.e. stacks can skip over the pattern in the upper stack and do damage above it. So, having both techniques is helpful.

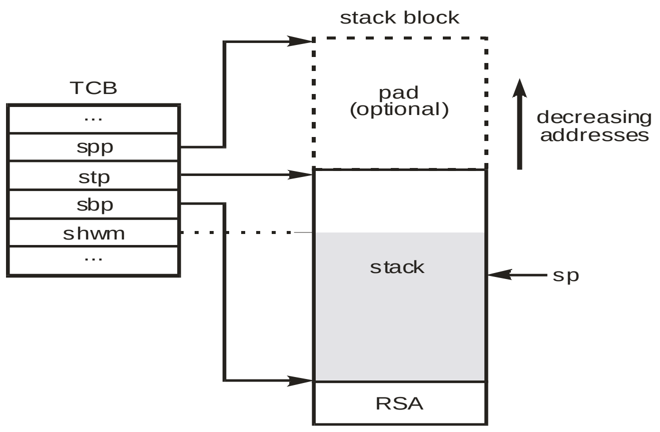
shared stacks

Shared, or unbound, stacks are a unique feature of smx intended to reduce RAM usage by sharing stacks between tasks, whenever possible. Basically, these are stacks which are obtained from a stack pool when a task is dispatched. All stacks are the same size. The stack pool is allocated from ADAR the first time a task is created. The configuration parameters NUM_STACKS, STACK_SIZE, and STACK_PAD_SIZE, defined in acfg.h, and SMX_RSA_SIZE, defined in xcfg.h determine the number and size of the stack blocks. The three size parameters are added together and the resulting block is up-sized to be a multiple of SB_STACK_ALIGN, which is defined in the smxBase processor architecture header file, e.g. barm.h, bcf.h. This is 4 for most processors, but 8 for ARM and ARM-M. This assures that all stacks from the stack pool will be properly aligned.

Tasks using shared stacks are called one-shot tasks. For more discussion of the benefits of using shared stacks see the One-Shot Tasks chapter.

stack control

Stacks are controlled by TCB fields. The following diagram shows the correlation between the TCB fields and the task's stack:



Note: Stacks grow "up" toward memory location 0. Hence "top" means the lowest address of the stack block and "bottom" means the highest address of the stack block. Also the stack pointer, sp, is pre-decremented on a push (write) and post-incremented on a pop (read).

tcb.stp points to the top word of the stack, and tcb.sbp points one word below the bottom of the stack. Initially sp = tcb.sbp. Hence, the first push is into the bottom of the stack. The stack is full when sp equals tcb.stp. tcb.sp stores the stack pointer when a task is suspended. tcb.sp == 0 indicates that the task has never been suspended or has been stopped.

tcb.spp points to the stack pad which is above the stack (i.e. at the "top" of the stack block).

tcb.sbp also points to the start of the Register Save Area (RSA). SMX_RSA_SIZE is defined in the smx processor header file (e.g. xarm.h for ARM). RSA is used to save non-volatile registers when a task is suspended. Volatile registers are not saved by SSRs since C compilers do not expect them to be saved. Volatile registers are saved in the stack by smx_ISR_ENTER() in smx ISRs. See the assembly include file for your processor architecture / tool chain (e.g. xarm_iar.inc).

The TCB contains additional stack fields, not shown above: tcb.ssz is the size of the stack (minus any loss for alignment, and not including the optional pad above the stack). tcb.shwm is the stack high water mark, indicating the maximum stack usage since startup. It is used by smxAware to generate the stack display. For more information, see the discussion on stack scanning, below.

stack pads

Stacks can be padded by setting STACK_PAD_SIZE in acfg.h to the desired pad size. All stacks (permanent and shared) are padded by this amount. See the diagram above to see where the stack pad is placed. Stack pads allow a system to keep running when a stack overflows, because the overflow will be contained in the pad and not harm other data above the stack.

Because stack overflows can lead to puzzling behavior and waste time to diagnose, using stack pads during development will speed up progress. We recommend using large stack pads during initial development, even if stacks must be put into external RAM. Stack usages of tasks invariably increase over time as complexity grows, so this saves hassling with stack overflows. Pad size can be reduced, later in the development process, in order to reduce memory required or to improve performance.

finalizing stack sizes

To finalize stack sizes, it is best to allow the system to run for an extended period of time using moderate-size pads, then use smxAware to inspect stack usage for each task or compare tcb.shwm to tcb.ssz for each task, using the debugger. Of course, it is important that the system perform all of its intended functions during the test period. It is also important to work with the release version, since optimization uses more stack space. Because stack scanning records exact maximum stack usage, this is a reliable way to fine-tune stack sizes.

Even so, unexpected sequences of operations in the field or software upgrades can cause stack overflows. For this reason, it is recommended to leave small stack pads in the final system — as little as 2 words is helpful. The advantage is that smx will detect and report any stack overflows, but hopefully the pad is large enough to allow the system to continue running. Then, if there is a reporting mechanism in place, it is possible to detect a stack overflow problem without a system failure.

creating and filling stacks

The stack pool is created the first time smx_TaskCreate() is called. If STACK_SCAN in acfg.h, is 1, the stacks are filled with the 32-bit SB_STK_FILL_VAL, defined in the smxBase processor-architecture header file (e.g. barm.h). This can be set to any desired pattern. Then the stacks are linked into the freestack list for use by the scheduler. The first two words of each stack are used for a link pointer to the next stack and to record the previous owner.

A permanently-bound stack is allocated from the heap and assigned to a task if the size parameter in smx_TaskCreate() is non-zero. If STACK_SCAN is true, the stack is filled with SB_STK_FILL_VAL.

stack overflow detection

Typically, there are a large number of task stacks in a system and it is desirable to minimize their sizes in order to reduce RAM usage. However, doing so can result in stack overflows, which can cause serious problems that are difficult to track down. Hence, detecting a stack overflow as soon as possible and taking appropriate action is of utmost importance. To do this, stack overflow is checked by the scheduler whenever a task is suspended or stopped. Stack checking is enabled by setting the tcb.flags.stk_chk flag and clearing the tcb.flags.stk_ovfl flags when a task is first created.

When stack checking is enabled, the scheduler compares the current task's stack pointer to its stack top and the stack's high water mark to its stack size. If overflow has occurred (sp < tcb.stp || tcb.shwm > (smx_ct->sbp - smx_ct->stp)), smx_EM() is called and a stack overflow error is recorded. Its error number is loaded into the smx_ct->err field and into smx_errno; in addition, smx_errctr and smx_errctrs[SMXE_STK_OVFL] are incremented. If smx_ct->flags.stk_ovfl is zero, error records are saved in EB and EVB and a STACK OVERFLOW message is sent to the console.

Since further stack overflow reports can be a nuisance, the ct->flags.stk_ovfl flag is set to disable further recording in EB and EVB and display of stack overflow errors for this task. Otherwise stack overflow reports might cause other errors to go unnoticed. This applies only to this particular task; other tasks are not affected. The errnos and counters continue to be updated for every detected stack overflow for this task.

stack overflow handling

After reporting stack overflow: If a stack pad is present and sp has gone only into it, the task will be allowed to suspend or stop, and resume or restart normally, at a later time.

If sp has gone or will go above the top of the stack block (smx_ct->spp) or if the stack high water mark has exceeded the stack block size, then whatever is above the stack block has been or will be damaged. In the case of a permanent stack, this would likely be the heap control block above the stack[2]. In the case of a shared stack, it would likely be the bottom of another stack. Because recovery is unlikely, smx_EMExitHook(errnum) is called. This is a user call back function, which can take appropriate action for the application. Currently it calls sb_Exit(), which reboots.

foreign stacks

Some third-party libraries change to foreign stacks — i.e. special stacks elsewhere in memory. (The PC BIOS is a well-known example of this.) It will have no effect on stack scanning, but if a preemption occurs while in a foreign stack, the scheduler will definitely report a stack overflow. To avoid this, wrap such a library call as follows:

```
smx_TaskSetStackCheck (smx_ct, FALSE);
/* call to function that switches stacks */
smx_TaskSetStackCheck (smx_ct, TRUE);
```

Except for this, it is recommended to keep stack checking on at all times.

Advanced Topics

stack scanning

The advantage of stack scanning vs. sp overflow detection is that stack writes that occur while the task is running are recorded. By contrast, testing smx_ct->sp detects only the furthest stack excursion at the time when a task is either suspended or stopped by the scheduler. Since nested subroutine calls can occur at other times, a task's shwm is a more reliable indicator of stack overflow than its sp — especially for higher priority tasks which may seldom be preempted.

Stack scanning is implemented by filling a stack with a pattern before the task starts, then scanning periodically to see how far the pattern goes down. The address of last word with the pattern is subtracted from task->sbp and stored in task->shwm. This can be compared to task->ssz to see how much stack has been used. The scan pattern is SB_STK_FILL_VAL, which is defined in the smxBase processor-architecture header file (e.g. barm.h).

Except for SS scan following startup, all other stack scanning must be enabled by setting SMX_CFG_STACK_SCAN, in xcfg.h, when the smx library is built. This causes the stack scan code to be included. For the code to operate, STACK_SCAN, in acfg.h, must also be set. Scanning is done from the idle task. If the application switches to lower power mode during idle time, turning scanning off will result in more time spent in the low-power state, so in these systems, it may be best to enable it only during development or when debugging a problem. Alternatively, the frequency of stack scanning may be reduced by calling smx_StackScan() less often from the idle task.

[2] For an exception to this, see the discussion above in the permanent stacks section.

As noted previously, smx supports two types of task stacks — permanent and shared — and a system stack, SS. Each requires different treatment. All are processed by smx_StackScan(), which is called from the idle task. During normal operation all stacks are scanned in rotation — one stack per pass through idle. Scanning SS is a special case, which is discussed later.

For task stacks, scanning starts at the top of the pad or the top of the stack if there is no pad. The result of stack scanning is to steadily increase a task's shwm, as the task uses more and more of its stack, until a stable value is reached, which should be the maximum stack needed by the task under all conditions. However, be aware of special conditions, which call special functions, and thus require more stack.

permanent stack scanning

The permanent stack case is simpler, so we will cover it first. A permanent stack is filled with the scan pattern when obtained from the heap and assigned to a task by smx_TaskCreate(). Scanning is performed by smx_StackScanB() (B = bound) and is from the top of the stack (pad) to the first non-pattern word. The address of this word is subtracted from task->sbp and loaded into task->shwm; the task->flags.stk_hwmv (high water mark valid) flag is also set. shwm represents the number of bytes of stack, including the stack pad, if any, that have been used, to date. If it is greater than task->ssz, which is the size of the stack proper, then there has been an overflow into the stack pad, and the stack size should be increased.

If a TCB is not in use, no scan is performed. For permanent stacks the scan will take less and less time as the stack grows upward. Hence scan times for even very large stacks may not take long after the system has run for a while, unless stacks or stack pads are greatly oversized.

shared stack scanning

Shared stacks are a bit more complicated because they are released by tasks when no longer needed. If stack scanning is enabled, such stacks are released to the smx_scanstack pool instead of the smx_freestack pool, where they are scanned from idle. When smx_StackScan() runs, it first checks for a stack in the scanstack pool. If one is found, smx_StackScanU() (U = unbound) is called to scan it and update the previous owner's shwm. If the previous owner is still stopped, its stk_hwmv flag is set. Then, the rest of the stack is filled with the fill pattern and it is moved to the freestack pool. The net result is that the entire free stack block, except for the two top words (reserved for link and onr fields) are filled with the scan pattern. If there are no stacks in the scanstack pool, then StackScan calls smx_StackScanB(), and the stack of the next bound task is processed, as described above. Note: a shared stack that is currently bound to a task is treated like a permanent stack.

Since scanning is done in idle, which is the lowest priority task, there is a possibility that a higher priority task may not be able to run because the freestack pool is empty, yet there are stacks in the scanstack pool. See the out of shared stacks section below for how this is handled.

stack scanning details

Because smx_StackScan() is called from the lowest priority task, it has been designed to tolerate tasks preempting and running during a scan. These could be other tasks or the same task for which the stack is being scanned. As a result, a stack can be released while being scanned or even a task can be deleted while its stack is being scanned. While this results in difficult code for smx_StackScan(), it is preferable to the alternative of locking smx_StackScan() or making it an SSR because that would increase the latency of all tasks.

Scanning is performed from the top of the stack block (pad) down. Hence, shwm can be used to determine that a stack overflow has occurred, but not how far the stack has overflowed, unless a stack pad is present. In particular, shwm cannot be used to determine if a stack has overflowed its stack block.

Stack usage information is displayed by the smxAware Graphical Analysis Tool as a bar chart and in the smxAware stack text window numerically. See the smxAware User's Guide for details and a screen shot of the stack display. smxAware uses the task's shwm, if the stk_hwmv flag is true; if not, smxAware scans the stack, itself, in order to present a more accurate value.

Stack scanning is very fast on most processors and there should be no problem enabling it even in a release build. Since stack scanning normally runs only from idle, it uses only idle time and should not interfere with more important tasks. The exception to this is if tasks requiring stacks cannot get them from the freestack pool because they are waiting to be scanned (see section out of shared stacks below). However, such tasks normally do not have tight deadlines. If they do, they should be given permanent stacks. As mentioned earlier, systems that go into low power mode during idle time should disable stack scanning in a release build, so the system spends more time in low power mode.

out of shared stacks

When the smx scheduler is ready to dispatch an unbound task, ctnew, it may find that the freestack pool is empty. In that case, it checks the scanstack pool to see if any stacks are waiting to be scanned. If so, the scheduler calls smx_StackScanU() to scan the first waiting stack and move it into the freestack pool, from which it is then given to ctnew. A potential problem occurs if idle was preempted in the middle of StackScanU(). This is indicated by smx_inssu == TRUE. In this case the scheduler performs an *idleup* operation, which runs idle in place of ctnew in order to complete running StackScanU(). When it is done, idle drops back to priority level 0 and restarts the scheduler, which now is able to dispatch ctnew. Using idleup avoids priority inversion. Without it, ctnew might have to wait for mid-priority tasks to run before idle could run and release a stack.

If the scanstack pool is also empty, then an SMXE_OUT_OF_STKS error occurs. After the first time, a TCB flag inhibits reporting this error again to avoid saturating the error buffer, event buffer, and console with this error. Also, running out of stacks might be normal, and not an error — see the stack pool size discussion in the One-Shot Tasks chapter.

system stack fill and scan

System stack (SS) overflow can lead to serious problems that rarely occur — e.g. when all interrupts happen at the same time. Detecting and protecting against it are important to system integrity, since ISRs, LSRs, the scheduler, and even the error manager, smx_EM(), depend upon correct operation of SS. This is all the more true since SS is a hidden stack that could be completely forgotten. The following safeguards are implemented for SS:

SS is filled during startup with the SB_STK_FILL_VAL, which is defined in the smxBase processor architecture file (e.g. barm.h). As shipped, this is the value expected by the debugger; the same value is used for task stacks. SS is filled prior to calling the C library initialization function, which clears uninitialized variables and loads initial values into initialized variables. This function will also call static initializers if C++ is being used.

At the start of main() in main.c, SS is scanned down to the first word of non-pattern and the number of unused bytes is reported. If this is less than 40 bytes, a debug trap occurs, so SS size can be increased by the programmer. The debug trap occurs only when running under the debugger, not in the released system. When working in C++, static initializers are added by the compiler without the programmer's knowledge of how much stack may be needed. Hence SS can easily be exceeded and a much larger cushion, such as 100 bytes may be better.

At the start of application initialization (ainit() in main.c), SS is again filled with the SB_STK_FILL_VAL all the way to the bottom (this is possible because SS is not being used at that time). SS is associated with the Nulltask so that it is scanned during normal operation by smx_StackScan() in idle. As shipped, if there are fewer than 40 bytes of unused space at the top of SS, a debug trap will occur in the debugger, and SS size should be increased. SS size can be tuned down late in a project, but it is worth keeping a good margin for safety — especially if nested interrupts are permitted. If there is no space in SS following a scan (implying a probable overflow) an SMXE_STK_OVFL error is reported and aexit() is called. Since recovery is unlikely, this should probably lead to a system reboot or stop.

Chapter 7 Intertask Communication

introduction

Once a system has been divided into tasks, communication between the tasks becomes important if the system is to do anything useful. Intertask communication consists not only of exchanging data but also of synchronization and control.

smx provides six mechanisms for intertask communication:

(1) semaphores

(2) mutexes

(3) event queues

(4) event groups

(5) pipes

(6) exchange messaging

Note that intertask communication also includes communication with *LSRs*. Since these are not discussed until a later chapter, mention of LSRs is omitted in most of what follows. LSRs are discussed in the Service Routines chapter.

smx provides a rich set of ITC operations. However one can choose a minimalist approach, if desired, such as messages only. Generally such an approach simply reflects more complexity into the application. It is better to choose mechanisms which fit the tasks at hand. The following chapters are intended to introduce you to mechanisms above and show how to use them.

All calls which suspend or stop the current task have a timeout parameter. The timeout is intended primarily as protection to prevent a task from waiting forever, but it can also be used as a delay. The resolution of timeouts is specified by TIMEOUT_PERIOD in acfg.h. Resolution can vary from 1 tick up to seconds or more.

stop SSRs

Every SSR discussed in this chapter that waits has an equivalent stop version. These have "Stop" appended to their names (e.g., smx_MsgReceiveStop()). They operate the same as their suspend counterparts, except that ct is always stopped and the result of the SSR is passed in as the task parameter, when the task is restarted. For this to work the task main function must be of the form:

```
void  taskMain(ptype par) or
rtype  taskMain(type par)
```

where ptype is the parameter type, such as u32, BOOLEAN, MCB_PTR, etc. See the Reference Manual for details. Stop SSRs are primarily used by one-shot tasks. See the One-Shot Tasks chapter, for more discussion.

using ITC mechanisms

The chapters that follow illustrate how ITC mechanisms can be used to achieve effective solutions to typical embedded systems problems. The advantages are:

(1) The mechanisms are clear and apparent, not buried in obscure code.

(2) Each piece of code (i.e. task) is largely spared the complexities of these real-world considerations and thus can be focused on its main mission.

(3) The mechanisms can be easily adjusted with minimal impact upon code.

The essence of this is that you can create the code for each task without too much concern for the what-if's of the real world. Hence the code for each task can focus solely on its mission. The what-if's can be handled by structuring with smx objects. (Of course, both parts of the design must be done together.)

Chapter 8 Semaphores

Semaphores regulate the flow of tasks.

```
BOOLEAN     smx_SemClear(SCB_PTR sem)
SCB_PTR     smx_SemCreate(SMX_SEM_MODE mode, u32 lim, const char *name)
BOOLEAN     smx_SemDelete(SCB_PTR *sem)
BOOLEAN     smx_SemSignal(SCB_PTR sem)
BOOLEAN     smx_SemTest(SCB_PTR sem, u32 timeout)
void        smx_SemTestStop(SCB_PTR sem, u32 timeout)
u32         smx_SemPeek(SCB_PTR sem, SMX_PK_PARM par)
```

introduction

Semaphores are used primarily for signaling acknowledgments and completions, and for mutual exclusion, although it is generally preferable to use a mutex for mutual exclusion (see the Mutex chapter, which follows). When used for these purposes, there is normally no associated data; otherwise it may be preferable to use messages and an exchange.

The smx semaphore is capable of operating in one of 6 modes:

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

The semaphore mode is determined when the semaphore is created, by the mode and lim parameters of the create call. For example, to create a binary resource semaphore:

```
SCB_PTR  sbr;
sbr = smx_SemCreate(RSRC, 1, "sbr");
```

It is recommended that the name chosen for a semaphore indicate its operating mode, in some manner. (For example, “br” for binary resource, in the above example.) This will help to avoid misuse. Semaphore modes are discussed in the sections that follow. NOTE: This manual and esmx use abbreviated names for constants such as RSRC. See esmx.h for definitions and the abbreviated names section of the Under the Hood chapter in this manual for an explanation.

resource semaphore

Resource semaphores are used to regulate access to resources, such as printers, and critical code sections, such as non-reentrant subroutines. The above binary resource semaphore could be used as follows:

```
TCB_PTR  t2a, t3a;
SCB_PTR sbr;
```

```
void tMain(void)
{
    sbr = smx_SemCreate(SMX_SEM_RSRC, 1, "sbr");
    t2a = smx_TaskCreate(t2a_main, P2, 500, NO_FLAGS, "t2a");  /* 500= stack size */
    t3a = smx_TaskCreate(t3a_main, P3, 500, NO_FLAGS, "t3a");
    smx_TaskStart(t2a);
}

void t2a_main(void)
{
    smx_SemTest(sbr, TMO)
    smx_TaskStart(t3a);
    /* use resource here */
    smx_SemSignal(sbr);
}

void t3a_main(void)
{
    smx_SemTest(sbr, TMO)
    /* use resource here */
    smx_SemSignal(sbr);
}
```

In the above example, t3a preempts t2a after t2a has sbr. Hence, it waits at sbr, thus allowing t2a to resume and finish using the resource. Then, t2a signals sbr that it is done, and t3a resumes and uses the resource. When done, t3a signals sbr, so another task can use the resource.

A **binary resource semaphore**, such as sbr, is used to share a single resource, as in the above example. Its internal counter can have only values of 1 (available) and 0 (unavailable). smx_SemTest() passes if the count is 1, and the current task is allowed to proceed, as does t2a above. If the count is 0, smx_SemTest() fails and the task is enqueued in the semaphore's task wait list, in priority order, as is t3a, above. When the semaphore is signaled, the first task (i.e. the longest waiting at the highest priority) is resumed, and the count remains 0. If no task is waiting, the count increases to 1. Once the count is 1, additional signals are ignored. Note: there should be no additional signals for binary resource semaphores — tests and signals should always be paired, as shown in the example. Also note that a resource semaphore starts with its count already set, as if it had been signaled.

The **multiple resource semaphore** is a generalization of the binary resource semaphore, intended to regulate access to multiple identical resources. It is commonly called a *counting semaphore.* A multiple resource semaphore is created as follows:

```
SCB_PTR  sr;
sr = smx_SemCreate(RSRC, lim, "sr");
```

Where lim is the number of resources, which must be > 1. lim == 0 will trigger an SMXE_INV_PARM error.

A good example of using a multiple resource semaphore is for a block pool consisting of NUM blocks of the same size:

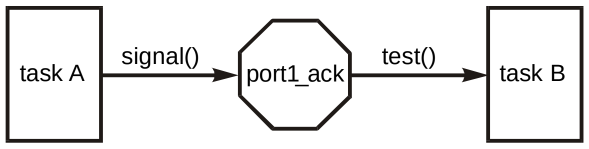
```
PCB_PTR  blk_pool;
SCB_PTR  sr;

void init(void)
{
    u8* p;
    p = (u8*)smx_HeapMalloc(1000);
    blk_pool = smx_BlockPoolCreate(p, NUM, SIZE, "blk_pool");
    sr = smx_SemCreate(RSRC, NUM, "sr");
}

void t2a_main(void)
{
    u8* bp;
    BCB_PTR blk;

    blk = GetBlock();
    bp = (u8*)smx_BlockPeek(blk, SMX_PK_BP);
    /* process block here using bp */
    RelBlock(blk);
}

BCB_PTR GetBlock(void)
{
    smx_SemTest(sr, TMO);
    return(smx_BlockGet(blk_pool, NULL, 0));
}

void RelBlock(BCB_PTR blk )
{
    smx_BlockRel(blk);
    smx_SemSignal(sr);
}
```

GetBlock() first tests sr. If its count > 0, the test passes and the next blk_pool block handle is returned to the caller. If not, the current task is suspended, in priority order, on sr's wait list. RelBlock() releases the block back to blk_pool and signals sr. If tasks are waiting for blocks, the first waiting task will get the block and resume. If no task is waiting, sr's count is incremented. Thus the count always equals the number of available blocks. The maximum possible count is lim. Signals after count == lim are ignored. This helps to protect resources if spurious signals occur, but it is not foolproof.

In the above example, t2a demonstrates using calls to the functions that get and release blocks. Note that the t2a code is unaware of the semaphore, yet the semaphore causes it to wait if no blocks are available.

event semaphore

Event semaphores provide a method to synchronize tasks with internal or external events and with other tasks. For example, one task can signal another:



where port1_ack is an event semaphore. An event semaphore is created as follows:

```
SCB_PTR  se;

se = smx_SemCreate(EVENT, mode, "se");
```

mode == M for a multiple event semaphore and == B for a binary event semaphore. In both cases, the semaphore's internal count starts at 0 — i.e. there are no events to report, yet.

The **multiple event semaphore** is the more common of the two. When a signal occurs for it, the first waiting task is resumed with TRUE. If no task is waiting, the count is incremented by 1 up to a maximum of 255. If a signal is received after count == 255, the count is not changed, and SEM_CTR_OVFL error is reported. Assuming that its counter does not overflow, a multiple event semaphore keeps an exact count of signals received, whether or not any task is waiting. Hence, it is good for situations where every event counts and missing events could have bad consequences.

Of course, a simple counter could keep an accurate count of events, but a task cannot wait at a counter. An important property of the multiple event semaphore is that it handles both the case where events get ahead of the processing task and the case where the processing task gets ahead of events. In the first case, the semaphore's internal count is incremented; in the second case, the task waits. The following is an example showing this:

```
SCB_PTR  sme;
TCB_PTR  t2a;

sme = smx_SemCreate(EVENT, M, "sme");

void ISR_EventA(void) {
    smx_LSR_INVOKE(LSR_EventA, 0);
}

void LSR_EventA(num) {
    smx_SemSignal(sme);
}

void t2a_main(void) {
    while (smx_SemTest(sme, TMO)) {
        /* process event A */
    }
    /* handle timeout or error */
}
```

This is a very simple example in which the occurrence of external event A causes an interrupt, which causes ISR_EventA() to run. It in turn, invokes LSR_EventA(), which signals semaphore, sme. Note that task t2a tests sme for every loop, including the first. If the internal count in sme is greater than 0, the count will be decremented, and t2a will process event A. Looping continues until the internal count of sme is 0, then t2a waits. Hence, no events will be lost if t2a gets behind and when all events have been processed, t2a will wait for the next event. This synchronization between external events and an internal task produces a solid design.

Note that t2a will only wait for TMO ticks, after which the while loop is exited and other code runs to deal with the timeout or error. It is always a good idea to specify reasonable timeouts on task waits in order to prevent tasks from being permanently suspended, due to unexpected combinations of events. This improves reliability.

A **binary event semaphore** is created as follows:

```
SCB_PTR  sbe;
sbe = smx_SemCreate(EVENT, B, "sbe");
```

A binary event semaphore comes into play when it is necessary to notify a task of only the first event in a string of events. In this case, subsequent events will be processed with the first event, and thus it is not necessary to record their occurrences. Binary semaphores are useful in *producer/consumer* applications where the producer signals more often than the consumer tests. The consumer loops to get all items the producer has made available, to date, then tests the semaphore again, rather than testing the semaphore for each item produced. A binary semaphore is appropriate for this situation because the task does not care how many times it was signaled; it only cares that it was signaled. The binary event semaphore does not have the potential count overflow problem of the multiple event semaphore, because its count is limited to 1, since it doesn’t record the number of signals occurred.

A good usage example is where received bytes are put into a buffer by t3a and processed by t2a, as demonstrated in the following example:

```
PICB_PTR inpipe;
SCB_PTR sbe;
TCB_PTR t2a, t3a;

void init(void)
{
    void* ppb;

    /* create inpipe and sbe binary event semaphore*/
    ppb = smx_HeapMalloc(IP_LEN);
    inpipe = smx_PipeCreate(ppb, IP_WIDTH, IP_LEN, "inpipe");
    sbe = smx_SemCreate(EVENT, B, "sbe");

    /* create t2a and t3a and start */
    t2a = smx_TaskCreate(es4_t2a_main, P2, 0, NO_FLAGS, "t2a");
    t3a = smx_TaskCreate(es4_t3a_main, P3, 0, NO_FLAGS, "t3a");
    smx_TaskStart(t3a);
    smx_TaskStart(t2a);
}

void es4_t3a_main(void)
{
    u8  ch;
    do
    {
        ch = *inport;
        smx_PipePut8(inpipe, ch);
        smx_SemSignal(sbe);
    }  while (ch);
}

void  es4_t2a_main(void)
{
    u8  ch;
    char*  dp;

    while (smx_SemTest(sbe, TMO))
    {
        dp = &dbuf;

        do
        {
            smx_PipeGet8(inpipe, &ch);
            *dp++ = ch;
        }  while (ch);
        sb_MsgVarDisplay(INFO, dbuf);
    }
}
```

In the above example, t3a gets characters from inport and puts them into inpipe, until it gets a NUL character, then it stops. When t3a is done, t2a is started. It tests sbe, which has a count of only 1, no matter how many characters t3a loaded into inpipe. Hence, t2a will run only once, then wait for the next signal. With a multiple event semaphore t2a would run once for every character put into inpipe. It would obviously be wasteful to write t2a_main() to process only one character at a time, if more are present in inpipe.

threshold semaphore

A threshold semaphore is useful when it is desired to respond only every Nth event. For example, it might be desirable to respond to every 10th revolution of a wheel, rather than to every revolution. Another example is to permit, a certain number of retries before taking a communication line out of service. A threshold semaphore is created as follows:

```
SCB_PTR  st;
st = smx_SemCreate(THRES, T, "st");
```

where T is the threshold, e.g. 10. The initial count is set to 0. The motivation for using a threshold semaphore is similar to that of using a binary event semaphore — namely to avoid unnecessary work. However, in this case, events are not discarded; they are diminished and delayed. If T == 1, a threshold semaphore is the same as a multiple event semaphore. lim must be > 0 or SMXE_INV_PARM error is reported.

The example above serves to illustrate the threshold semaphore, with “st” substituted for “se”. t3a signals st every time it finishes processing some data. This increments the signal counter in st and causes it to be compared to the threshold, T. If it is greater than or equal to the threshold, the first task waiting at st (t2a) will be resumed and the signal counter will be reduced by T. However, count is not allowed to be incremented over 255. It this occurs, count stays at 255 and SMXE_SEM_CTR_OVFL error is reported.

gate semaphore

A gate semaphore allows starting all waiting tasks simultaneously. One signal opens the gate. To create a gate semaphore:

```
SCB_PTR  sg;
sg = smx_SemCreate(GATE, 1, "sg");
```

The lim parameter must be set to 1. Tasks are suspended and resumed in FIFO order in the wait list. Hence, when the tasks start running, priority and waiting order will be preserved.

Gate semaphores are useful for starting multiple tasks at once. This example also shows use of a gate and a threshold semaphore, in combination, to regulate work, so that all work is completed before starting the next cycle.

```
TCB_PTR  t2m;           /* master task */
TCB_PTR  t2s1, t2s2;  /* slave tasks */
SCB_PTR  sg, st;        /* gate and threshold semaphores */

void init(void)
{
     /* create semaphores */
    sg = smx_SemCreate(GATE, 1, "sg");
    st = smx_SemCreate(THRES, 2, "st");

     /* create tasks and start */
    t2s1 = smx_TaskCreate(es5_t2s1_main, P2, 0, NO_FLAGS, "t2s1");
    t2s2 = smx_TaskCreate(es5_t2s2_main, P2, 0, NO_FLAGS, "t2s2");
    t2m = smx_TaskCreate(es5_t2m_main, P2, 0, NO_FLAGS, "t2m");
    smx_TaskStart(t2s1);
    smx_TaskStart(t2s2);
    smx_TaskStart(t2m);
}

/* slave task 1 */
void t2s1_main(void)
{
    while(1)
    {
        smx_SemTest(sg, INF);   /* wait at gate */
        /* do process 1 here */
        smx_SemSignal(st);      /* tell master done */
    }
}

/* slave task 2 */
void t2s2_main(void)
{
    while(1)
    {
        smx_SemTest(sg, INF);
        /* do process 2 here */
        smx_SemSignal(st);
    }
}

/* master task */
void t2m_main(void)
{
    while(1)
    {
        /* initialize process 1 */
        /* initialize process 2 */
        smx_SemSignal(sg);      /* start both slaves at once */
        smx_SemTest(st, INF);   /* wait for both slaves to finish */
    }
}
```

In this example, t2s1 and t2s2 are slave tasks. They start by waiting at the gate semaphore, sg, for work to do. t2m is the master task. It starts by initializing process 1 and process 2. When t2m is done initializing the processes, it signals sg. This starts t2s1 and t2s2, simultaneously. Each task performs its process. These processes may involve waiting for other events or resources, hence their execution times may not be predictable. By launching both tasks simultaneously, one can run while the other is waiting, thus making more efficient use of the processor. When each task has finished, it signals the threshold semaphore, st, then waits at sg. When both slave tasks have signaled st, the master task, t2m, resumes and starts the next cycle of processing.

In this example, the master task has the same or lower priority than the slave tasks. Otherwise, as soon as t2s2 signals st, the master task will preempt, prepare the processes, then signal sg. However, only t2s1 will be waiting there. t2s2 will resume due to the master task suspending on st; then t2s2 will suspend on sg. Hence t2s2 will miss the sg signal from the master task and not perform its processing. This illustrates the importance of getting priorities right.

other semaphore services

smx_SemTestStop(sem, tmo) is the stop variant of smx_SemTest(), intended for use by one-shot tasks. It operates the same as smx_SemTest(), except that it stops and restarts the current task, rather than suspending and resuming it. See the One-Shot tasks chapter for discussion of how to use one-shot tasks.

smx_SemClear(sem) resumes all waiting tasks with FALSE, and resets count to its initial value. It is useful in recovery situations, such as SMXE_SEM_CTR_OVFL.

smx_SemDelete(*sem) resumes all waiting tasks with FALSE, clears and releases the Semaphore Control Block (SCB) back to the SCB array, so it can be reused, and clears the sem handle so that sem cannot be used again, by mistake.

Normally a semaphore’s wait list should be empty before it is cleared or deleted, except possibly in recovery situations. Resuming all waiting tasks with FALSE, is done to prevent losing tasks that were waiting with INF timeouts. This illustrates the importance of testing return values, before proceeding. For example, if a resource semaphore is deleted and tasks that were waiting do not test for TRUE before proceeding, then resource conflicts are likely to occur. Likewise, if an event semaphore is deleted, tasks waiting for it might proceed even though no event actually occurred. Testing SSR return values is good defensive programming.

Other than the clear function, smx provides no way to change a semaphore because it is risky to do so. It is better to delete the semaphore, then recreate it with the desired parameters.

summary

The smx semaphore operates in one of 6 modes: The binary resource semaphore is adequate to regulate access to critical software sections and resources for simple systems. Mutexes should be used in complex systems. The multiple resource semaphore offers the unique capability to regulate access to multiple identical resources, such as blocks from a block pool. The binary event semaphore is useful when only the first of a series of events needs to be recorded. The multiple event semaphore is used when each event is important. It will record up to 255 events. The threshold semaphore is used when notification of every Nth event is desired — for example, N slave tasks reporting that they are done. The gate semaphore serves to start all waiting tasks on one signal, such as a master task starting all slaves for the next cycle.

exercises

1. Show how a gate semaphore can be used for cooperative run-time limiting based upon task run-time counts.
2. Demonstrate how smx_SemClear() can be used to recover from a SMXE_SEM_CTR_OVFL error. (This requires designing tasks to handle the error when FALSE is returned.)

Chapter 9 Mutexes

```
BOOLEAN     smx_MutexClear(MUCB_PTR mtx)
MUCB_PTR    smx_MutexCreate(u8 pi, u8 ceiling, const char *name)
BOOLEAN     smx_MutexDelete(MUCB_PTR *mtx)
BOOLEAN     smx_MutexFree(MUCB_PTR mtx)
BOOLEAN     smx_MutexGet(MUCB_PTR mtx, u32 timeout)
void        smx_MutexGetStop(MUCB_PTR mtx, u32 timeout)
BOOLEAN     smx_MutexRel(MUCB_PTR mtx)
```

introduction

Mutexes "mutual exclusion" semaphores offer a safer method of mutual exclusion than binary resource semaphores. They are used to limit access to non-reentrant sections of code or to system resources that cannot be shared. Mutexes have two states: free and owned. A mutex can be owned by only one task at a time. This task is called the owner. Because of the ownership property, terminology is slightly different for mutexes than for semaphores. Get and release for a mutex are similar to test and signal for a semaphore.

comparison to binary resource semaphores

Binary resource semaphores are useful in simple systems, but they have serious shortcomings for use in more complex systems:

(1) Task lock-ups can occur due to more than one test of the same semaphore from the same task. Timeouts on semaphore tests help to overcome this problem.

(2) Unbounded priority inversions. There is no owner concept for a binary semaphore, thus no way to promote the priority of the task using the resource when a higher-priority task needs it.

(3) Any task can signal a binary resource semaphore, thus permitting an access conflict.

(4) There is no way to release a binary semaphore if the task which owns it is deleted, stopped, or suspended.

Mutexes are safer than binary resource semaphores for the following reasons:

(1) Nested testing by the same owner is permitted.

(2) Priority promotion of the owner is automatic.

(3) They cannot be released by a non-owner.

(4) smx_TaskDelete() automatically releases any mutexes owned by the task being deleted. Prior to stopping or suspending a task, it is possible to search for mutexes owned by it and free them.

Nested testing of semaphores is a problem in larger projects. For example, if there are two functions that both test the same semaphore and one function calls the other, then the test will be done twice. The second test by the same task will cause the task to lock up. This is particularly a problem with libraries in which functions that use the same binary resource semaphore may call each other. A mutex has an internal nesting counter that simply increments for each get and decrements for each release. Hence, no task lockup will occur.

Priority promotion of the mutex owner is necessary to prevent unbounded priority inversion of higher priority tasks that share the mutex. (Bounded priority inversion occurs when a low priority task keeps a high priority task waiting while it uses a resource. Unbounded priority inversion occurs when the low priority task can be preempted by one or more mid-priority tasks, thus keeping the high priority task waiting for an indeterminate amount of time.) This can cause glitches in a system's performance. The smx mutex provides both ceiling and inheritance priority promotion. These are discussed below.

A binary resource semaphore, being used for mutual exclusion, will accept a signal from any task—not only the one using the resource. Again, mutual exclusion breaks down. By contrast, a mutex may be released only by its owner.

Not being able to release owned semaphores when a task is deleted, suspended, or stopped by another task obviously can cause serious problems — other tasks will become permanently blocked at those semaphores.

For the above reasons, mutexes are recommended over binary resource semaphores if memory space is available. Adding mutexes to a design that is already using semaphores adds a minimum of 2000 bytes of code. A mutex control block is 28 bytes vs. 16 for a semaphore. In addition, there is more run-time overhead on mutexes.

creating and deleting mutexes

A mutex is created as follows:

```
MUCB_PTR   mtx;

void  function(void)
{
     mtx = smx_MutexCreate(pi, ceil, "mtx");
}
```

It is created in the free state. ceil is the ceiling priority of the mutex. A task is immediately promoted to this priority when it becomes the owner of the mutex. (Note: promote means to increase task priority, or leave it the same; demote means to reduce it, or leave it the same.) If non-zero, pi enables priority inheritance, which means, if a higher priority task waits on an owned mutex, the owner's priority is temporarily promoted to its priority. Also, if it is waiting in another queue, the owner is moved to the proper position in that queue. Priority propagation occurs if the owner is waiting on another mutex and that mutex owner's priority is less than the new priority and pi is non-zero for that mutex. This chain continues until an owner is found that does not meet these conditions.

The two forms of priority promotion (ceiling and inheritance) each have advantages and disadvantages. Ceiling is simplest and results in the fewest task switches. Also, it prevents task deadlock. However, it requires knowing the top priority among the tasks sharing the resource, and it may unnecessarily block the same or lower priority tasks that do not need the resource. In simpler systems, these may not be a concern, and therefore ceiling protocol is recommended. Inheritance does not require knowing the top priority. Hence, it is better for complex systems and it is safer if task priorities change dynamically or are being changed to fine-tune system operation. See the Mutex white paper for more discussion of this topic.

The smx mutex allows a combination of ceiling and inheritance. The ceiling can be set according to the top priority level of all users, or to a lower level, and inheritance can be enabled to take care of tasks above that level. Ceiling can be suppressed by ceil = 0; inheritance can be suppressed by pi = 0.

To delete a mutex:

```
smx_MutexDelete(&mtx);
```

getting and releasing mutexes

To get a mutex:

```
if (smx_MutexGet(mtx, tmo))
    /* critical section */
else
    /* error or timeout */
```

mtx is the mutex handle; tmo is the timeout, in ticks. If the mutex is free, the current task, smx_ct, will become its owner, and smx_MutexGet() will return TRUE. Otherwise, smx_ct will be placed in the mtx wait queue, in priority order. If inheritance is enabled, the current mtx owner will be promoted to smx_ct's priority, and promotion may be propagated to other tasks, as discussed above. If an error is encountered or a timeout occurs, smx_MutexGet() returns FALSE.

To release a mutex:

```
smx_MutexRel(mtx);
```

Returns TRUE or FALSE. smx_ct may release mtx only if it is the owner. If the nesting count is greater than one, it is simply decremented. If the nesting count is one, smx_ct releases the mutex and the top waiting task becomes the new owner; if no task is waiting, the mutex is freed. If smx_ct does not own any other mutexes, its priority is demoted to its normal priority. Otherwise, when a task releases a mutex, its priority is demoted to the highest priority required by its remaining owned mutexes (i.e. either the highest ceiling priority or the highest waiting task priority, if priority inheritance is enabled). This is called staggered priority demotion. If a non-owner or an LSR attempts to release mtx, or it is already free, an error is reported.

freeing and clearing mutexes

To free a mutex:

```
smx_MutexFree(mtx);
```

Acts like smx_MutexRel() except that any task can call it and the nesting count is cleared. smx_MutexFree() is intended for recovery situations and deleting, suspending, or stopping a task. It should not be used to release a mutex. Staggered priority demotion is implemented for the owner task.

To clear a mutex:

```
smx_MutexClear(mtx);
```

Acts like smx_MutexFree() except that the mtx wait queue is also cleared, and all waiting tasks are resumed with no error indication. Hence, each will appear to have timed out. This function is intended for recovery situations and normally should not be used. It is used by smx_MutexDelete(). Staggered priority demotion is implemented for the owner task.

impact upon other smx functions

smx_TaskCreate(): The normpri field in the TCB is initialized to the same value as the pri field, when the task is created. normpri contains the priority of the task prior to its being promoted. Tasks are demoted to this level. The mtxp (mutex pointer) field in the TCB is initially NULL. It is used to link the mutex control blocks (MUCBs) of all mutexes owned by the task.

smx_TaskDelete(task): All mutexes owned by task are freed.

smx_TaskBump(task, p): The normpri field in the TCB is changed to p. The TCB pri field is changed to p, unless the task owns a mutex. In the latter case, it will be raised to p if p > tcb.pri, but not lowered to p if p < tcb.pri. The task is moved up or down in rq or to a new position in a priority queue, as determined by its new priority. If it is waiting for a mutex, the mutex owner is promoted, if its priority is less than the new priority and priority inheritance is enabled for the mutex (mtx->pi > 0). Priority promotion may propagate to other tasks, as discussed above.

smx_MsgReceive(PXCHG), smx_MsgReceiveStop(PXCHG), and smx_MsgSendPR(PXCHG): Similar to smx_TaskBump() except that the receiving task cannot be waiting for a mutex since it is either in an exchange wait queue or it is running. Hence, priority promotion is not needed.

A task should never be stopped nor suspended when it owns a mutex. At a minimum, this may result in unbounded priority inversion for higher-priority tasks waiting at the mutex. Since a stopped task restarts from the beginning of its code, it will probably get the mutex again and never release it. This of course, is even worse than the suspend case.

In general, other resources, such as messages, should be obtained before getting a mutex, and all mutexes should be freed before suspending or stopping a task:

```
while (task->mtxp != NULL)
   smx_MutexFree(task->mtxp);
```

This is automatically done when a task is deleted.

library function example

The following example shows using a mutex to protect a non-thread-safe library call.

```
MUCB_PTR    in_clib;
TCB_PTR     lo_task, hi_task;

void  appl_init(void)         /* non-preemptible */
{
   in_clib = smx_MutexCreate(ENPI, 0, "in_clib");   /* ENPI = enable priority inheritance */
   lo_task = smx_TaskCreate(lo_task_main, PRI_LO, 0, NO_FLAGS, "lo_task");
   hi_task = smx_TaskCreate(hi_task_main, PRI_HI, 0, NO_FLAGS, "hi_task");
   smx_TaskStart(lo_task);
   smx_TaskStart(hi_task);
}

void  lo_task_main(void)
{
   smx_MutexGet(in_clib, INF);
   print_msg("This is lo_task.");
   smx_MutexRel(in_clib);
}

void  hi_task_main(void)
{
   smx_EventQueueCount(smx_TicksEQ, 2, INF);
   print_msg("This is hi_task.");
}

void  print_msg(char *msg)
{
   smx_MutexGet(in_clib, INF);
   printf(msg);
   smx_MutexRel(in_clib);
}
```

Although lo_task is started first, hi_task runs first because appl_init() is non-preemptible. hi_task immediately suspends for 2 ticks on smx_TicksEQ, thus allowing lo_task to run. lo_task gets the in_clib mutex and calls print_msg(), which gets in_clib again. If this were a semaphore, lo_task would hang, at this point. However, since it is a mutex, all that happens is that the nesting counter in in_clib is incremented. (Of course it is unnecessary to get the mutex twice, but assume that the lo_task programmer does not realize that print_msg() is protected by the in_clib mutex, because it is in a library developed by another programmer.)

Assuming the hi_task delay ends while printf() is running, hi_task preempts lo_task and calls print_msg(). Since in_clib is owned by lo_task, hi_task suspends on it. Since priority inheritance is enabled, lo_task is promoted to PRI_HI priority. Thus, no medium priority task can preempt lo_task thus causing hi_task to wait even longer (unbounded priority inversion). When printf() finishes, lo_task releases in_clib. It must do this twice before hi_task can get in_clib, preempt, and run printf().

Note: printf() is used here only as an example. Functions in the printf() family are not recommended because they use excessive stack space. Instead, use the sb_Con functions in XBASE\bcon.h.

summary

In this chapter, we have compared the advantages of using a mutex to using a binary resource semaphore. Mutex services have also been discussed, followed by a usage example.

.

Chapter 10   Event Groups

BOOLEAN	smx_EventGroupClear(eg, init_mask)
EGCB_PTR	smx_EventGroupCreate(init_mask, *name)
BOOLEAN	smx_EventGroupDelete(*eg)
u32	smx_EventGroupPeek(eg, par)
BOOLEAN	smx_EventFlagsPulse(eg, set_mask)
BOOLEAN	smx_EventFlagsSet(eg, set_mask, pre_clear_mask)
u32	smx_EventFlagsTest(eg, test_mask, post_clear_mask, timeout)
void	smx_EventFlagsTestStop(eg, test_mask, post_clear_mask, timeout)

introduction

Event groups are like gate semaphores on steroids. They allow simultaneously waiting on the AND, OR, or AND/OR of more than one event. smx event groups contain a 16-bit flags field. Each flag can be set or reset when its corresponding event occurs. Both *mode* and *event* flags are supported. A mode flag indicates a mode of operation (e.g. startup). An event flag indicates that an event has occurred (e.g. motor on).

Previously discussed smx objects (exchanges, semaphores, and event queues) do not permit a task to wait on more than one event at a time. However, this capability is necessary in many situations. For example, it might be desirable to qualify waiting on event X with mode M. This is logically represented by MX (or M && X). Alternatively, it may be desirable to start a task if either event X or event Y occurs. This is logically represented by X + Y (or X || Y). Event groups can handle this kind of logic. smx extends flag testing to AND/OR logic. For example, it may be desirable to start a task if event X occurs in mode M or event Y occurs in not mode M. This is logically represented by MX + nMY (or M&&X || !M&&Y). Note: nM means ~M.

Another use for event groups is to implement state machines. An example of a state machine is given later in this chapter.

Multiple tasks can test and wait upon different combinations of flags at the same event group. Tasks wait in FIFO order, and all tasks which match a test condition are resumed together. (Priority order is not meaningful in a situation where priority is not the resumption criterion and where multiple tasks may be resumed at the same time.)

smx event groups add the capability to pre-clear and post-clear flags, which simplifies operations and makes them atomic.

terminology

An **event** is something that happens either externally or internally. A **flag** represents an event and is stored locally inside of an event group. A **mode** is a mode of operation, such as startup, normal, or recovery. A mode remains set or cleared for a relatively long time, whereas event flags are frequently set and cleared. A **mask** represents flags to set, clear, or initialize. Flags are represented by bits in 16-bit words.

Logical Combination Bits (logic bits in short) are bit 16 and bit 17 in the event group **testing mask**. They specify the combinatorial logic that should be applied to the flag bits when testing. The following logic are allowed by the smx event group testing:

```
#define OR        SMX_EF_OR        /* 0x0000****  bit 16 and 17 both clear */
#define AND       SMX_EF_AND       /* 0x0001****  bit 16 set, bit 17 clear */
#define ANDOR     SMX_EF_ANDOR     /* 0x0002****  bit 16 clear, bit 17 set */
```

With OR logic, the testing will pass when **any** one of the bits specified in the test mask is set in the event group flag.

With AND logic, the testing will only pass when **all** of the bits specified in the test mask is set in the event group flag.

AND/OR logic triggers a more complicated testing mechanism that is explained in the **AND/OR Testing** section.

naming event flags

Choosing a good naming scheme for flags will help to make your code more readable and to avoid errors.

Hexadecimal numbers can be used to define event flags. For example:

```
smx_EventFlagsTest(eg, 0x10105, 0x0004, tmo);
```

However, this does little to help aid understanding and to avoid errors. **Abstract** names are a better way is to name flags, such as:

```
#define M     0x0100
#define X     0x0004
#define Y     0x0001
```

where M is a mode and X and Y are event flags (presumably M, X, and Y mean something). Note that these names do not indicate bit positions, thus permitting easily moving them to different bits, if necessary. Now the above test is:

```
smx_EventFlagsTest(eg, AND+M+X+Y, X, tmo);
```

which gives a clearer picture of what is happening. If you prefer more descriptive names, such as:

```
#define Mode  0x0100
#define FlagX 0x0004
#define FlagY 0x0001
```

then it may be necessary to use a "cut and bleed" format for event service calls, such as:

```
smx_EventFlagsTest(ega,      AND
                        + Mode     /* mode */
                        + FlagX    /* flag X  */
                        + FlagY,   /* flag Y */
                          FlagX,   /* auto clear flag X on match */
                          tmo);
```

This permits commenting the use of each flag, which may be helpful for complicated logic.

Abstract flag naming is theoretically nice, but it does not help during debug, when one is forced to look at flags as hexadecimal or binary numbers. For that reason, you may find it helpful to include the **bit number** in the name. For example:

```
#define M8    0x0100
#define X2    0x0004
#define Y0    0x0001
smx_EventFlagsTest(eg, AND+M8+X2+Y0, X2, tmo);
```

Then, 0x10105 shown by the debugger for the set mask is more easily decipherable. The downside is that if a flag is moved, it must be renamed. We find this notation to be helpful in the examples that follow.

Another way is to always assign flags **in order**, from bit 0 up:

```
#define M      0x0004
#define X      0x0002
#define Y      0x0001
```

and always use them in order:

```
smx_EventFlagsTest(eg, AND+M+X+Y, X, tmo);
```

Then 0x10007 for the set mask and 0x2 for the pre-clear mask are fairly easy to relate to the source code above. This may be the best approach in most cases. It is up to you to decide which of the above schemes works best for you.

creating and deleting event groups

An event group consists of an Event Group Control Block, EGCB, which contains 16 flags and heads a FIFO task wait queue. It is created as follows:

```
EGCB_PTR  eg;
eg = smx_EventGroupCreate(init_mask,  name);
```

eg is the event group handle, init_mask specifies the initial states of its flags, and name is an ASCII name, such as “eg”, which is assigned to the event group for identification when debugging and testing. Create() gets an EGCB from the QCB pool, initializes it, and loads its handle into eg.

An event group can be deleted by:

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

In this case, all tasks waiting at eg are first resumed with 0 return values (for their Test() calls), then the EGCB is returned to the QCB pool, and its handle, eg, is cleared so it cannot be used again.

testing flags

Testing flags is performed with smx_FlagsTest() or smx_FlagsTestStop(), as follows:

```
if (smx_EventFlagsTest(eg, AND+M+F1, F1, tmo))
    /* process F1 */
else
    /* recover from timeout or error */
```

will test eg for M and F1 both TRUE. If so, it will clear F1, return M+F1, and continue. (In this case, it is not desired to clear M because it is a mode.) The if() statement tests for a non-zero return and finding it, processes F1.

If the test condition is not met, ct is suspended on the eg wait queue in FIFO order. The test_mask parameter (including AND) and the post_clear_mask parameter are stored in the task’s TCB for later comparison and post-clearing during a Set() operation. If both M and F1 become set before the tmo timeout expires, the task will be resumed and M+F1 will be returned. Otherwise, Test() fails and returns 0 so that a timeout recovery will occur.

Another common situation is to test for one of multiple events to occur:

```
#define F3     0x0008
#define F5     0x0020

u32 flags;

while (flags = smx_EventFlagsTest(eg, OR+F5+F3, F5+F3, tmo))
{
    if (flags & F3)
        /* do action 3 */
        else if (flags & F5)
            /* do action 5 */
        else
            /* process timeout or error */
    }
```

In this case, either F3 or F5 TRUE causes Test() to pass and whichever flag caused the match will be cleared. If both caused the match, both will be cleared. The subsequent code deals with one or both flags being TRUE. When the action or actions are completed the task will again test F5 | F3.

Note that the post_clear_mask operates to limit which flags causing a match are automatically cleared. Hence, in the first example, the M flag was excluded from being cleared, whereas in the second example, both F3 and F5 are allowed to be cleared. If this line of code were changed to:

```
while (flags = smx_EventFlagsTest(eg, OR+F5+F3, 0, 5))
```

neither flag would be cleared and it would be necessary to do **manual resets**, such as:

```
smx_EventFlagsSet(eg, 0, F3);
```

This would normally be called after F3 processing; it obviously must occur before the next Test(). Manual reset is useful if ct wants to ignore additional F3 events until it finishes its processing. For this to be successful ct could be locked, as follows:

```
while (flags = smx_EventFlagsTest(eg, F3, 0, 5))
{
    smx_TaskLock();
    ProcessF3();
    smx_EventFlagsSet(eg, 0, F3);    /* manual reset */
}
```

This code forces ct to wait until F3 is set again. It operates, as follows: ct is locked, so it cannot be preempted. It processes F3, then clears the F3 flag in eg. Since ct is locked, F3 cannot be set by another task. When ct calls Test(), the ct lock is lifted, but Test(), itself, cannot be preempted until it has suspended ct on eg. Now, another task can set F3, resulting in the above actions being repeated.

The reason for doing something like this would be to reduce task switching due to rapidly occurring events. This assumes that some number of events can be buffered without loss (e.g. the input channel of a UART with a FIFO buffer). ct could process several events at once, then go back to waiting for more.

smx_FlagsTestStop() operates similarly to smx_FlagsTest(). For a usage example, see the state machine example near the end of this chapter.

setting and clearing flags

Flags are set and cleared with smx_EventFlagsSet(), as follows.

```
smx_EventFlagsSet(eg, set_mask, pre_clear_mask);
```

The flags specified in the pre_clear_mask are cleared, before setting the flags specified in the set_mask. This is useful for mutually exclusive flags, such as F1 and F2:

```
smx_EventFlagsSet(eg, F1, F2+F1);
```

which assures that only F1 will be set. So, if the flags started out == 0x2, they would end up == 0x1.

There is no clear flags function, so to clear all flags, use:

```
#define ALL    0xFFFF
smx_EventFlagsSet(eg, 0, ALL);
```

(Note: smx_EventGroupClear() clears all flags, but it also resumes all waiting tasks.)

Next, assuming that the set_mask is non zero, the flags specified in it are set in eg. Then, if at least one new flag has been set, the task wait queue is searched for matches, as follows: Each task's test_mask (including AND or OR) is obtained from its TCB. The test mask is compared to eg->flags and if there is a match, the task is resumed. The flags causing the match are recorded in the rv field of the TCB, which is returned when the task actually starts running (i.e. as the return value of the Test() operation that caused the task to wait).

Then the matching flags are ANDed with the post_clear_mask, also saved in the TCB. For example: if the flags causing a match are MA and the post_clear_mask is A, then their AND is A. This allows auto clearing event flags, like A, without auto clearing mode flags, like M. The result of the AND is the task's *flag clear mask*. This mask is inverted and then ANDed with the eg flags to clear them.

If there are multiple tasks waiting, the above procedure is repeated for each. Obviously this takes time and therefore, very long task wait queues are not recommended. When all tasks have been processed, their flag clear masks are ORed; then the result is inverted and ANDed with the event group's flags. Thus all flags causing matches, after filtering by corresponding clear masks, are reset. For example:

```
smx_EventFlagsTest(eg, AND+F5+F3, F3, tmo)      /* in task t2a */
smx_EventFlagsTest(eg, AND+F6+F2, F2, tmo)      /* in task t2b */
smx_EventFlagsSet(eg, F3+F2, 0);                /* in task t1a */
```

The Set() results in t2a and t2b being resumed (assuming F5 and F6 were already set) and flags F3 and F2 being cleared. F5 and F6, which were previously set, would not be cleared because they are not in the clear masks of the Test()s. This example is shown to illustrate how flag clearing works. Having tasks wait upon groups of flags, with flags in common, is probably not too useful, but it is possible.

Note that the following case is probably an error:

```
smx_EventFlagsTest(eg, F3, 0, tmo)      /* in  task t2a */
smx_EventFlagsTest(eg, F3, F3, tmo)     /* in task t2b */
smx_EventFlagsSet(eg, F3, 0);           /* in task t1a */
```

because t2a clearly does not want F3 cleared, yet t2b clears it, anyway. This is an example of how sharing flags between tasks waiting at the same event group can cause trouble. To avoid this problem, use separate event groups. See the practical usage section, below, for more discussion of good usage.

inverse flags

Sometimes one really wants to use the inverse of a flag, such as in ~MB. This can be handled with an inverse flag. For example:

```
#define M     0x0008
#define nM    0x0004
#define Y     0x0002
#define X     0x0001
```

where nM is the inverse of M. Now:

```
smx_EventFlagsTest(eg, AND+M+X, X, tmo);
smx_EventFlagsTest(eg, AND+nM+Y, Y, tmo);
```

The first test passes for MX and the second test passes for ~MY. For reliability, M and nM must never be TRUE at the same time outside of critical sections. Hence, it is necessary to reset one before setting the other:

```
smx_EventFlagsSet(ega, M, nM);     /* reset nM then set M */
```

to change mode from nM to M. When the event groups are created:

```
ega = smx_EventGroupCreate(nM, "eg");
```

assures starting correctly in ~M.

AND/OR testing

It is also possible to test AND/OR combinations of flags, for example:

```
#define M       0x8
#define E       0x4
#define nM      0x2
#define F       0x1
#define ME      (M+E)
#define nMF     (nM+F)

void eeg4(void)
{
    eg = smx_EventGroupCreate(nM, "eg");
    t2a = smx_TaskCreate(eeg4_t2a_main, P2, ESS, NO_FLAGS, "t2a");
    smx_TaskStart(t2a);

     /* in mode nM: result == F */
    smx_EventFlagsSet(eg, E+F, 0);

     /* change to mode M: result == E */
    smx_EventFlagsSet(eg, M, nM);

    /* cleanup */
    smx_TaskDelete(&t2a);
    smx_EventGroupDelete(&eg);
}

/* wait for ME + nMF */
void eeg4_t2a_main(void)
{
    u32 flags;

    while (flags = smx_EventFlagsTest(eg, ANDOR+ME*2+nMF, E+F, 3))
     {
        switch (flags)
        {
            case ME:  /* E cleared */
                result = E;
                 break;
             case nMF: /* F cleared */
                 result = F;
         }
    }
}
```

In this example, ANDOR (bit 17) is the and/or flag, which overrides the AND (bit 16) flag. The AND/OR test requires that flags in each AND term be adjacent. As shown above, the ME term is bits 2 and 1, nMF term is bits 1 and 0. For good performance, the terms should be as close to the least significant end, as possible. In the example above, M is a mode and nM is not mode M. When eg is created, nM is set. (We are assuming M is the normal run mode and nM is the startup mode.) t2a tests for ME + nMF — i.e. if in mode M it waits for event E and if not in mode M it waits for event F. Setting E + F, while in nM, causes the t2a test to pass and return nMF. Note that E remains set, because it did not cause the pass. Thus, when the mode switch from nM to M occurs, the t2a test again passes and returns ME.

Any AND/OR combinations may be tested, up to the limit of 16 bits. For example:

```
ABC + C + EF + GHI + J + K == 0b1110 1011 0111 0101
```

requires 16 bits (11 flag bits + 5 zero bits, corresponding to +'s). Pre-clear and post-clear flags work the same as for other comparisons. Theoretically one task could wait on an AND/OR comparison while other tasks waited on AND or OR comparisons of the same flags at the same event group. However, more likely to be useful would be:

```
AM1 + BM2 + CM3
```

In this case, a task would be waiting on different flags in each of the 3 different modes. As previously discussed to make the modes mutually exclusive, use the init_mask and pre_clear_mask as follows:

```
eg = smx_EventGroupCreate(M1, "eg");      /* start in M1 */
smx_EventFlagsSet(eg, M2, M1+M3);         /* switch to M2 */
```

Another potential use example is as follows:

```
AM1 + BM1 + CM2
```

which allows a task to wait on A + B in mode M1 or on C in mode M2. Note that the form

```
(A + B)M1
```

is not supported.

Creating an AND/OR mask is fairly simple. For example, for flags ABCD in bits 3 - 0, to test for A +CD, create preliminary mask: b1011. Then insert a spacer 0 between terms, so the final mask = b10011 = 0x13; then add the ANDOR flag to get 0x20013. An alternative way similar to the large example above is = ANDOR+A*2+CD.

other event group services

smx_EventGroupClear(eg, init_mask) resumes all waiting tasks with FALSE, and sets eg->flags = init_mask. It is useful in recovery situations.

smx_EventGroupPeek(eg, arg) returns the value of the specified argument. Valid arguments are:

SMX_PK_FLAGS	flags
SMX_PK_TASK	number of tasks waiting
SMX_PK_FIRST	handle of first task waiting
SMX_PK_NAME	name of event group

Note that the first task may not be the highest priority task because tasks are enqueued in FIFO order.

smx_EventFlagsPulse(eg, set_mask) is like smx_EventFlagsSet() except that it does not leave the flags in set_mask set, unless they were already set. It is useful in situations where it is desired to resume only tasks that are already waiting for specified flags — i.e. to pulse eg. This might be useful to resume a task if it is idle, but otherwise leave it alone. (This is similar to waking up a watchman, but leaving him alone if he is already on his round.) Note that this service does not have a pre-clear flag.

smx_EventFlagsTestStop(eg, test_mask, post_clear_mask, timeout) is the stop variant of smx_EventFlagsTest() and is intended for use by one-shot tasks. It operates the same as smx_EventTest(), except that it stops and restarts the current task, rather than suspending and resuming it. In the above discussion, if TestStop() has been used, replace "resume" with "restart", wherever it occurs. Tasks that are to be restarted can wait at the same event group with tasks that are to be resumed. This is a property of the task, not the event group. See the One-Shot Tasks chapter for discussion of how to use one-shot tasks. See the state machine example, below, for how to use one-shot tasks with event groups.

a few words concerning practical usage

smx event groups have been designed to meet a wide variety of requirements in many diverse applications.

Since multiple tasks can wait on different combinations of flags, some of which may be in common, coupled with pre-clearing and post-clearing, things can get too complicated in a hurry. This is not the way that smx event groups are intended to be used — it is not desirable for reliability of the system and for the sanity of the programmer.

Event groups are small and take very little space. Hence, it is practical to use more event groups in order to keep things simple. This has the downside that more event group operations will be required. However, that may be a small price to pay.

In a case where pre-clearing or post-clearing of flags is being used, it is generally better for only one task to wait at the particular event group. Generally, multiple tasks waiting at an event group should be reserved for gating operations, where it is desired for all waiting tasks to resume simultaneously when the a condition is met. When this happens, the tasks will be put into their rq priority levels in the order that they waited at the event group. Hence, neither priority nor wait order is lost.

Tasks waiting at the same event group for disjoint groups of flags is probably not useful and is best avoided by using separate event groups. Tasks waiting at the same event group for overlapping sets of flags may be useful, but post-clearing of the flags may cause trouble. Post-clearing should be used by all or none of the tasks. If tasks run in a fixed sequence, it may be better to use manual clearing of flags.

Generally speaking, keep event group usage simple.

maintaining atomic operation

When an event group is split into multiple event groups, in order to simplify operation, there is potential loss of atomicity, such as:

```
void t2aMain(0)
{
    while (smx_EventFlagsTest(ega, F1, F1, tmo))
    {
        OperationA();
    }
}

void t3aMain(0)
{
    while (smx_EventFlagsTest(egb, F1, F1, tmo))
    {
        OperationB();
    }
}

void t1aMain(0)
{
      smx_EventFlagsSet(ega, F1, 0);
      smx_EventFlagsSet(egb, F1, 0);
      ...
}
```

A potential problem in the above code is that the Set() operations are not atomic. Hence, t2a will resume immediately after the first Set() (because it has higher priority than t1a) and perform OperationA(). Then, t1a will resume and cause t3a to resume, which will perform OperationB(). Since t3a has higher priority, you might expect OperationB to occur before OperationA, but it will not. To make that happen, lock the Set() operations:

```
void t1aMain(0)
{
        smx_TaskLock();
        smx_EventFlagsSet(ega, F1, 0);
        smx_EventFlagsSet(egb, F1, 0);
        smx_TaskUnlock();
}
```

This makes them atomic. Now t3a will run ahead of t2a, as expected. (Of course, we could have set egb ahead of ega, but this is intended to be an example of good usage, not of getting lucky!)

pros and cons of event groups

Event groups are useful when there is a bonafide need for a task to wait on multiple events simultaneously. On the downside, events can be missed if their corresponding flags are not tested before being set a second time. Hence, for reliable operation, events should be interlocked with processing so they do not occur randomly. For example, in many communications systems the sender waits for acknowledgement before sending the next message. Hence events (acknowledgements) are interlocked with processes (sending messages). If interlocked operation is not possible, then event semaphores may be a better choice, because they do not lose events.

An exception to the above is if lost events are not important. For example, if each event signals the receipt of a byte that is put into a buffer, then if service task removes all bytes from the buffer whenever it runs, a few lost events are not important.

state machine example

A state machine is an ideal example for an event group and one-shot tasks. Only one state should be true at a time in a state machine. This fits one-shot tasks sharing one stack. In the example, below, the states are represented by A = t2a, B = t2b, C = t2c, and the state transitions are A->B->A->C->A, etc. The state matching stops after 100 ticks (1 sec) and the counters show the number of times each task ran. For a 400 MHz AT91SAM9G20: actr = 58,074 and bctr = cctr = 29,037. Hence, a total of 116,148 state transitions were achieved per second. Of course, a real state machine would do much more useful work, than incrementing counters. However, it is clear that the overhead for this approach is low and that it is a practical way to implement a state machine or something equivalent.

Note that each one-shot task is started with par = 0; this causes it to initially wait on eg for its flag. Then setting flag F1 starts t2a and the state machine begins running. Each state does very little work; in this example: t2a increments actr, and depending upon whether actr is odd or even, sets F2 or F3, which causes either t2b or t2c, respectively, to run next. t2b and t2c increment their counters and set F1 to cause t2a to run again. The state machine is stopped when tmr times out and invokes LSR, which stops t2a, and thus stops the machine.

```
static void eeg8_t2a(u32 par);
static void eeg8_t2b(u32 par);
static void eeg8_t2c(u32 par);
static void eeg8_LSR(u32 par);
static u32 actr;
static u32 bctr;
static u32 cctr;
static TMCB_PTR tmr;

void eeg8(void)
{
    actr = bctr = cctr = 0;

    /* create objects */
    eg = smx_EventGroupCreate(0, "eg");
    t2a = smx_TaskCreate((FUN_PTR)eeg8_t2a, P2, 0, NO_FLAGS, "t2a");
    t2b = smx_TaskCreate((FUN_PTR)eeg8_t2b, P2, 0, NO_FLAGS, "t2b");
    t2c = smx_TaskCreate((FUN_PTR)eeg8_t2c, P2, 0, NO_FLAGS, "t2c");

    /* get tasks waiting and start the timer */
    smx_TaskStartPar(t2a, 0);
    smx_TaskStartPar(t2b, 0);
    smx_TaskStartPar(t2c, 0);
    smx_TimerStart(&tmr, 100, 0, eeg8_LSR, 0, "LSR");

    /* start the state machine running*/
    smx_EventFlagsSet(eg, F1, 0);

    /* machine done, cleanup */
    smx_TaskDelete(&t2a);
    smx_TaskDelete(&t2b);
    smx_TaskDelete(&t2c);
    smx_EventGroupDelete(&eg);
}

/* state A */
void eeg8_t2a(u32 par)
{
    if (par == F1)
    {
        actr++;
        if (actr & 1)
            smx_EventFlagsSet(eg, F2, 0);
        else
            smx_EventFlagsSet(eg, F3, 0);
    }
    smx_EventFlagsTestStop(eg, OR+F1, F1, INF);
}

/* state B */
void eeg8_t2b(u32 par)
{
    if (par == F2)
    {
        bctr++;
        smx_EventFlagsSet(eg, F1, 0);
    }
    smx_EventFlagsTestStop(eg, OR+F2, F2, INF);
}
```

```
/* state C */
void eeg8_t2c(u32 par)
{
    if (par == F3)
    {
        cctr++;
        smx_EventFlagsSet(eg, F1, 0);
    }
    smx_EventFlagsTestStop(eg, OR+F3, F3, INF);
}

/* stop the machine */
void eeg8_LSR(u32 par)
{
    smx_TaskStop(t2a, INF);
}
```

This example is available in the ESMX directory so you can step through it, if you wish. The best way to follow the action is to put breakpoints at the first Create(), at the if() statements in the tasks, at the LSR, and at the first Delete().

summary

Event groups contain 16 flags, which can be set or reset when corresponding events occur. A task can test any AND, OR, or AND/OR combination of the flags. Hence it can wait upon the occurrence of events, multiple events, or qualified events. If its test does not match the event group flags, the task will wait for up to the specified timeout for a match. When a match occurs, the Test() returns the flag(s) that caused the match so the task can perform the appropriate operations for the flag(s). The Test() can specify what flags causing a match are to be cleared. Pre-clearing and post-clearing of flags simplify operations and make them atomic.

Multiple tasks may test and wait upon different combinations of flags at the same event group. Tasks wait in FIFO order and all tasks whose test masks match the event group flags are resumed together. This is similar to gate semaphore operation.

exercises

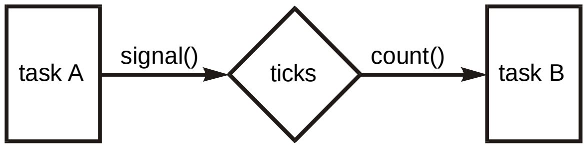
1. Try working with modes and events.
2. Develop a state machine that does something useful.

Chapter 11   Event Queues

```
BOOLEAN     smx_EventQueueClear(EQCB_PTR eq)
BOOLEAN     smx_EventQueueCount(EQCB_PTR eq, u32 count, u32 timeout)
void        smx_EventQueueCountStop(EQCB_PTR eq, u32 count, u32 timeout)
EQCB_PTR    smx_EventQueueCreate(const char *name)
BOOLEAN     smx_EventQueueDelete(EQCB_PTR *eq)
BOOLEAN     smx_EventQueueSignal(EQCB_PTR eq)
```

introduction

Event queues allow tasks to wait for precise numbers of events. Events can be signaled by tasks or LSRs:



A typical use of an event queue might be to count cans passing a photocell on a conveyer belt. Each passing can would trigger a count. TaskQC might wait for every 100th object and shuttle it off to the quality test station. TaskBox might switch to a different boxing machine after every 12th can.

Event queues are intended to make it easier to design systems which respond to specified counts of external events. A single task might wait at an event queue with different counts at different times. For example, after boxing 1200 cans in 12-can boxes, it might switch to the 24-can boxers and start counting off 24 cans. The previous example describes two tasks waiting at the same event queue.

The event semaphore allows one task to keep precise track of events. The event queue allows multiple tasks to keep track of events. A principal use of an event queue is to keep track of ticks (i.e. for delays). However there are many other kinds of recurring events that an embedded system may need to track, such as: rotations, passing objects on a conveyer belt, waves, vibrations, and other physical phenomena. An important difference between an event queue and an event semaphore is that an event queue will lose events if a task is not waiting, whereas an event semaphore will not lose events (unless its internal counter overflows). If losing events is not acceptable, then it may be necessary to use multiple event semaphores and to signal each one per event.

creation, deletion, and clearing

An event queue consists of a queue of tasks which are in order by event count. An event queue is created by:

```
EQCB_PTR  eq;

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

To delete an event queue:

```
smx_EventQueueDelete(&eq);
```

This results in resuming all waiting tasks with FALSE, releasing the EQCB back to its pool, and clearing the eq handle. Any attempt to use eq, after this, will result in an SMXE_INV_EQCB error.

Alternatively, an event queue can just be cleared:

```
smx_EventQueueClear(ticks);
```

In this case, all waiting tasks are resumed with FALSE, but the event queue is not deleted and can continue to be used. This might be useful if starting monitoring of a certain event over.

counting and signaling

Tasks are enqueued at an event queue via smx_EventQueueCount():

```
ECB_PTR  events;

void  atask_main(void)
{
    while (smx_EventQueueCount (events, 10, SMX_TMO_INF))
    {
        fun();
    }
}
```

To work reliably, fun() should require much less time than the time between events, else an event may be missed, which could result in an equipment malfunction. A way to overcome this would be for the eventISR() to maintain an event count and for atask to adjust the count parameter in each call to smx_EventQueueCount() accordingly.

Event queue count stop is used as follows:

```
ECB_PTR  events;

void  atask_main(void)
{
    fun();
    smx_EventQueueCountStop (events, 10, SMX_TMO_INF);
}
```

atask is a one-shot task. This might be a more reliable implementation because atask would probably be run locked so it could not be preempted until fun() was done and atask was back in the events queue. See the One-Shot Tasks chapter for discussion of one-shot tasks.

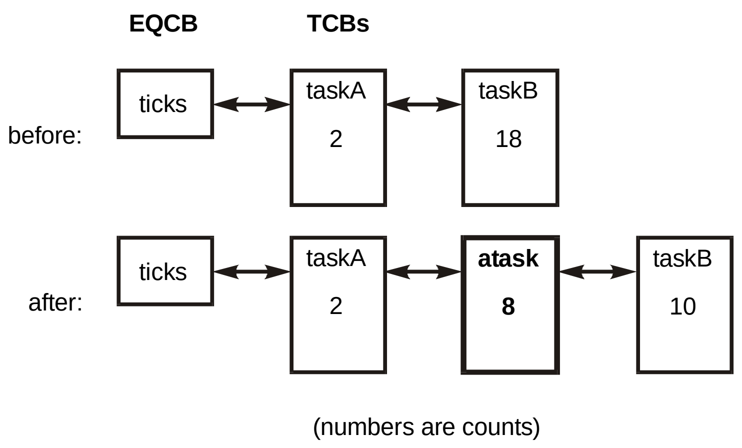
Signaling an event queue is done similarly to signaling a semaphore:

```
smx_EventQueueSignal(ticks);
```

enqueueing a task

```
smx_EventQueueCount (events, 10, SMX_TMO_INF);
```

enqueues the current task at the events queue with a count of 10. Suppose that two other tasks are already in the events queue with counts of 2 and 20. atask would be put between them and a differential count of 8 would be loaded into atask->sv. Also, the differential count in the next task's sv would be changed from 18 to 10:



Each subsequent smx_EventQueueSignal() will decrement the count of the first task. When it reaches 0, that task will be resumed and atask will become first.

The time required to enqueue a task in an event queue with smx_EventQueueCount() is variable. It can be long, if the task is placed near the end of a long queue. Signaling an event queue is usually fast, but can be variable. It can be slow if several tasks become ready, simultaneously. This is possible if several tasks, following the first task in the queue, have 0 diff counts, meaning that they must be resumed with the first task. In such a case, the tasks are resumed FIFO and go into rq in FIFO order. (Of course, higher priority tasks will go to higher levels and run sooner.)

accurate timing

smx_EventQueueCount() permits timeouts which are accurate to a tick. To use it for this purpose, use the event queue called smx_TicksEQ, which is created by smx_Go(). It is signaled every tick from smx_KeepTimeLSR in xtime.c. Thus, to delay the current task, for 10 ticks:

```
smx_EventQueueCount (smx_TicksEQ, 10, INF);
```

For convenience, a macro using the above, allows specifying delays in milliseconds:

```
smx_DelayMsec(N);
```

This waits at least N milliseconds, rounded up to the nearest tick. Accuracy is one tick.

Tick rates of 1000 Hz are normally practical for typical embedded processors. Hence, accuracies of 1 millisecond are possible. Faster processors can, of course, support faster tick rates and thus achieve greater timing accuracies. Note, however, that accuracy applies only to putting the task into the ready queue. How long it waits after that, will depend upon its relative priority, upon whether the current task is locked, and upon the amount of foreground activity occurring.

Once a task is enqueued, the overhead to decrement the counter of the first task is minimal. However, the smx_EventQueueCount() call can be slow to enqueue tasks in very large event queues. smx_TicksEQ() is good for systems where only a few tasks require accurate timeouts. Then enqueueing times will not be significant.

Using tick-accurate timeouts is another way to achieve accurate delays. e.g.:

```
smx_Suspend(atask, 10);
```

will delay atask for 10 ticks. However, if there is very a large number of tasks, the overhead of tick-accurate timeouts may be too great. See Timing Chapter, timeouts.

summary

There are four important things to be aware of concerning event queues:

(1) The count, specified in smx_EventQueueCount(), applies to events (i.e. signals received) after the task is enqueued. Unlike tasks using a semaphore, tasks using an event queue may miss signals when not in the queue.

(2) The number of counts for which a task waits is independent of other tasks in the event queue.

(3) smx_EventQueueCount() may be slow for long event queues.

(4) smx_EventQueueSignal() can be slow, if many tasks count out together.

Chapter 12 Pipes

```
BOOLEAN     smx_PipeClear(PICB_PTR pipe)
PICB_PTR    smx_PipeCreate(void *ppb, u8 width, u16 length, const char *name)
VOID_PTR    smx_PipeDelete(PICB_PTR *pipe)
BOOLEAN     smx_PipeGet8(PICB_PTR pipe, u8 *b)
u32         smx_PipeGet8M(PICB_PTR pipe, u8 *bp, u32 lim)
BOOLEAN     smx_PipeGet(PICB_PTR pipe, void *pdst)
void        smx_PipeGetPkt(PICB_PTR pipe, u8 *pdst);
BOOLEAN     smx_PipeGetWait(PICB_PTR pipe, void *pdst, u32 tmo)
void        smx_PipeGetWaitStop(PICB_PTR pipe, void *pdst, u32 tmo)
BOOLEAN     smx_PipePut8(PICB_PTR pipe, u8 b)
u32         smx_PipePut8M(PICB_PTR pipe, u8 *bp, u32 lim)
BOOLEAN     smx_PipePut(PICB_PTR pipe, void *psrc)
void        smx_PipePutPkt(PICB_PTR pipe, u8 *psrc);
BOOLEAN     smx_PipePutWait(PICB_PTR pipe, void *psrc, u32 tmo)
void        smx_PipePutWaitStop(PICB_PTR pipe, void *psrc, u32 tmo)
BOOLEAN     smx_PipeResume(PICB_PTR pipe)
u32         smx_PipeStatus(PICB_PTR pipe, PSS *ppss)
```

introduction

Pipes are intended primarily for low-speed, asynchronous, serial i/o such as input from keypads and strip readers or output to character printers and scrolling displays. For these kinds of devices, pipes are easier to use than block i/o. Pipes are also useful for self-regulated data transfers between tasks. Hence, pipes serve two purposes:

(1) I/O

(2) Asynchronous inter-task communication

I/O is characterized by one end of the pipe being served by an ISR and the other end by a task. Since ISRs cannot call SSRs, smx has special non-SSR put and get functions for use from ISRs. These functions cannot wake up tasks waiting at the other end of pipes, because they are not SSRs. However, smx provides a variety of methods to do that, which are discussed, in more detail, below.

Intertask communication is characterized by tasks servicing both ends of a pipe. In this case each task can wait on the pipe for a put or a get operation by the task at the other end of the pipe. Hence, pipes serve a synchronization function between tasks. smx permits multiple tasks to wait on one end of a pipe. However tasks can be waiting at only one end of a pipe at a time — all waiting tasks are waiting to do either a put or a get. Task operations are discussed in more detail later in this chapter.

Pipes are used to transfer packets of information. The packet size corresponds to the width of the pipe.

structure

Pipes are composed of cells, which hold packets. The cell size equals the pipe width, which can be 1 to 255 bytes. Packet size cannot exceed cell size. The pipe length is the number of cells that it contains. The maximum number of packets that a pipe can hold is one less than its length, because one cell must be sacrificed to distinguish between a full pipe and an empty pipe. Pipe width and length are determined when the pipe is created and cannot be changed during operation.

operation

Writing a byte or a packet into a pipe is called a put. Getting a byte or packet from a pipe is called a get. A pipe has an input end and an output end. Puts put bytes into the input end; gets get bytes from the output end.

As packets are put into a pipe, the pipe's internal write pointer is advanced until it is one cell behind the read pointer. At that point the pipe is full and a further put will result in failure or a task waiting on the pipe, depending upon the put service. As packets are removed from a pipe, the pipe's internal read pointer is advanced until it is equal to the write pointer. At that point, the pipe is empty and a further get will result in failure or a task waiting on the pipe, depending upon the get service.

smx provides both wait and non-wait puts and gets. The former are to be used from tasks and will cause tasks to wait on pipes, when a put or get cannot be completed. They also can be used from LSRs, but SMX_TMO_NOWAIT should be specified, else an error will occur. Non-wait puts and gets are intended for use from ISRs. If the put or get operation cannot be completed, the operation is aborted and returns FALSE. ISRs must be written to handle this condition.

API services

The smx_PipeGetPkt(), Get8(), Get8M(), PutPkt(), Put8(), and Put8M functions are ordinary functions that can be used from ISRs. smx provides no protection for them and they do not wake up a task waiting on the pipe. Two types of put and get functions are provided. The Get and Put packet functions are for general usage and support packet sizes from 1 to 255 bytes. The Get8, Get8M, Put8, and Put8M functions are stripped-down, high-speed functions intended to deal with byte streams.

smx_PipeGet() and smx_PipePut() are ordinary functions intended for use from LSRs and tasks. All other services are SSRs intended for use from tasks or LSRs. A wait function causes a task to wait if its put or get cannot be completed; a stop function causes a task to stop and wait, or to restart. Create and Delete serve obvious purposes. Clear clears all packets from a pipe and resumes all tasks waiting on the pipe. Resume is used to resume any task waiting on a pipe. (Note: timeout, resume, and start will also resume a task waiting on a pipe and will auto complete, if possible — see below.) Status returns the number of packets in the pipe and other status information, if desired.

waking waiting tasks

Note: if a task has been stopped on a pipe, resuming it has the same effect as restarting it. Hence, here and in other discussions, the term resume should be interpreted as restart, if the task was stopped.

For task to task communication, when a task gets a packet from a pipe, if another task is waiting to put a packet into the pipe because it is full, its packet will be put into the pipe and the waiting task will be resumed; similarly, when a task puts a packet into a pipe, if another task is waiting to get a packet from the pipe because it is empty, it will be given the new packet and the waiting task will be resumed.

When one end of the pipe is being serviced by ISRs, the put and get functions cannot wake up a task waiting at the other end of the pipe. As noted above, there are three ways to deal with this:

(1) The ISR can invoke an LSR at appropriate times (e.g. x bytes or packets received or a full message received or sent). The LSR, in turn, can do a PipeResume(), which resumes a task waiting on the pipe.

(2) A timeout value can be selected, such that the pipe will not overflow. The task will automatically wake up and complete its get or put operation, if possible. If the get or put is completed, it will return with TRUE; if not it will return with FALSE.

(3) A software timer can be set such that the pipe will not overflow, and the LSR that it invokes can do a PipeResume(), as in #1.

A pipe overflow would occur if an ISR is putting packets into a pipe and the pipe became full; it could result in lost data. Pipe underflow would occur if an ISR is getting packets from a pipe and the pipe became empty; this might impact performance, but would not result in lost data. Normally, the pipe should be long enough and data rates and task rates matched well enough so that these problems do not occur. In any event, higher-level protocols should handle any problems.

multiple waiting tasks

Multiple tasks can wait on a pipe. This is not likely to be used much, but it is included for completeness. Normally, a pipe would be an information conduit between two specific tasks or between and ISR and a task. However, it is possible that multiple tasks might process an incoming data stream or multiple tasks might contribute to an outgoing data stream (e.g. sending error or status messages to a console). Also, a pipe could be used to control access to multiple resources; each packet being a token that permits access to a specific resource.

Note: multiple task waits are better handled using messages and exchanges because priorities can be used.

auto-completion of waiting puts and gets

As noted above, if a task is waiting on a pipe and its complementary function occurs, the put or get operation of the task will be completed, if possible. This also is true if the task times out, is resumed, or is restarted. Auto-completion becomes more complicated if more than one task is waiting at a pipe. What happens then, is that auto-completion occurs for each task in the order it is enqueued until no more auto-completions are possible or no more tasks are waiting. This preserves FIFO resumption order among waiting tasks. If the task that timed out or was resumed (i.e. the target task) did not auto complete its put or get, it will still resume, but with a FALSE return. Hence, for example, a task that times out may not get a packet, even though there was a packet in the pipe, because a preceding task got it. A waiting task, other than the target task, will continue waiting if auto completion is not possible for it.

packet size

The unit of data transfer is called a packet. All transfers in and out are on a packet basis. As noted above, packet size is determined by cell size, which is determined by pipe width. Pipes may be of any width from 1 to 255 bytes. The width is determined when the pipe is created. Input and output buffers must be full cell size – i.e. a full cell’s worth of data will be read into or written out from the pipe. A pipe is considered empty if it contains less than a full packet. This can occur if a put packet operation is interrupted. If the invoked ISR attempts to get a packet, the get will fail.

Pipes having a width of 1 are called byte pipes. Byte pipes are useful for byte streams, especially for I/O. smx provides very efficient byte functions, Get8(), Get8M(), Put8(), and Put8M() for use by ISRs.

Greater width pipes provide a means to pass short messages efficiently. However, the messages must be copied into and out of pipes. This can become inefficient for longer messages. smx provides packet functions for use on wider pipes by ISRs and it provides packet SSRs for use by tasks. At some size, it is more efficient to put a packet into a block and send a pointer to the block via a pipe. Of course, blocks have a certain amount of overhead to get and release them and they introduce more complexity. It is up to the user to decide where to cross over from pipe packets to pointers to blocks.

queues vs. pipes

Word pipes, used in the above manner, are generally referred to as “message queues” in the Industry, and blocks are referred to as “messages”. We prefer to use the term “queue” to mean a doubly-linked list of objects.

When used in this manner, pipes pass pointers to either base blocks or smx blocks. Since there is no message control block, the receiving task must know the size of the block, where to return it, and who sent it, if a reply is needed. For these reasons, smx messages are preferable and safer. However, passing pointers via pipes may be desirable in scarce memory systems or for emulating other kernels. See the Exchange Messaging chapter for more information.

pipe buffer

The PipeCreate() SSR could automatically get a pipe buffer from the heap. However, this is not a good idea because it could lead to system failure if pipes are being created when needed and deleted, when not, in order to free memory. Even if this is being done infrequently, heap fragmentation might eventually cause a pipe create to fail. That would mean that a system function could not be performed and the system, or at least part of it, would fail. Hence, we decided that it is best for the user to allocate or statically define a block for the pipe buffer, then pass its pointer to PipeCreate(). Of course, the user must allocate a block that is big enough. For that reason, it is recommended to use (pipe_width * pipe_length) for the allocation, which will automatically adjust to changes in the pipe width or length.

The user can allocate the pipe buffer from:

- Static memory — e.g. u8 pb[pipe_width * pipe_length];
- Heap
- DAR
- Block pool

If a pipe is deleted, the user is responsible to deal with the pipe block. Block pools are best for dynamic pipes. It is up to the user to make sure that the block is big enough for the pipe and to release the block back to its pool after deleting the pipe.

For best performance, pipe buffers should be located in on-chip SRAM or cache-aligned, if in external RAM. For pipe buffers in external RAM, best performance will be achieved if cells are cache-aligned. For example if a packet is 7 bytes, 8-byte cells will work well for cache line lengths of 8, 16, or 32 bytes. If a packet is 12 bytes, chose a cell size of 16 bytes. Performance will more than make up for inefficient memory usage, even for long pipes.

Note: be sure that input and output buffers are cell-size, not packet-size.

restrictions on usage

(1) Bare functions are not protected from preemption and thus should not be used in tasks unless preemption is not possible. Bare put and get functions could be used on a pipe between two tasks of equal priority in order to achieve faster operation. However, this loses the synchronizing action of pipe SSRs and is not recommended.

(2) Bare functions must be protected from other ISRs accessing the same pipe. (Note: this is not recommended.) This is normally done by disabling interrupts. Bare functions do not disable interrupts, themselves.

(3) It is not permitted to mix bare functions and SSRs on the same end of a pipe (e.g. having an ISR and a task putting packets into the same pipe).

(4) SSRs may not be used in ISRs.

In general, pipes should be used for task to task communication with SSRs or ISR to task communication, with the ISR using a bare function and the task using an SSR. LSRs can be used in place of either tasks or ISRs and can use either SSRs or bare functions. However, such usage is rarely needed. Normally LSRs will be used to resume tasks waiting on pipes.

An example of using an LSR, in place of a task, is its use to process a high-speed data stream (e.g. audio or video) in order to avoid possible task blocking. Since an LSR cannot wait on a pipe, it would need to be invoked from the ISR or from an smx cyclic timer. Generally speaking, smx pipes are designed to be rugged and can be used in unusual ways, such as this example, as long as their limitations are understood.

safe operation

A pipe SSR, called from a task, may be safely interrupted by a complementary pipe function called by an ISR or LSR. A pipe SSR, called from an LSR, may be safely interrupted by a complementary pipe function called by an ISR. These statements are not true for non-complementary pipe operations, which are prohibited.

example

This example shows the difference between pipe transfers from an ISR to a task and from task to task.

```
#define  ETX   3
#define IP_LEN         200
#define IP_WIDTH     1
#define TP_LEN         50
#define TP_WIDTH    1
PICB_PTR      inpipe, tpipe;
TCB_PTR       xfer, pro;
SCB_PTR       read_inpipe;

void  ep2(void)
{
    void *ppb;

    /* create inpipe, tpipe, and read_inpipe semaphore */
    ppb = smx_HeapMalloc(IP_LEN);
    inpipe = smx_PipeCreate(ppb, IP_WIDTH, IP_LEN, "inpipe");
    ppb = smx_HeapMalloc(TP_LEN);
    tpipe = smx_PipeCreate(ppb, TP_WIDTH, TP_LEN, "tpipe");
    read_inpipe = smx_SemCreate(EVENT, 0, "read_inpipe");

    /* create xfer and pro tasks and start them */
    xfer = smx_TaskCreate(ep2_xfer_main, P2, ESS, NO_FLAGS, "xfer");
    pro = smx_TaskCreate(ep2_pro_main, P2, ESS, NO_FLAGS, "pro");
    smx_TaskStart(xfer);
    smx_TaskStart(pro);
}

void  rxISR(void)
{
    u8  ch;

    smx_ISR_ENTER();
```

```
          ch = input(IN_CHAN);
          if (smx_PipePut8(inpipe, ch))
          {
               if (ch == ETX)
                    smx_LSR_INVOKE(rxLSR, TRUE);
          }
          else
               smx_LSR_INVOKE(rxLSR, FALSE);
          smx_ISR_EXIT();
     }
     void  rxLSR(u32 ok)
     {
          if (ok)
               smx_SemSignal(read_inpipe);
          else
               report_overflow();
     }
     void  xfer_main(void)
     {
          u8  ch;
          while (smx_SemTest(read_inpipe, INF))
          {
               while (smx_PipeGetWait(inpipe, &ch, NO_WAIT) && (ch != ETX))
                    smx_PipePutWait(tpipe, &ch, SMX_TMO_INF);
          }
     }
     void  pro_main(void)
     {
          u8  ch;
          while (smx_PipeGetWait(tpipe, &ch, INF))
          {
               if (ch[i++] == ETX)
                    break;
               else
                    process_ch();
          }
     }
```

ep2 creates inpipe for incoming characters and tpipe for transfers between tasks. The pipe buffers are allocated from the heap using predefined constants (e.g. IP_LEN). Using such constants helps to avoid a mismatch between what is actually allocated to a pipe and what the pipe expects. Numbers can get out of sync due to code modifications. Pipe buffers should be allocated from block pools for pipes that are created and deleted often.

rxISR gets a character, ch, from IN_CHAN and puts it into inpipe, using smx_PipePut8(), which is designed to be used from ISRs. If the character received is end of text (ETX), rxLSR() is invoked with TRUE. Note that ETX is put into inpipe. If inpipe overflows, PipePut() returns FALSE, and rxLSR() is invoked with FALSE.

To prevent inpipe from overflowing, its pipe buffer should be large enough to handle peak data rates vs. available processing bandwidth. In this case, inpipe size is quite large, indicating high peak rates vs.

expected minimum processing rates. If inpipe overflows, incoming characters are lost. This example shows reporting the overflow via rxLSR calling report_overflow(). Using LSRs to report errors is a good approach, since it keeps error processing overhead out of ISRs — in this case calling report_overflow().

If ok, rxLSR signals the read_inpipe semaphore, and this wakes up xfer task, which is waiting on read_inpipe. Another way to accomplish the same result would be for xfer task to wait on inpipe and for rxLSR() to use smx_PipeResume(inpipe) to wake it up.

xfer task, once awakened, moves a character at a time from inpipe to tpipe, until the ETX character is reached. xfer task then goes back to waiting on the read_inpipe semaphore.

The xfer and pro task code illustrate task to task serial communication via pipes. Note that tpipe is short compared to inpipe. If there were no restriction on message length, tpipe could apparently overflow. But this is not the case because, if tpipe is full, xfer will simply wait. Conversely, if tpipe is empty, pro task will wait. Such task synchronization is a valuable characteristic of pipes. By contrast, rxISR cannot wait because it must handle characters as they come in. Hence, if xfer task waits too long for pro task to run, inpipe may overflow.

Assume that messages are relatively short compared to the length of inpipe and that pro task has low priority. Then several messages might build up in inpipe, and the read_pipe semaphore might receive many signals while xfer task waits on tpipe. Since the read_pipe semaphore is an event semaphore, no signals will be lost, and xfer task will run an equal number of times to transfer all of the messages waiting in inpipe.

Data transfer between the tasks could be made more efficient, if tpipe were a packet pipe, and xfer task accumulated characters into packets, then put packets into tpipe.

Chapter 13 Exchange Messaging

kinds of messaging

Messaging is the preferred method for exchanging data between tasks. There are three kinds of messaging:

(1) pipes — unprotected

(2) mailboxes — unprotected

(3) exchanges — protected

The first is the only type of messaging offered by many kernels. The term *queue* or *message queue* is commonly used by them. We think pipe is a more accurate description for this kind of object. Pipes permit bytes or packets to be exchanged between tasks. A 4-byte packet can be used to pass a pointer to a block containing a message. Pipe messaging is simple, but it offers no protection, and it has other limitations. However, it may be appropriate for simple systems with minimal resources or for porting legacy code from another kernel to smx. See the Pipes Chapter for more information on this kind of messaging.

Some kernels permit sending messages directly to tasks. Normally this is done in the form of sending pointers. Often only one message can be received at a time and the object receiving the message is usually called a *mailbox*. smx offers no equivalent for this other than dedicating an exchange to a task.

exchange messaging

Exchange messaging is the preferred messaging technique for smx. smx_MsgSend() sends a message to an exchange; smx_MsgReceive() gets a message from an exchange. An *exchange* is an object that can have either a message queue or a task queue, depending upon which is waiting for which. Its name was derived from its similarity to a telephone exchange. Messages are exchanged by passing their handles, which are actually pointers to their message control blocks (MCBs).

Exchange messaging provides many advantages over other techniques:

(1) Anonymous receiver. The receiver's name need not be hard-coded into the sender's code.

(2) Dynamic change of receiver merely by causing a different task to wait at an exchange.

(3) Receiver autonomy. The receiver controls when it receives and from which exchange.

(4) Unlimited queues. Exchanges can accept and enqueue any number of messages or tasks.

(5) An exchange can be used to control resource sharing via *token* messages. (See exchanges in the Resource Management chapter.)

(6) Multiple receivers can tap into a single message stream by waiting at the same exchange.

(7) Greater safety provided by validating MCBs upon receipt and embedding data pointers within MCBs.

(8) Broadcasting, multicasting, and distributed assembly are supported.

exchange API

```
BOOLEAN     smx_MsgXchgClear(XCB_PTR xchg)
XCB_PTR     smx_MsgXchgCreate(SMX_XCHG_MODE mode, const char *name)
BOOLEAN     smx_MsgXchgDelete(XCB_PTR *x)
u32         smx_MsgXchgPeek(XCB_PTR x, SMX_PK_PARM par)
```

Exchanges are capable of operating in one of three modes:

mode	**exchange mode**
SMX_XCHG_NORM	Normal exchange
SMX_XCHG_PASS	Pass exchange
SMX_XCHG_BCST	Broadcast exchange

The mode is determined when the exchange is created.

A **normal exchange** is used to exchange messages between tasks. Most messages received by a normal exchange will have 0 priority and will be enqueued in FIFO order. FIFO enqueueing is much faster than priority enqueueing — especially for a large number of messages. An occasional high priority message will be put at the start of the queue and will be processed first. This allows sending an urgent message, which might result in the other messages being discarded. A normal exchange is created as follows:

```
XCB_PTR  nx1;
nx1 = smx_MsgXchgCreate (NORM, “nx1”);
```

nx1 would usually be a global variable so this exchange could be accessed by other tasks. An exchange is headed by an exchange control block (XCB), which is a type of queue control block (QCB), and is allocated from the QCB pool. The size of this pool is controlled by NUM_QCBS in acfg.h; it is automatically allocated from SDAR the first time a QCB is needed.

Tasks are always enqueued in priority order. The highest priority task enqueued is the first to receive a message. Priority task wait queues ensure that there will be no unintended priority inversions.

A **pass exchange** is used like a normal exchange to exchange messages between tasks, but in addition, it passes the priority of the message to the task. Pass exchanges are most often associated with server tasks which wait at pass exchanges for work. When a message is sent to a pass exchange, its priority is assumed by the server task. Hence, the message, itself, determines the priority at which it will be processed. A pass exchange is created as follows:

```
XCB_PTR  px1;
px1 = smx_MsgXchgCreate (PASS, “px1”)
```

The messages waiting at px1 are enqueued in priority order. It is unlikely that this queue will get very long — else the server task is falling behind in its work!

Task and message priorities can be altered with smx_TaskBump() and smx_MsgBump(), respectively, even when they are waiting at an exchange. These services result in the task or message being moved immediately after other tasks or messages of the same priority in the queue.

A **broadcast exchange** accepts only one message at a time. This message “sticks” to the exchange. All waiting tasks and any subsequent receiving tasks get copies of the msg handle and message block pointer, but none get the message, exclusively. When no message is stuck to the exchange, receivers are enqueued in FIFO order and all are resumed when a message does arrive. This is similar to a gate semaphore. Broadcasting is discussed in a later section of this chapter.

message API

BOOLEAN	smx_MsgBump(MCB_PTR msg, u8 pri)
MCB_PTR	smx_MsgGet(PCB_PTR pool, u8 **bpp, u16 clrsz)
MCB_PTR	smx_MsgMake(PCB_PTR pool, u8 *bp)
u32	smx_MsgPeek(MCB_PTR msg, SMX_PK_PARM par)
MCB_PTR	smx_MsgReceive(XCB_PTR xchg, u8 **bpp, u32 timeout)
void	smx_MsgReceiveStop(XCB_PTR xchg, u8 **bpp, u32 timeout)
BOOLEAN	smx_MsgRel(MCB_PTR msg, u16 clrsz)
u32	smx_MsgRelAll(TCB_PTR task)
BOOLEAN	smx_MsgSendPR(MCB_PTR msg, XCB_PTR xchg, u8 pri, void *reply)
u8 *	smx_MsgUnmake(PCB_PTR *pool, MCB_PTR msg)

An smx message consists of a data block, called its *message body* and a message control block (MCB), which smx uses to handle the message. Message bodies and smx blocks use the same data block pools. In fact, the smx_BlockPool: Create(), Delete(), and Peek() functions are used for messages, as well as for blocks — see the Memory Chapter for information on these functions. The only difference is that messages use MCBs and smx blocks use BCBs.

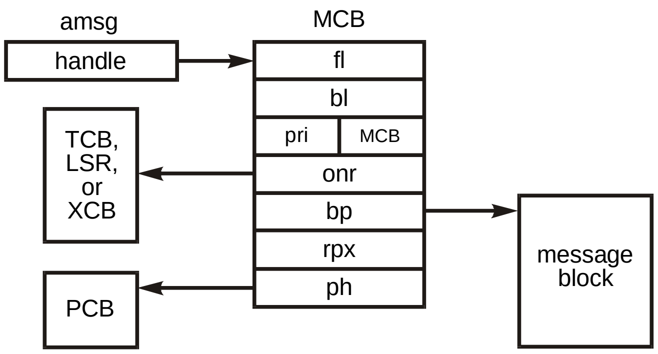
getting and releasing messages

smx_MsgGet() is used to get a message from a pool:

```
MCB_PTR msg;
u8  *mbp;
PCB_PTR  poolA;

msg = smx_MsgGet(poolA, &mbp, 4);
/* fill message using mbp */
```

A data block is obtained from poolA to use as the message body, an MCB is obtained from the MCB pool, and the MCB is linked to the data block. The MCB address is returned as the msg handle, msg. This handle will be used by all subsequent smx services to deal with the message. In addition, the first 4 bytes of the message body are cleared and the message body starting address is loaded into the message body pointer, mbp, supplied by the user. mbp is a working pointer which is used by the application to load the message with data.

The structure of an smx message is as follows:



Note that an MCB has 24 bytes — twice as large as a BCB. Hence, the overhead per message is higher, which tends to favor using smx blocks for small packets and smx messages for large messages. However, in most cases, the benefits of exchange messaging may overrule the overhead consideration — even for small packets.

smx_MsgGet() can be called from a task or an LSR. The handle of the requesting task (smx_ct) or the address of the LSR (smx_clsr) is stored in the onr field of the MCB. This indicates that the message is in use. (It is worth noting that whether in use or free, an MCB is still physically in the MCB pool. The most reliable way to distinguish a message in use from a free message is via the onr field, which is NULL for a free message.)

The message body pointer, bp, and the pool handle, ph, are also stored in the MCB. ph points to the PCB for the pool. Additional information such as NUM and SIZE is stored in the PCB. An MCB has forward and backward links, which allow it to be linked into an exchange queue. It also has a cbtype, priority, reply index, and an owner. All smx control blocks except the BCB, have cbtypes, which are checked to verify that handles are correct. Priority and reply are discussed below. The owner is the handle of the task, LSR, or exchange that currently owns the message.

smx_MsgGet() aborts, with a NULL return, if the pool is invalid, or if either the data block pool or the MCB pool is empty. In either of the latter two cases, a task cannot wait at the pool for a message. Instead, the task should wait at a resource semaphore, as shown in the receiving and sending messages section, below.

The reason that a task cannot directly wait at a pool is because PCBs do not have forward and backward links. To do so would create too much difference between base pools and smx block pools. If it is not desired to use a resource semaphore, then the other alternative is to fail and possibly retry later:

```
void t2a_main(void)
{
    u8  *mbp;
    if (msg = smx_MsgGet(poolA, &mbp, 4) != NULL)
        /* use mbp to write msg */
    else
        /* fail */
}
```

Note that in the above example, mbp is defined as a local variable for the task. This is a good practice because mbp is only needed within t2a. In many cases, it is helpful to declare mbp as a pointer to a message structure:

```
#define DSZ 10
typedef struct {
    u32     time;
    u8  data[DSZ];
} MSGA;

void t2a_main(void)
{
    MSGA*       mp;
    MCB_PTR  msg;

    msg = smx_MsgReceive(xa, (u8**)&mp, TMO);
     /* process msg here using mp */
    smx_MsgRel(msg, sizeof(MSGA));
}
```

Then the message body can be more easily accessed via mp->time and mp->data[i].

smx_MsgRel() is used to release a message, using its handle, msg:

```
smx_MsgRel(msg, NUM);
```

msg block is released to its pool, if it has one, and its MCB is released to the MCB pool. MsgRel() will fail and return FALSE if msg is invalid or is not owned by the current task or LSR. (Required ownership is a safety feature.) smx_MsgGet() and smx_MsgRel() are interrupt-safe with respect to sb_BlockGet() and sb_BlockRel(). This means that these smx services can be used from tasks or LSRs at the same time that the base functions are being used on the same pool from ISRs. So, for example, ISR1 could get a block from poolA at the same time that t2a was returning a message to poolA.

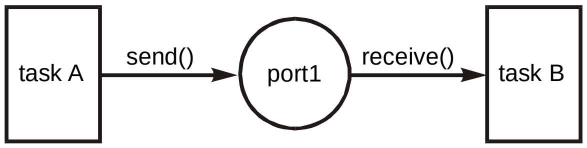
smx_MsgRelAll() releases all messages owned by a task and returns the number released. To do this it searches the MCB pool for an MCB whose owner is the task, then calls smx_MsgRel() to release that message. This process is repeated until all MCBs have been checked:

```
u32     num;
num = smx_MsgRelAll(taskA);
```

Released messages are not cleared because messages owned by a task may be of various sizes. smx_MsgRelAll() will fail if the task handle is invalid. This service is used when a task is deleted by smx_TaskDelete(). It may also be useful when a task is stopped in order to release messages not be needed until the task is restarted, and it may be useful in recovery situations. smx_MsgRelAll() is interrupt-safe.

sending and receiving messages

Sending and receiving messages between tasks and between tasks and LSRs is done via exchanges. As a consequence, neither the sending nor the receiving task knows the other's identity. This increases task independence and it allows tasks to easily be replaced with other tasks.



To send a message to an exchange:

```
PCB_PTR  msg_pool;
XCB_PTR  port1 = smx_MsgXchgCreate(NORM, "port1");

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

    msg = smx_MsgGet(msg_pool, &mbp, 0);
    /* load msg here using mbp */
    smx_MsgSend(msg, port1);
}
```

msg is sent to port1 with priority 0 and no reply is expected. It will be delivered to the top waiting task at port1 and that task will be resumed. If there is no waiting task, msg will be enqueued at port1. Since its priority is 0, it will be enqueued in FIFO order.

To receive a message from an exchange:

```
void  taskB_main(void)
{
    u8  *mbp;
    MCB_PTR  msg;
```

```
        while ((msg = smx_MsgReceive (port1, &mbp, TMO)) != NULL)
        {
            /* process msg here using mbp */
            smx_MsgRel(msg, SIZE);
        }
        /* deal with timeout or error */
    }
```

taskB waits at port1 until it receives a message or TMO ticks elapse. When a message arrives at port1 it is passed to taskB, if taskB is the first waiting task. The message's handle is assigned to msg, and the address of the message body is loaded into mbp. The latter is used to process the message. Then taskB releases the message back to its pool and goes back to port1 to get the next message. If one is waiting, taskB receives it immediately and processes it; otherwise taskB waits again for up to TMO ticks.

Note that the wait occurs inside of smx_MsgReceive(), before the assignment operation. taskB will be resumed when a message is received or TMO ticks elapse. This is transparent to the code (unless it is keeping track of elapsed time). If taskB times out, it exits the while loop and the code which follows deals with the problem. The same is true if MsgReceive() detects an error, which could happen, for example, if port1 were cleared by another task.

If INF is specified, instead of TMO, taskB would wait forever for a message. If NO_WAIT is specified, taskB would return immediately with or without a message. If there were no message at port1, msg == NULL, and the task exits the while() loop.

If port1 were a pass exchange, then the priority of taskB would depend upon the priority of each message it received. This assures that a high-priority message is processed at a high priority and that it is not delayed by a low-priority message (unless the latter's processing has already begun, since taskB would have to finish the current message before starting the next). Usually only one task waits at a priority exchange. Such a task is usually a *server task*. The above example applies to a server task as well as to a non-server task.

The following example shows the use of a resource semaphore to regulate getting and releasing messages between two tasks.

```
PCB_PTR  msg_pool;
SCB_PTR  sr;
TCB_PTR  t2a, t2b;
XCB_PTR  xa;

msg_pool = smx_BlockPoolCreateDAR(sb_adar, NUM, SIZE, ALIGN, "msg_pool");
sr = smx_SemCreate(RSRC, NUM, "sr");
xa = smx_MsgXchgCreate(NORM, "xa");

void t2a_main(void)
{
    u8 *  mbp;
    MCB_PTR   msg;

    while (smx_SemTest(sr, INF))
    {
        msg = smx_MsgGet(msg_pool, &mbp, 4);
        /* fill msg here using mbp */
        smx_MsgSend(msg, xa);
    }
}
```

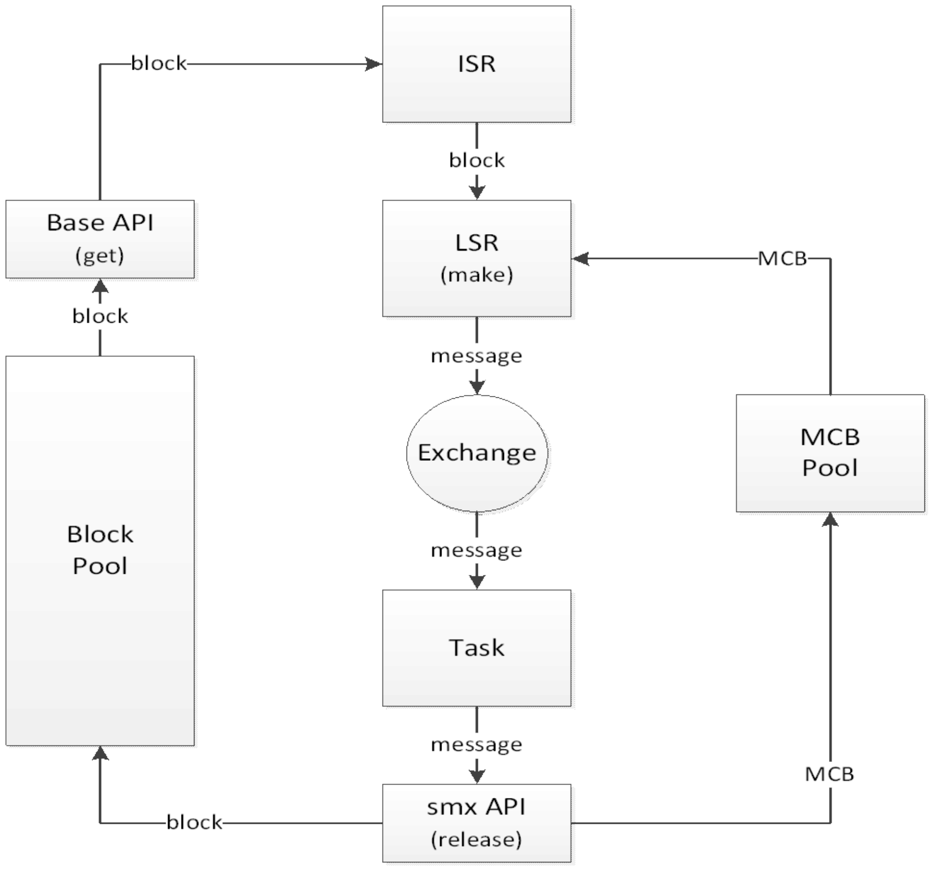
```
void t2b_main(void)
{
    u8 *  mbp;
    MCB_PTR  msg;

    while ((msg = smx_MsgReceive(xa, &mbp, INF)) != NULL)
    {
        /* read msg here using mbp */
        smx_MsgRel(msg, SIZE);
        smx_SemSignal(sr);
    }
}
```

In the above example, the resource semaphore, sr, is initialized to a count equal to the number of messages, NUM, in msg_pool. When a task, such as t2a, needs a message, it tests sr. If a message is available, sr's internal count will be greater than 0, so the test will pass, and the internal count will be decremented. If no message is available, t2a will be suspended and wait at sr for one to become available. When t2b gets a message, it fills it and sends it to xa, where t2b waits. t2b receives and processes the message. When t2b is done with the message, it releases it back to msg_pool and signals sr. This allows the top waiting task at sr to get a message from msg_pool.

In this particular example, t2a will exhaust the message pool before allowing t2b to run, since they have the same priority. When t2b starts running, there will be NUM messages waiting for it at xa. It will read and release each message and signal sr. However t2a cannot run until t2b has read and released every message it sent, even though t2a passed sr on the first message and was put into the ready queue. This is because t2b is ahead of t2a in rq[2]. When t2b has processed every message waiting in xa, the process repeats.

The above illustrates the importance of task priorities. If, for example, t2b were t3a, then every time t2a sent a message, t3a would preempt and free it — quite a different sequence of events due to a small difference in priorities!

making and unmaking messages



This section discusses a technique we call *block migration*, which allows using smx’s exchange messaging capability for I/O blocks.

smx allows any block to be made into a message, which then can be propagated to tasks via exchanges. This brings the full exchange messaging capabilities of smx to bear on I/O blocks.

smxBase provides interrupt-safe block pools for use from ISRs, so that an ISR can obtain an input block from a base pool and fill it with incoming data. When full, the ISR invokes an LSR and passes the block pointer to it. The LSR makes the block into an smx message and sends the message to a message exchange, where a task waits to process it.

The message may be partially processed by the first task, then sent up to the next layer of the software stack via another exchange, and so forth. When the last task is done with the message, it simply releases it, and the data block automatically goes back to the correct base pool. This entire process requires no copying of the message; hence it is efficient and fast.

The reverse process can be used for block output: A message is obtained by a high-level task, partially filled, and then passed down to the next software level via an exchange. The task waiting at that exchange adds more information (e.g. a header) and passes the message down to the next level, etc.

The lowest level of the software stack (which could be a task or an LSR) unmakes the message into a bare block, loads its block pointer and pool handle into ISR global locations, and starts the output process. The ISR outputs all of the data, then releases the block back to the pool it came from. Like input, this entire process requires no copying of the message.

smx block migration provides considerable flexibility in that blocks can come from anywhere. If they are either base or smx pool blocks they will automatically be released back their correct pools. Whatever thread (ISR, LSR, or task) is releasing a block need not know where it came from. If the block is not from a pool, the pool parameter is NULL, and no action is taken to release the block to a pool.

smx_MsgMake() converts a *bare block* (i.e. one with no MCB nor BCB) to an smx message. The bare block can be from a base pool, a DAR, the heap, or it can be a static block. For example:

```
u8*  mbp;
MCB_PTR  msg:
PCB  poolA;

mbp = sb_BlockGet(&poolA, 4);
msg = smx_MsgMake(&poolA, mbp);
```

In this example, a base block is obtained from poolA, then made into an smx message. The result is no different from:

```
msg = smx_MsgGet(&poolA, &mbp, 4);
```

BlockGet() + MsgMake() allows an ISR to get a base block, fill it, then pass it to an LSR with mbp as the LSR parameter. The LSR makes the base block into a message and sends it to an exchange, where a task receives and processes it.

Whichever way msg is obtained above,

```
smx_MsgRel(msg, SIZE);
```

produces the same result — i.e. it will be cleared and released back to poolA and its MCB will be released to the MCB pool (smx_mcbs). It is important to note that the message block was obtained by an ISR, in the first case, and released by a task. The former used an smxBase function and the latter used an smx service (SSR). Hence an ISR and a task are able to easily share a block pool, which facilitates no-copy input.

In the case of a block that is not in a pool, MsgMake() is used as follows:

```
u8  smsg[NUM];
msg = smx_MsgMake(NULL, smsg);
```

NULL is loaded into the pool handle of the MCB to indicate that the message has no pool. In this case,

```
smx_MsgRel(msg, NUM);
```

releases the MCB and clears the block (due to passing NUM), but does not attempt to release it to any pool.

Note that a static block could be located in ROM as well as in RAM. In that case, it would be a read-only message and the size parameter should be 0, when releasing it. A ROM message may not seem to be of much use, but it could contain a table of information for a process:

```
typedef struct {
    /* table fields */
} T1 *tp;

msg = smx_MsgReceive(xchg, (u8**)&tp, TMO);
```

The task could then use tp->field operations to tell it how to process data messages that it is receiving from other sources. In this way, a master task might control the operation of another task by sending it canned messages telling it what to do.

smx_MsgUnmake() converts an smx message to a bare block. It does so by releasing its MCB back to the MCB pool. Unmake() can be used as follows:

```
PCB_PTR  poolA;
XCB_PTR  xa;
PCB_PTR  ppi;
u8*  bpi;

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

    msg = smx_MsgGet(poolA, &mbp, 4);
    /*  fill msg here using mbp */
    smx_LSRInvoke(LSRA, (u32)msg);
}

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

    bpi = smx_MsgUnmake(&ppi, msg);
    /* start message output  using bpi */
}
```

In this case, msg is obtained and loaded by taskA, then passed to LSRA, via the invoke parameter, where it is unmade into a bare message block and output is started. Note: LSRA can unmake msg because it is not required to be its owner.

smx_MsgUnmake() loads the globals, ppi and bpi, for use by the ISR performing output (not shown). It also releases the MCB back to its pool, smx_mcbs. The ISR uses bpi to access bytes in the message block. When all bytes have been sent, the ISR uses ppi to return the block to its pool:

```
sb_BlockRel(ppi, bpi, 0);
```

This will release the message block back to poolA. It is important to note that the message block was obtained by a task and released by an ISR. The former used an smx service (SSR) and the latter used an smxBase function. Hence a task and an ISR are able to easily share a block pool, which facilitates no-copy output.

smx_MsgMake() and smx_MsgUnmake() are complementary — one reverses the other's actions. smx_MsgGet() and smx_MsgRel() are also complementary. All four services are compatible with one another and may be used in any reasonable sequence. Message bodies may originate from any pools and will be returned to their correct pools, when released. The net result is a flexible no-copy method to move blocks of data from ISRs to LSRs to tasks and back.

peeking at messages and exchanges

The **smx_MsgPeek(msg, par)** and **smx_MsgXchgPeek(xchg, par)** allow obtaining information concerning messages and exchanges. To obtain information concerning a message pool, use the **smx_BlockPoolPeek()**. Although it is possible to read MCB and XCB fields directly, this is discouraged because future versions of smx are expected to utilize a software interrupt (SWI) API in order to run smx in privileged mode and application code in user mode. Then, smx objects will no longer be accessible from application code. In the interim, it is preferable to use peeks, rather than direct field accesses, because they are task-safe, and smx object field names may change.

Available message peek parameters are as follows:

SMX_PK_BP	body (block) pointer
SMX_PK_NEXT	next object in queue. NULL, if none.
SMX_PK_ONR	owner
SMX_PK_POOL	pool
SMX_PK_PRI	priority
SMX_PK_REPLY	reply handle
SMX_PK_SIZE	message body size
SMX_PK_XCHG	exchange where msg is waiting. NULL, if not waiting.

This function can be used only on messages that are in use. Otherwise, the MCB is cleared except for the free list link and 0 will be returned. 0 is returned for POOL if there is no pool.

Available message exchange peek parameters are as follows:

SMX_PK_TASK	first waiting task. NULL if none.
SMX_PK_MSG	first waiting message. NULL if none.
SMX_PK_MODE	exchange mode.
SMX_PK_NAME	name of exchange.

Example:

To find how many messages are waiting at xchgA:

```
u32  ctr = 0;
CB_PTR   m;

m = (CB_PTR)smx_MsgXchgPeek(xchgA, SMX_PK_MSG);
while (m != NULL)
{
    ctr++;
    m = smx_MsgPeek(m, SMX_PK_NEXT);
}
```

If no messages are waiting, m will be NULL. When the end of the message wait list is reached, m = NULL. Note that the search is started with MsgXchgPeek(), but it loops on MsgPeek(). This is a good example of combining peek operations.

To timestamp all messages in use:

```
MCB_PTR  max, msg;
u32  *mbp;
u32  ts = smx_etime;

msg = (MCB_PTR)sb_BlockPoolPeek(&smx_mcbs, SMX_PK_MIN);
max = (MCB_PTR)sb_BlockPoolPeek(&smx_mcbs, SMX_PK_MAX);

for ( ; msg <= max; msg++)
{
    if (smx_MsgPeek(msg, SMX_PK_ONR))
    {
        mbp = (u32*)smx_MsgPeek(msg, SMX_PK_BP)
        *mbp = ts;
    }
}
```

Messages in use have MCBs, so in the above example, the MCB pool is searched for MCBs with owners.(Free messages do not have owners.[3]) When an owned MCB is found, its message body pointer is obtained and it is used to load a timestamp into the first word of the message.

Note that msg and max are loaded by the base block pool peek. This is because smx_mcbs is a base pool. These peeks are not task-safe, but they are obtaining the handles of the first and last MCBs in smx_mcbs, which do not change. Otherwise base functions should be protected by locking tasks and possibly LSRs to prevent access conflicts.

message owner

Messages have owners. The owner handle is stored in mcb.onr. When a message is sent to an exchange, except a broadcast exchange, the exchange becomes the new owner. When the message is received, except from a broadcast exchange, the receiving task or LSR becomes the new owner. This is important because a task cannot release, send, or unmake a message, unless it is the owner. This is done for safety. LSRs can perform these operation because they are necessary for no-copy I/O. In the case of a broadcast exchange, the sender retains ownership. See broadcast messages, below.

using the reply field

The sender can tell the receiver where to reply simply by passing a fourth parameter, reply, in smx_MsgSendPR(). This parameter is stored in mcb.rpx. The allowable reply types are: exchange, semaphore, event group, or event queue. The reason that only these objects can be used for replies is that mcb.rpx is a 16-bit index into the QCB pool, which contains the control blocks for these objects. The rpx index is used in order to save space in the MCB.

The receiving task can peek at the reply field of the MCB, use smx_SysWhatIs() to find its type, then send the appropriate response. The reply field is useful for client/server designs, such as shown in the following example:

client/server example

```
XCB_PTR  ack_xchg, data_xchg;
PCB_PTR  msg_pool;

ack_xchg = smx_MsgXchgCreate(NORM, "ack_xchg");
data_xchg = smx_MsgXchgCreate(PASS, "data_xchg");

void clientA_main(void)
{
    u8*  mbp;
    MCB_PTR  msg;
    u8  pri = (u8)smx_TaskPeek(smx_ct, SMX_PK_PRI);

    if  ((msg = smx_MsgGet(msg_pool, &mbp, 0)) != NULL)
    {
        *mbp = ACK;    /* get started */
        do
        {
            if ((BOOLEAN)*mbp == ACK)
                load_new_msg(mbp);
            else
                load_old_msg(mbp);
```

[3] MCBs are cleared when returned to the MCB pool. Hence, a zero owner field indicates a free MCB.

```
                smx_MsgSendPR(msg, data_xchg, pri, ack_xchg);
            }  while ((msg = smx_MsgReceive(ack_xchg, &mbp, TMO)) != NULL);
        }

        /* take corrective action */
    }
    void serverX_main(void)
    {
        u8*   mbp;
        MCB_PTR  msg;
        u32  reply, type;

        while ((msg = smx_MsgReceive(data_xchg, &mbp, INF)) != NULL)
        {
            /* reuse msg for reply -- process_msg() returns TRUE if ok */
            *mbp = process_msg(mbp);
            reply = smx_MsgPeek(msg, SMX_PK_REPLY);
            type = smx_SysWhatIs(reply);
            switch (type)
            {
                case SMX_CB_XCHG:    /* clientA */
                    smx_MsgSend(msg, (XCB_PTR)reply);
                    break;
                case SMX_CB_SEM:       /* not clientA */
                    if (*mbp == ACK)
                        smx_SemSignal((SCB_PTR)reply);
                default:
                    smx_MsgRel(msg, 0);
            }
        }
    }
```

In this example, the same message is recycled — first to carry data to serverX, then to carry ACK or NAK back to clientA. After sending a data message, clientA waits for an ACK or NAK, then sends new data or resends the old data, respectively. Meanwhile, serverX receives the data message. It processes the message, then sends ACK, if successful, or NAK, if not successful, to the client's reply exchange. Since the serverX message requires no priority nor reply, the short send is used.

The data exchange is a pass exchange, so the client sends the data message with its own priority. (The priority could be higher or lower depending upon the urgency of the message.) This enqueues the message at its appropriate priority in the server queue (i.e. the message queue at the exchange) and causes the server to run at the client's priority while it processes the message.

When the server is ready to send an ACK or NAK, it obtains the reply handle from the message. It then uses smx_SysWhatIs() to determine what kind of reply is expected. The following switch statement either sends an ACK or NAK message to ack_xchg, which is the correct response for clientA, or it signals a reply semaphore for ACK and does nothing for NAK. In the latter case, the client would wait at the reply semaphore with a timeout; no signal within the timeout would be interpreted as NAK. Note that in this and the default cases, msg is released to its pool.

Note that the client task does not know the identity of the server task, nor does the server task know the identity of the client task. Both are sending to intermediate exchanges. In the case of the server task, it sends ACK or NAK or a signal to the reply object. It has no idea where things are coming from or going to, hence it is operating in a sheltered environment. This creates an adaptable structure, permitting client tasks and server tasks to be changed for different product models or conditions.

serverX uses an infinite timeout. If no one wants its services, why should it complain? The situation is different for clientA, because it wants to know when it can send more data. So, a timeout, TMO is specified and if a reply has not been received, within TMO ticks, the do loop is exited and recovery code executes. In the recovery code, it might be a good idea to peek at the message to see who owns it, now. If serverX owns it, it is probably best to wait a little longer. However, if it is stuck at data_xchg, the message could be bumped to the next higher priority:

```
u8 pri = smx_MsgPeek(msg, SMX_PK_PRI);
smx_MsgBump(msg, ++pri);
```

If the message has been returned to its pool (msg->onr == NULL), with no acknowledgement, something is clearly amiss. Sending it again may be necessary.

Note that it is not assumed that the first smx_MsgGet() succeeds. This is defensive programming. It is particularly important to not write to mbp if smx_MsgGet() fails because mbp will be pointing at some random location and damage may result. As previously noted, a good approach is to use a resource semaphore to protect against exhausting the msg_pool. This would force clientA to wait for a message block. However, it is still possible to run out of MCBs, in which case smx_MsgGet() would fail.

broadcasting messages

A new element has been added to smx v4.2; it is the broadcast exchange. A broadcast exchange accepts only one message at a time and the message “sticks” to it; a receive operation does not transfer exclusive use of the message to the receiving task. Instead, the receiver gets only its handle and a pointer to its message body. Furthermore, the task queue is a FIFO (i.e. non-priority) queue. When a message is sent to a broadcast exchange, all waiting tasks get the message handle and message body pointer and are resumed, immediately. Any subsequent task receiving from the broadcast exchange will also get the message handle and message body pointer and it will continue running (i.e. not wait at the exchange). Tasks receiving from a broadcast exchange are referred to as *slave* tasks, in what follows.

All slave tasks have access to one message’s MCB and its message body. This is the essence of a broadcast. Normally, slave tasks will only read broadcast messages, not modify them. However, each slave could be assigned a section of the message body that it is permitted to alter. This fosters what we call *distributed message assembly*.

The single task which sends the message is called the *master* task. The master task maintains access to the broadcast message since it still has the message handle. When all slaves have signaled that they are done with the message, the master task can then do one of three things:

(1) Release the message back to its pool.

(2) Send a new message to the exchange.

(3) Change the message body and resend it to the broadcast exchange, effectively resulting in a new message being broadcast.

The following example illustrates a simple message broadcast:

```
void em10(void)
{
    u8*  mbp;
    MCB_PTR  msg;

    t2a = smx_TaskCreate(em10_t2a_main, P2, 500, NO_FLAGS, "t2a");
    t3a = smx_TaskCreate(em10_t3a_main, P3, 500, NO_FLAGS, "t3a");
    xbd = smx_MsgXchgCreate(BCST, "xbd");
    smx_TaskStart(t2a);
    smx_TaskStart(t3a);
    /* get msg, fill, and send to broadcast exchange */
    msg = smx_MsgGet(msg_pool, &mbp, 0);
    memcpy(mbp, "em10 Test Message", sizeof("em10 Test Message"));
    smx_MsgSend(msg, xbd);

    smx_MsgRel(msg, 0);
}

void em10_t2a_main(void)
{
    u8*  mbp;
    MCB_PTR  msg;
    TCB_PTR  onr;

    msg = smx_MsgReceive(xbd, &mbp, INF);
    onr = (TCB_PTR)smx_MsgPeek(msg, SMX_PK_ONR);
    if (onr == esmx)
        sb_MsgVarDisplay(INFO, (char*)mbp);
}

void em10_t3a_main(void)
{
    u8*  mbp;
    MCB_PTR  msg;
    TCB_PTR  onr;

    msg = smx_MsgReceive(xbd, &mbp, INF);
    onr = (TCB_PTR)smx_MsgPeek(msg, SMX_PK_ONR);
    if (onr == esmx)
        sb_MsgVarDisplay(INFO, (char*)mbp);
}
```

In the above example two tasks of priorities 2 and 3 are created and started by a priority 1 master task. Each task waits at the xbd broadcast exchange. t2a is first because it ran first and xbd has a FIFO wait queue. The master task gets a message, fills it, and sends it to xbd. This causes both tasks to be resumed and t3a runs first, since it has higher priority than t2a. Note that both slave tasks are able to access the MCB via the msg handle and to display the message via the mbp pointer. However, neither can release it, send it, or unmake it. The master task resumes when both slaves are done and it releases the message.

Interlocking of tasks is necessary to assure that all intended tasks receive a broadcast message and that none receive it more than once. After sending the message, the master waits at a threshold semaphore, semT. The threshold is set to the expected number of slave readers. As each slave finishes with the message, it signals semT and waits at a gate semaphore, semG. semT reaches its threshold when all slaves are done, and wakes up the master. The master performs one or more of the above operations, then signals semG, thus permitting the slaves to come to the broadcast exchange for the next message.

Multiple broadcasters can send to a normal or pass exchange that is serviced by a single master broadcast task. The normal exchange can prioritize messages and messages can accumulate in its priority message queue. MCBs include a reply handle so that the master task can tell a broadcaster when its message has been broadcast — e.g. by signaling the event semaphore at which the broadcaster waits.

Caution: A message at a broadcast exchange must be released before deleting the broadcast task, because the message owner is the broadcast task. When a task is deleted, all messages that it owns are automatically released. This causes an error for a broadcast message which is linked to a broadcast exchange, because normally the exchange would be the owner.

proxy messages and multicasting

It is possible to make multiple messages that share one message block. The first such message is called the *real* message and additional messages are called *proxy* messages because they represent, but are not, real messages. The operation might proceed as follows:

```
MCB_PTR  rm, pm[N];
u8  *bp;
u32  i;

rm = smx_MsgGet(poolA, &bp, 0);
MsgFill(bp);
for (i = 0; i < N, i++)
    pm[i] = smx_MsgMake(NULL, bp);
```

smx_MsgGet() is used to get a real message, rm, from its pool; MsgFill() loads data into the message body; MsgMake(NULL, bp) is called N times to make multiple proxies of rm. The proxy handles are stored in the pm[] array. The NULL is used for the pool parameter because proxy messages have no pool. After this process, there is one real message and N proxies of it. The proxies can be sent to other exchanges where server tasks wait:

```
XCB_PTR  xchg[N];

for (i = 0; i < N, i++)
    smx_MsgSend(pm[i], xchg[i]);
```

This process is called *multicasting* to distinguish it from broadcasting, which was discussed previously. Multicasting is more selective than broadcasting because the proxies are sent to specific exchanges where specific tasks are expected to be waiting. Normally, a multicast might be to only a few tasks, whereas a broadcast might be to many tasks. A broadcast is inherently more efficient, since only one message send is required, whereas a multicast requires a message send per receiver. However the multicaster theoretically has better control over which tasks receive its messages.

Receiving task n would typically look like:

```
void tn_main(void)
{
    u8*  mbp;
    MCB_PTR  pmsg;

    while (pmsg = smx_MsgReceive(xchg[n], &mbp, INF))
    {
        /* process pmsg using mbp */
        smx_MsgRel(pmsg, 0);
    }
}
```

The above is similar to receiving a normal message, except that the message block of pmsg is not actually released by tn. It must be released by the master task:

```
smx_MsgRel(rm, 0);
```

It might be clearer in tn to use:

```
smx_MsgUnmake(NULL, pmsg);
```

which does the same thing as smx_MsgRel().

For distributed message assembly, we call the multicaster the *client* task in order to be consistent with the client/server paradigm. Using proxy messages is a bit better than broadcasting for distributed message assembly. The client gets an empty message from a pool, creates and sends proxies out to server exchanges, with different message body pointers. For example, a protocol header task would receive a header section pointer, whereas a data task would receive a data section pointer. Each server loads its section, then signals completion to the client via the reply handle in the proxy message. The client, which retained the real message, could then send it to another exchange, perhaps to be output. Each server releases its proxy back to the MCB pool, after it finishes with it. This example illustrates how a single client task could assemble messages with data and header loaded from different protocols using proxy messaging:

```
SCB_PTR  st;
TCB_PTR  t3m, t2h, t2d;
XCB_PTR  xa, xb, xout;
st = smx_SemCreate(THRES, 2, "st");
t3m = smx_TaskCreate(t3m_main, P3, ESS, NO_FLAGS, "t3m");
t2h = smx_TaskCreate(t2h_main, P2, ESS, NO_FLAGS, "t2h");
t2d = smx_TaskCreate(t2d_main, P2, ESS, NO_FLAGS, "t2d");

void t3m_main(void)
{
    MCB_PTR rm, pma, pmb;
    u8  *mbp, *hp, *dp;

    /* get real msg and set pointers */
    rm = smx_MsgGet(tpool, &mbp, 0);
    hp = mbp;
    dp = mbp + HDR_SZ;

    /* make proxy msg a using the header section pointer and send to xa */
    pma = smx_MsgMake(NULL, hp);
    smx_MsgSend(pma, xa);

    /* make proxy msg b using the data section pointer and send to xb */
    pmb = smx_MsgMake(NULL, dp);
    smx_MsgSend(pmb, xb);

    /* start msg builder tasks and wait for them to finish */
    smx_TaskStart(t2h);
    smx_TaskStart(t2d);
    smx_SemTest(st, TMO);

    /* Forward the assembled message */
    smx_MsgSend(msg, xout);
}

void t2h_main(void)
{
    MCB_PTR  pma;
    u8*  hp;

    /* get  proxy msg a and load header */
    pma = smx_MsgReceive(xa, &hp,  2);
    LoadHeader(hp);

    /* release proxy msg and signal done */
    smx_MsgRel(pma, 0);
    smx_SemSignal(st);
}
```

```
void t2d_main(void)
{
    MCB_PTR pmb;
    u8*   dp;

    /* get proxy msg b and load data */
    pmb = smx_MsgReceive(xb, &dp,  2);
    LoadData(dp);

    /* release proxy msg and signal done */
    smx_MsgRel(pmb, 0);
    smx_SemSignal(st);
}
```

Multicasting can also be used for information distribution, like broadcasting. The advantage is that it is more selective and multicast messages can build up at an exchange and be handled in priority order just like real messages. In this case an interlock is necessary so that the multicaster does not release the message before all viewers have seen it. That could be accomplished by each server signaling a threshold semaphore specified in the proxy's reply field.

Proxy messages are a result of the capability to make messages from message body pointers. The importance of this is that it provides a means to distribute information to multiple viewers without actually duplicating the information; it also allows assembling information from multiple servers into one message and then pass it on in a no-copy manner.

CAUTION: Interlocks are required to insure that all proxies are released before the real message is released, else remaining proxies will point to an invalid message body.

other message services

smx_MsgReceiveStop(xchg, **bpp, tmo) is the stop variant of smx_MsgReceive(); it is intended for use by one-shot tasks. It operates the same as smx_MsgReceive(), except that it stops and restarts the current task, rather than suspending and resuming it. See the One-Shot tasks chapter for discussion of how to use one-shot tasks and stop services.

smx_MsgXchgClear(sem) resumes all waiting tasks with NULL or releases all waiting messages and clears exchange fields. It is useful in recovery situations. Other than the clear function, smx provides no way to change an exchange because it is risky to do so. It is better to delete the exchange, then recreate it with the desired parameters.

smx_MsgBump(msg, pri) bumps msgafter the last message of the specified priority in its current queue. Does nothing if pri == SMX_PRI_NOCHG, msg is not in an exchange queue, or it is the only message in an exchange queue. Also changes the priority field in the msg MCB. This is useful if a message is languishing at an exchange due to higher-priority messages being received — it allows the message's priority to be increased.

summary

smx offers both pipe (AKA "queue") and exchange messaging. Exchange messaging is safer and more powerful. Tasks send messages to exchanges using MsgSend() and receive messages from exchanges using MsgReceive(). This provides anonymity for both sender and receiver. Either tasks or messages may wait at exchanges. There are three types of exchanges: normal, pass, and broadcast. For normal and pass exchanges, tasks and messages wait in priority queues. In addition to operating like normal exchanges, pass exchanges pass a message's priority to the receiving task.

Message bodies come from smx data block pools. The MsgGet() function combines a message body with a message control block (MCB) from the MCB pool. The MsgRel() function reverses this process. The MsgMake() function can make a message from any bare block, including a base pool block. This is useful to *migrate* input packets to tasks for processing. The MsgUnmake() function reverses this process, which is useful for migrating messages to blocks for output.

A broadcast exchange accepts one message at a time. The message “sticks” to the exchange until it is either released or another message is sent, causing it to be released. All waiting tasks are FIFO enqueued and all receive the message handle and the message body pointer when a message is sent. All tasks receive the same message until it is removed from the exchange. It is helpful to use threshold and gate semaphores to ensure that all intended tasks receive a message and none receives it twice.

By making use of the message make feature, proxy messages can be created. These can be sent to different exchanges to achieve multicasting or for distributed message assembly. Message migration, broadcasting, and proxy messages offer more efficient ways to structure I/O and the assembly and dissemination of information within multitasking systems.

Additional services are available to peek at messages and exchanges, to bump message priorities, to release all messages owned by a task, and to clear exchanges.

Chapter 14 Tasks

BOOLEAN	smx_TaskBump(TCB_PTR task, u8 pri)
TCB_PTR	smx_TaskCreate(FUN_PTR code, u8 pri, u32 stksz, u32 flags, const char *name)
BOOLEAN	smx_TaskDelete(TCB_PTR *task)
BOOLEAN	smx_TaskHook(TCB_PTR task, FUN_PTR entry, FUN_PTR exit)
void *	smx_TaskLocate(const TCB_PTR task)
BOOLEAN	smx_TaskLock(void)
BOOLEAN	smx_TaskLockClear(void)
BOOLEAN	smx_TaskResume(TCB_PTR task)
BOOLEAN	smx_TaskSetStackCheck(TCB_PTR task, BOOLEAN state)
BOOLEAN	smx_TaskSleep(u32 time)
void	smx_TaskSleepStop(u32 time)
BOOLEAN	smx_TaskStart(TCB_PTR task)
BOOLEAN	smx_TaskStartNew(TCB_PTR task, u32 par, u8 pri, FUN_PTR fun,)
BOOLEAN	smx_TaskStartPar(TCB_PTR task, u32 par)
BOOLEAN	smx_TaskStop(TCB_PTR task, u32 timeout)
BOOLEAN	smx_TaskSuspend(TCB_PTR task, u32 timeout)
BOOLEAN	smx_TaskUnhook(TCB_PTR task)
BOOLEAN	smx_TaskUnlock(void)
BOOLEAN	smx_TaskUnlockQuick(void)

introduction

Having covered the basics of memory management and inter-task communication, it is now appropriate to look deeper into smx task services. Task types, scheduling, handling, states, creation, main function formats, and starting have been covered in the Introduction to Tasks chapter.

task priority

Task priority can be any value from 0 to 126 where 0 is the lowest and 126 is highest. 127 is reserved for SMX_PRI_NOCHG, which is used in some smx services, when a priority change is not desired.

A large number of priorities is not necessarily desirable. Often a few are sufficient and often it is more important that the longest waiting task runs first, among tasks of equivalent importance. This corresponds to everyday experience — among equals, we expect the first in line to be served first. This tends to be true in embedded systems also — if a task is kept waiting too long, it may miss its deadline.

In most systems ten or fewer priority levels is adequate. It may be difficult to break task importance down finer than this. We suggest starting with a few levels and adding levels as the need for them becomes evident. It is best to name the levels, such as: PRI_MIN, PRI_LO, PRI_NORM... Then it is easier to add intermediate levels such as PRI_LO1. These names are already defined in the PRIORITIES enum in xcfg.h and are used in SMX middleware. You can change the names, if you wish, by doing search-and-replace operations on the middleware that you are using.

In a large system, you might want to prioritize major functions and to prioritize tasks within each function. For example: FILE_LO, FILE_MID, FILE_HI, TCP_LO, ... If your system had 10 such functional areas and you needed about 4 levels, each, it would not be unreasonable to end up with 40 plus system priority levels. This might create a sane system out of what would otherwise be an unmanageable one. The cost of having more priority levels is just one ready queue control block in the ready queue per level — 16 bytes. Hence and additional 40 levels costs only 640 bytes. As discussed below, there is minimal performance penalty.

Priority 0, called PRI_MIN, is used by the idle task, smx_Idle. See the discussion, below. The highest defined priority is called PRI_SYS and is reserved for system tasks. Within smx, SMX_MAX_PRI is used, which has the same value.

changing a task's priority

To change a task's priority, use task bump:

```
smx_TaskBump (task, NEW_PRIORITY);
```

This will put it at the end of the NEW_PRIORITY level or the end of the same level, if the priority is unchanged. As previously discussed, the current task may bump itself, which is useful for round-robin scheduling. It is not recommended to change the priority field in task's TCB directly. This could cause serious problems. (Note: In general smx control blocks should not be altered.)

Bumping is also useful for increasing the priority of a task that is being starved for processor time or reducing the priority of a task that is hogging processor time. Run time counts in TCBs can be used to determine this, if profiling is enabled:

```
count = smx_TaskPeek(task, PRI);
```

TaskStartNew() can also be used to change a task's priority:

```
smx_TaskStartNew(task, 0, NEW_PRIORITY, t2a_run);
```

This would normally be done as part of changing the task's main function.

idle task

smx_Idle has priority 0. It is the only task required by smx; it is created by smx_Go(). smx_Idle runs only when no other task is ready[4]. The idle task performs services such as stack scanning, etime rollover, profile display, operator message display, and other low-priority functions. When the idle task completes a pass, it bumps itself at priority 0. Hence, other priority 0 tasks can run in a round-robin manner. It is recommended that priority 0 application functions be implemented this way, rather than by adding them to smx_IdleMain().

task flags

The behavior of each task is controlled by several flags within its TCB. Three of these flags are user-controlled:

(1)	stk_chk	Enables stack checking.
(2)	hookd	Enables hooking entry and exit routines into context switches for the task.
(3)	strt_lockd	Starts the task locked.

The stk_chk flag is set when a task is created. It may be set or cleared by:

```
smx_SetStackCheck(t2a, ON/OFF);
```

[4] The smx scheduler is capable of looping internally if there is no task to run.

If true, the scheduler will check for stack overflow whenever the task is suspended or stopped. The hookd flag is off by default and set by:

```
smx_TaskHook(task, task_entry, task_exit);
```

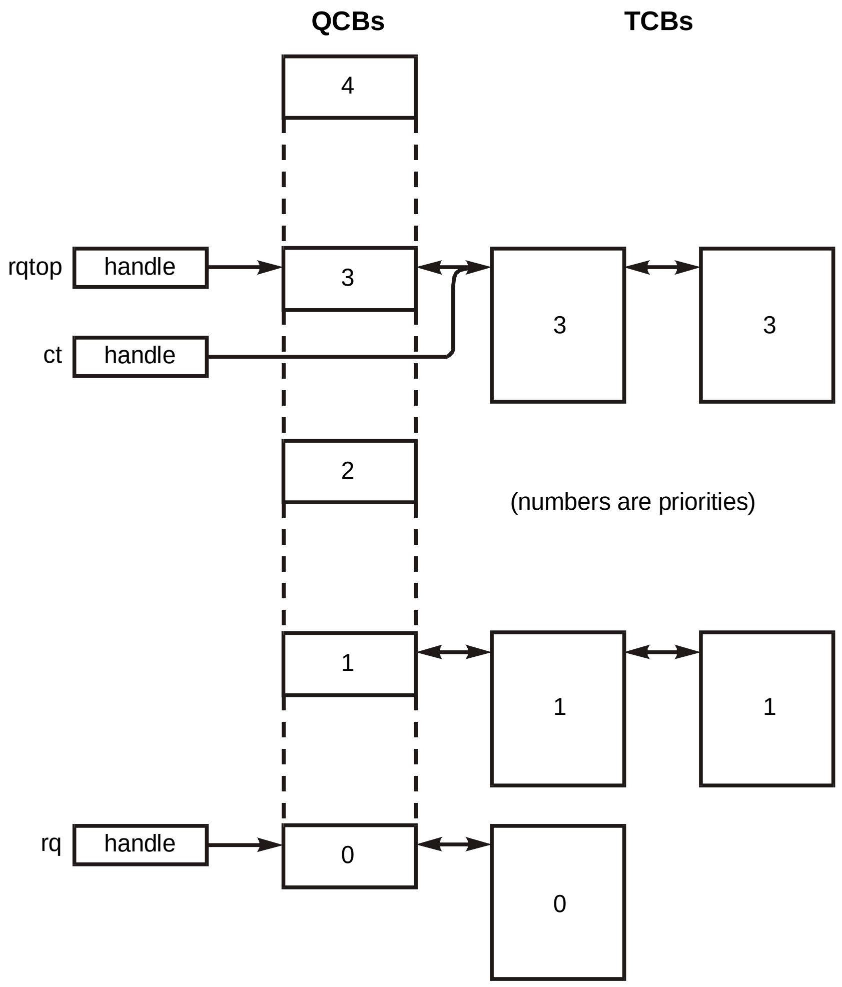
It is cleared if NULL functions are specified. The strt_lockd flag is set by using SMX_FL_LOCK at the time the task is created:

```
t2a = smx_TaskCreate(t2a_main, PR2, 200, SMX_FL_LOCK, "t2a");
```

This is similar to starting an ISR with interrupts disabled. In this case, it prevents preemption during the task initialization code.

ready queue

The ready queue is a queue of all tasks that are ready to run. It is designed to support large numbers of tasks without impairing performance. The ready queue is defined as a static array in xglob.c of ready queue control blocks (RQCBs), named smx_rq. There is one rq level per priority level, including 0, so the size of the array is SMX_MAX_PRI + 1. The levels are in increasing priority order. Each level has an RQCB which heads the subqueue of tasks at that level's priority.



In the above diagram, rq (smx_rq) points to the beginning (lowest priority level) of the ready queue and rqtop (smx_rqtop) points to the highest occupied level. ct (current task) points to the current task's TCB.

Even though the current task is running, its TCB remains in the ready queue. Also, its TCB is typically the first at the level rqtop (as shown), but not always. For example, a locked task could bump itself to the end of its priority level (with smx_TaskBump()) but keep running since it is locked.

rq is structured for speed. To enqueue a task, the desired level of rq is accessed by using the task's priority as the index, and the task's TCB is linked to the end of the queue at this level. No searching is required, since the back link (bl) in each RQCB points to the last TCB in its level.

When the current task is stopped or suspended, the scheduler dispatches the top task in the ready queue to run. To find the top task, the rqtop pointer is used. It points to the highest occupied level. The top task is the first task in this level. rqtop is increased or decreased as the top occupied level of rq changes.

If rqtop is not pointing to a valid rq level or if the rq level is empty, the task scheduler will attempt to reset rqtop and/or repair rq and try again. It reports SMXE_RQ_ERROR, then reports SMXE_Q_FIXED, if it is able to fix the ready queue and continue.

smx_rq can easily be viewed in a watch window by entering smx_rq and clicking on it. This shows the rq array. Clicking on a level shows the RQCB for that level. Clicking on fl for the level shows the TCB of the first waiting tasks. The entire task queue can be traced by clicking on successive fl's.

current task (ct)

When the top task is selected, it becomes the *current task* and its handle is loaded into smx_ct. If the current task is locked (i.e. preemption inhibited), another task, which becomes the top task, will not run until the current task is unlocked, suspended, stopped, or deleted. Then, the new top task will become the current task. If the current task is *unlocked* and another task becomes the top task, the new top task will run. This process is called *preemption*.

Note: smx_ct can be reliably used in a task to give the handle of the task, itself. However it is not reliable in an LSR because, what was top task when the LSR started running, may no longer be the top task due to an SSR (e.g. smx_SemSignal()) called by the LSR. So, when the LSR finishes, the top task will become the new ct, unless ct is locked.

inter-task operations

smx offers inter-task operations to start, stop, resume, suspend, and delete tasks. These operations can be performed by one task upon another task. Hence, they are useful in startup, shutdown, and recovery situations. They may also be useful to abort operations that have already been started or to restart operations with newer data. They might be used in a manner similar to branch prediction in a processor. They can may be used to abort waits for events that have taken too long or will not occur.

starting and stopping tasks

When a task is first created, it is in a dormant state (i.e. timeout inactive and not in any queue). It will stay in this state, potentially forever, until another task starts it with:

```
TCB_PTR atask;
smx_TaskStart (atask);
```

This puts atask into the ready queue. It does not actually run the task. That is done by the scheduler. Two variants of smx_TaskStart() are available:

```
smx_TaskStartPar (atask, par);
smx_TaskStartNew (atask, par, priority, new_main);
```

These are used primarily for one-shot tasks — see the One-Shot Tasks Chapter for further discussion of them.

A task can also be directly stopped by another task:

```
smx_TaskStop (atask, timeout);
```

This removes atask from any queue it may be in (e.g., waiting for a signal at semaphore) and sets its timeout as follows:

```
smx_timeout[tn] = smx_etime + timeout;
```

where tn is atask's index into the TCB array, and smx_etime is the current elapsed time. Three special values of timeout are:

(1) INF (SMX_TMO_INF): smx_timeout[tn] is made inactive and atask will wait forever.

(2) NO_CHG (SMX_TMO_NOCHG): atask is stopped, but its timeout is not changed.

(3) NO_WAIT (SMX_TMO_NOWAIT): smx_timeout[tn] is made inactive, and atask restarts immediately.

smx_TaskStart() and smx_TaskStop() function similarly. For example, if atask is waiting in a queue, smx_TaskStart(atask) will remove it from the queue, disable its timeout, and enqueue it in rq. smx_TaskStop(atask, tmo) does the same, but waits tmo ticks to enqueue atask in rq. The final result is the same, except for the delay.

For normal tasks, smx_TaskStart() is used primarily to start a new task. However, it and smx_TaskStop() may also be useful to abort a task, and possibly start it over. (When doing this, one must be careful to not get the same resources again.)

The current task may start or stop itself. Starting itself causes a restart — i.e., it is moved to the end of its rq level. If it is the only task in this level and if there is no task in a higher level, it will immediately be dispatched again, but it will start from the beginning of its main function. If the current task stops itself, it is removed from rq and its timeout is set, as explained above.

An important aspect of both smx_TaskStart() and smx_TaskStop() is that they cause a task with a temporary stack to release the stack back to the stack pool. This is discussed further in the One-Shot Tasks chapter.

resuming and suspending tasks

A task may be resumed with:

```
smx_TaskResume(atask);
```

which does the same as smx_TaskStart(atask), except that atask continues execution where it left off rather than at the beginning of its main function. Also it does not cause a one-shot task to release its temporary stack. smx_TaskResume() can be used by another task to cause a task to stop waiting for a resource and to resume running, as if it had timed out.

A task may be suspended with:

```
smx_TaskSuspend(atask, timeout);
```

which does the same as smx_TaskResume(task), but with a timeout before enqueueing atask in rq. The same special timeouts apply as smx_TaskStop(). A suspended task can be either resumed or restarted.

deleting tasks

Sometimes, it may be necessary to go beyond merely stopping a task. It may be necessary to purge the task from the system with:

```
smx_TaskDelete(&atask);
```

This clears the atask TCB so that it is available for use by another task. If atask has a temporary stack, it is returned to the stack pool; if atask has a permanent stack, it is freed to the heap. Also all owned blocks, messages, and timers are released. Finally, the atask handle is cleared so that it cannot be mistakenly reused.

smx_TaskDelete() is useful for infrequently running tasks that can be created when needed and deleted when not. Examples might be tasks that do initialization following boot up, produce a daily report, or run only when a technician shows up such as maintenance or code-update tasks.

Deleted tasks cannot wait for an event and cannot be seen in smxAware. They are in the NULL or non-existent state. If these are not desirable, one-shot tasks should be used, instead. A one-shot task gives up its stack, when stopped. This may be 200 bytes, or more. To delete the same task only releases another 84 bytes of TCB and timer space for reuse. See the One-Shot Tasks Chapter for more discussion.

locking and unlocking tasks

```
BOOLEAN    smx_TaskLock(void)
BOOLEAN    smx_TaskLockClear(void)
BOOLEAN    smx_TaskUnlock(void)
BOOLEAN    smx_TaskUnlockQuick(void)
```

smx allows the current task to lock the scheduler, in order to protect itself from preemption. Locking is useful for short sections of critical code and for accessing global variables because its overhead is small. ISRs and LSRs are not blocked from running so there is no foreground impact. Normal usage is as follows:

```
smx_TaskLock();
/* perform critical section */
smx_TaskUnlock();
```

Locking is implemented by the smx_lockctr, which prevents the current task from being preempted if it is nonzero. Task locking is comparable to disabling a processor’s interrupt enable flag to prevent the current code from being interrupted. smx_TaskUnlock() checks to see if a higher priority task is ready to run. Hence, a task can be locked for a relatively long time.

The smx_TaskUnlockQuick() does not check for preemption; it is intended for use where low overhead is needed (e.g. accessing a single global variable) and the likelihood of preemption is low:

```
smx_TaskLock();
a = globalA;
smx_TaskUnlockQuick();
```

If, for any reason, the current task stops running, smx_lockctr is automatically cleared, so that other tasks can run. This is not a problem if a task is stopped or deleted, because there is no code left to execute. However, if the current task causes itself to be suspended, lock protection will be lost. This could be a problem and care must be taken to avoid it. Basically, waits or stops are not permitted during a locked section. For example:

```
smx_TaskLock();
if (smx_SemTest(semA, tmo))
    /* perform function */
smx_TaskUnlock();
```

is ok if tmo = NO_WAIT, but not otherwise. It is important to note that smx_SemTest(semA, !NO_WAIT) will break the lock, even if no wait actually occurs. This is because actions must be consistent — the same call cannot break the lock one time and not another. Also, smx_SemTestStop() is not ok, because it always stops the task and whether the task starts locked or not is governed by task->flags.strt_lockd. Here, again, consistent operation is necessary.

It should be noted that it is really the scheduler which is locked and not tasks. However, this seems to be a feature associated with tasks, so it is named accordingly.

lock nesting

Lock nesting is supported because it is needed for libraries in which functions do locking and also may call each other. For example:

```
void funA(void)                 void funB(void)
{                               {
    smx_TaskLock();                 smx_TaskLock();
    funB();                         /* operationB */
    /* operationA */                smx_TaskUnlock();
    smx_TaskUnlock();           }
}
```

If locking were implemented with a simple lock flag, operationA would not be protected because the task unlock in funB() would have cleared the lock. The lock function increments the lock counter and the unlock function decrements it; when it reaches zero, the task becomes unlocked. Hence operationA is protected.

To guard against the lock counter becoming permanently non-zero and thus no other tasks running, smx_TaskLockClear() can be used at the known end of a locked interval to assure that the locked counter is forced to zero and thus other tasks can run. An additional safety feature, is the user can specify a lock count limit that cannot be exceeded (SMX_CFG_LOCK_NEST_LIMIT in xcf.h).

uses for task locking

The strt_lockd flag can be set in a task, when it is created. If set, the task will always start locked. This is useful to prevent preemption during task initialization and for non-preemptible one-shot tasks. (See the One-Shot Tasks chapter.)

Another important use for task locking is to avoid unnecessary task switches. In the following example, task t3a (priority 3) is waiting on semA. When task t2a (priority 2) signals semA, t3a will preempt immediately and run:

```
void t3a_main(void)
{
    while (1)
    {
        smx_SemTest(semA, TMO);
        funA();
    }
}

void t2a_main(void)
{
    while (1)
    {
        funB();
        smx_SemSignal(semA);
        smx_SemTest(semB, TMO);
    }
}
```

When t3a finishes it will wait on semA again, and t2a will resume, only to wait on the semB. This is a wasted task switch. A more efficient way to code t2a is as follows:

```
void t2a_main(void)
{
   while (1)
   {
      funB();
      smx_TaskLock();
      smx_SemSignal(semA);
      smx_SemTest(semB, TMO);
   }
}
```

Now t3a cannot preempt until t2a tests semB. This prevents the unnecessary task switch. Note that the lock counter is automatically reset on the semB test, whether it passes or not. The only exception to this is if TMO = NO_WAIT.

making tasks act as expected

When you look at code incorporating kernel services, it does not necessarily behave in the sequential manner to which you are accustomed. For example, in the following code, which task runs first — higher priority t3a or t2a?

```
void  t1a_main(void)
{
   while (1)
   {
      smx_TaskStart(t2a);
      smx_TaskStart(t3a);
      smx_SemTest(sema, tmo);
   }
}
```

The answer is t2a. When t2a is started, it immediately preempts t1a and runs. When t2a completes, t1a resumes and starts t3a, which also immediately preempts t1a and runs. When t3a completes, t1a resumes, only to wait at sema. Not only are the tasks running in different order than expected by their priorities, but note that two hidden task switches (t2a -> t1a, and t3a -> t1a) occur, which waste processor time. To correct these:

```
void  t1a_main(u32 par)
{
   while (1)
   {
      smx_TaskLock();
      smx_TaskStart(t2a);
      smx_TaskStart(t3a);
      smx_SemTest(sema, tmo);
   }
}
```

Now, t1a is locked so it cannot be preempted. t1a suspends and auto-unlocks on SemTest(). At this point, both t3a and t2a are in rq and t3a will run first.

Advanced Topics

extending the task context

It sometimes happens that a task has an extended context which must be preserved when the task is suspended and restored when the task is resumed. A good example is a bank switching register which points to different memory banks for different tasks. Another example is saving and restoring coprocessor registers.

The smx_TaskHook(atask, entry, exit) call is used to hook entry and exit routines into the scheduler for atask. These routines can be unique to atask or used by other tasks, also. The exit routine is automatically called by the scheduler, whenever the task is suspended. It can be used to save an extended task context or to perform exit functions. The entry routine is automatically called whenever the task is resumed. It can be used to restore an extended task context or to perform entry functions.

Hook routines operate as extensions of the smx scheduler, so care must be taken when writing them. They should be kept short since their execution times add directly to task switching times for hooked tasks. Do not call smx SSRs from hook routines.

If hook routines are written in assembly, or if inline assembly is used in a C hook routine, you must preserve non-volatile registers (those the compiler expects to be unchanged across a function call).

If a resume task operation is aborted after the entry routine has run, the exit routine will be run, even though the task, itself, did not run. Such an abort would occur, for example, if a higher priority task became ready to run while the entry routine was running and the task being resumed was unlocked. This process is called *resume flyback*.

saving coprocessor context

Floating point and other coprocessors have registers which must be saved when switching to another task that also uses the coprocessor. This can be done by hooking appropriate exit and entry routines into each such task. Then the coprocessor registers will be saved and restored automatically when each task is suspended and resumed, respectively.

On an x87 coprocessor, for example, the FSAVE and FRSTOR instructions are used to save and restore its registers. These can be called from exit and entry routines. They move 108 bytes to or from the stack, which nearly doubles task switching time. Note that this impacts not only switching to the coprocessor task, but also switching from it to possibly a high-priority, urgent task. Hence, if task switching time is critical, it is advisable to minimize the number of tasks using floating point. Note, however, that even a simple assign

```
double x;
x = 1.0;
```

may invoke the coprocessor.

Another way to deal with this is to bracket just the floating point operations,

```
smx_TaskHook(smx_ct, fp_entry, fp_exit);
/* floating point operations */
smx_TaskUnHook(smx_ct);
```

Then, the extra coprocessor overhead is encountered only in these areas of the task. If this is still unacceptable, use

```
smx_TaskLock();
/* floating point operations */
smx_TaskUnlock();
```

or a semaphore, and avoid saving the coprocessor registers entirely.

Cortex-M4 FPU

The Cortex-M4 presents a better solution, than the foregoing, called *automatic state saving.* If enabled, the FPCA bit in the CONTROL register is set if an FPU instruction is executed by the current task. In this case, when the current task is interrupted, additional stack space on its stack is allocated for FPU registers S0-15. Also bit 4 of EXC_RETURN is set to 0. (Thus, it is 0xFFFFFFED instead of 0xFFFFFFFD.) The last byte of EXC_RETURN is saved in the exret field of the TCB of the current task, by smx_PreSched (which is the smx PendSV_Handler() for Cortex-M processors). When this task is resumed, its exret field is used by smx_PreSched() to determine which size stack to unstack.

Cortex-M4 automatic state saving is coupled with its *lazy stacking mechanism*, which defers saving S0-15, unless the FPU is actually used by a preempting thread (ISR, LSR, or task). This is implemented wholly in hardware. If an ISR or LSR using the FPU is interrupted all operations are handled by hardware, since ISRs and LSRs run under the System Stack (the Main Stack for Cortex-M processors). Lazy stacking is not currently supported. See the ARM-M section of the SMX Target Guide for more information.

ideal task structure

Ideally, tasks should have entry, run, and exit sections, as shown in the following example:

```
#define ENTER     -1
#define EXIT      -2

void  taskMain(u32 par)
{
    switch (par)
    {
        case EXIT:
            taskExit();
            break;
        case ENTER:  /* enter, then run */
            taskEnter();
        default:
            while (1)     /* normal task */
            {
                /* run code */
            }
            -- OR --      /* one-shot task */
            /* run code */
    }
}

void     taskEnter(void)
{
    /* obtain and initialize all resources needed by task */
}

void     taskExit(void)
{
    /* release all resources used by task */
    smx_TaskDelete(self);
    -- OR --
```

```
          /* release smx resources released by delete */
          smx_TaskStop(self, INF)
      }
```

To start task from a dormant state following create or stop infinite:

```
smx_StartPar(task, ENTER);
```

To end a task:

```
smx_StartPar(task, EXIT);
```

Note that in the above, control passes from case ENTER directly into the run code, hence a separate RUN mode is not needed. For normal tasks, there is no problem assigning mode values. However, one-shot tasks use the par parameter to return results of stop functions. The latter are limited to u16 values or valid handles, so choosing large values, as shown, should be fine. A potential problem for both task types is from task main: return n followed by smx_Stop(self, tmo) or autostop. These can result in any value of n permitted by the application, so care is needed.

The taskEnter() function may initialize and power on I/O resources as well getting and initializing application and smx objects.

The taskExit() function frees all application resources and smx resources and may power off I/O resources, then either deletes task or puts it into a dormant state. It never returns, so the break is actually not necessary. smx_TaskDelete() releases smx resources owned by smx. The advantage of designing taskMain() this way is that an error recovery or power monitoring task does not need an exit function pointer table to shut down tasks — it uses StartPar(task, EXIT), as above.

This structure works nicely in a multi-programmer environment. Each programmer controls how his tasks are entered and exited and other programmers do not need to know the details. Other cases can be added such as for powering on and powering off, without full task shutdown and resumption. For example:

```
switch (par)
{
    case EXIT:
        taskPwrOff()
        taskExit();
        break;
    case PWROFF
        taskPwrOff();
        break;
    case ENTER:
        taskEnter();
    case PWRON:
        taskPwrOn();
}
```

Systems not needing error recovery or low power can use simpler task structures, as shown elsewhere in this manual.

third main function type

The third task main function supported by smx is as follows:

```
u32  task_main(u32 par)
```

This is used so that a task can return a value to itself, the next time it starts. When a task does a return it autostops and the return value is passed in as a parameter the next time the task starts. Hence, a task can return a value to itself, as shown in the following example:

```
 /* create and first start */
taskA = smx_TaskCreate((FUN_PTR)et7_taskA_main, P2, ESS, NO_FLAGS, "taskA");
smx_TaskStartPar(taskA, ENTER);

/* subsequent starts */
smx_TaskStart(taskA);

u32  taskA_main(u32 par)
{
    switch (par)
    {
        case EXIT:
            /* release resources owned by task here */
            break;
        case ENTER
            /* initialize task here */
            while (1)
            {
                /* do normal task function here */
                if (catastrophic_error)
                    return EXIT;
            }
    }
    return 0;
}
```

When first started, ENTER causes task initialization followed by the normal infinite loop where the task does its normal work. If a catastrophic error occurs, taskA returns EXIT, and autostops. When restarted by another task (e.g. error manager task), EXIT is passed into taskA_main(), which causes it to release owned resources and prepare to become dormant. Hence, the error manager, in this example, does not need to know how to shut down this task.

using what you have learned

This is a good time to stop and look at the Protosystem. Refer to the SMX Quick Start manual and on-disk documentation for details concerning installing, making, and running smx and the Protosystem

Using a debugger, go to ainit(), then step through the code and observe how tasks are created and started and how they run. To observe a task, break on its first statement, then step through it. If a task has an infinite loop, remember to break inside the loop next time since execution will not return to the top of the task main function unless it is restarted; you can break on the first statement only once.

Now, add your own code to app.c to experiment with the smx services discussed in this chapter such as creating new tasks, starting and stopping them, etc. smxAware is particularly helpful at this stage since it allows you to view the ready queue, TCBs, and other smx objects symbolically. The event timelines view in smxAware GAT is helpful to see the order in which tasks, LSRs, and ISRs run.

Note: Instead of altering the Protosystem files, we suggest that you copy them to a new, parallel subdirectory. This way, if you should have trouble, you can easily revert to the original Protosystem to verify that it does not have the same problem, then work forward to find what change caused the problem.

Chapter 15   One-Shot Tasks

introduction

One-shot tasks are useful to reduce RAM usage by sharing stacks between tasks. Since one-shot tasks are unique to smx and not commonly understood, this chapter has been created to describe them.

stack RAM usage

In general, every active task requires its own stack. There are two concerns with this (1) total RAM usage and (2) performance. For best performance, stacks should be in fast RAM. This is especially true if auto variables are used extensively to achieve reentrancy.

The System Stack frees task stacks of ISR, LSR, scheduler, and error manager stack requirements. However, depending upon coding style, project standards, and other factors, task stacks can still get quite large. For example, a project standard may be to make all subroutines reentrant in order to avoid accidental reentrancy problems. Or it might be necessary, for example, to make a floating point library or a graphical display library reentrant so they can be shared between tasks. Such libraries might require large arrays and large variables to be stored in each task's stack.

Reentrancy is usually achieved through exclusive use of auto variables and function parameters, which are stored on the current stack. Hence, a function of moderate complexity can require significant stack space (this is illustrated by common C library functions, such as printf()). Coupled with deep function nesting this can result in large stack requirements in the thousands of bytes. Even for moderate stacks of 500 bytes, a 20 task system requires 10K bytes. This could be a problem if trying to use only on-chip SRAM to reduce cost or to increase performance.

For these reasons, the tendency on the part of many programmers is to reduce the number of tasks below that which is optimum for the application. When that happens, many benefits of multitasking are lost because operations that could be handled by kernel services become internal operations within tasks. Not only must new code must be created and debugged to implement them, but also they are hidden from debug tools such as smxAware.

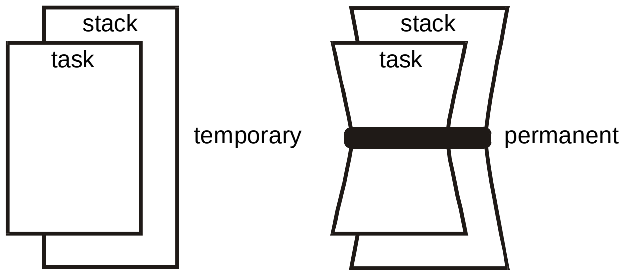
Deleting tasks while not needed is another way to reduce stack RAM usage. However, such tasks cannot wait for events. Thus, recreating such tasks and getting them restarted, when needed, is relatively complex. Also, when in the NULL state, they drop off the radar as far as the debugger and smxAware are concerned.

one-shot tasks offer a solution

One-shot tasks provide a way to reduce stack RAM usage. They offer a stack-sharing capability that is unique to smx. A *one-shot* task is a task that runs once, then stops. When it stops, it is done with its current run and has no information to carry over to its next run. Hence, it does not need a stack. smx permits such tasks to release their stacks while they wait for events.

One-shot tasks are created and started without a stack. A one-shot task is given a temporary stack from the stack pool, when it dispatched (i.e. actually starts running). When the task stops, it releases the stack back to the stack pool.

Previous discussion has centered on normal tasks, which have permanent stacks. These are bound to the tasks when they are created and remain bound until the tasks are deleted. Normal tasks are always bound to their stacks, even after being stopped:



While running, there is no operational difference between one-shot tasks and normal tasks. Both can be preempted and both can be suspended to wait for events. Both retain their stacks, in these situations. smx permits one-shot tasks to be stopped to wait for the same events as normal tasks. This is advantageous for one-shot tasks, because it enables them to give up their stacks. Thus, a one-shot task can, for example, stop at an exchange and wait for the next message, without consuming a stack. When a one-shot task is restarted and dispatched, it is given a new stack from the stack pool, and it starts from the beginning of its code. (This is why it is called a one-shot task.) There is very little performance penalty for starting a task with a new stack, versus resuming a task that already has a stack. (The time for getting a new stack is balanced by not needing to restore registers.)

stack pool size

In order to not impact operation, the stack pool should have as many stacks as the maximum number of one-shot tasks that might run at the same time. To minimize the size of the stack pool, one-shot tasks should generally stop when waiting, rather than suspend. However, it may be necessary to suspend if getting a resource midstream — i.e. before the one-shot task is done. This causes no harm, but may necessitate a larger stack pool.

For many systems, some tasks are mutually exclusive. Hence, only a subset of those tasks can run simultaneously. If there are enough stacks in the stack pool for the largest subset, then there never will be a shortage of stacks. In other systems, the probability of more than a certain number of tasks running simultaneously may be small and this number could be used as the stack pool size.

Should the stack pool run out of stacks, the smx scheduler will skip to the next ready task that can be run. Each time the scheduler is entered it will try again to run the stalled one-shot task (assuming that it is still the top task). Eventually, a stack will become available and the task will run. If RAM is at a premium and the foregoing performance hit is acceptable, this would be an acceptable way to design a system. As a result, substantial late-project recoding due to RAM overflow might be avoided with the only consequence being reduced idle time.

If we make a rule that one-shot tasks may not suspend themselves (i.e. they can wait only in the stopped state), then the number of stacks required in the stack pool is equal to the number of different priorities of one-shot tasks. For example, if there are 10 one-shot tasks having 3 different priorities, then only 3 stacks are required in the stack pool!

In the case where one-shot tasks must wait for stacks, they provide a method to automatically scale performance to available RAM by simply changing the number of stacks in the stack pool. So, in the above example, if the number of stacks were changed from 3 to 2, operation would be slower, but possibly acceptable for a smaller system.

applying one-shot tasks

One-shot tasks can be ideal for large functions, which seldom run and which require large stacks. Why tie up a large block of precious SRAM for a task that is idle most of the time? Examples are: infrequent operator input, infrequent downloads, I/O, background analyses, GUI processing, data encryption/decryption, data compression/decompression, error handling and recovery, report generation — hourly, daily, weekly, etc. (Of course, whether these kinds of tasks are infrequent or frequent depends upon the application.) The foregoing tasks are in the nature of server tasks. smx messaging may work well to pass them work to do and pass out results. Having done its job, the task has no history to remember and thus is a good one-shot candidate.

One-shot tasks are also good for mutually exclusive operations, which cannot happen simultaneously (e.g. a task that starts an engine versus a task that stops the engine) and they are good for sequential operations. It often is helpful to break complex operations into simpler steps. Implementing the steps as tasks results in easier to write and more flexible code. For example, a step might be skipped or replaced with an alternate step, depending upon results from a prior step. Using kernel task services makes this easy to do.

Tasks corresponding to steps, wait at different exchanges for messages containing the information for them to process. A step can be skipped simply by sending a message to its output exchange instead of its input exchange; an alternate step can be invoked by sending the message to its input exchange, rather than to the normal step's input exchange. Instead of internalizing this complexity inside of a single, complicated task, breaking it out into many tasks allows tools such as smxAware to be used to debug problems. In the future, new step tasks can be added to easily achieve different behaviors.

When created, a task can be specified to start locked. If never unlocked, the task will be non-preemptible for as long as it runs. Short, non-preemptible, one-shot tasks are useful for infrequent, short operations, such as sounding an alarm, opening an emergency valve, etc. Typically they are high-priority tasks. They can wait at semaphores, or other objects, do their thing when started, then they quit. This kind of operation might be done by LSRs, except that LSRs cannot wait. Although the amount of stack required by this kind of task is likely to be small, if there are several such tasks in a system RAM usage might add up to 1K byte or more. In combination with an event group, non-preemptible, one-shot tasks might be used to implement state machines.

writing one-shot tasks

One-shot tasks are created by passing a 0 stack size parameter to smx_TaskCreate(). They can be stopped, and therefore give up their stacks, by calling any Stop SSR. These are SSRs that have "Stop" in their names (e.g. smx_MsgReceiveStop()). smx_TaskStart() stops a task that has been previously started, then restarts it. A task can also be stopped by smx_TaskStop() or by auto stopping (i.e. running through the end of its main function). smx_TaskStop(task, tmo) restarts the task after the tmo timeout. smx_TaskStop(task, SMX_TMO_INF) and autostop stop the task permanently.

When a stopped task is restarted, the return value from the SSR that stopped it is passed in as the parameter to its main function. For example if the task called smx_SemTestStop(), then the parameter would be TRUE if the semaphore test passed and FAIL otherwise. Due to stopping then restarting, one-shot tasks require a different code structure than normal tasks:

```
void  atask_main(u32 par)
{
    /* process par */
    /* calculate new par */
    return (par);
}
```

The parameter is also useful for passing an initial value to atask via smx_TaskStartPar() or smxTaskStartNew().

Note: For a processor having separate data and address registers (e.g. ColdFire) and used with a compiler that passes parameters via registers (e.g. CodeWarrior), par will always be put into a data register. To counter this, typecast pointers (e.g. smx handles) to u32, then typecast them back to pointers within the task main function by assigning par to a local pointer of the correct type.

When using the one-shot task main function, it is necessary to use a FUN_PTR typecast in smx_TaskCreate():

```
atask = smx_TaskCreate((FUN_PTR)atask_main, PRI_NORM, 0, NO_FLAGS, "atask");
```

To start a task and pass a parameter to it, use:

```
smx_TaskStartPar (atask, par);
```

where par is a u32 value. To restart a task with a parameter, new main function and priority, use:

```
smx_TaskStartNew (atask, par, priority, new_main);
```

examples

A normal task is created and started and after initialization it goes into an infinite loop, as follows:

```
TCB_PTR   taskA;
XCB_PTR   xa;
void taskA_main(void);

void ug_main(void)
{
    taskA = smx_TaskCreate(taskA_main, P2, 500, NO_FLAGS, "taskA");
    smx_TaskStart(taskA);
}

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

    /* initialize taskA here */

    while (msg = smx_MsgReceive(xa, &dp, TMO))
    {
        /* process msg here using local dp */

        smx_MsgRel(msg, 0);
    }
    /* handle error or timeout  */
}
```

The comparable code for a one-shot task is as follows:

```
u8*  dp;
TCB_PTR  taskB;
void  taskB_init(void);
void  taskB_main(MCB_PTR msg);
void  ug_main(void)
{
     taskB = smx_TaskCreate(taskB_init, P2, 0, NO_FLAGS, "taskB");
     smx_TaskStart(taskB);
}

void   taskB_init(void)
{
     /* initialize taskB here */

     smx_ct->fun = (FUN_PTR)taskB_main;
     smx_MsgReceiveStop(xa, &dp, TMO);
}

void  taskB_main(MCB_PTR msg)
{
     if (msg)
     {
          /* process msg here using global dp */

          smx_MsgRel(msg, 0);
          smx_MsgReceiveStop(xa, &dp, TMO);
     }
     /* handle timeout, or error */
}
```

Note that taskB is created with no stack and function taskB_init(). This code runs the first time taskB runs and it initializes taskB (the same as is done before the while loop for taskA). When taskB initialization is complete, taskB->fun is changed to taskB_main, and taskB does a receive stop on xa. When taskB restarts, its parameter is a message handle, and it processes the message, releases it, and does a receive stop on xa to wait for the next message. If msg is NULL, a timeout or error has occurred, which is handled by the code after the if statement.

taskB does exactly the same thing as taskA, but it does not consume a task stack while waiting at xa. The structure of the two tasks is significantly different: (1) taskB requires a separate initialization function, (2) it has no internal while() loop, (3) it restarts every time a new message is received (or a timeout or error occurs), (4) the msg handle is passed in as a parameter, and (5) dp is a global pointer. Other than these, the code is the same.

A simpler way to write taskB is as follows:

```
#define INIT 1
void ug_main(void)
{
     taskB = smx_TaskCreate((FUN_PTR)taskB_main, PRI_NORM, 0, NO_FLAGS, "taskB");
     smx_TaskStartPar(taskB, INIT);
}

void  taskB_main(MCB_PTR msg)
{
     if  (msg)
     {
          if (msg == (MCB_PTR)INIT)
           {
               /* initialize taskB here */
          }
```

```
                    else
                    {
                        /* process msg here using global dp */
                        smx_MsgRel(msg, 0);
                    }
                    smx_MsgReceiveStop(xa, &dp, TMO);
                }
                /* handle timeout or error */
            }
```

For this case, the first time taskB runs, msg == INIT, so taskB is initialized and then wait stops on xa for the first message. This keeps initialization code with the task main function, like a normal task. The downside is that the extra msg test code must run every time taskB runs.

The following code creates a non-preemptible, one-shot task:

```
void ug_main(void)
{
    t2a = smx_TaskCreate((FUN_PTR)t2a_main, P2, 0, SMX_FL_LOCK, "t2a");
    t2b = smx_TaskCreate(t2b_main, P2, 0, NO_FLAGS, "t2b");
    t3a = smx_TaskCreate(t3a_main, P3, 0, NO_FLAGS, "t3a");
    xa = smx_MsgXchgCreate(NORM, "xa");
    smx_TaskStartPar(t2a, INIT);
}

void t2a_main(MCB_PTR msg)
{
    if (msg > 0)
    {
        smx_TaskStart(t2b);
        smx_TaskStart(t3a);
        smx_MsgReceiveStop(xa, &dp, NO_WAIT);
    }
}

void t2b_main(void)
{
}

void t3a_main(void)
{
}
```

Note that task create sets t2a->flags.strt_lockd. Thus, t2a always starts locked and it remains locked until it stops for the next message. Between smx_MsgReceiveStop() and restart, a task of higher priority can run, even though NO_WAIT is specified. The reason for this is that when t2a is stopped, the lock is broken. However, t2a retains its position in rq, thus an equal or lower priority task cannot run.

The above examples show stopping on an exchange. It is also possible to stop on a semaphore, an event queue, an event group, a pipe, or just stop. In the above example, t2a is stopped whether a message is available at xa, or not. If there is a message available, t2a will immediately restart. If a timeout were specified and no message were available, t2a would wait at xa until timeout ticks had passed. If a message were received before then, t2a would restart with msg passed as the parameter to t2a_main(), else msg = NULL.

I/O tasks

I/O tasks are a prime candidate for nonpreemptible one-shot tasks. Such tasks may be high- priority and very short. It often is the case that there are multiple channels of the same type (e.g. D/A converters or UARTs). In such a case, one task per channel may present the simplest implementation. The tasks may or may not all be of the same priority and they may or may not all share the same code, but they may share some globals or non-reentrant code. Making the tasks non-preemptible solves possible conflicts (even if some tasks have higher priority). By using one-shot tasks, only one stack is required for the group, yet one gains the advantages of multitasking, such as simpler code, greater flexibility, use of proven kernel services, and the ability to monitor operation via smxAware.

For input, LSRs may work well, but for output it usually works best to have tasks that wait at exchanges or pipes. Then outputs are naturally queued up, if the processor should fall behind. Message can have priorities so more important outputs will take precedence.

a further example

will help to illustrate another advantage of , one-shot tasks. Suppose that messages can appear at either mxchgA or mxchgB. Suppose further that message processing is exactly the same, that receiving messages at either exchange is random, and that messages must be processed very soon after arrival. One task cannot wait at two exchanges and the rapid-response requirement prohibits timer-driven polling of each exchange in rotation, by a single task. This is a natural application for two one-shot tasks:

```
taskA = smx_TaskCreate((FUN_PTR)taskA_main, PRI_NORM, 0, NO_FLAGS, "taskA");
taskB = smx_TaskCreate((FUN_PTR)taskB_main, PRI_NORM, 0, NO_FLAGS, "taskB");
mxchgA = smx_MsgXchgCreate(NORM, "mxchgA");
mxchgB = smx_MsgXchgCreate(NORM, "mxchgB");

void taskA_main(MCB_PTR msg)
{
    if (msg)
    {
        et13_process(msg);
        smx_MsgReceiveStop(mxchgA, NULL, TMO);
    }
    et13_exception();
}

void taskB_main (MCB_PTR msg)
{
    if (msg)
    {
        et13_process(msg);
        smx_MsgReceiveStop(mxchgB, NULL, TMO);
    }
    et13_exception();
}
```

Whichever exchange receives a message will immediately start its waiting task (i.e. taskA at mxchgA or taskB at mxchgB). Thus, the rapid-response requirement is met. Observe that the two tasks share most of their code — both processing and exception — and share a single stack. (Note that they are mutually exclusive because they have the same priority. Hence one stack is guaranteed to be sufficient.) As a consequence, having two tasks instead of one task incurs minimal extra overhead — only the little unique code shown above and a TCB.

This concept can be extended to multiple waits on any smx objects. For example, taskA could operate as above, whereas taskB could wait at semA and perform an appropriate action when it is signaled. Due to stack sharing between one-shot tasks and small TCBs, this is a practical, and possibly simpler way to implement multiple waits.

For further discussion of advantages and usage of one-shot tasks see the *One-Shot Tasks Reduce RAM Usage* white paper.

return to self example

This subject was briefly discussed under normal tasks. It presents an interesting way to write one-shot tasks, as shown in the following example:

```
void  t1a_main(void)
{
    t2a = smx_TaskCreate((FUN_PTR)t2a, P2, 200, NO_FLAGS, "t2a");
    smx_TaskStartPar(t2a, 0);
    smx_TaskStart(t2a);
    smx_TaskStart(t2a);
}

u32  t2a_main(u32 par)
{
    switch (par)
    {
        case 0:
            /* initialize task */
            return 1;
        case 1:
            /* perform operation 1 */
            return 2;
        case 2:
            /* perform operation 2 */
            return 1;
        default:
            return 0;
    }
}
```

In this example, task t1a creates the one-shot task t2a and starts it with par == 0 to initialize it. Task t2a preempts, initializes itself, passes par = 1 to itself, and stops. At some later time, t1a starts t2a. Then t2a restarts with par == 1, performs operation 1, passes par == 2 to itself, and stops. The next time t2a is started it will perform operation 2, pass par = 1 to itself, and stop. The above example illustrates how a one-shot task can pass information to itself. In this case, t2a is determining how it will next run, so par is effectively a mode or state.

Alternatively, t2a could be a jack-of-all-trades task which other tasks call upon to perform needed functions via smx_TaskStartPar() calls. In this case, t2a should always run locked so it cannot be restarted while it is running:

```
t2a = smx_TaskCreate((FUN_PTR)t2a, P2, 200, LOCK, "t2a");
```

Chapter 16 Service Routines

ISRs are first responders, LSRs helpers, and SSRs tools

introduction

Service routines are task-like objects which operate at a higher priority level than tasks. Service routine are distinct from subroutines, which are simply extensions of the code that calls them. Service routines have unique properties, which are discussed in this chapter.

There are three types of smx service routines:

(1)	SSR	system service routine
(2)	ISR	interrupt service routine
(3)	LSR	link service routine

system service routines (SSRs)

System service routines implement most smx calls. An SSR starts with smx_SSR_ENTERn() (n is the number of parameters) and it ends with smx_SSR_EXIT(). In between, operations are performed upon smx objects. SSRs are task-safe and LSR-safe. Unless you create SSRs of your own (see custom SSRs in this chapter), you will not be dealing directly with SSRs. Hence, it is necessary only to be aware of their existence.

smx_SSR_ENTERn(call_id, par1, par2, ... parn) logs the call into the event buffer, if SMX_CFG_EVB is set in xcfg.h, then increments smx_srnest. The main purpose of the smx_SSR_ENTER() and smx_SSR_EXIT() macros is to transfer control to the scheduler when the system service routine completes and to prevent interruption by another SSR or LSR.

CAUTION: SSRs enable interrupts**.**

interrupt service routines (ISRs)

Interrupt service routines handle interrupts. smx ISRs start with smx_ISR_ENTER() and end with smx_ISR_EXIT(). Interrupt handling code is put in between. If it is desired to log an ISR, smx_EVB_LOG_ISR() and smx_EVB_LOG_ISR_RET() can be placed inside of the enter and exit macros, respectively — see Event Logging for more information. ISRs have no control blocks and thus are identified by their starting addresses. However, they can be assigned names in the handle table for use by smxAware with smx_SysPseudoHandleCreate().

Interrupt handling is discussed in more detail, below.

link service routines (LSRs)

Link service routines are normally invoked by ISRs to perform deferred interrupt processing and to call SSRs in order to interact with tasks. smx_EVB_LOG_LSR() and smx_EVB_LOG_LSR_RET() can be placed at the start and end of LSR code to log its start and end. LSRs have no control blocks and thus are identified by their starting addresses. However, they can be assigned names in the handle table for use by smxAware with smx_SysPseudoHandleCreate().

rules of behavior

Like any happy family, service routines have rules of behavior:

(1) ISRs may not call SSRs, but can invoke LSRs using smx_LSR_INVOKE(), to do so.

(2) LSRs must be invoked, not called directly.

(3) LSRs must allow SSRs to finish.

(4) LSRs do not preempt other LSRs.

(5) LSRs can call SSRs, but cannot wait.

These rules comprise the backbone of how smx works. There benefits are as follows:

(1) SSRs run with interrupts enabled.

(2) LSRs run with interrupts enabled, unless disabled by the user.

(3) Very low interrupt latency — interrupts are disabled by smx only in small sections of the scheduler and a few other places.

(4) LSRs handle deferred interrupt processing, with interrupts enabled, thus allowing ISRs to be short.

(5) Multiple LSRs can be invoked, each multiple times, if necessary, with different parameters to handle heavy interrupt loads and maintain temporal integrity. This is limited only by the size of the LSR queue.

(6) LSRs can operate as top-priority, non-blockable tasks — i.e. they are not subject to priority inversions.

(7) LSRs can do everything tasks can do except wait and call limited SSRs.

background vs. foreground

The term *foreground* is used here to mean interrupt handling code and the term *background* is used to mean non-interrupt handling code. In non-real-time systems, the term foreground may mean operator interface code and background all else (which could include interrupt handling).

The previous chapters are concerned primarily with background processing. Tasks are strictly background objects — switching from one task to another is generally too slow for foreground servicing. Objects such as messages, blocks, exchanges, etc. are used in the foreground as well as in the background.

smx provides a powerful, two-level structuring of foreground via interrupt service routines (ISRs) and link service routines (LSRs).

two ISR types

smx ISRs are ISRs that invoke LSRs. Such ISRs must be preceded with an smx_ISR_ENTER() macro and followed with an smx_ISR_EXIT() macro. The former increments smx_nest. The latter decrements smx_nest and passes control to the scheduler when smx_nest becomes 0. This allows smx ISRs to be nested.

At some point, an smx ISR will invoke an LSR using smx_LSR_INVOKE(lsr, par). This results in smx_ISR_EXIT() causing the LSR scheduler to run, which runs the lsr and passes par to it. lsr may call one or more SSRs, which result in scheduler flags being set. When all LSRs have run, control passes to the task scheduler, which acts upon the scheduler flags.

non-smx ISRs are also permitted. These do not use the above macros and therefore do not directly link to the background. They generally perform some quick foreground function. Such ISRs must meet one of two criteria:

(1) Higher priority than all smx ISRs.

(2) Never enable interrupts.

If neither criterion is met, then a non-smx ISR may be interrupted by an smx ISR. If the latter invoked an LSR, control will go to the scheduler at the end of it and the non-smx ISR may be left dangling for an unpredictable period. (This is because it is using the current task's stack and if the scheduler switches tasks, then another stack will come into use, which has no record of the non-smx ISR).

CAUTION: smx ISRs must be inhibited during initialization since the LSR queue has not been created nor initialized. It is best to wait until ainit().

interrupt handling

There are two processor structures for interrupt handling:

(1) All interrupts are vectored to one ISR.

(2) Each interrupt is vectored to its own ISR.

For processors of the first type, smx_irq_handler(), in the processor-tool assembly module (e.g. xarm_iar.s) is linked into the interrupt. It, in turn, calls sb_IRQDispatcher(), which is in the BSP of the specific processor. sb_IRQDispatcher() performs software interrupt vectoring to the ISR for each interrupt. The call to sb_IRQDispatcher() is embedded between smx_ISR_ENTER() and smx_ISR_EXIT() assembly macros so that they do not need to be included in the ISRs.

Most processors are of the second type. There are two processor variants of this type:

(1) Those that enable interrupts after the first machine instruction.

(2) Those that do not.

ColdFire 5208 is an example of the first variant. For this kind of processor, it is necessary to create an assembly shell, which is linked to the interrupt, instead of the ISR. It disables interrupts in its first instruction, then calls the ISR. Assembly shells are put in the processor-tool isrshells assembly module (e.g. isrshells_cw.s). For example, _smx_TickISRShell() handles the tick interrupt and calls smx_TickISR(). Assembly shells call smx_ISR_ENTER() and smx_ISR_EXIT() macros before and after calling the ISR, so it is not necessary to include these macros in the ISRs.

For the second processor variant, ISRs are directly linked to interrupts. In this case, ISRs must start with smx_ISR_ENTER() and end with smx_ISR_EXIT() macros.

In all of the above cases, ISRs can be written in assembly language or in C. It is generally possible to obtain better performance with assembly, but C is easier to use and usually adequate. ISRs written assembly language must use the assembly versions of smx_ISR_ENTER() and smx_ISR_EXIT(). These are located in the processor include file (e.g. xcf.inc). C ISRs must use the C versions, which are located in the processor header file (e.g. xarmm.h). Assembly shells, of course, must also use assembly versions.

If this whirlwind tour of the interrupt world has left your head spinning, please see the SMX RTOS Target Guide for your specific processor architecture and tools.

smx ISR and SS operation

As noted above, the structure of smx ISRs depends strongly upon the processor being used. In general, the ISR code must be embedded between smx_ISR_ENTER() and smx_ISR_EXIT() macros. In addition to incrementing smx_nest, smx_ISR_ENTER() saves volatile registers and switches to the system stack (SS), if in a task stack, unless the processor already has already done these (e.g. Cortex-M3). (The volatile registers are those that the compiler does not expect to be preserved by a function call — e.g. r0-3 and r12 for ARM).

If the ISR is nested (smx_srnest > 1) smx_ISR_EXIT() decrements smx_srnest, pops the registers and returns to the interrupted ISR. Otherwise, if no LSR has been invoked, it sets smx_srnest = 0, switches to the task stack, pops the volatile registers, and returns to the interrupted task. If an LSR has been invoked, it decrements smx_srnest and calls the scheduler.

Operation is similar if the scheduler, an LSR, or the error manager was interrupted. In these cases, smx_srnest > 1 and control returns to the point of interrupt, after decrementing smx_srnest and popping the volatile registers.

Note in the foregoing, that all smx ISRs, LSRs, and the scheduler run in SS. For processors which do not automatically switch into SS when interrupts occur, the code is complex. Since interrupts may occur when already in SS, the code must check that SS is not already being used before switching to it. Also the error manager, smx_EM, runs in SS, so switching to SS is necessary when an error occurs. A combined flag and counter, smx_ssnest, is used to handle deciding when to switch into or out of SS.

For processors that do not automatically switch to SS, non-smx ISRs run in the current task's stack. For this reason, they should use minimal stack, and it is best if they do not nest.

writing smx ISRs

If an ISR is written in C, the interrupt keyword or pragma, provided by the C compiler, is not necessary because smx_ISR_ENTER() saves the volatile registers and, if a task switch is made, the smx scheduler saves all registers in the register save area (RSA). Since the ISR is a C function, the compiler will save any non-volatile registers used. Hence, nothing special needs to be done. The ISR can be defined as follows:

```
void xxxISR(void)
{
    smx_ISR_ENTER();   /* if needed */
    ...
    smx_LSRInvoke(lsr, par);
    ...
    smx_ISR_EXIT(); /* if needed */
}
```

Note that the interrupt keyword is not used.

ISRs can be written in assembly language to theoretically achieve better performance and to more easily control hardware. If doing so, it is not necessary to save the volatile registers, since smx_ISR_ENTER() does that, but any additional registers used must be saved and restored by the assembly ISR. Also, smx_ISR_EXIT replaces the interrupt return instruction. Currently, we do not provide smx_LSR_INVOKE() in assembly language for any but the ColdFire processor, so it would need to be created.

In addition to the above macros, ISRs may use only bare functions such as: smx_PipeGetPkt() and smx_PipePutPkt(), including the 8 and M variants, and sb_BlockGet() and sb_BlockRel(). No SSRs may be called from ISRs.

more on link service routines

LSRs serve four purposes:

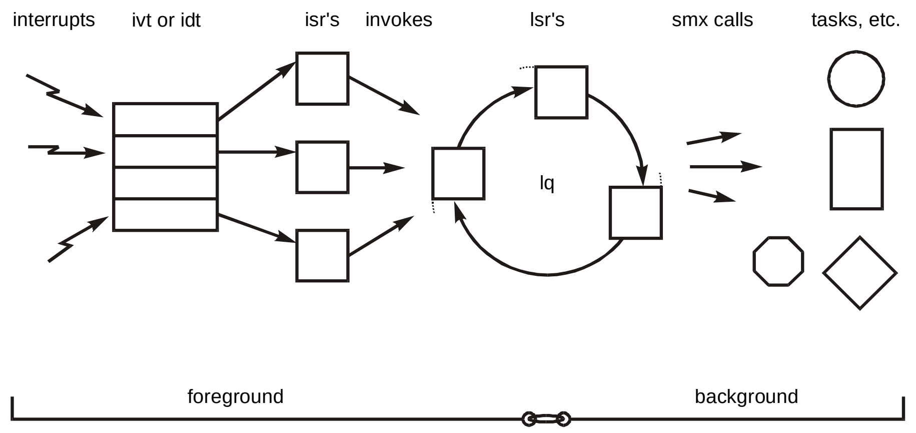
(1) Minimize interrupt latency.

(2) Deferred interrupt processing.

(3) High-priority, non-preemptible processing.

(4) Timer processing.

Conceptually, LSRs fit in as follows:



Link service routines are so named because they link the foreground to the background. They can be invoked from ISRs, as follows:

```
smx_LSR_INVOKE (send_LSR, par);
```

The address of send_LSR is placed at the end of the LSR queue (lq) and the parameter, par, is stored after it. When actually run, par is passed to the LSR via a register or the current stack, as would be done for a normal function call parameter.

An LSR can also be invoked from a task or another LSR:

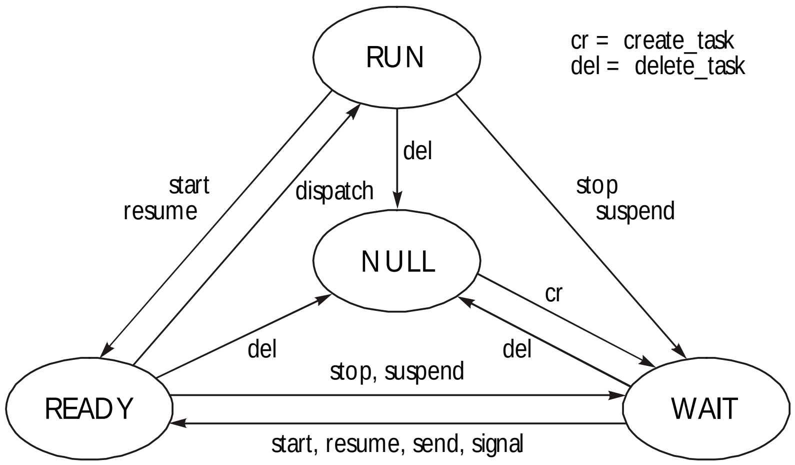
```
smx_LSRInvoke (send_LSR, par);
```

This can be useful to emulate an interrupt, or break a long LSR execution into pieces so that other LSRs can run. In addition, LSRs may be invoked by tasks and LSRs to avoid resource conflicts. This is discussed in the Resource Management chapter.

smx calls from LSRs

Whereas ISRs can perform very few smx calls, LSRs can perform most smx calls (all but limited SSRs). However, they have no priority, they are scheduled in FIFO order, and they cannot wait at exchanges, semaphores, etc. LSRs are designed deliberately for speed so that they can be used in the foreground. In practice, the fact that LSRs have no priority and execute in FIFO order has very little system impact because all LSRs execute ahead of all tasks. Where foreground priority is a concern, more processing can be done in the ISR, if necessary.

The following state diagram shows the transitions which can be caused by representative system services called from an LSR:



To help interpret the above diagram, remember that the task in the run state is the current task. By comparing this diagram to that for system services (see task states in the Tasks chapter), it is apparent that the only restriction on LSR calls is in causing the current task to leave the run state. This can be done by LSRs only through smx_TaskResume(), smx_TaskStart(), smx_TaskStop(), or smx_TaskSuspend(). Indeed, an LSR is like a foreground task — it can call most system services, and it has higher priority than any background task. Much of the discussion in previous chapters applies to LSRs as well as to tasks. During the design process, some tasks may end up as LSRs for performance or other reasons.

Because LSRs operate in the foreground of the current task, they are not permitted to wait. If an LSR were to wait at an exchange, what would actually happen is that the LSR and current task would be suspended on the exchange together. Since the current task could be any task, the highest priority task in the system could unexpectedly stop running.

To avoid this problem, if an LSR makes a call with a non-zero timeout and the call cannot be immediately satisfied, an SMXE_WAIT_NOT_ALLOWED error occurs, and the call fails. Consider the following example:

```
SCB_PTR  done;
XCB_PTR  data_out;

void  send_LSR(u32 arg)
{
    MCB_PTR  msg;
    u8 *mbp;

    if (msg = smx_MsgReceive (data_out, &mbp, tmo))
        /* send msg */
    else
        smx_SemSignal(done);
}
```

In this example, if no message were available at the exchange, a wait not allowed error would be reported and msg would be NULL. It is best to use NO_WAIT timeouts in smx calls from LSRs, which would avoid the error report, in this example. Then only the done signal would occur, allowing the problem to be handled without an error report.

Also, LSRs are not allowed to make limited SSR calls. These are calls which either stop or always suspend the current task. Making a limited call from an LSR causes the SMXE_OP_NOT_ALLOWED error, and the call is aborted. This is considered to be a programming error. For example:

```
void  receive_LSR(MCB_PTR msg)
{
    /* process msg */
    msg = smx_MsgReceiveStop(data_out, &mbp, NO_WAIT);
}
```

This is an error because smx_MsgReceiveStop() would stop the current task, then restart it when a message was available. Unexpectedly stopping the current task is obviously not correct. Hence, smx stop SSRs are limited to tasks.

Exceptions to the above rule are smx_TaskStop(), smx_TaskStart(), smx_TaskStartPar(), smx_TaskStartNew(), smx_TaskSuspend(), smx_TaskResume(), and smx_TaskBump(). These are allowed from LSRs, and are not classified as limited calls, because they are task specific. It makes sense, for example, that an LSR might resume or restart a task.

CAUTIONS:

(1) Unlike calling these functions from a task, the task switch does not take effect immediately, and the LSR continues to run. This must be taken into account in the code following the call. For safety, it is recommended that these calls be made at the ends of LSRs.

(2) Do not use smx_ct in LSRs, because it may have been changed by a preceding SSR called by the current task or by this or another LSR that ran before. Although the actual task switch does not occur until all LSRs in lq have run, SSRs may change smx_ct.

example

```
typedef  enum {INIT, START, STOP}  mt;

tmode = smx_TaskCreate((FUN_PTR)tmode_main, P2, 0, SMX_FL_LOCK, "taskA");

void modeISR(void)
{
    mt  mode;

    smx_ISR_ENTER();
    mode = read_mode_switch();
    smx_LSR_INVOKE(modeLSR, (u32)mode);
    smx_ISR_EXIT();
}

void modeLSR(u32 par)
{
    smx_TaskStartPar(tmode, par);
}

void  tmode_main(u32 par)
{
    switch ((mt)par)
    {
        case INIT:
            sys_init();
            break;
        case START:
            sys_start();
            break;
            case STOP:
                sys_stop();
                break;
            default:
                sys_alert(MODE_ERROR);
                break;
        }
    }
```

This example assumes that changing a mode switch on a control panel generates an interrupt to modeISR(), which reads the mode switch and passes the reading on to modeLSR(), which starts the tmode task and passes the mode on to it. tmode is a one-shot task, which performs system initialization, system start, system stop, or alerts the operator that a mode error has occurred. This is a good use for a one-shot task — it obviously runs very seldom, it performs a simple function, and then it releases its stack and becomes dormant. Note that tmode runs locked, so that no other task can run while it is changing the system mode.

tips on writing ISRs and LSRs

(1) ISRs should be short and do only the minimum to keep the system running. They should invoke LSRs to do any work that can be deferred and to call smx services. This helps to avoid missed interrupts.

(2) Do not do end of interrupt handling in LSRs. Remember that an LSR does not run immediately after the ISR which invokes it — other ISRs and LSRs can run in between. All I/O necessary to re-enable the hardware to generate the next interrupt must be done in the ISR.

(3) Although smx permits nested interrupts, if you can keep an ISR very short, it may be best to leave interrupts disabled until the end. This helps to avoid access and other problems and simplifies debug.

(4) The LSR queue should be treated as a work queue. Some systems allow a hundred or more LSR requests to pile up during periods of peak interrupt activity. The same LSR can be invoked over and over, usually with different parameters. When the burst of activity is over, LSRs will run in the order they were invoked, thus maintaining what we call "temporal integrity". The importance of this is that system may become sluggish, but it will not break. (We like to think that this might enable control to be maintained through a period of chaos.)

(5) Keep LSR code with the ISR code which invokes it, if possible. After all, these really are two pieces of the same handler — one immediate and one deferred.

(6) Do not use smx_ct in LSRs. It is unreliable at the LSR level since smx_ct may be changing at the time the LSR runs. It is safe to use smx_ct if the smx global variable smx_sched == 0, but generally smx_ct should be used only by tasks.

(7) LSRs written in assembly language must preserve non-volatile registers (those the compiler expects to be unchanged across a function call).

(8) LSR functions must accept one u32 argument, which is the par argument in smx_LSRInvoke() or smx_LSR_INVOKE (). It need not be used.

(9) For a processor having separate data and address registers that is used with a compiler that passes parameters via registers (e.g. ColdFire with CodeWarrior), the par in invoke will be put into a data register since it is defined as a u32. To pass a

pointer or handle, typecast it to u32 in the invoke, then assign it to a pointer of the correct type in the LSR, as follows:

```
void LSR(u32 par)
{
    TCB_PTR task = (TCB_PTR)par;
    smx_TaskStart(task);
}
```

Advanced Topics

how do LSRs work?

If an SSR is interrupted and an LSR is invoked by the ISR, the LSR will not run until:

(1) The ISR and all nested ISRs have completed.

(2) Any SSR that was interrupted has completed.

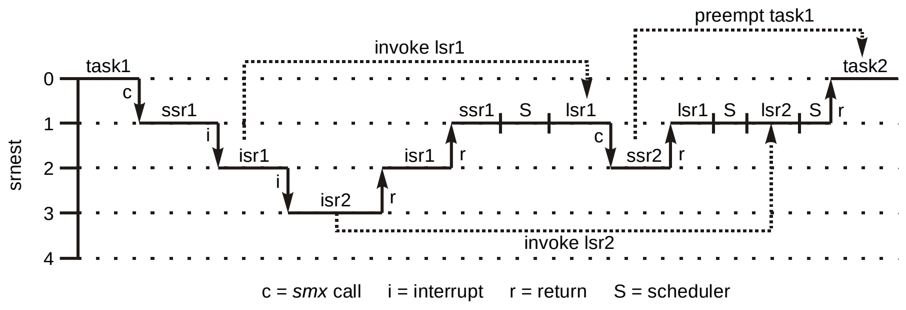
(3) Any LSR that was interrupted has completed.

(4) The scheduler, if interrupted, has completed.

(5) All LSRs ahead of it in the LSR queue have run.

LSRs cannot become nested; nor can they cause SSRs to become nested. As a result, it is not necessary to disable interrupts while an LSR or an SSR is running. LSRs cannot cause access conflicts for smx objects. This is a key aspect of the design of smx.

These points are illustrated in the following diagram. smx_srnest is the service routine nesting level:



At time 0, task1 is running. Tasks run at nesting level 0. task1 makes an smx call which causes ssr1 to run at level 1. Before ssr1 can complete, it is interrupted by isr1, which runs a level 2 and invokes lsr1. Notice that lsr1 does not begin running immediately. Instead, it waits in the LSR queue, lq. isr1, itself, is then interrupted by isr2, which runs at level 3 and invokes lsr2. lsr2 is placed in lq after lsr1. isr2 completes and returns to isr1. isr1 completes and returns to ssr1. ssr1 completes and passes control to the scheduler (S). Note that, at this, point smx_srnest == 1.

The scheduler dispatches lsr1, which makes an smx call resulting in ssr2 running. ssr2 causes a higher priority task, task2 to become ready. This task will preempt the current task, task1. ssr2 and lsr1 complete. Then the scheduler then dispatches lsr2. Finally, the LSR queue is empty and the scheduler checks the ready queue. It finds that task2 is now the top task. Assuming that task1 is unlocked, it is preempted and task2 is dispatched.

The scheduler always operates at level 1 in case it is interrupted. This forces the interrupting ISR to return to the point of interrupt in the scheduler rather than reentering the scheduler, which, of course is not correct.

Careful examination of the above figure reveals that neither SSRs[5] nor LSRs are nested at any time (i.e., there is no time when two SSRs are running simultaneously nor when two LSRs are running simultaneously). There is, however, no restriction upon nesting ISRs, as is shown in the figure.

Not shown in this diagram are stack switches. A switch is made to SS when isr1 starts running. A switch is made back to task stack1 (TS1) when ssr1 resumes running. A switch is made to SS when S starts running and continues until a switch to TS2 is made prior to task2 running. As a consequence, only SSR stack usage adds to task stack requirements. For more information see choosing stack sizes in Coding.

Although LSRs cannot cause access conflicts for smx objects, there can be access conflicts between LSRs and the current task for other system resources such as global variables or peripherals. See foreground/background access conflicts in the Resource Management.

custom SSRs

To convert a function to a system service routine (SSR), start it with smx_SSR_ENTER() and end it with smx_SSR_EXIT(). For example:

```
ret_type  NewService(par1, ... parn)
{
    smx_SSR_ENTERn(NewServiceID, par1, ... parn);
    /* new service */
    return (smx_SSR_EXIT(ret_val));
}
```

smx_SSR_ENTERn() is a macro that increments smx_srnest. If logging is enabled (SMX_CFG_EVB == TRUE), it also causes the NewServiceID and its i parameters to be logged into the event buffer, EVB, whenever NewService() runs. NewServiceID can be defined in the Function ID list in xdef.h. This allows smxAware to recognize NewService and to show it when it occurs. Alternatively, you can simply ignore this feature, by using smx_SSR_ENTER0(0).

The maximum size of the return type from an SSR is sizeof(int). If it is necessary to return a larger data type or additional values, pass return variable(s) by reference. For example, a double-precision floating point value could be returned through a parameter, like this (parameters 1 through n eliminated for simplicity):

```
NewService (double *dptr);
```

Assigning the return value to a variable in the same statement that makes the function call,

```
dvar = NewService();
```

can be achieved by making NewService() a shell function that calls the SSR and returns the value the SSR passed back. The SSR is given a slightly different name:

[5] It is possible for one SSR to call another. We are getting away from this, but it still occurs in a few places in smx and could occur in custom SSRs. It obviously is necessary to insure that access conflicts do not occur between the callee and the caller.

```
void  NewServiceSSR(double * dptr)
{
    smx_SSR_ENTER(SSR_ID, dptr)
    /* Code here to load value into location pointed to by dptr. */
    smx_SSR_EXIT(0);
}

double  NewService(void)
{
    double  dval;

    NewServiceSSR(&dval);
    return(dval);
}
```

It is recommended that you create custom SSRs rather than modifying smx SSRs, because the latter may be modified in future smx releases, thus introducing errors into your code. This can be done by starting with the smx SSR, giving it a different name, then modifying it as you wish. Custom SSRs can also be used to add new functionality to smx. Of course, the advantage of using SSRs instead of functions is that they are task- and LSR-safe.

Chapter 17 Timing

Timing, of one sort or another, plays a vital role in most real-time systems. smx supports a variety of timing requirements.

etime and stime

Within smx, two times are maintained:

(1) etime	elapsed time
(2) stime	system time

etime (smx_etime) is the elapsed time, in ticks, since the last cold start. It is a 31-bit counter. The 32nd bit, when set, is used to trigger etime rollover, which handles rolling over etime and all task timeouts from 0x80000000 + n to n. For a 100 Hz tick rate, etime has a range of 248 days. etime is used for timeouts.

stime (smx_stime) is the 32-bit elapsed time, in seconds, since either the last cold start or since some time origin such as 00:00:00 1/1/70 GMT. It is a 32-bit counter and has a range of over 100 years. stime is used by smx_TaskSleep() and smx_TaskSleepStop(); it may also be used for time stamping files in the file system or for similar purposes.

stime is initialized by sb_StimeSet(), called from ainit(). As delivered, stime is initialized to 0. To initialize it to a standard time origin, implement sb_StimeSet() for your target. It is common for operating systems to maintain a 32-bit seconds counter, so it is likely that your C run-time library (RTL) has a function that will convert a date and time structure into a value that can be stored in this counter and vice versa. The C library mktime() and localtime() functions are examples. Since time functions typically consider the time zone and daylight savings time, there may be some dependence of your C RTL on a particular operating system's functions to return these values. You may need to modify or emulate such routines.

If your system has a hardware clock, we recommend using it to timestamp files and to set stime.

tick rate

All timing in smx depends upon a periodic tick interrupt generated by a hardware counter or timer. The tick rate dictates the minimum resolution of timeouts in smx. It is specified by SB_TICKS_PER_SEC in bsp.h. sb_TickInit() initializes the hardware timer to this rate. Typical tick rates vary from 10 to 1000 Hz, depending upon system timing requirements.

Since it is usually more meaningful in a real-time system to specify time in milliseconds or seconds rather than ticks, conversion macros are provided in xapi.h. For example, smx_ConvMsecToTicks() converts milliseconds to ticks, rounding up to the nearest tick. See xapi.h for others.

timeouts

All SSRs which put a task into the wait state permit a timeout to be specified. When such a call is executed, the task's timeout is loaded with:

```
smx_timeout[tn] = smx_etime + timeout;
```

where tn is the task index = tcb.indx. timeout has a maximum value of (2exp31-1). The task timeouts in smx_timeout[] are in the same order as TCBs.

smx_TimeoutLSR() compares the smallest task timeout to etime. If it is less than or equal to etime, its task is resumed or restarted. TimeoutLSR then searches the timeout array to find the next smallest timeout and re-invokes itself to allow other LSRs to run. It continues this process until the smallest timeout is greater than etime. Unless there is a very heavy load of other LSRs, smx_TimeoutLSR() can handle many tasks timing out together, within a tick time.

smx_TimeoutLSR() is invoked from smx_KeepTimeLSR() every TIMEOUT_PERIOD ticks, which is defined in acfg.h. This may be as often as once per tick to achieve accurate timeouts, or it may be many ticks, if timeouts are used solely for safety.

Timeouts are intended primarily as a safety feature to assure that tasks do not permanently stop running. For reliable operation, it is best to avoid using SMX_TMO_INF for timeouts. When a timeout does occur, all SSRs are designed to return 0 or FALSE. The return value should be tested and appropriate action taken.

As noted above, it is also possible to use the timeout mechanism for accurate (within a specified number of ticks) timing. This can be useful in protocols and control operations and often makes code simpler. In this case, a timeout would spur a retry or other normal operation. For this case, TIMEOUT_PERIOD would probably be set to one or a few ticks. That, of course, applies to all timeouts — safety and accurate. However, this should not be a problem because the smx timeout mechanism is very efficient. For example, in a system with 40 tasks running on a 20 MHz processor with a 100 Hz tick rate, the total timeout overhead is < 0.1% for an average of 4 timeouts per second and TIMEOUT_PERIOD == 1.

NOTE: In most examples in this and other smx manuals, timeouts are ignored (i.e. set to SMX_TMO_INF). This is done to simplify the examples. It should not be taken as good coding practice.

special timeout values

In some cases, there is no reasonable upper bound for a timeout. Or, perhaps other tasks will detect the problem first, thus it is unnecessary for this task to have a timeout. An infinite timeout is specified by SMX_TMO_INF. For example:

```
msg = smx_MsgReceive (port1, SMX_TMO_INF);
```

This causes the timeout for ct to be made inactive, even though the task is waiting for a message at port1. Inactive and SMX_TMO_INF both equal 0xFFFFFFFF.

There are times when no wait is appropriate. For these cases use:

```
msg = smx_MsgReceive (port1, SMX_TMO_NOWAIT);
```

It creates what is called a non-blocking call. Non-blocking calls are used where there is other useful work for a task to do, while it waits, or from LSRs. Tasks can use non-blocking calls to poll for an event. Note: polling is normally better implemented with timers — see the Timers chapter. Another important usage is when it should not be necessary to wait, such as when getting a message that is supposed to be waiting at an exchange. In such a case the smx_MsgReceive() should fail so that a corrective action can be taken, such as boosting the priority of the task creating messages.

A third special timeout value is SMX_TMO_NOCHG (0xFFFFFFFE) which is used when it is not desired to change a task's timeout. This is used only in smx_TaskBump() and smx_TaskStartNew().

handling timeouts

All smx SSRs return 0 if a timeout or an error has occurred. Hence, you can test for a timeout and take corrective action as in the following example:

```
if (msg = smx_MsgReceive (port1, &dp, tmo))
    /* process msg */
else
    if (SMX_ERR == SMXE_TMO)
        /* handle timeout */
    else
        /* handle SMX_ERR */
```

Usually timeouts must be dealt with at the point of call, whereas errors may be delt with centrally by the error manager, smx_EM. See Error Management chapter.

If a timeout is being used for an accurate delay, then it is probably the expected result. For example:

```
set_speed();
smx_TaskSuspend(self, 50);
test_speed();
```

could be used to wait 50 ticks after setting a speed before testing it. Note: This assumes that TIMEOUT_PERIOD in acfg.h has been set to 1.

time delay options

A wide range of delays are needed by embedded systems:

(1) very fast / precise — microseconds

(2) fast / precise — milliseconds

(3) medium / accurate — ticks

(4) slow / accurate — many ticks

(5) date-time / non-accurate — seconds

very fast delays

A very fast delays usually require using a hardware timer that is set to an initial count, then count down to 0 and interrupt. Depending upon the processor the timer clock rate can vary from one to many processor clocks. This has the advantage of providing a precise delay with low overhead.

smx implements ptime (precise time), via sb_PtimeGet(), which takes readings from the tick counter and offers speed and precision equivalent to the foregoing. This can be used with sb_DelayUsec(usecs) for very fast delays. The advantage vs. the hardware timer approach is that it does not require another hardware timer; the disadvantage is that nothing else can run during a delay, except ISRs. See the smxBase manual for usage information.

fast delays

A fast delay can be implemented with a software timer, which is driven by a 1000 Hz interrupt. Because smx imposes very low interrupt latency, it is reasonable to handle interrupt rates of 1000 Hz, or more, on moderate speed processors. In this case, a software counter would be loaded with the desired delay and would be decremented on every interrupt. When it reached 0, an LSR would be invoked to resume or restart a task or to perform a timeout function directly.

The 1000 Hz interrupt, of course, requires an additional hardware timer, unless ticks are derived from it (e.g. call smx_TickISR() every 10th interrupt). However, it can drive many software timers.

medium delays

Ticks rates are normally about 100 Hz. For fast processors, they can be set higher.

Timers are the primary means for generating medium-speed delays, accurate to a tick. See the next chapter, for discussion.

smx_TicksEQ provides a simpler method for tasks to delay themselves. It also permits timeouts, which are accurate to a tick. See accurate timeouts in Event Queues.

smx timeouts can also provide tick-accurate timing, if TIMEOUT_PERIOD is set to 1 in acfg.h. This may be useful in systems where most tasks require accurate timeouts and there are only a moderate number of tasks. However, as discussed above, using timeouts may result in excessive overhead for very large numbers of tasks. In such systems, if only a small number of tasks require accurate timeouts, it may be preferable to use smx_TicksEQ.

NOTE: Using timeouts for accurate medium task delays is attractive because it is simple and natural. Unfortunately a project may start using timeouts, then end up with significant timeout overhead as the number of tasks increases. The reason for this is that each time a timeout occurs, the smx_TimeoutLSR() must search the entire timeout[] array for the next minimum timeout. If this becomes a problem, it is possible to implement a more efficient search algorithm, or use assembly language.

slow delays

As discussed previously, task timeouts are provided primarily for safety, and as such, usually have coarse resolution (e.g. 10’s of ticks). In that case, task timeouts provide slow, non-accurate time delays. A task can delay another task by using the smx_TaskStop() or smx_TaskSuspend() calls:

```
smx_TaskStop(atask, tmo);
smx_TaskSuspend(atask, tmo);
```

A task can delay itself with either of these or any other call having a timeout parameter.

CAUTION on ticks: Never use 1 tick for a delay. If near the end of the current tick period, this could result in an actual delay of 0. In general, specifying n ticks delay will produce an actual delay of (n - 1) to n ticks.

date-time delays

smx_TaskSleep() and smx_TaskSleepStop() permit delaying a task until a specified date and time. Precision is one second. Accuracy is the same as for timeouts. Date and time can be up to 248 days into the future (the same as for timeouts), for a 100-Hz tick rate. The current task may put itself to sleep until a specified time as follows:

```
void  hourly_main(void)
{
    u32  next_hour;

    /* perform hourly function */
    next_hour = ((smx_SysStimeGet()/3600)+1)*3600;
    smx_TaskSleepStop (next_hour);
}
```

hourly might be a low priority task, and some ticks might go by before it runs. The method of computing next_hour assures that there is no cumulative error.

smx_TaskSleep() or smx_TaskSleepStop() merely computes an equivalent etime and loads that value into smx_timeout[tn]. Hence, a sleep is actually a timeout, and the accuracy of sleeps is the same as the accuracy of timeouts. Normally, this is good enough. In fact, for sleeps, usually the most important thing is that there be no cumulative timing error, which can be assured by careful programming, as shown above.

Advanced Topics

using a hardware timer for a fast timeout

Task timeouts can be precise to one tick. However overhead can be fairly high if there are many tasks. It is possible to use a dedicated hardware timer to gain more resolution for tasks that need it.

Suppose that the tick time is 10 msec, and it is desired to have a 0.1 ms resolution for atask. To do so, it is possible to use a hardware timer which generates an interrupt when it times out. The resulting ISR/LSR can simply resume the task — the waiting call will return with 0, indicating a timeout. The following is an example for testing a semaphore using a timeout via a hardware timer:

```
void  atask_main(void)
{
    //...
    smx_TaskLock();
    start_hdw_timer(counts);
    if (smx_SemTest (sem, SMX_TMO_INF))
    {
        stop_hdw_timer();
        smx_TaskUnlock();
        /* do function */
    }
    else
        /* handle timeout */
}

void hto_ISR(void)
{
    smx_ISR_ENTER();
    smx_LSR_INVOKE(hto_LSR, 0);
    smx_ISR_EXIT();
}

void hto_LSR(void)
{
    smx_TaskResume(atask);
}
```

The foregoing works correctly under all race conditions: If the hto interrupt occurs during the signal to sem, which resumes atask, hto_LSR will not run until after the signal SSR completes. At that point, the return value is already TRUE and no harm is done in resuming an already resumed task. The same is true if the hto interrupt occurs after the signal but before stopping the hardware timer. On the other hand, if the hto interrupt beats the signal, then atask will be resumed with a FALSE return value, and atask will not be waiting at sem when the signal arrives.

Chapter 18 Timers

```
BOOLEAN    smx_TimerDup(TMCB_PTR *tmrbp, TMCB_PTR tmr, cc *name)
u32        smx_TimerPeek(TMCB_PTR task, SMX_PK_PARM par)
BOOLEAN    smx_TimerReset(TMCB_PTR tmr, u32 *tlp)
BOOLEAN    smx_TimerSetLSR(TMCB_PTR tmr, LSR_PTR lsr, SMX_TMR_OPT opt, u32 par)
BOOLEAN    smx_TimerSetPulse(TMCB_PTR tmr, u32 period, u32 width)
BOOLEAN    smx_TimerStart(TMCB_PTR *tp, u32 delay, u32 period, LSR_PTR lsr,  cc *name)
BOOLEAN    smx_TimerStartAbs(TMCB_PTR *tp, u32 time, u32 period, LSR_PTR lsr, cc *name)
BOOLEAN    smx_TimerStop(TMCB_PTR tmr, u32 *tlp)
```

introduction

Software timers provide a low-overhead mechanism for tick-accurate delays and cyclic operations. A timer is an smx object which provides a timed action. smx supports three types of timers:

(1) one-shot

(2) cyclic

(3) pulse

A one-shot timer runs for a specified number of ticks (its delay), then invokes an LSR and stops. A cyclic timer starts the same, but after its initial delay, it continues running and timing out at fixed intervals (its period). It invokes an LSR each time it times out. A pulse timer does the same, but offers two delays per period. These are the pulse delay and the inter-pulse delay. Its LSR is invoked for each.

smx timers are volatile. No control block exists before a timer is started, nor after it stops.

why use software timers?

Software timers are appropriate for:

(1) one-shot timeouts

(2) cyclic timing requirements

(3) pulse timing requirements

(4) tick-accurate timing

(5) ease of use

Timers work well to provide timeouts for operations, such as protocol waits and machinery start-up and shut-down sequencing. They run in the background and do not interfere with normal task, ISR, or LSR operations.

Timers work especially well for cyclic requirements. Once started, a timer will invoke an LSR at the end of every period, with minimal overhead. There is no cumulative error because the timer is immediately restarted by smx_KeepTimeLSR() when it times out. In general, this is well before the next tick can occur.

Pulse timers provide an additional capability to generate pulses for controlling things such as blinking lights and to support control algorithms such as pulse width modulation for motor control.

A timer produces more accurate time delays than a task timeout because it invokes an LSR rather than restarting or resuming a task. There is less overhead in this, and the LSR cannot be blocked by other tasks. This benefits one-shot timers in having greater accuracy and benefits cyclic and pulse timers in having less jitter (i.e. less time variation from period to period or pulse to pulse).

Because of the low overhead of timers (only a control block, no code, and no stack), applications with many timing requirements will benefit from using timers and LSRs rather than small tasks with timeouts. Furthermore, since LSRs invoked by timers require less switching overhead than tasks, timers require less processor bandwidth than using small tasks.

Finally, timers permit more powerful control of timing, since they permit control of multiple tasks. For example, one task can start several timers, each of which can start other tasks via LSRs. A task timeout, on the other hand, can restart only the current task.

software vs. hardware timers

Hardware timers are necessary for sub-tick timing, and they offer much finer resolution. However, hardware timers have some drawbacks:

(1) Limited number per processor or SoC.

(2) Usually must be concatenated for moderate delays and may not permit long delays.

(3) More difficult to use — may require assembly language and often are complicated to understand and control.

(4) Asynchronous timing can result in race conditions.

The last point is particularly important. Although developers do not like to admit it, developing bullet-proof code with asynchronous timing is hard to do. If all timing is synchronized to the tick, one stands a better chance of success. If finer resolution is needed for software timers, it is possible to do so without appreciably increasing overhead. This is discussed in the Advanced section at the end of this chapter.

starting a timer

A timer is created and started as follows:

```
TMCB_PTR   TimerA;

smx_TimerStart(&TimerA, delay, period, LsrA,  “TimerA”);
```

This creates TimerA and enqueues it in the timer queue (tq) at (etime + delay). *etime* is the elapsed time since the system was started. To create a timer, smx gets a timer control block (TMCB) from the timer control block pool (smx_tmcbs) and initializes its fields. In particular, it loads the above parameters into corresponding TMCB fields and default parameters into the other fields. The LSR parameter defaults to 0 since it is expected to seldom be used. Note that unlike smx create calls, the timer handle is loaded into the location specified by the first parameter and it is not returned by the function. It is done this way so the address of the timer handle can be saved in the TMCB. This is necessary in order to clear the handle when a one-shot timer times out.

When a timer is enqueued in tq, its timer control block (TMCB) is linked between TMCBs with less and greater expiration times. Each TMCB stores a differential count in its diffcnt field. Only the diffcnt of the first TMCB in tq is decremented when a tick occurs. Thus, the overhead for many timers waiting is no more than for one. However enqueueing a timer requires searching from the beginning of the timer queue and deriving a differential count at each step, until the right position is found. Obviously, this delay is a function of how many timers are in tq.

When a timer times out, if its period is zero (i.e. a one-shot timer), its LSR is invoked and the timer is deleted; its control block is cleared and returned to its pool and its handle is cleared. If the timer's period is not zero (i.e. a cyclic or pulse timer), it is immediately requeued in tq at (etime + nxtdly) and its LSR is invoked. Due to immediate requeueing, no ticks are lost and this type of timer will remain tick-accurate over arbitrarily long times. For a cyclic timer, nxtdly == period. For a pulse timer nxtdly == width, if the pulse state is LO, or == (period - width) if the pulse state is HI. Width and pulse state are TMCB fields. width == 0 designates a cyclic timer, and width != 0 designates a pulse timer. Pulse timers are discussed more later.

The timeout mechanism, itself, is controlled by smx_KeepTimeLSR(), which is invoked by smx_TickISR(). If many timer LSRs are invoked at once, there is a remote possibility that one or more may still be in lq when the next tick occurs and smx_KeepTimeLSR is invoked again. If this happens, smx_KeepTimeLSR() will be enqueued in lq after the waiting timer LSRs. They will run, then smx_KeepTimeLSR() will run, possibly invoking more timer LSRs. No LSRs will be lost as long as lq does not overflow[6], and the worst that will happen is that some LSRs will be delayed.

With modern processors, the above timer problems are very unlikely to occur. Many processors achieve 1,000,000 instructions per tick. Hence, designs with large numbers of software timers should work just fine.

Timers are unusual objects in that when they stop running, they disappear. To determine if a timer is still running, check its handle. If the handle is NULL, then the timer has timed out, was stopped, or never started running.

absolute start

A timer can also be started with:

```
smx_TimerStartAbs(&TimerA, time, period, LsrA,  "TimerA");
```

Unlike the normal start, this service accepts an absolute time from system start rather than a delay from now. Otherwise, operation is exactly the same. The advantage of this service lies in starting synchronized timers. If started with delays, there is a possibility that a tick may intervene and the timers will not be synchronized as expected. Absolute starts are also useful for sequencing the startup of machinery when a rigorous schedule of control actions is needed. If absolute time parameter is already less than etime, the timer is not started and SMXE_INV_PARM is reported.

stopping a timer

Timers can be stopped as follows:

```
TMCB_PTR  TimerA;
u32  time_left;

smx_TimerStop(TimerA, &time_left);
```

This stops and deletes TimerA, and stores its remaining delay time in time_left, unless time_left is NULL. Stopping a timer consists of dequeueing it from tq and adding its differential count to that of the next timer, if any. Deleting a timer consists of clearing its TMCB and returning it to its pool and clearing the timer handle so the timer cannot be reused.

If the timer has already timed out or has been stopped, TRUE is returned, and 0 is loaded into time_left. If TimerA is not a valid timer handle, smx_TimerStop() returns FALSE and an SMXE_INV_TMCB error is reported.

[6] If it does, SMXE_LQ_OVFL error is reported.

Timers are frequently used in situations where they are not supposed to time out. An example would be waiting for a message acknowledgement after sending it. Normally, the acknowledgement would arrive in time and the timer would be stopped. In a case like this, the time left might be useful to adjust future timeouts or as a measure of server loading.

cyclic timer example

```
TMCB_PTR  SampleTimer;
TCB_PTR  SampleTask;
#define RATE 100;

void  ainit(void)
{
    smx_TimerStart(&SampleTimer, RATE, RATE, SampleLSR,  "SampleTimer");
    SampleTask = smx_TaskCreate((FUN_PTR)SampleTaskMain, P2, 0, NO_FLAGS, "SampleTask");
}

void  SampleLSR(u32 par)
{
    u32 sample;
    sample = TakeSample();
    smx_TaskStartPar(SampleTask, sample);
}

void  SampleTaskMain(u32 sample)
{
    /* process sample */
}
```

In this example, the cyclic SampleTimer times out every RATE ticks and invokes SampleLSR(), which takes a sample and starts SampleTask. This is a good way to make a one-shot task run at regular intervals. SampleTask processes the sample, then autostops. Note that because the sample is taken from an LSR, which is invoked by a timer, the sampling time is fairly precise. Of course, other LSRs could intervene, but this is likely to cause much less uncertainty than that for starting a task.

Processing of the sample need not be as timely, so it is relegated to a medium priority (P2) task. However, processing must occur between samples — i.e. SampleTask must be done before it is restarted by SampleLSR, else data will be lost. If this cannot be guaranteed, then samples should be put into a pipe. See esmx etm12 for a more complete example.

duplicating a timer

Duplicating a timer is another way to create a timer. It is done as follows:

```
TMCB_PTR  TimerA, TimerB;

smx_TimerStart(&TimerA, 10, 10, LsrA, "TimerA");
smx_TimerDup(&TimerB, TimerA, "TimerB");
```

In this example, TimerA is first created with a delay of 10 ticks and a period of 10 ticks. Then a duplicate timer, TimerB, is created from TimerA. TimerB is identical to TimerA, except for its name, handle, and possibly its owner. (If another task or LSR duplicates an active timer, that task or LSR would be the owner.) TimerB is then enqueued in tq, immediately after TimerA with 0 differential count, so that the two timers will time out together.

If TimerA is not a valid timer handle, SMXE_INV_TMCB is reported. If TimerA has already timed out or been stopped (TimerA == NULL), SMXE_TMR STOPPED is reported. If &TimerB == NULL, or

TimerB already exists (!= NULL), SMXE_INV_PARM is reported. If the TMCB pool is empty, SMXE_OUT_OF_TMCBS is reported. In all of these cases, the operation fails, FALSE is returned.

Assuming TimerB is successfully created and started, its characteristics can then be changed using other timer services. In particular it is likely that the LSR or the parameter being passed to the LSR will be changed. smx_TimerDup() is useful for creating additional timers that are similar to a root timer. For example:

```
TMCB_PTR  TimerR, Timer[N];

smx_TimerStart(&TimerR, 10, 0, LsrR, "TimerR");
for (i = 0; i < N;  i++)
{
    smx_TimerDup(&Timer[i], TimerR, NULL);
    smx_TimerSetLSR(Timer[i], LsrP, SMX_TMR_PAR, i);
}

void LsrR(u32 n) {}

void LsrP(u32 n)
{
    switch (n)
    {
        case 0:
            /* handle timer 0 */
            break;
        case 1:
            /* handle timer 1 */
etc.
```

In this example, a set of N one-shot timers, is created from TimerR, which does nothing else and disappears when it times out. LsrP is designed to provide different handling for each of the N timers. These timers might be used to provide timeouts for N similar operations and could be individually reset, if operations completed in time, as described in the next section.

resetting a timer

Resetting a timer is similar to stopping it, except that it continues running with its current delay. Timers can be reset as follows:

```
TMCB_PTR  TimerA;
u32  time_left;

smx_TimerStart(&TimerA, 10, 0, LsrR, "TimerR");
...
smx_TimerReset(TimerA, &time_left);
```

smx_TimerReset() removes TimerA from tq. The time remaining for TimerA is loaded into time_left, unless &time_left is NULL. Then, TimerA is requeued in tq using its current delay.

If the timer is a one-shot timer, its current delay is its initial delay (i.e. the delay it was started with). For cyclic and pulse timers, the current delay is the initial delay until the first period starts. Then, for a cyclic timer, the current delay is its period and for a pulse timer, the current delay is the delay until the end of the current HI or LO period — i.e. the time until the next timeout.

If the Timer has already timed out or been stopped, FALSE is returned and 0 loaded into time_left. If TimerA is not a valid timer handle, operation is aborted with a FALSE return and SMXE_INV_TMCB is reported.

Timers are frequently used in situations where they are not supposed to time out. An example would be waiting for an event. Using a timer allows putting an upper limit upon how long the system will wait for the event. When the event occurs, the timer is reset and the timeout begins anew. If the event fails to occur in time, the timer will timeout and an LSR will be invoked to deal with the timeout. Time left may be useful to track maximum or average waits.

For cyclic or pulse timers, reset could be used to restart a timeout or pulse train.

timer LSR control

A timer's LSR and/or the parameter passed to it can be altered as follows:

```
smx_TimerSetLSR(atmr, LsrA, opt, par);
```

This permits changing the LSR for the timer, the LSR option, and the parameter to pass to the LSR. LSR options are defined as follows in xdef.h:

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

These options add convenience for timer LSR design.

The smx_SetLSR() service is typically used in combination with starting a timer, as follows:

```
smx_TimerStart(&atmr, 10, 10, LsrA, "atmr");
smx_TimerSetLSR(atmr, LsrA, SMX_TMR_PAR, atmr);
```

In this example, the LSR stays the same, but the parameter is set to be the timer's handle and the LSR option is set to pass it to LsrA.

```
void LsrA(TMCB_PTR tmr)
{
    /* get sample and send to task */

    if (smx_TimerPeek(tmr, SMX_PK_COUNT) > 50)
        smx_TimerStop(tmr, NULL);
}
```

This example shows passing the timer's own handle to the LSR so it can peek to get more information about the timer. In this case, it stops the timer after 50 samples. Note that if the timer has timed out, the peek will return 0. However, in this case, tmr is a cyclic timer, so that will not happen.

pulse timer control

A pulse timer is created and started as follows:

```
smx_TimerStart(&TimerA, 2, PERIOD, LsrA,  "TimerA");
smx_TimerSetPulse(TimerA, PERIOD, WIDTH);
```

This is a two-step process of first starting the timer, then setting its pulse width, which must be less than the period. Normally, the period is not changed by TimerSetPulse(). In this case, after two ticks, the first pulse of WIDTH ticks will occur. During this time, the pulse state is HI. After WIDTH ticks, the timer goes into its inter-pulse interval of (PERIOD - WIDTH) ticks and the pulse state is LO. The 2 tick delay in TimerStart() is to assure that TimerSetPulse() is able to run before the first pulse, in order to get off to a clean start. On each transition (i.e. timeout), the timer's LsrA is invoked.

An example of using a pulse timer is as follows:

```
smx_TimerSetLSR(TimerA, LsrA, SMX_TMR_STATE, 0);
```

```
void LsrA(u32 state)
{
    if (state == HI)
        control_signal = 1;
    else
        control_signal = 0);
}
```

In this example, the LSR option is changed to STATE, meaning its state is passed as the parameter to LsrA (and par parameter is unused), so LsrA will know if the new pulse state (i.e. after the transition) is HI or LO and can take action accordingly.

pulse width modulation (PWM)

Pulse width modulation is a frequently used method to control DC motors, voltages, etc. Normally hardware timers are used, but with modern processors, it is quite possible that software timers will be fast enough and have adequate resolution — especially for larger machinery and more slowly changing signals. To implement pulse-width modulation for TimerA and LsrA shown above:

```
smx_TimerStart(&TimerB, 1, PERIOD, (LSR_PTR)LsrB,  "TimerB");

void LsrB(void)
{
    u32 volts, width;

    volts = ReadVoltage();
    width = smx_TimerPeek(TimerA, SMX_PK_WIDTH);
    if (volts > (set_volts + LIM))
        smx_TimerSetPulse(TimerA, PERIOD, --width); /* reduce pulse width by one unit */
    if (volts < (set_volts - LIM))
        smx_TimerSetPulse(TimerA, PERIOD, ++width); /* increase pulse width by one unit */
}
```

Where set_volts is an externally-controlled variable and voltage output is proportional to the pulse width of control_signal. This code is in LsrB, which periodically samples the input voltage with ReadVoltage(). LsrB is triggered by a cyclic TimerB with the same period as TimerA, but running one tick ahead of it. Hence, pulse width changes are made just ahead of the next pulse, thus responding quickly to changes in set_volts. Since the period remains constant, pulses will be sent out at a constant rate. If the pulse width goes below 0 or greater than PERIOD - 1, no change will occur and SMXE_INV_PARM will be reported.

pulse period modulation (PPM)

Alternatively, pulse-period modulation could be implemented as follows:

```
smx_TimerStart(&TimerC, 1, PERIOD, (LSR_PTR)LsrC,  "TimerC");

void LsrC(void)
{
    u32  freq, period;

    freq = ReadFrequency();
    period = smx_TimerPeek(TimerC, SMX_PK_PERIOD);
    if (freq > (input_freq + LIM))
        smx_TimerSetPulse(TimerC, ++period, WIDTH); /* decrease frequency */
    if (freq < (input_freq - LIM))
        smx_TimerSetPulse(TimerC, --period, WIDTH); /* increase frequency */
}
```

In this example, width remains constant and period is adjusted to track the input frequency, input_freq. The measured frequency, freq is derived from the pulse train. LsrC is invoked periodically to read freq and to compare it to input_freq, then adjust the pulse output period to track it. In this case, if period becomes less than or equal to width, no change will occur and SMXE_INV_PARM will be reported.

frequency modulation (FM)

Modifying width and period simultaneously permits frequency modulation (FM) as follows:

```
smx_TimerStart(&TimerD, 1, 2*WIDTH, LsrD,  “TimerD”);
smx_TimerSetPulse(TimerD, 2*WIDTH, WIDTH);

void LsrD(void)
{
    u32  period, width, volts;

    volts = ReadVoltage();
    period = smx_TimerPeek(TimerD, SMX_PK_PERIOD);
    width = smx_TimerPeek(TimerD, SMX_PK_WIDTH);
    if (volts > (input_volts + LIM))
        smx_TimerSetPulse(TimerD, ++period, ++width); /* decrease frequency */
    if (volts < (input_volts - LIM))
        smx_TimerSetPulse(TimerD, --period, --width); /* increase frequency */
}
```

This example generates a variable-frequency square wave, which can be converted to a sine wave. The resulting frequency is proportional to the input voltage.

timer peek

Active timer parameters can be obtained as follows:

```
val =  smx_TimerPeek (TimerA, par);
```

For example:

```
count = smx_TimerPeek(TimerA, SMX_PK_COUNT);
```

where count is the number of timeouts since the timer was started. Note: the timer counter is only 16 bits, so the count may not be accurate for long-running cyclic or pulse timers. See the smx_TimerPeek() in the Reference Manual for the complete list of parameters.

If the timer handle is invalid, SMX_INV_TMCB is reported. If the timer has timed out, SMXE_TMR_STOPPED is reported. If par is not recognized, SMXE_INV_PARM is reported. In all three cases, 0 is returned.

Advanced Timer Topics

timers vs. tasks

Timers permit sophisticated solutions to task management problems.

Because timers are foreground objects, they have priority over tasks and can be used to regulate tasks. For example, a task could start a timer to suspend itself, after a delay. Even when the task is running, the timer

LSR can run and can suspend the task, because it is driven by interrupts. This could be used to implement time slicing within a group of tasks (see esmx etm10).

The fact that timers invoke LSRs is useful because LSRs can be context sensitive. Hence, timer expiration need not be limited to a predetermined action; the action could depend upon what task is currently running and what it is doing. An LSR can peek at the current task or any other task. See RM smx_TaskPeek(). This could be used to log what the current task is doing, taking a snapshot of queues, or for other purposes, such as restarting hung tasks.

more precise timers

It may be undesirable to speed up the tick interrupt, because of its overhead. However, more precise timers may be needed. The timer queue, tq, need not be driven by the tick interrupt. The code for processing smx timers in smx_KeepTimeLSR (see xtime.c) could be moved to another LSR and invoked due to a higher-frequency periodic interrupt. Then, profiling, TicksEQ, time slicing, and task timeouts would still be driven the slower tick, but timers would operate at the higher rate. Or the tick interrupt could be sped up, but these other functions could be put into a separate LSR that was invoked every Nth time.

SECTION III   DEVELOPMENT

Understanding the theory of multitasking kernels and applying that knowledge successfully are two different things. A multitasking kernel provides a significantly different environment from the superloop approach, and it takes some getting used to. This section briefly presents how to structure your system, presents a design methodology well-suited to multitasking, presents coding suggestions, followed by debugging discussion and tips. We hope this will get you off to a good start without too much reading.

Chapter 19   Structure

Having studied the basic services and structure of smx, it is time to get started. In this chapter, we discuss how to structure your application. In the next chapter, we discuss how to code it. In both, we advocate a top-down approach

function vs. structure

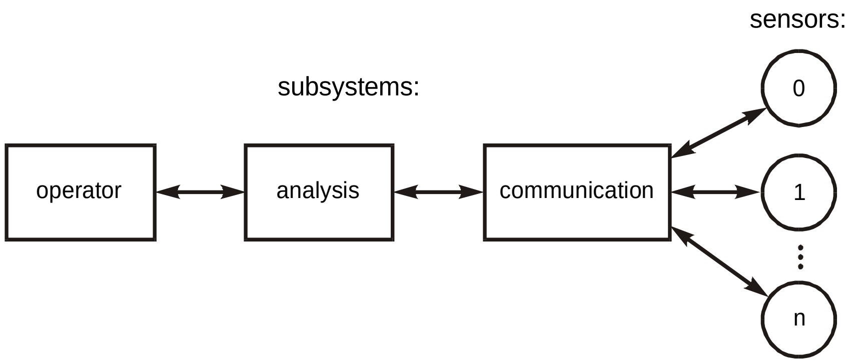
There are two aspects of a system: The functional aspect is concerned with what the code does and how it does it (e.g., the algorithms used). The structural aspect is concerned with how the code operates.

smx is more concerned with the structural aspect of application software than it is with the functional aspect. smx provides a method of structuring which is independent of function. This method involves objects such as tasks, messages, and exchanges. Using smx causes the system to be structured in a way which makes it easier to develop, debug, fix, and change.

Traditional flowcharts are good for implementing functions, but are not of much use for structuring multitasking systems. A better tool is a flow diagram, which shows the flow of data and control from object to object within a system.

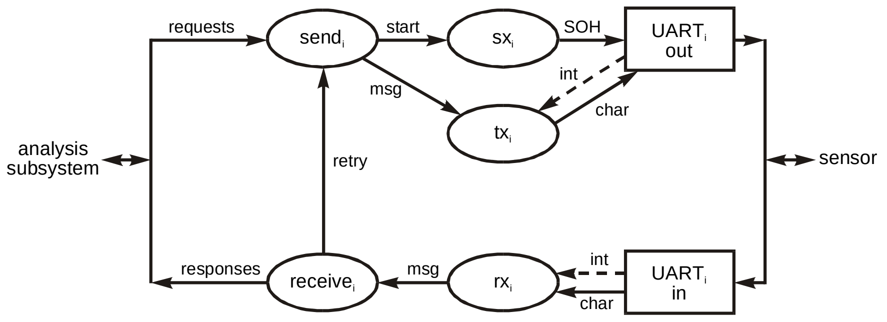
subsystems and tasks

We call the total software system, which you are about to create, the application. The first step is to break the application into subsystems. For example, there might be an operator subsystem, an analysis subsystem, and a communication subsystem. Let us further assume that the communication subsystem connects to some intelligent sensors:



Consider the communication subsystem. It must send queries for the desired data to the sensors and receive the data from them. Assume that the analysis subsystem determines what data is required from which sensor and when. Hence, the communication subsystem is subservient to the analysis subsystem. We will assume that there significant delays at the sensors, and hence requests to the sensors and responses from are overlapped with those from other sensors. This is more complex than a round-robin scheme, which could be handled from a single function.

To make things even more complex, assume that the data messages from the sensors have error-detection codes and that the communication subsystem is expected to resend a query if a data message is found to be in error. It is clear that each port must operate asynchronously with respect to the others. The following would be a possible design for the communication subsystem for one port:



For simplicity, assume that all sensors interface the same way and that this diagram applies individually to each port. The objects shown in this diagram are as follows:

(1) a send task which prepares query messages and initiates transmission.

(2) sx which initializes the UART send channel and causes it to begin transmission with the SOH (Start Of Header) byte. This might be a subroutine or an LSR.

(3) tx which outputs a byte to the UART each time a send interrupt occurs. This must be an interrupt service routine (ISR).

(4) rx which accepts a byte from the UART receive channel each time a receive interrupt occurs. This also must be an ISR.

(5) a receive task which accepts a complete or partial message, checks it, and requests a retry if anything is wrong; or, if there is no error, forwards the message to the analysis subsystem.

approach

The preceding diagram is a data flow diagram showing the flow of data and control between various objects. It is not necessary to be overly specific on the first pass — details will work themselves out. The lines can represent either data or control or both. We are using ovals to represent software and rectangles to represent hardware.

We have a pretty good idea of which block is a task, which block is an ISR, and so forth. This is typical, but not essential. In fact, we could be wrong about some objects. Since it is not important, at this point, we will refer to software objects as “tasks” unless known to be otherwise.

It is important to recognize that code is not a concern now. These tasks might require much code or very little — it doesn’t matter. In fact, it doesn’t really matter whether the tasks are going to be performed by software or hardware. It also does not matter if they are actual tasks or service routines, or whether they are in the foreground or the background. What is important is to identify what objects are necessary and how they are linked.

Of course, we must have some idea of what the various tasks do. For this, writing brief object descriptions is helpful. As design progresses, the descriptions will become more detailed and more precise. Eventually, the transition to code is relatively easy.

how many tasks?

smx fosters creating large numbers of tasks, for the following reasons:

(1) modularity
(2) code reuse
(3) using proven inter-task communication mechanisms
(4) more explicit structure
(5) smxAware debug help
(6) smx error detection help
(7) profiling
(8) smx safety mechanisms

We are all pretty good at estimating the time to write code, but we are generally poor at estimating the time to debug it. Debug time can consume as much as 70% of project development time. Heavy use of tasks reduces new code and places more reliance on field-proven kernel code, so why not use smx as much as possible? Doing so helps to deliver a reliable product on time and on budget. One reason for this is that small tasks are easier to design, code, and debug — small steps foster faster progress. Focusing on writing one task, at a time, is facilitated by the predefined interfaces provided by the kernel.

guidelines

The primary concern at this point is system structure. The criterion of whether a task is needed or not is largely based upon how you want to structure your system. This a creative step. There are no hard rules, but the following guidelines may help:

(1) Functions which are asynchronous with respect to each other must be separate tasks. Functions which might become asynchronous in the future should also be separate tasks.

(2) Functions which are unrelated should be put into separate tasks.

(3) Distinctive processing such as filtering, encrypting, Fourier transforms, report generation, and data analysis merit separate tasks — especially if they involve special hardware.

(4) Relate tasks to physical objects which must be monitored or controlled. Generally, each peripheral will require at least one separate task. Often, separate tasks are required for the input and output functions of the same peripheral. Also, tasks may mirror objects which are outside of the system, such as devices on an assembly line or a magnetic card being swiped through a reader.

(5) If there are multiple sensors, ports, actuators, or whatever, of the same type, each should have its own task — even if they all use the same code and do not operate asynchronously with respect to each other. This facilitates efficient processor usage — while one task is waiting on its device, another task can be running.

(6) Don’t be concerned, initially, whether a task is a true task, an LSR, or just a routine, and don’t be concerned about foreground vs. background.

(7) Connect tasks to other tasks and to physical objects with arrows to indicate the direction that data or control must flow.

(8) Divide each subsystem into small tasks. Tasks can always be merged later, when it becomes apparent that there is no need for more than one task. It is a common mistake to initially define tasks too large; this may mask important interactions and

result in complicated tasks. This is probably due to these programmer's experience with bulky, slow OSs. smx is very fast, capable of performing 90,000 task switches per second on a low-end Cortex-M3 processor and over 400,000 task switches per second on a medium speed ARM9. You want to use smx as much as possible to reduce the new code which you must create and debug.

(9) It is generally easier to work on some subsystems than on others. For some, it is easy to create detailed diagrams and task outlines. Others remain a blob. That's ok. Obviously, you know more about the former and you should concentrate on them. The idea that you have to do everything to the same level may not be productive. Quite likely, those subsystems you do know the most about are the most critical for your application, anyway. (If not, you're in a heap of trouble!).

benefits

Division of an application into many independent domains is one of the main benefits of multitasking. It helps to minimize the propagation of changes. If you have worked with "spaghetti code," you know that even a small change can propagate throughout this kind of code, spawning unexpected new bugs and undermining confidence in the system.

An important part of the process is not just dividing the application into smaller pieces, but also realizing what communication must exist between these pieces. In so doing, you are beginning to define interfaces between tasks, which may assume greater permanence than the code, itself. They may even dominate the design after it has progressed to a certain stage. smx provides a rich set of intertask communication services to support good interfaces.

The more structure that is implemented in smx, the easier it is to change the structure. For example, task priorities can be changed, tasks can be divided or merged, intertask communication can be altered, etc. — all at the stroke of a wand.

using what you have learned

Identify the subsystems of your application and pick one which is moderately complex. It should be a subsystem which you understand and which has some evident multitasking requirement.

Start drawing tasks and interconnections. As you work, you may feel like an artist doing a charcoal sketch. Miraculously, a structure will begin to emerge, and meaningful relationships will unfold. This may surprise you, for it is happening without a flowchart and without code. You are working in a different medium, namely objects. As you progress, it helps to outline what each task does. It also helps, at this stage, to begin defining data structures.

As you proceed, your structure will evolve. Some tasks will split; others will merge. One of the main advantages of a multitasking structure is that it allows these adjustments to occur quickly — even late in the design cycle.

Some of the foregoing is aimed at preventing "writer's block" — that initial queasy feeling of "how do I start?" This is undeniably a difficult part of the design process — possibly because we get so little practice at it — not as much, for example, as we get at working around the problems caused by poor system design!

Chapter 20 Method

Using the guidelines in the previous chapter you now should have a preliminary application structure. Rather than starting at one corner of it and generating code, it is more productive to continue top-down development by creating a skeleton.

skeletal development

A kernel like smx, allows you to rough out your main tasks, LSRs, and ISRs with stub code and to create the intertask communication and control mechanisms between them. Operational times can be simulated with delays. As opposed to a paper design, a system skeleton actually runs, and allows you to work with your system before 95% of the code has been written! The event timeline display in smxAware is especially helpful to view what is happening.

It is possible to experiment with different structures supported by smx to determine which works best, with minimal wasted code. A skeleton also helps to identify sections of the code which may be bottlenecks. Test routines can be written to gain more accurate time estimates for various algorithms; these can be plugged into the skeleton to determine their impact. If not favorable, this may lead to hardware changes, such as picking a different processor, reduction of scope, or other alternatives.

Skeletal development is especially helpful in multi-programmer projects because it permits focusing on interfaces between subsystems and defining how the subsystems interact, up front, instead of after thousands of lines of code have been written. This alone can save considerable time and cost. It may be, for example, that a particular interface method is found to not operate as expected and must be scrapped. Needless to say, the less code written at this point, the better.

The skeletal stage is also a good time to bring in the middleware that you plan to use. This allows you not only to get familiar with the middleware, but also to determine its impact upon your grand plan. There could be problems. If you are attempting to use outside middleware, you may find that integration problems are more difficult than expected or the middleware is not reliable. (A skeleton is a good place to hang test programs and performance measurement programs.) Better to find these out sooner than later. Conversely, you may have to restructure your application to fit around limitations in middleware or a device — perhaps it is slower than you expected.

example

The Protosystem has been provided to give you a starting point. If you have not already, this is a good time to read about it in the smx Quick Start manual and to get it running on your test or evaluation board. Then proceed as follows:

Starting from your flow diagram and object descriptions, decide how to implement each object and each connection between objects. First create tasks:

main.c:

```
TCB_PTR  Op, Anlyz, Comm;      /* main tasks */

void ainit(void)
{
    Op = smx_TaskCreate(Op_main, P3, 200, NO_FLAGS, "Op");
    Anlyz = smx_TaskCreate(Anlyz_main, P3, 200, NO_FLAGS, "Anlyz");
    Comm = smx_TaskCreate(Comm_main, P3, 200, NO_FLAGS, "Comm");
    smx_Start(Op);
    smx_Start(Anlyz);
    smx_Start(Comm);
}

void Op_main(void)
{
}

void Anlyz_main(void)
{
}

void Comm_main(void)
{
}
```

This code should compile and run. Of course, it does nothing. However, we have identified the main tasks for each subsystem, created and started them. Now let's expand upon the Comm task:

```
main.c:
#define N 4             /* number of ports */
TCB_PTR rec[N], send[N];
XCB_PTR rx[N], sx[N];
const char *rtname[N] = {"rec0", "rec1", "rec2", "rec3"]}
const char *stname[N] = {"send0", "send1", "send2", "send3"]}
...
comm.c:
void Comm_main(void)
{
    u32  i;

    for (i = 0; i < N; i++)
    {
        rec[i] = smx_TaskCreate((FUN_PTR)rec_main, P2, 300, NO_FLAGS, rtname[i]);
        send[i] = smx_TaskCreate((FUN_PTR)send_main, P2, 300, NO_FLAGS, stname[i]);
        smx_TaskStartPar(rec[i], i);
        smx_TaskStartPar(send[i], i);
        rx[i] = smx_MsgXchgCreate (NORM, rxname[i]);
        sx[i] = smx_MsgXchgCreate (NORM, sxname[i]);
    }
}
```

In the previous chapter, it looked like having separate receive and send tasks for each port was a good idea, so we are implementing that idea. If it does not work, there is not much code to change, so far. The above illustrates using arrays of handles for tasks sharing common code. Since Comm runs at P3 priority, it will complete before any of the receive or send tasks run. Note that the port number is passed to each receive and send task. Continuing:

```
main.c:
PCB_PTR  spool[N];
u8  *sdp[N];
```

comm.c:

```
void send_main(u32 pn)
{
    u8  *mbp;
    MCB_PTR  msg;

    while ((msg = smx_MsgReceive(sx[pn], &mbp, INF)) != NULL)
    {
        sdp[i] = smx_MsgUnmake(&spool[pn], msg);
        start_UART(pn);
    }
}
```

The N send tasks share send_main(). The task is identified by the port number, pn, passed to it by smx_StartPar() in Comm_main(). Send task n waits at the sx[n] exchange for a message to send. When it receives a message, it unmakes it and starts the UART. It is assumed that there is a send ISR for each port. The pool and data pointers are passed via arrays to the ISRs by smx_MsgUnmake(). The code above will compile and you can step through it with your debugger, but not much will happen. At least you can get an idea of whether or not you like arrays of tasks to deal with multiple ports.

The next step would be to write stub ISR functions and a test task to send canned messages to the sx exchanges. This is something the communications programmer could do to test his or her skeleton while the analysis and operator programmers are working on their skeletons. Then the skeletons can be put together to see how this very primitive system operates. At this very early stage, problems could be found, which dictate a different approach.

The main point to glean from the foregoing is the idea of working downward from the top and adding code only as necessary to get to the next step. Note that most of the actual code is smx calls and that as subroutines are identified, only stubs are written for them — e.g. start_uart(). This is different from starting from one corner and writing detailed code form the get go. At every step, code will compile and run and smxAware can be used to determine what is happening. Since very little code is actually written and what is there can be cut and pasted into different configurations, it is possible to experiment with different implementations without wasting much time or effort. As implementation progresses downward, it of course will be necessary to flush out ISRs and implement crucial calculations in order to get more accurate operational data. But in most cases adding reasonable delays and canned return values to stubs will go a long way toward verifying a design before fully implementing it.

It is also interesting to note that the skeleton can be run on any processor at any time and various other scenarios can be tried with minimal effort. This should appeal to management as well as their being able to see the design unfold via smxAware and to be able to review assumed execution times for unimplemented routines and to see the impact of different values.

By now it is apparent that a skeletal design can be carried a long way before starting detailed coding. In fact, the further the better. Using smx allows you to create skeletal tasks and to fit them together via smx calls. Then as you add blocks of code, each block already has a defined place in the system, including its interface to the rest of the system.

incremental development

Unfortunately time often does not permit skeletal design (or so management thinks!). Or we don't think we understand the whole system well enough. (So why are we doing detailed coding?) Furthermore, we seldom have the luxury of firm specifications. (But skeletal designs are easier to change than is final code.) So, smx supports the incremental approach to coding, too. The basic, incremental design approach is:

(1) Divide the application into subsystems. (But don’t spend time on a skeletal structure.)

(2) Then, subsystem by subsystem:

 (a) structure

 (b) code

 (c) test

This approach typically looks good in the beginning when rapid apparent progress is being made, then falls apart at the end when subsystems do not fit together.

evolutionary development

Frequently, projects start with legacy code, which runs ok, but needs to be extended. Typically, the background code can be initially run as a single task (including the superloop). ISRs will require some minor restructuring as explained in the chapters on ISRs and LSRs. The resulting code should be merged into the smx Protosystem. The goal is to get the old code running properly in the smx environment, with minimal work. Once this goal is accomplished it is then possible to begin dividing the main task into smaller tasks which use smx intertask communication mechanisms to communicate with each other. The old code need not be broken down too finely before it is possible to begin adding new tasks to implement new capabilities. This evolutionary approach avoids falling off a cliff and provides good visibility at each step and it can be combined with the skeletal approach.

summary

Of the three methods discussed above, skeletal development is likely to produce the best results. It is similar to agile software development, but it uses kernel resources to make the job easier. The objective is a well-defined place and interface for every nugget of your new code. During the journey to get there, application code can be simulated with stubs that produce typical results and use time delays to simulate execution times. This permits looking at high-level system operation using smxAware to confirm that operation is as expected or to identify problems and fix them.

Since multitasking is complicated and there are many inter-task communication mechanisms plus task structures to choose from, it makes sense to get the basic structure right before generating detailed code. In a multi-programmer project this is a good job for the lead programmer.

Chapter 21 Coding

Do simple things simply and complex things elegantly.

The numerous examples in previous chapters and the Reference Manual should suffice to show how to use smx services. This chapter is intended to provide additional helpful guidelines. It is possible to muddle through misusing a kernel and, if the processor is fast-enough, no one may notice, except for occasional hiccups and redlining the processor. Our hope is that you will use the many advanced features of smx to achieve an effective design, with no hiccups and plenty of headroom for future growth.

Note: As used herein, the term “task” includes LSR, and the term “task-safe” includes LSR-safe.

design techniques

Summarized below are several design techniques to help you flush out your skeleton design. This is a compendium of useful techniques, from previous chapters, with the objective of helping you through the difficult concept design phase. The goal is to achieve the best fits of form to function as the skeleton takes shape. References are to prior sections and chapters.

(1) Client/Server designs are one of the best ways to avoid access conflicts for special processing units and for peripherals. They also are a good way to perform common processing between tasks, rather than using subroutines. The reason for this is that such accesses and processing can often be performed at a lower priority level, thus keeping high-priority tasks short so that other important tasks can run sooner. Exchange messaging works very well for client/server designs. Clients can assign priorities to messages. Servers can run at fixed priorities or message priorities can be passed to them. See Exchange Messaging.

(2) Pipes can be used for serial I/O. Input ISRs put bytes into pipes as they are received. When a complete packet or message has been received, an LSR is invoked to resume a task to process it. For output, a task can load a pipe, then invoke an LSR to start the output process, or start it directly. Then an ISR will complete outputting the pipe contents, then possibly invoke an LSR to get more data. The disadvantage of pipes vs. exchanges is that pipes require copy in and copy out, whereas exchanges permit no-copy transfers of data. See Pipes.

(3) Block Migration Input allows base blocks obtained and filled by ISRs to be made into smx blocks or messages by LSRs, then passed to tasks, either via pipes or exchanges. Tasks can release the smx blocks or messages back to their base pools for reuse by ISRs. Pipes might be preferred for small systems since they support serial I/O also. However, exchange messaging provides much more capability. See Pipes and Exchange Messaging.

(4) Block Migration Output reverses the above: tasks obtain smx blocks or messages and pass them to LSRs via pipes or exchanges, which unmake them into base blocks and pass them to ISRs, which output the information, and release the blocks back to their smx pools.

(5) State Machines are normally implemented within tasks, but that need not be the case. States may be implemented with tasks, and one or more event groups used to control transitions. See the state machine example in Event Groups. This is feasible,

especially using one-shot tasks. If a single stack is shared, then the RAM cost is only 84 bytes per state. The advantages of this approach are: (1) more visibility — can see state transitions via the timeline in smxAware, and (2) more flexibility — can co-mingle with other tasks.

(6) Interrupt Events Important events typically trigger interrupts, which in turn invoked ISRs. In some cases, events require very little action (e.g. just incrementing a counter) and ISRs can handle them. In other cases, more processing is required and an ISR defer this processing by invoking an LSR. If only moderate processing is required (e.g. calculating a new set point and outputting it) the LSR can perform it. In other cases, the LSR will cause a waiting task to resume or restart. This might be done by signaling an event semaphore, setting an event group flag, sending a message to an exchange, directly resuming or restarting a task, etc. See Service Routines.

(7) Polled Events Less frequent and less important events may be best served via polling. Polling is best done by starting a periodic timer, which invokes an LSR to test for the event. This requires minimal processor time. An alternative is a 0 priority task which tests for one or more events, then bumps itself to the end of rq level 0. Such a task will alternate with the idle task. See Timers and idle task in Tasks.

(8) Resource Sharing. Sharing resources between tasks without conflicts is very important. See Resource Management for a list of techniques to do this.

(9) Gating may be likened to starting a horse race, where each horse represents a task. This can be useful when it is necessary to make sure that every task has completed its assignment or has received information before going on. It is also useful to regulate tasks. See gate semaphore in Semaphores.

(10) Gathering is the opposite. For it, a master task is restrained until every slave task reports completion. Then it is allowed to start the next cycle. See threshold semaphore in Semaphores.

(11) Periodic Operations are best implemented using cyclic timers. When such a timer expires, it immediately restarts itself so no ticks will be lost, then invokes an LSR to perform the desired operation. Since LSRs cannot be blocked by tasks, this results in low-jitter operation. Another approach is for tasks to timeout. For example, a task may suspend of stop itself for N ticks. This may be ok, if precision is not required. Note that the task may be blocked by higher-priority tasks and that there is likely to be significant cumulative error. At a clock rate of 100 Hz, either kind of timeout cannot exceed 248 days. See Timers and timeouts in Timing.

(12) Date/Time Operations are best implemented with task sleeps. See clock calendar time in Timing.

(13) Broadcasting can be performed using broadcast exchanges. This is a no-copy method to make the same data available to several tasks at once. See broadcasting messages in Exchange Messaging.

(14) Multicasting is similar to broadcasting but provides better control over which tasks receive messages. It uses proxy messages, which are sent to specific exchanges. Multicasting can be used not only to disseminate information, but also to do distributed assembly of messages. See proxy message and multicasting in Exchange Messaging.

(15) Aborts may be necessary if conditions change and it desirable to abort previous operations, which are no longer needed, in order to free processor time for new operations, which are needed. Using either start or stop functions, tasks can be

aborted and restarted. See starting and stopping tasks in Tasks and also see ideal task structure in Tasks.

(16) Intertask Communication is essential if tasks are to do anything useful. Event semaphores provide a simple way to alert tasks that events have occurred. If a task processes all waiting events each time it runs, then a binary event semaphore will prevent waking it up unnecessarily. Sending messages to tasks waiting at exchanges is probably the best way for one task to communicate with another, because it is possible to send data and control information in each message. This results in more encapsulated operation vs. using global buffers and variables. Other mechanisms such as pipes, event groups, and event queues can be used, as well as directly starting tasks with parameters. See Intertask Communication.

(17) Avoid Global Variables. These frequently lead to trouble. It is best to pass everything a task needs via a message or a block, if possible. It still is necessary to make sure that there has been a clean handoff — i.e. that the sending ISR, LSR, or task does not fiddle with the block or message after sending it. However, this is generally easier to do than dealing with access conflicts to globals.

keep it simple

With its many calls and system objects, smx may seem overwhelming — just like a new processor or a new programming language. To a degree, smx is both of these, because it creates an environment for applications and it has what amounts to its own language for that purpose. It is best to focus on what you need. smx is designed to serve a wide range of small to large systems with differing performance and functional requirements. Consequently, it is likely that you will need only a subset of smx features for your application. What you do not use is not linked in, hence there is no reason to be concerned about it.

It works best to start simple and introduce more sophistication as needed. For this, it is beneficial to build and run your skeleton, as it develops. This helps to see what really works and what does not work. Some features may seldom be needed, but they can be real lifesavers, when they are needed. If you have carefully read the material, so far, you are in a position to decide what features suit the needs of your application. Concentrate on using those features. Later, if new problems develop, take a second look to see if smx offers solutions for those problems.

many small tasks

Fine-grained task structures produce simpler systems and make better use of smx. The reasons for this are:

(1) Tasks tend to break along functional lines.

(2) Greater use familiar kernel services.

(3) Coding seems more natural.

As a task grows in size, it tends to develop a more and more complex internal structure. Such a task should be broken into smaller tasks in order to make better use of smx services, rather than re-inventing the wheel by developing equivalent internal services. Tasks do not need to be a regular size. Some tasks may need only a small amount of code; others may require pages of code. It is best if each task has a single well-defined function. This makes it easier to keep the task simple and focused.

The ideal task waits for work, performs a straight-forward operation on it, then waits for more work from the same source. If you have a task waiting for an event, doing an operation, waiting for another event, doing another operation, and so on, you probably have implemented a superloop in a task! This is not the right idea — that task needs to be broken up. The advantages of doing so are:

(1) Each task will be simpler.

(2) The tasks can run in the order that events occur, not in a fixed sequential order, as a superloop does.

When you impose artificial ordering, you lose some of the effectiveness of multitasking. Rely more on smx services and keep your tasks simpler. Remember that smx is designed to support large numbers of tasks, so take advantage of it.

infinite task loops

As discussed in previous chapters, normal tasks have infinite internal loops as follows:

```
void taskA_main(void)
{
    BOOLEAN ok;

    /* task initialization */

    while (1)     /* infinite loop */
    {
        ok = smx_SemTest(semA, tmo);   /* wait for signal */
        if (ok)
            /* process signal */
        else
            break;
    }
    /* handle timeout or error */
}
```

Using a while (1) statement is the customary way to create an infinite loop. In addition, the above follows the rule of one statement per line. However smx permits a simpler alternative:

```
void taskA_main(void)
{
    /* task initialization */

    while (smx_SemTest(semA, tmo))     /* infinite loop -- wait for signal */
    {
        /* process signal */
    }
    /* handle timeout or error */
}
```

Note that if () and break statements are eliminated. Also, we have fudged on the one statement per line rule to achieve a simpler appearance.

smx_SemTest() returns a BOOLEAN, so it is correct to test it as shown. If there is no signal at the semaphore, smx_SemTest() will suspend taskA, unless an error is detected, in which case it will abort and return FALSE, immediately. If a timeout occurs, FALSE is also returned. If a signal occurs, TRUE is returned and the loop executes once.

The same test can be done for smx calls that return handles:

```
void taskA_main(void)
{
    u8 *mbp;
    MCB_PTR msg;

    /* task initialization */
```

```
        while (msg = smx_MsgReceive(xchgA, &mbp, tmo))  /* infinite loop -- wait for msg */
        {
            /* process msg */
        }
        /* handle timeout or error */
    }
```

This is acceptable to most compilers, but it might be fudging too much for code test tools and the following should be used instead:

```
        while ((msg = smx_MsgReceive(xchgA, &mbp, tmo)) != NULL)
```

Which of the foregoing you use, depends upon your coding standard and style.

C statements are not atomic

We all know this, yet it is easy to forget that C statements are implemented with many processor instructions. An interrupt can occur between any two processor instructions, unless interrupts are inhibited. What is different in a multitasking environment is that an interrupt could result in a higher priority task preempting the current task and running while the latter is suspended. The higher priority task could access a shared global variable, change it, then suspend, and allow the preempted task to run. If the latter were in the process of using the variable, an error is likely to occur.

Typically application code is not task-safe meaning that it is not protected from preemption. Conversely, SSRs are task-safe and can be used without concern for preemption. See the Resource Sharing Management Chapter for discussion of ways to protect application code.

invalid handles

All system objects are dynamically created by smx create calls. An object handle must not be used until the object has been created. This can be a problem, because object handles are defined at compile time and allocated to RAM at link time. As a consequence, they exist before application code actually creates the corresponding objects. Hence, it is best to declare handles as global variables in an area of memory that is cleared on startup (e.g. bss). This way handles are initially NULL and smx calls using them will fail with reported errors (e.g. SMX_INV_TCB).

Generally speaking, code such as the following is not good programming practice:

```
    t2a = smx_TaskCreate(...);
    smx_TaskStart(t2a);
```

The t2a handle is not being tested and would be NULL, if the task create failed. If so, task start would fail and this, too, would not be detected. Hence a system failure is bound to occur somewhere down the line. Better coding practice is:

```
    if ((task = smx_TaskCreate(...)) != NULL)
        smx_TaskStart(task);
    else
        /* handle error */
```

In the above, failure to create t2a is caught immediately and no attempt is made to start a non-existent task. In addition, error recovery is attempted.

using task arrays

Sometimes it is necessary to have several tasks that do the same thing and use the same code, and it is not convenient to give them separate names — especially if the number of tasks may vary. This situation can be handled by creating an array of tasks as follows:

```
#define N 3
TCB_PTR  t2[N];
const cha * tn[ ] = {"t2a", "t2b", "t2cf"};

void app_init(void)
{
    for (i = 0; i < N; i++)
    {
        t2[i] = smx_TaskCreate((FUN_PTR)task_main, P2, 200, NO_FLAGS, tn[i]);
        smx_TaskStartPar(t2[i], i);
    }
}

void  task_main(u32 n)
{
    /* initialize task n here */
    while (1)
    {
        /* task loop */
    }
}
```

In this example, N tasks share the same code, task_main(). A task handle array, t2[N] is defined and a for loop is used to create and start N tasks. Task main function type 2 is used so that the task number can be passed to task_main() in order to initialize each task (e.g. it might wait at xchg[n]). Then task n enters its infinite loop. The above example also shows how to name the tasks in an array, although doing so is optional — NULL could be used, instead.

This example illustrates clearly that there is not a 1:1 relationship between tasks and main functions. (In this case, there is an N:1 relationship.) This tends to be conceptually difficult, at first. A task is defined by both its code and its context. In this case, the code is identical for all tasks, but the contexts are not. Each task’s context is defined by its:

(1) TCB

(2) timeout

(3) stack

This example is also a good illustration of the tradeoff between more simple tasks vs. fewer complex tasks. In this case, assume that parameter n is used to cause task t2[n] to wait at exchange x[n] for messages. Each exchange constitutes a different message stream, so the purpose of having N tasks is for each to process a separate message stream. This improves processor utilization because while other tasks are waiting for messages, a task that has a message can run.

The same could be accomplished by one task waiting at one exchange for all messages and all message streams being directed to the one exchange. In that case it would be necessary to add a message header to identify a message’s stream and task_main() would become somewhat more complex. For this example, the main benefits of the multiple task approach are: (1) the number of streams can easily be changed and (2) the parallel stream structure is more apparent — especially to tools such as smxAware. If more requirements are added, such as giving streams different priorities, then performance of the one-task approach will be impacted and task_main() will become more complicated. Thus it will more difficult to debug and verify that it is correct. On the other hand, the smx code is proven through use in many designs and should not be a problem unless misused.

The cost for the parallel task structure is low: 84 bytes for the TCB + timeout and perhaps 120 bytes for the stack (i.e. 284*N = 2840 for 10 tasks).

This can be reduced further by using N one-shot tasks:

```
TCB_PTR rec[N];
XCB_PTR x[N];

for (n = 0; n < N; n++)
{
    rec[n] = smx_TaskCreate((FUN_PTR)rec_init, P2, 0, NO_FLAGS, NULL);
    x[n] = smx_MsgXchgCreate(NORM, NULL);
    smx_TaskStartPar(rec[n], n);
}

void rec_init(u32 n)
{
    smx_ct->fun = rec_main;
    smx_MsgReceiveStop(x[n], NULL, TMO);
}

void  rec_main(u32 m)
{
    u8*          bp;
    MCB_PTR msg = (MCB_PTR)m;
    XCB_PTR xchg;

    bp = (u8*)smx_MsgPeek( msg, SMX_PK_BP);
    /* process msg using bp */
    xchg = (XCB_PTR)smx_MsgPeek( msg, SMX_PK_REPLY);
    smx_MsgRel(msg, 0);
    smx_MsgReceiveStop(xchg, NULL, TMO);
}
```

In this simplified example, N receive tasks are created with rec_init() main function and started. Note that each is a one-shot task since no stack is allocated. As each task runs, its main function is changed to rec_main() and it does a receive stop at exchange x[n], where n is its task number. When a message is received at x[n], rec[n] is restarted with rec_main() as its main function and the message handle is passed to it. The message block pointer is obtained with a peek and the message is processed. The exchange to wait for more messages is obtained with another peek. For this to work, the sending task must specify the exchange as the reply:

```
smx_MsgSendPR(msg, x[n], 0, x[n]);
```

Then the message is released back to its pool and receive task waits for another message in the stopped condition with no stack.

Now the cost would be 84*N + 200 = 1040 for 10 tasks, if one stack could be shared among the 10 tasks, which might be reasonable if messages seldom were received and were received randomly.

priority inversion

Priority inversion is said to occur if a high priority task is waiting for a low priority task to finish an operation. Unbounded priority inversion occurs when one or more mid-priority tasks preempt the low priority task and keep the high priority task waiting even longer. Normal priority inversion is said to be bounded because theoretically the time lost to the low priority task can be calculated, making it possible to verify that the high-priority task will meet its deadline.

The unbounded case cannot be calculated because it is unknown what mid-priority tasks might preempt and for how long. Hence, in a system where some high-priority tasks have tight deadlines, unbounded priority inversion must be avoided. Mutexes provide good protection against unbounded priority inversion. See Mutexes for further discussion.

Any type of priority inversion is undesirable since best results are achieved if the highest priority task always runs. Priority inversions occur if:

(1) A lower-priority task is locked.

(2) A task shares a resource with a lower-priority task.

Obviously, locking should be used sparingly and you should verify that it will not cause a higher-priority task to miss its deadline. This can be done using rate monotonic analysis (RMA), which will not be discussed here. It is most easily done by running the system and measuring results. This is especially good to do at the skeletal level, with best-estimate delays plugged in. If a big problem is found it is possible to re-architect or change hardware, with minimal loss of time.

The second problem area can be delt with in a number of ways. First you might consider why a high-priority task is sharing a resource with a low-priority task, or alternatively, why the two tasks have different priorities. If there are good reasons, then a good alternative is to use a server task — see prior client/server discussion under design techniques in this chapter. The nice thing about this solution is that it gets the high-priority task off the hook and allows it to go about its business. This is tantamount to splitting it into two pieces, which might be another viable solution.

If a server task won't work, we suggest reviewing Resource Management for other ideas. Possibly using a mutex will be the best solution

avoiding deadlocks

A *deadlock* occurs when two tasks require the same two resources and each has one of the resources and cannot complete for lack of the other. Since neither task can complete, neither can release the resource it has, so the tasks are said to be deadlocked. To avoid this problem, always get resources in the same order. For example: If task1 and task2 both get resourceA before resourceB, then task2 cannot get resourceB if task1 already has resourceA. When resourceB becomes available (e.g. released by task3), task1 will get it and run. task1 will then release both resources and task2 will get them and run. Using ceiling priority mutexes also solves the deadlock problem.

For various reasons, the above rule may be broken and a system lockup could occur, which could result in serious damage. Hence, reasonable timeouts should always be specified on waits and when they occur, appropriate diagnostic and recovery actions should be taken.

choosing stack sizes

Thanks to the system stack (SS) determining stack sizes for tasks is fairly simple. You do not need to be concerned about the stack usages of ISRs, LSRs, the scheduler, nor the error manager. These are handled by SS. You need only to consider the stack depth of each task main function plus stack required for maximum nesting of functions it calls. IAR EWARM provides a tool to help with this — maximum stack usage per function is provided in .lst files. Unfortunately it is not documented and does not seem to include subroutines. Using this tool, the SSRs with the greatest stack depths are:

smx_BlockPoolCreateDAR()	88 bytes
smx_BlockPoolCreate()	80
smx_EventFlagsSet()	72
smx_EventFlagsPulse	64
smx_TaskCreate()	64
smx_TimerStart()	56

All others are 48 bytes, or less. Functions such as smx_SemTestStop() need 40 bytes and smx_SemSignal() need only 24 bytes. Hence, it is practical to use stack sizes as small as 100 bytes for very simple tasks, such as one-shot tasks.

It is generally best to initially choose stack sizes that are larger than expected to be necessary, add stack pads, then tune stack sizes late in the project. See Debugging and Stacks for more information.

configuring smx

Configuration constants that control the basic operation and size of smx are defined in xcfg.h and smx must be re-compiled if any of these constants is changed. Examples are enabling profiling, stack scanning, handle table, time slicing, lock nest limit, etc.

Configuration constants that are application-orientated, such as the numbers of various system objects (e.g. NUM_TASKS), handle table size, LSR queue size, stack pool stack size, timeout period, etc. are defined in acfg.h. As delivered, acfg.h is set to adequate values to begin work. As application code is developed, settings will generally need to be increased.

The application must be recompiled for changes in acfg.h to take effect, but smx need not be recompiled. This is because the values in acfg.h are loaded into the smx_cf structure, which is located in main.c and which is used by smx, during initialization. Hence, development can proceed without recompiling the smx library. This makes smx evaluation kits useful to start actual development work and is convenient during development.

Some configuration is done from the Protosystem make file. This is necessary when an option controls whether certain files are included in the build. For example, the make file allows specifying whether to link component libraries, such as smxFS and smxUSBH. Enabling a product also passes a define on the compiler command line to enable any conditional code for that product. See the SMX Quick Start manual for details about the Protosystem, its make file, and configuration.

Note: For safety, smx_cf can be located in ROM and we advise doing so for release versions. Most of these configuration constants are used only during initialization and thus damage done to them while running should not matter, but it is better to be safe.

user access to smx objects

smx peek services are the recommended way to obtain information about smx objects. They are currently provided for several system objects. Peek services are implemented with SSRs and thus are task-safe. For example,

```
first_task = (TCB_PTR)smx_SemPeek(sem, FIRST);
```

will return a valid handle of the first task waiting at sem, or NULL if none is waiting.

If a peek service is not available, accessing an object's control block by its handle can be done as follows:

```
smx_TaskLock();
first_task = (TCB_PTR)sem->fl;
smx_TaskUnlock();
```

However, the following is safer:

```
smx_LSRsOff();
first_task = (TCB_PTR)sem->fl;
smx_LSRsOn();
```

It is important to lock the task or inhibit LSRs while accessing sem->fl, because the C assignment is not task-safe and a meaningless value could be loaded into first_task, if a preemption occurred that changed sem->fl.

naming objects

We recommend naming smx handles (i.e. control block pointers) after the system objects which they represent. Handles are usually globally defined and frequently used. Main task function names are seldom used, so it works well to form their names by adding "_main" to the task names (e.g., atask_main).

Choosing meaningful names for system objects enhances readability. For example:

```
PCB_PTR empty_sectors;
MCB_PTR next_sector;
u8 *mp;

next_sector = smx_MsgGet(empty_sectors, &mp, 0);
```

empty_sectors is a message pool of sector-size messages; next_sector message is the next to be used.

All caps are used in smx for manifest (i.e. compile time) constants, C macros, typedefs[7], directives, and keywords. In this manual, a name in all caps is usually either an smx data type or a manifest constant

comments and style

C does not comment well due to the free-form nature of its code (as compared, for example, to assembly code). The thing that seems to be the most helpful is to use descriptive names in the code. We find it also works well to describe what a function does in a heading, prior to the start of the function. If the function is long, subheadings are helpful.

tips for reliable code

This chapter summarizes programming practices used in smx. It may help you to decide upon your own programming standards. Writing code for real-time, multitasking systems requires a different discipline than writing code for other systems. The code must run unattended for lengthy periods time, often years, and failure often is not an option.

(1) The resting or idle state of every control variable should be 0. This permits putting the system into a safe state merely by clearing RAM, which is normally done on startup for variables in the bss segment.

(2) Avoid forbidden states:

```
switch (var)
    case 1:
        /* action 1 */
    case 2:
        /* action 2 */
    default:
        /* action 0 */
```

In the above, var cannot be put into a forbidden state due to noise or a bug. This is viable, as long as action 0 will not cause harm.

(3) Alternatively detect, report, and correct forbidden states:

```
switch (var)
    case 0:
        /* action 0 */
    case 1:
```

[7] With the exception of the basic data types such as *u32* and *u16*. These are lower case for consistency with standard basic data types such as *char*, *int*, etc

```
    /* action 1 */
case 2:
    /* action 2 */
default
    /* report and correct forbidden state */
```

(4) Error recovery code should not mask problems, leaving no indication that they occurred. There should be at least an error counter. During debug, it is recommended to report errors. See reporting errors in Debugging.

(5) Implement high-level checks. For example, report timeouts. This helps to find bugs which are not showing up during normal testing.

(6) Retry whenever feasible. (But log the error.)

(7) Use the techniques discussed in the Resource Management chapter to avoid access conflicts.

(8) It is best to interlock operations so that nothing can possibly go wrong, rather than to rely on open-ended operation. The latter may work — for a while. See the broadcasting example in Exchange Messaging for how interlocks may be implemented.

(9) Limit-check variables before using — especially those from outside the system. This helps to avoid malfunctions and makes the system more robust against environmental factors (e.g. noise) and malware. Some design standards require that every function parameter be checked before using it.

(10) Use parity, checksums, or CRC for data channels known to be noisy, or even if they aren't noisy now.

(11) Clear message handles immediately after sending their messages and clear a message and block handles immediately after releasing them. For example:

```
BCB_PTR blk;

MCB_PTR msg;

smx_MsgSendPR(msg, xchg, PR0, NO_REPLY);
amsg = 0;
smx_BlockRel(blk, 0);
blk = 0;
```

This prevents a handle from being used again, by another smx call, and thus increases task isolation. All smx delete calls pass the locations of handles and the handles are cleared so that corresponding objects can no longer be used. This has not been done with send and release operations because of concern that it might cause errors.

(12) Don't make assumptions — check for exceptions. Making assumptions, not love of money, is the root cause of evil. Purge them!

(13) Use timeouts on all task waits. Time spent determining reasonable upper bounds on event waits, rather than just using INF, will be well-rewarded. When timeouts occur they indicate a problem or a lack of understanding, either of which should be corrected. If it not possible or too time-consuming to pick a timeout for each event, define a reasonable MAX_TMO to use for all events. This is a timeout that you would think would never happen in your system, but if it did you would want to know about it. Note: INF is correct, when there is no appropriate upper limit.

Error recovery from timeouts should also be implemented in order to break deadlocks and enable the system to recover from missed events. Task death is not a good thing and probably will lead to a system crash or compromised operation. Correcting a timeout is obviously less drastic and may suffice to keep the system running normally.

(14) Avoid preemption when not necessary. Short tasks, such as one-shot tasks can be started locked and kept locked, until done. They should be unlocked only if necessary to achieve performance goals for higher-priority tasks. It is also advisable to start with a small number of priorities and add priorities as necessary. Lock tasks to avoid unnecessary preemptions, e.g.:

```
smx_TaskLock();
smx_SemSignal(semA:
smx_MsgReceive(xchgA, &dp, tmo);
```

will prevent an unnecessary task switch if a higher priority task is waiting at semA. Although smx task switching is fast, unnecessary preemptions waste processor time. They also can lead to unnecessary resource conflicts.

(15) When initializing structures set every field to its proper initial state; do not assume that it is already in the proper state or that it will not be used. Someone writing new code or modifying existing code will probably assume that all fields have been initialized.

(16) When disabling interrupts, it is recommended to save the state, then restore it later:

```
CPU_FL ps;

ps = sb_ISD();
/* perform operation with interrupts disabled */
sb_IR(ps);
```

sb_ISD() saves the processor state in the local variable, ps, then disables interrupts. (It is faster to use a register than the stack — hopefully the compiler assigns ps to a register.) sb_IR() loads ps into the processor state register in order to restore interrupts to their initial state. The problem with using sb_ID() and sb_IE() is that you must be absolutely certain that interrupts are enabled to start with. If not, interrupts will end up enabled when they should not be. (It should be noted, however, that smx, itself, uses sb_ID() and sb_IE() to maximize performance, and this is why interrupts must be enabled whenever an SSR is called.

(17) Group related objects into structures rather than defining individual, alphabetized variables. This makes better use of processor addressing mechanisms and data caching in order to improve performance and reduce code. Aligning structures on cache lines is even better:

```
#pragma data_align = SB_CACHE_LINE

struct  x
   {
      /* fields */
   }
```

(18) Try to make structures fit cache lines evenly. For example if the processor cache line is 16 bytes, try to define structures to be 8, 16, or 32, etc. bytes, then align on cache-line boundaries. The objective is to minimize the number of cache line accesses needed to get the whole structure into the data cache. Very often, fields can be reduced from 32 to 8 or 16 bits. For example, counters often need to count to less than 256. With some compilers (IAR EWARM, for example) enums can restricted to 8 bits (see SMX_ERRNO in xdef.h, for an example); this is usually plenty.

Addresses or handles can often be reduced to 8- or 16-bit indices. Up to 8 flags can be compressed into a single byte. Some compilers implement bit fields as efficiently as defines. Of course, these compressions and decompressions require more processor cycles, but processor cycles are cheap compared to external memory accesses.

If it is not possible to change fields enough consider adding padding. This is very important for large arrays of structures and can make a significant improvement in performance. External memory is cheap, so padding is not much of a concern, except for huge arrays of structures.

(19) Define configuration constants in a single header file in order to make changing them less error-prone. It also helps code readability to have all constant definitions in one place, where they can be seen together.

(20) It is usually worth the effort to range test counters, indices, and pointers before using them. Out-of-range values can cause serious harm. Since fetching variables usually takes many processor cycles, the additional time to test them vs. constants may add little or no overhead..

(21) Minimize comments because they invariably get out of step with the code and are useless. Let the code speak for itself and write it so it speaks clearly.

(22) Change symbol names as their functions change. Appropriate names improve understanding (yours as well as others). Short names are more memorable than long names.

(23) Make simplifications as opportunities arise. Each time you pass through the code, you are likely to understand it better and to be able to simplify it. This is commonly called *refactoring* and it is probably the best way to get clean code.

(24) Desk checking finds more bugs per hour than any other form of testing. However, programmers don't write code nor print it, anymore. So, stepping through code with the debugger is the next best thing — but a foggy head won't do. Wait until you can rethink the code and consider all possibilities. Otherwise, do your email.

(25) Sections of code which do the same should look the same. Take the time to modify code to make it identical. Even better, replace multiple instances with a subroutine or macro. This reduces future errors by requiring changes to be made in only one place.

(26) One C source statement per line produces more readable code and works better with C source-level debuggers.

(27) Distinguish between true constants and initialized variables. The *const* keyword should be applied to true constants and is useful to detect programming errors. Set the compiler to put constants in a separate section so that they can be located in ROM.

(28) Use the *volatile* keyword for peripheral registers and any other global variables which may change due to an interrupt or preemption. Otherwise, the compiler may not reload the variable each time it is accessed.

(29) See tips on writing ISRs and LSRs in Service Routines.

(30) Use interlocks to guarantee that things happen as expected, rather than assuming that they will happen as expected. Semaphores are good for this.

(31) Do not declare a timer as an auto variable. When it times out or is stopped, the timer handle location will be cleared. This will cause a mysterious error if the function that declared timer has returned and this location in the stack being used by another function.

(32) A similar restriction applies to handles because deleting objects clears their handles.

(33) Allocate buffers from the heap to separate them from globals — especially smx globals. A common problem with buffers is overflow, which can damage nearby global variables. If allocated from heap debug chunks, fences provide some protection from buffer overflow. Running smx_HeapScan() periodically can detect small overflows before serious damage is done.

(34) Be careful about calling smx services when a task is locked — many of them will break locks. See Tasks, locking and unlocking tasks. To be safe, it is best to avoid any service which may suspend or stop the task. If it is essential to test a semaphore or something, be sure to specify NO_WAIT.

(35) It is generally best to delete an object only from the lowest priority task using it and not from LSRs. Not following this rule can result in unexpected invalid object errors.

Chapter 22 Debugging

Unfortunately computers do what we tell them, not what we mean.

Because tasks can preempt each other and thus run in random order, multitasking systems tend to be more difficult to debug than non-multitasking systems. This chapter is intended to help you to debug your multitasking code more easily. It presents information on smx debug tools and features, followed by debug tips.

debug tools & features

The following smx tools are available to help you:

(1) smx error manager detects and reports about 70 error types, which can be very helpful during debug. You should read the Error Manager Chapter before starting debug.

(2) smxAware is a DLL that adds kernel-aware capabilities to debuggers. It has text displays and graphical displays (for some debuggers). The text displays show lists of tasks, semaphores, the ready queue, stacks, event buffer, memory usage and other smx objects. The graphical displays show event timelines, profiling, stack usage, and memory map overview. The event timelines allows seeing when each ISR, LSR, and task ran. A detail window shows why. It is possible to zoom in for detail and out for overview. The same information is available in tabular form. The memory map overview shows the layout of memory graphically with named objects, such as stacks and heap blocks. It also supports zoom. smxAware is a very powerful debug tool — especially during the skeleton development stage. See the smxAware User's Guide for full details about features and debuggers supported.

(3) handle table associates assigned names with handles. Most create calls allow assigning names to system objects. These names are stored in each object's control block (no longer in the handle table, as of v4.2). Names can be helpful during debug — they are visible when looking at control block structures. smxAware creates its own handle table from smx control blocks. For objects that do not have control blocks or which do not have name fields in their control blocks, it is possible to make entries in the smx handle table using smx_HT_ADD(). smxAware also uses this handle table for its displays. It is discussed more below.

(4) profiling can be turned on or off by SMX_CFG_PROFILE in xcfg.h. When on, it keeps a precise run-time count in the rtc field of each task and it keeps run-time counts for all ISRs combined in smx_isr_rtc and for all LSRs combined in smx_lsr_rtc. Run time counts are used by smxAware profile displays; they can also can be viewed via the debugger. See Profiling in the Advanced Section.

(5) event logging can be turned on or off by SMX_CFG_EVB in xcf.h. When on, specified events are logged into the event buffer, EVB, and time stamped with precise times. smxAware uploads the EVB when stopped at a breakpoint and uses it for the event timelines graph and table. Event logging is discussed below.

(6) time measurement macros are provided for precise time measurements of up to one tick. See smxBase User's Guide.

(7) stack checking. Stack usage is automatically monitored by smx to help you tune stack sizes and detect stack overflow. Stack scanning is enabled by SMX_CFG_STACK_SCAN in xcfg.h and performed by the idle task. A stack high-water mark is stored in tcb.shwm, for each task. This information is used by smxAware to display stack usage. See stack scanning in Stacks.

(8) heap checking. smx_HeapScan() is provided for checking the integrity of the heap. It is normally called regularly from the idle task and scans fnum nodes each time. Also, the global smx_heap_hwm is provided to indicate the heap usage high-water mark. This is the maximum number of bytes used at any time since system startup. See Heap.

(9) print ring buffer is a global data structure allocated in the Protosystem if SMXAWARE is defined in the tools prefix include file (e.g. iararm.h). It is displayed by smxAware. This provides “printf-style” debugging. That is, you can add calls to sa_Print() in your code to write strings at various points so you can see the order in which things are executing. Instead of printing these to the console, they are stored in the ring buffer. The size of this buffer is set in APP\smxaware.c. The definition and this function are there too. See the smxAware User’s Guide for more information about this feature.

(10) smx_TickISRHook(), if enabled, is useful to piggy-back ISRs on the tick interrupt during early system debugging, when the final hardware is not available. See esmx.c for how to use it. It also can be used to add functionality to the tick ISR without altering it, or to make sure that a chain of action is correct before connecting to the real interrupt.

application events

Reporting or stopping on application events, including errors, is helpful during debugging. The following are available:

(1) Print ring buffer — see above.

(2) User events — see Event Logging. These appear in smxAware.

(3) sb_DEBUGTRAP — halts execution in debug mode, only. Can be used to stop execution if a wrong path is entered. Use instead of assert.

important smx variables

When debugging an application, it can be helpful to get a feeling for what smx is doing. The following are important smx variables:

(1) smx_ct — current task handle.

(2) smx_clsr — current LSR address, if not 0.

(3) smx_sched — scheduler flags: CTSTOP, CTSUSP, & CTTEST. Determine what the scheduler will do.

(4) smx_srnest — service routine nesting level. 0 when in application code. 1 when in ISR, LSR, SSR, or scheduler. Higher when service routines become nested.

(5) smx_ssnest — system stack (SS) nesting and most significant bit indicates in system stack, if set. When SS nesting = smx_srnest, exit SS and return to task stack.

(6) smx_rq — points at the lowest level, rq[0], in the ready queue array.

(7) smx_rqtop — points at the top occupied level in rq.

(8) smx_lqctr — number of LSRs in lq.

(9) smx_etime — elapsed time since system start.

(10) smx_stime — system time relative to a reference date and time.

looking at smx objects

The following discussions apply to objects in a debugger watch window. Discussion applies to IAR C-SPY and may not be correct for other debuggers.

smx_rq is statically defined as an array of RQCBs. Clicking on it will show all levels. Clicking on a level will show the RQCB for that level. If the level is occupied, its tq is set and its fl points to the first task. Clicking on fl shows the full TCB of the task. The task can be easily identified from its name field. Clicking on the TCB fl will show the next task, and so on. When fl points to the RQCB, there are no more tasks at the level. To see what task is running or will run next, click on smx_rqtop, then fl.

control blocks are allocated from their respective pools and assigned handles by create functions. If the handle is 0, then the object has not been created or it has been deleted. When it an object does exist, click on its handle, such as t2a. This shows the full control block for the object. The name field shows the name of the object, the cbtype field identifies the object type (e.g. SMX_CB_TASK). The fl and bl fields show if the object is in a queue. Other information is object-type dependent.

LSR queue is allocated from SDAR. It consists of lq cells. smx_lqi points to the first cell, smx_lqx points to the last cell, and smx_lqout points to the next cell that is running or will run next.

smx_cf contains the current configuration constants. It is easily viewed as a structure.

error buffer (EB) is allocated from smx_ebi, smx_ebx, and smx_ebn point to the first, last, and next to fill error records. Clicking on these will show the corresponding error records. To see the record of the last error click on smx_ebn-1.

event buffer (EVB) is allocated from SDAR. smx_evbi, smx_evbx, and smx_evbn are word pointers and event records are of variable size. Hence, it is only practical to look at the event buffer is via smxAware.

debug tips

(1) Use a console. smx is quite good at catching programming errors such as out of control blocks, stack overflow, incorrect service parameters, etc. When an error is detected, an error message is output to the console. Seeing such a message can save a much time and frustration tracing what looks like a tough bug, but which, in fact, is very easily fixed. To add a console, connect the RS232 port on your test board to a USB port on your PC, via a serial adapter, and run a terminal emulator, such as TeraTerm® on your PC. (It helps to have a dual display so that the console and other less important windows can be put on the side display.) Check the BSP notes to see which UART port on the test board to connect to.

(2) Use breakpoints in tasks to determine order of operation. When you are not sure what will happen next, it useful to put breakpoints on all tasks that might run. Then pressing the debugger run button repetitively shows quickly the order of task operations. (smxAware shows this even better.) Note: when setting breakpoints in tasks, set them at the start of the infinite loop, not at the start of the main function, which probably has already run and will not run again.

(3) Confusing results when stepping. It is easy to forget that C statements are not atomic — each consists of many machine instructions and interrupts can occur between them. When you step, interrupts may occur, ISRs may run, LSRs may run, and higher priority tasks may run.

A lot can happen in the blink of an eye. All of this hidden activity may cause results that are confusing to you as you step through your code. One way to simplify things is to disable interrupts in sections of code that you are debugging. Or you can lock tasks that you are stepping through by setting smx_lockctr or turn off LSRs. If none of these work you will have to debug your task or LSR in isolation.

(4) Looking at task timeouts. All task timeouts are grouped into the timeout[] array to make them cache-friendly for good performance. The timeouts are in the same order as TCBs in the TCB array. Due to the way tasks may have been created and deleted, TCBs and hence timeouts are not in any particular order. To see a task's timeout, get its index from its TCB, then enter smx_timeout[indx] into the debugger's Watch or Globals window. If the value shown is 0xFFFFFFFF, the timer is inactive. Otherwise, subtract smx_etime from the timeout value to get the number of ticks to go. smx_tmo_min is the value of the next timeout to expire and smx_tmo_indx is its index into timeout[]. Note: if the corresponding task has resumed for another reason, these values may temporarily be out of sync. You can also use smx_TaskPeek(task, TMO) to get the time to go for task.

(5) Finding where an error occurred. Set a breakpoint on smx_EMHook() in main.c to stop the debugger on any error. You can determine where the error occurred by using the debugger's call stack window or by stepping out of smx_EMHook() and smx_EM() to the point of error. Instead of a breakpoint, you can enable sb_DEBUGTRAP(), which also causes the processor to halt in smx_EMHook() on every error.

(6) Assigning names to system objects is recommended. Most create services allow assigning names to objects. Name pointers are stored in control blocks and names appear in the debugger Watch window when looking at the control blocks. These names are also used by smxAware. See handle table in Miscellaneous Topics and see the smxAware User's Guide.

(7) Use stack pads Stack pads are put above the top of the stack. During debug, it is helpful to have stack pads of 10 or more words, so the application can continue to run if a stack overflow occurs. Without a stack pad, it may not be possible to suspend a task or the stack may overflow into another stack or heap chunk, causing a failure. Stack overflow is detected and reported as normal. In addition it is shown by smxAware as a red tip on the stack bar. STACK_PAD_SIZE in acfg.h controls the size of stack pads. See stack pads in Stacks for more information.

(8) When a puzzling problem is encountered, increasing STACK_PAD_SIZE is a quick way to determine if stack overflow is causing it.

(9) Task lockup (i.e. stops running) can occur if the task is using a binary resource semaphore and tests it twice. This problem can occur when using a library which uses a binary resource semaphore to protect non-reentrant functions and one such function calls another. It is preferable to use a mutex, or at least tight timeouts on semaphore tests.

(10) Triggering on a particular task resume or start (i.e. running to where a suspended/stopped task will resume/start execution again) is possible by setting smx_tt to the handle of the task to run to, in the debugger watch window, and then running the application. First set a breakpoint or uncomment sb_DEBUGTRAP() in smx_TTHook() in main.c. The debugger will then break right at the point where that task is dispatched next. It is necessary to step a number of times to get through the tail of the scheduler and smx_SSR_EXIT() or smx_ISR_EXIT() code. Once back in C, the call stack window will work, and you can use it or continue stepping through the task code.

SECTION IV   ADVANCED TOPICS

Prior sections have been aimed at presenting smx functionality and how to design using smx. This section covers more general topics such as error management, event logging, resource management, profiling, etc. which draw upon smx services and features previously presented, as well as including new features. Our recommendation is that you learn how to use smx and bring your skeleton design fairly far along before delving into these other areas

Chapter 23 Heap Management

u32	smx_HeapBinPeek(u32 binno, SMX_PK_PARM par)
BOOLEAN	smx_HeapBinScan(u32 binno, u32 fnum, u32 bnum)
BOOLEAN	smx_HeapBinSeed(u32 num, u32 bsz)
BOOLEAN	smx_HeapBinSort(u32 binno, u32 num)
u32	smx_HeapChunkPeek(void* vp, SMX_PK_PARM par)
u32	smx_HeapPeek(SMX_PK_PARM par)
BOOLEAN	smx_HeapScan(CCB_PTR cp, u32 fnum, u32 bnum)
BOOLEAN	smx_HeapSet(SMX_ST_PARM par, u32 val)

xcfg.h:

SMX_HEAP_FENCES	number of fences above & below data blocks.
SMX_HEAP_SAFE	enable all safety checks.
SMX_HEAP_STATS	enable min and max allocation time measurements
SMX_HEAP_DATA_FILL	data block fill pattern.
SMX_HEAP_DTC_FILL	donor and top chunk fill pattern.
SMX_HEAP_FENCE_FILL	fence fill pattern.
SMX_HEAP_FREE_FILL	free chunk fill pattern.

acfg.h:

HEAP_MERGE_OFF	threshold for chunk merge OFF.
HEAP_MERGE_ON	threshold for chunk merge ON.

introduction

This chapter is concerned with heap tuning, debugging heap problems, and self-healing. Methods are first presented for dealing with out of heap. As shipped, eheap is configured for normal requirements. However, if performance is not adequate, tuning the heap for the application will improve it. Debug aids help to find difficult heap problems. Self-healing is implemented via heap scanning and fixing from the idle task in order to improve heap reliability.

Heap Tuning

the need for tuning

Theoretically, it is not possible to design one allocator that will work well for all applications. For every allocation strategy, some applications can be found that will cause excessive fragmentation or other serious problems. A general-purpose allocator, such as dlmalloc, is good at satisfying the needs of most applications, but when push comes to shove, it goes to the operating system for more memory. dlmalloc has a fixed, highly optimized structure, which can be tuned to the operating system, but not to the application.

This situation is different for most embedded systems. Although there is a wide variation of characteristics from one embedded system to the next, a given embedded system is a typically a single application with constrained characteristics. Thus it is feasible to tune its heap to get good performance without serious risk of fragmentation failure or other serious problems. Also, variable structure and tunability help to shoe-horn heaps into tight spaces, while achieving necessary performance. Tunability is a valuable characteristic for embedded system heaps.

bin configuration

The SMX_HEAP_BIN_SIZES (xcfg.h) constants array is used to initialize smx_bin_size[]. This is done by the compiler. Bins can easily be added, removed, or resized simply by changing the constants in SMX_HEAP_BIN_SIZES and recompiling to adjust smx_bin_size[] to the new configuration. Then running smx_HeapInit() initializes the bins and internal heap variables from it. Hence, the entire heap configuration can be changed by changing a few constants in SMX_HEAP_BIN_SIZES. This makes it easy to experiment with different bin configurations to see which produces the best performance..

The following is the standard bin configuration shipped with smx. It is intended to provide an general purpose heap for most systems to start with:

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

The upper numbers are bin numbers and the lower numbers are the bin sizes. Bins 0 through 12 comprise a small bin array, SBA, which has bin sizes from 24 bytes to 120 bytes, in 8-byte increments. Above the SBA, starting at bin 13, are 15 bins covering the range from 128 bytes to 2040 bytes (the last size in bin 27). Each of these bins covers a range of 128 bytes with 16 chunk sizes. The top bin starts at 2048 bytes and covers it and all larger sizes. The last entry, -1, terminates the bin size array.

optimizing bins

To illustrate optimizing the bin array, consider a small embedded system that has a bin size array of: 24, 32, 40, 48, 128, 256, 512, then the SBA would consist of bin 0, 1, and 2, with chunk sizes 24, 32, and 40; bin3 would contain sizes 48 to 120, bin4: 128 to 248, bin5: 256 to 504, and bin6: 512 and up. Suppose it is found that packets of exactly 128 bytes (136-byte inuse chunks.) are in high demand by several tasks. Then, a small bin could be added as follows: 24, 32, 40, 48, 136, 144, 256, 512. The new bin4 contains only 136-byte chunks, so chunks are taken off the front, which eliminates bin search time and thus permits faster allocation for 128 byte blocks. This is an example of how the bin structure can be matched to the needs of a specific embedded system.

If certain large chunks are frequently used, their allocations can be optimized by creating bins with their sizes as the bin sizes. For example, if sizes 1504 and 1536 are popular, create bins with sizes 1504 and 1536. The 1504 bin would also hold sizes 1512, 1520, and 1528, but 1504 chunks would always be freed to the start of this bin. Thus, if a 1504 chunk is needed, the first chunk in the 1504 bin would always be taken because it would be either exact or big enough. The same applies to the 1536 bin. If there were a large number of free 1504 chunks, search times for 1512, 1520, and 1528 chunks could become too long because of the need to search through all of all the 1504 chunks ahead of them. This could be solved by creating a 1512 bin. Now 1512, 1520, and 1528 have their own large bin and 1504 has its own small bin. An alternative solution would be to turn cmerge on while excess 1504 chunks were being released. This should coalesce them into larger chunks put elsewhere.

For a system using a large variety of chunk sizes, an evenly spaced bin array, such as the standard bin configuration, is the best solution. However, if a system uses certain large sizes much more frequently than others, creating large bins that start with those sizes can greatly improve performance. For example, say a system uses predominantly 200, 400, 800, and 1200-byte blocks and a scattering of other sizes. Then the above heap bins could be optimized, as follows, for allocation of inuse chunks:

```
#define SMX_HEAP_BIN_SIZES \
/* small bins  0    1     2    3    4     5     6     7     8    9    10    11    12 */ \
            {24,  32,  40,  48,  56,  64,  72,  80,  88,  96, 104, 112, 120, \
/* large bins 13    14    15    16    17    18    19    20    21    22    23    24    25    26    27    28 */ \
            128, 208, 408, 512, 640, 768, 808,1024,1152,1208,1408,1536,1664,1792,1920,2048, -1}
```

Bold numbers indicate the bin sizes that have been changed. Due to the way that free() works, chunks of these sizes will always be put at the fronts of their bins. Hence, the next access for one of these chunk sizes is as fast as a small upper bin, even though these bins also contain other sizes. Due to taking last-freed chunk first, this also favors cache hits for access inside of these chunks, further improving application performance.

smaller bin arrays

For a smaller system, the following bin array might be adequate:

```
#define SMX_HEAP_BIN_SIZES     /* large bins  0    1      2      3      4  */ \
                                          {24, 512, 1024,1536, 2048, -1}
```

This covers the same range, as the standard array, with only 5 bins and the top bin (4) covers 2048 bytes and up, as in the standard array. Note that these are all large bins and that there is no SBA. Also dc is only 24 bytes and is not used. However tc still exists. Bins 1 thru 3 each cover 512 bytes and have 64 chunk sizes. Bin 0 is slightly smaller. Bins 0 thru 3 act like 4 small heaps. Finding the right bin takes up to 3 comparisons. Compared to the previous heap, it saves 288 bytes of memory for the bin array. If the bin sizes are chosen to be equal or slightly less than popular chunk sizes, then combined with sorting, this heap should be much faster and more deterministic than a simple linear heap.

To go even smaller, consider:

```
#define SMX_HEAP_BIN_SIZES     /* large bin   0   */ \
                                          {24, -1}
```

This defines a one bin heap. bin 0 handles all sizes from 24 bytes and up. This heap would be appropriate for a very small system, with a tiny heap space, such as 10-50KB. Like the previous heap, there is no SBA nor is dc used, but tc still exists. A one-bin heap has many of the eheap advantages over a simple linear heap:

(1) Only free chunks are linked into the bin, so it is not necessary to search through both inuse and free chunks. This alone should produce 5x faster allocations.

(2) tc provides a fast startup and is still the source of last resort.

(3) Bin sorting and deferred merging assure that small chunks can be found faster than large chunks.

(4) Debug and safety mechanisms are available.

With regard to (3) it can be argued that processing large blocks takes longer than processing small blocks, hence longer allocation times for them do not necessarily reduce system performance, overall. free() will put very small chunks at the start and all other chunks at the end of the free list. However, smx_BinSort() will quickly pop small to medium chunks back to where they belong. Hence, operation of the one bin heap could be pretty good.

Note that the bin size array, smx_bin_size[] is located in RAM, during debug, but should be in flash for deliverable systems, in order to improve reliability. (Unless, of course, flash is too slow and there is no cache for accesses to it.)

getting heap modes

Throughout Chapter 5 there is frequent mention of heap modes. These are single bit flags in smx_heap.mode. eheap provides direct access to and control of these heap modes. See the Reference Manual for usage examples. Accessing and controlling heap modes can be critical for customizing heap behavior and for debugging.

smx_HeapPeek(par) is used to get heap modes. par is of type SMX_PK_PARM. Available parameters are:

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

The return values are ON (1) and OFF (0). HeapPeek() returns 0 and reports SMXE_INV_PARM, if par is not one of these. This service is an SSR. Using it is highly recommended over directly reading chunk parameters, which may result in incorrect readings, due to preemption by another task.

It is also recommended to use smx_HeapPeek() to check modes, after setting, to verify that they are as expected. This service may also be of use in recovery operations.

setting heap modes

smx_HeapSet(par, val) is used to control heap modes. par is of type SMX_ST_PARM. Available parameters are:

SMX_ST_AUTO	Automatic free chunk merge control.
SMX_ST_DEBUG	Debug mode control.
SMX_ST_FILL	Pattern fill mode control.
SMX_ST_MERGE	Free chunk merge control.
SMX_ST_USE_DC	Use donor chunk control.

and the available values are ON and OFF. These modes are discussed in detail in other places. Briefly: amerge mode enables automatic control of the cmerge mode; debug mode causes allocations to create debug chunks; fill mode enables filling chunks or blocks with patterns by initialization, allocation, and free operations; cmerge mode enables merging free chunks by smx_HeapFree(), and use_dc mode enables using the donor chunk.

This service is an SSR. Using it is highly recommended over directly setting internal heap modes, which may result in incorrect settings, due to preemption of the current task.

Heap Debugging

need for strong debug support

Embedded systems are typically designed by small teams of engineers who have too much to do. However, delivering bug-free software is important because bugs in embedded systems can have serious, if not tragic, consequences. Debugging often makes or breaks projects and heap usage may frequently be the crux of the problem. Understanding how the heap works and having good visibility into its operations is important, because heap operations can be complicated and difficult to understand.

debug mode

To aid debugging, eheap has a debug mode. When debug is ON, allocations create *debug* chunks instead of *inuse* chunks. A debug chunk is a special form of an allocated chunk (in fact, its INUSE flag is set). Since debug chunks are typically much larger than inuse chunks it may not be possible for all allocated chunks to be debug chunks – especially in small heaps. For that reason they can be selectively controlled by turning the debug mode flag ON to create debug chunks and OFF to create inuse chunks. This can be done using smx_HeapSet(), as described above.

A heap may contain any mixture of debug, inuse, and free chunks. This can cause confusion when looking at the heap via the memory window of a debugger. If there is enough memory debug mode can be set permanently ON for debugging and permanently OFF for release. Then all allocated chunks will be debug chunks while debugging and will all look the same.

This solves another problem for inuse chunks because their headers are defined the same as for free chunks, to save code complexity. The header type, in both cases, is the Chunk Control Block, CCB:

fl	physical forward link.
blf	physical backward link + flags.
sz	chunk size, in bytes.
ffl	free forward link – used to link into bin free list.
fbl	free backward link – used to link into bin free list
binx8	bin number x 8.

As noted elsewhere, in an inuse chunk, all but the first two words are overwritten with data. Hence one must be careful to not be misled by values in the other words, which are data, not metadata, as the debugger suggests.

Debugging a heap with mixed debug and inuse chunks may be challenging, but it may be the only game in town, in a memory-constrained system. It is likely that inuse chunks will be used for debugged code and debug chunks for new code. Debug chunks, of course, are slower, so even if there is ample memory, it may still be preferable to have a mixture of inuse and debug chunks while debugging.

The header for a debug chunk consists of a Chunk Debug Control Block, CDCB:

fl	physical forward link
blf	physical backward link + flags
sz	chunk size, in bytes
time	time created — i.e. etime
onr	task or LSR that allocated this chunk
fence	SMX_HEAP_FENCE_FILL pattern (0xAAAAAAA3, as shipped)

fl and blf are common to all chunks. sz is the size of the chunk, not of the block. time is the value of etime when the chunk was allocated and onr is the task or LSR that allocated the chunk. Time and onr can be useful to track down memory leaks. For example the following chunks are suspect:

(1) An old chunk, unless it is a permanently allocated chunk.

(2) A chunk allocated by a task that has been deleted or stopped.

(3) An old LSR chunk, since it should have been passed on to another thread and freed by that thread.

Even if above are legitimate chunks, they may indicate poor coding practices that should be corrected.

The final field in the CDCB is the *fence*. It is a known pattern. Any pattern may be used, but bits 1 and 0 in the pattern must be 1, because they are the alternate DEBUG and INUSE flags, which are used when a chunk is accessed via its data block pointer (e.g. smx_HeapFree(bp)). These flags are necessary to determine the chunk type in order to determine where the chunk starts.

In addition to starting with a CDCB, a debug chunk has fences below and above its data block. Each fence is a word in size and has the same known pattern defined by SMX_HEAP_FENCE_FILL (xcfg.h). The number of fences, is controlled by SMX_HEAP_FENCES (xcfg.h). With the fence in the HDCB, there are FENCES + 1 fences before the data block and FENCES fences after it. These fences serve three purposes:

(1) To detect block and stack overflows.

(2) To show the overflow footprint, in order to help identify its source.

(3) To protect heap pointers so the system will continue functioning, if the overflow does not exceed the extent of the fences.

Since fences appear only in debug chunks, their number can be large. For a tough block overflow problem, it might be necessary to surround affected blocks with 10 or more fences in order to keep the system running and to see the footprint of the intruder. This could result in 200 bytes of overhead per debug chunk, which may not allow normal heap operation in the RAM available for the heap. In this case, debug mode would probably be turned ON only for one or two suspected tasks or functions.

One way to do this is to turn debug ON at the start of the task or of a function in the task. debug can be turned OFF in the *exit routine* hooked to the task and ON in the *enter routine* hooked to the task. This way debug is ON only when the task or function is actually running. If the same function is used in other tasks, then those tasks must also be hooked to the same enter and exit routines. The debug mode can be safely set or reset directly in hooked routines, since they cannot be preempted. For example:

```
smx_heap.mode.debug = ON/OFF;
```

Alternatively, SMX_HEAP_FENCES can be 0, in which case the debug chunk overhead is only the 24 bytes of CDCB. This would be useful if the problem at hand is a memory leak, not a block overflow.

If SMX_HEAP_FENCES is odd, the data block will be 4-byte aligned, instead of 8-byte aligned. This could cause a problem for code that expects the block to be 8-byte aligned. It could also alter cache performance. Such a problem could show up for one type of chunk and not for the other.

Also be aware that since debug chunks are bigger than inuse chunks, they may be put into higher bins, when freed, even though the block size has not changed. This could result in lower performance due to higher bins not being populated or of a different type (e.g. large bin vs. small bin). This may also cause confusion when looking at bins via a debugger window — i.e. chunks not being in their expected bins. This could cause wasted time trying to find a problem that does not even exist. On the positive side, the heap fences tend to clearly delineate data blocks. This is helpful when looking at a heap via a debugger memory window.

smx_HeapRealloc() poses another complexity, as follows: The resulting chunk type, when Realloc() is called, is determined by the value of debug mode then, not the value when the original chunk was allocated. Hence, an inuse chunk might be reallocated as a debug chunk or vice versa. Then, depending upon the debug chunk overhead, reallocating a block in a debug chunk to a larger size block in an inuse chunk may not require changing chunks.

Double free(): free() cannot detect a situation where a block is freed by task A, allocated to task B, double-freed by task A, then allocated to task C. As a result, tasks B and C will unknowingly be sharing a block. Only careful coding will avoid this problem. Use of debug chunks will help to identify the problem, because the debug chunk's owner will be task C, not B. Thus, if looking in task B to see why data is spuriously changing, check the chunk owner – B may not own the chunk.

heap fill

Sometimes there is no alternative but to look directly at the heap in order to track down heap usage problems. eheap has a fill mode, which fills unique patterns into data blocks when allocated, free blocks when freed, and dc and tc when the heap is initialized. These are helpful when viewing a heap in a debugger memory window. It is easier to see free, inuse, and debug chunks and to also see how much of an inuse data block has been used. It is also helpful to see how dc and tc are faring – i.e. plenty left or almost dry?

fill mode can be turned ON or OFF using smx_HeapSet(). Thus, heap fill is selective, like debug mode. It can be applied only to chunks of interest. Three heap fill patterns are defined in xcfg.h:

SMX_HEAP_DATA_FILL	0xDDDDDDDD
SMX_HEAP_FREE_FILL	0xEEEEEEEE
SMX_HEAP_DTC_FILL	0x88888888

smx is shipped with these values. They can be changed to whatever is preferred.

For the debug version (SMX_BT_DEBUG defined), fill mode is temporarily turned on in smx_HeapInit() so that dc and tc will be filled with the SMX_HEAP_DTC_FILL pattern. This is helpful during debug to easily find dc and tc in the debugger memory window and to see how they are faring, as the system runs. This is not done for released systems because filling dc and tc would increase boot time.

When fill mode is ON, the data blocks of inuse and debug chunks, are filled with SMX_HEAP_DATA_FILL when they are allocated. The fl and blf fields of an inuse chunk or the CDCB of a debug chunk are followed with the data fill pattern. This is helpful to see what chunks are allocated and how much of each data block is being used. The latter is useful to spot potential or actual data block overflows, or for downsizing data blocks. Not that if the chunk is a debug chunk, the onr field identifies the task or LSR that allocated it.

If a chunk is freed with fill mode ON, SMX_HEAP_FREE_FILL fills the rest of the free chunk after its CCB. So in the memory window, the CCB will be followed by the free fill pattern, making it easier to identify free chunks and their CCBs.

These patterns are helpful when looking at a heap through a debugger memory window — they make it a little easier to understand what is being seen. Having different fills enables quickly spotting what is free and what is inuse, while tracking down heap-related problems. Filling, of course, does reduce performance and is not recommended for released systems. However, since it is selective, it could be helpful for debugging heap problems in new code, without impacting other parts of a system.

heap error checks

As a further aid for debugging and system reliability, all heap service parameters are checked and invalid parameters reported. In most cases, services abort if an invalid parameter is found. These checks can be disabled for malloc() and free() if higher performance is desired, by defining SMX_HEAP_SAFE as 0 in xcfg.h. However this is not recommended, even for released systems.

The following errors are detected and reported:

SMXE_HEAP_BRKN	a heap error has been detected and cannot be fixed
SMXE_HEAP_ERROR	a double free has been attempted.
SMXE_HEAP_FENCE_BRKN	a broken fence has been found.
SMXE_HEAP_FIXED	a heap error has been detected and fixed
SMXE_INV_PARM	one or more parameters are invalid.

During debug putting a breakpoint at the start of smx_EM() to catch any of the above errors, when they first occur, is highly recommended. It can save wasted debug time chasing what appear to be serious errors, which in fact are just uninitialized pointers, a wrong sizes, etc.

getting heap information

When facing difficult heap debug problems, it may be helpful to write routines that take snapshots of the heap and report abnormalities. This is made easier by the services discussed below, which allow getting useful information about chunks and bins.

smx_HeapChunkPeek(vp, par) can be used to get information about chunks. vp is a chunk pointer in all cases, except for par == SMX_PK_CP, in which case it is a block pointer. If vp is out of heap range or not 4-byte aligned HeapChunkPeek() reports SMXE_INV_PARM and returns 0. The parameter, par, is of type SMX_PK_PARM. Available parameters are:

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

HeapChunkPeek() reports SMXE_INV_PARM and returns 0, if par is not one of the above. If the chunk is sc, PREV will return 0; if the chunk is ec, NEXT will return 0. If a chunk is the last chunk in a bin, NEXT_FREE will return 0. Similarly, if the chunk is the first chunk in the bin, PREV_FREE will return 0. If a chunk is inuse, it cannot be in a bin, thus 0 is returned. Since 0 is a valid bin number, the chunk should be tested for free.

This service is an SSR. Using it is highly recommended over directly reading chunk parameters, which may result in incorrect readings, due to preemption by another task or due to attempting to read an invalid field for the chunk type. As shown above, smx_HeapChunkPeek() returns 0 in the latter case. It usually is advisable to read the chunk type first to make sure that the expected chunk information is actually available. It also is advisable to check that the return value is not 0 before using it, except for bin number and chunk type, where 0 returns are valid.

The following example shows finding the bin number of a free chunk:

```
int  bin_num;
void *bp;
CCB_PTR cp;

bp = smx_HeapMalloc(100);
cp = smx_HeapChunkPeak(bp, SMX_PK_CP);
smx_HeapSet(SMX_ST_MERGE, OFF);
smx_HeapFree(bp);
bin_num = smx_HeapChunkPeek(cp, SMX_PK_BINNO);
```

In this example, a block is allocated from the heap, and the chunk pointer, cp, is obtained from the block pointer. Chunk merging is turned off so smx_HeapFree() does not merge the chunk with another free chunk, but rather puts it into the correct bin for its size. Then smx_HeapChunkPeek() is used to find out which bin the chunk was put into. This would not be a trivial exercise for a debug chunk.

smx_HeapChunkPeek() is useful for heap integrity checking and heap maintenance. For example, if a debug chunk is owned by a task that has been deleted or stopped then a heap leak has been found. If a block that is about to be freed is already free then a double free has been detected. Implementing tests like these can significantly reduce debug time – especially when adding *SOUP* to projects. This kind of testing may also be of value in released systems to improve their reliability.

smx_HeapBinPeek(binno, par) can be used to obtain information concerning bins. binno is the bin number. If it is not in the range 0 to smx_top_bin, SMXE_INV_PARM is reported and 0 is returned. The parameter, par, is of type SMX_PK_PARM. Available parameters are:

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

HeapBinPeek() reports SMXE_INV_PARM and returns 0, if par is not one of the above values. It also returns 0 if the bin is empty, except for SIZE. This service is an SSR. Using it is recommended over directly reading bin parameters. The latter may result in incorrect readings due to preemption by another task.

Determining chunk counts in bins is useful in order to keep bins from becoming empty, as in the example in the bin seeding section of Chapter 5, or too full. Totaling up free space in all bins or in all small bins might be used to control cmerge.

heap statistics

The following are accumulated as the heap runs:

smx_heap_hwm	Heap high water mark. Records maximum number of bytes used.
smx_heap_used	Number of bytes of the heap currently allocated.

These are globals accessible via a debugger and also via smxAware. If the high water mark becomes nearly equal to SMX_HEAP_SPACE (acfg.h), the latter should be increased, if possible. It is advisable to do long system runs in order to make this adjustment.

If SMX_HEAP_STATS (xcfg.h) is 1, the following heap statistics are accumulated:

smx_bnum[]	Number of chunks in each bin
smx_bsum[]	Sum of chunk sizes in each bin

These arrays can be viewed through a debugger watch window to provide some of the same information as the smxAware heap window. (See smxAware User's Guide.)

Heap Reliability

There is obviously a big overlap between debugging and reliability because the better debugged a system is, the more reliable it is likely to be. *Self-healing* goes beyond this, in that it provides protection from environmental factors and malware.

self-healing

With increasing IoT deployment, self-healing is becoming more necessary. General-purpose systems are generally housed within concrete buildings that provide protection against environmental factors. In contrast, embedded systems are often deployed at high altitudes or high latitudes, where high-energy particle fluxes are large. Also embedded systems are likely to be less protected, possibly right out in the open, subject to temperature extremes and EMI from thunderstorms, sunspots, etc. Ever smaller semiconductor feature sizes are exacerbating this problem.

Heaps are especially vulnerable because of their large concentrations of pointers and control information in the data area. For a heap, with an average chunk size of 48 bytes, control information constitutes about 20% of the memory space. A block pool, by comparison, is about 8%. The reason this is important is that one bit flip in a pointer or size field is likely to put the system into the weeds and cause a system crash. A bit flip in a data word is not likely to be so catastrophic because data can be bounds checked and can be rejected if invalid.

In addition to bit flips, heaps are highly vulnerable because control information (i.e. the CCBs) is sandwiched between data blocks. Data block overflows damage the control information, again sending the system into the weeds. This kind of damage usually result from programming errors or malware. Typically data buffers overflow in the up direction and stacks overflow in the down direction, so neither end of a data block is safe.

We are accustomed to reloading our applications or rebooting our desktop computers whenever misbehaviors occur. This not an option for most embedded systems because they are unattended and expected to run forever. Hence, heap self-healing is desirable for embedded systems. eheap accomplishes self-healing by continuously scanning the heap and bins, fixing broken pointers, wrong sizes, etc., whenever possible, or sounding an alarm if not possible. The latter helps to achieve a soft landing by fixing the problem before a malloc() or free() encounters it and crashes.

Assuming a modest heap of 10,000 chunks, a malloc() or free() will not use more than 6 pointers and one size, so the probability of it encountering a damaged pointer or size field is 7 in 26,000 = ~.027%. If 100 mallocs and 100 frees occur in the time needed for one scan, the probability of failure is about 5% — not perfect, but better than 100%. Of course, this can be improved by increasing the scan rate.

heap scan

smx_HeapScan(cp, fnum, bnum) is like a night watchman – a slow, but trusted patrol looking for trouble. It is intended to perform continuous forward heap scans and to fix heap problems, or report ones it cannot fix. To accomplish this, it is called once per pass of the idle task, so that it will not consume valuable processing time. It scans each chunk from the start of the heap to the end, fixing broken backward links, flags, sizes, and fences, as it goes.

In the debug version (SMX_BT_DEBUG defined), the scan stops on a broken fence so that the fence can be studied for clues as to what happened. In the release version, the fence is fixed. Fixes are reported and saved in the event buffer, for later analysis. This can be valuable for systems in the field, in order to monitor stresses and behaviors. During scans, all pointers are *heap-range tested* before use, in order to avoid *data abort exceptions*, or equivalent, if they are broken.

If a broken forward link is found and the chunk is either a free chunk or a debug chunk, the sz field is used to attempt a fix. (Chunk size is effectively a redundant forward link.) Otherwise, a backward scan is started. It proceeds from the end of the heap until the break is found and the forward link is fixed. Then the forward scan is resumed until the end of the heap is reached.

While back scanning, other broken forward links are fixed, if encountered. However a broken backward link stops the back scan and a *heap bridge* is formed between the chunk with the broken forward link and the chunk with the broken backward link. When this happens, many chunks of all kinds may be bridged over. Bridging serves to allow the scan to complete and may give the system a chance to limp along and to possibly fix itself, but generally it does not fix the problem. Therefore SMXE_HEAP_BRKN is reported.

High-priority tasks must not be blocked from running for too long, else they may miss their deadlines. Since smx_HeapScan() is an SSR, it cannot be preempted and if the heap is big, it could run for a very long time. Consequently smx_HeapScan() operates incrementally. When forward scanning, it moves *fnum* chunks per call; when backward scanning, it moves *bnum* chunks per call. These are called *runs*. Thus a *scan* is normally consists of many runs and a run consists of examining fnum or bnum chunks. bnum is usually larger than fnum because backward scanning is faster and, once a break has been found, there is urgency to fix it. So, for example fnum might be 2 and bnum might be 100.

A run will go the specified number of chunks then return FALSE, unless it is done or encounters an error that it cannot fix. smx_HeapScan() can be called repetitively, as follows:

```
while (!smx_HeapScan(NULL, 2, 100))
```

This will scan from the start of the heap to the end of the heap, one run at a time, unless an unfixable error is encountered, in which case it stops and returns TRUE. It keeps going for all fixable errors. The NULL parameter means to start from *smx_hsp* (heap scan pointer), for each run. At the end of each run, smx_hsp, is set to point to the next chunk to scan. Should a preempting free() merge the chunk pointed to by smx_hsp with a lower chunk, the free() changes smx_hsp to point to the lower chunk. Then the next run will start with it, rather than with potential garbage. Other frees and mallocs do not effect smx_hsp. During initialization and when a scan is completed, smx_hsp is set to NULL, causing the next smx_HeapScan() to start from the beginning of the heap.

In order to not consume too much idle time, it be desirable to space heap scans out – e.g. one per second or one per minute, etc. smx_HeapScan() stops and returns TRUE, when it reaches the end of the heap, in order to facilitate this.

smx_HeapScan() is normally called once per pass through smx_IdleMain(). Hence, heap scanning is a slow, continuous process that makes use of idle time to increase heap reliability. fnum can be adjusted upward to assure a higher probability that a break will be found by the scan before it is found by a heap service. Yet it should be possible to adjust fnum low enough that the impact on task latency is negligible. A value of 1 or 2 will probably suffice in most situations.

smx_HeapScan() can be called directly from an application task, as well as from idle. This might be done when another heap service reports an error, as follows:

```
smx_HeapScan(cp, 10000, 10000);
while (!smx_HeapScan(NULL, 10000, 10000))
```

The scan will start from cp, which might point at the chunk in question. As shown, fnum and bnum are likely to be high values so scanning will finish quickly. smx_HeapScan() will report SMXE_HEAP_FIXED, if it succeeds. An alternative to the above code is:

```
smx_HeapScan(cp, 1000, 1000));
while (smx_ct->err != SMXE_HEAP_FIXED)
    smx_HeapScan(NULL, 1000, 1000));
```

This code makes shorter runs and stops when the problem is fixed (which could be a different problem, but that is unlikely).

Because it is expected to run frequently, smx_HeapScan() makes no entries in the event buffer, other than those due to reported errors or fixes. Whenever a fix is made, SMXE_HEAP_FIXED is reported. This can be used to monitor how often problems are being found and fixed.

heap bin scan

Whereas bins are in local memory and thus may be less susceptible to bit flips, most of the bin free list pointers are in the free chunks, themselves, out in the heap, and thus are just as susceptible to damage as the heap forward and backward links. So, bin free list scanning is necessary to provide a consistent level of heap protection. **smx_HeapBinScan(binno, fnum, bnum)** performs this function. It is similar to smx_HeapScan(): it is also an SSR, incremental, scans doubly-linked chunk lists, and fixes broken links, when it can. It has three parameters: binno, the bin number, and fnum and bnum, the forward and backward run limits. Like heap scan it returns FALSE until it is done with the bin or an unfixable error is encountered. See RM smx_heapBinScan() for a usage example.

If binno is greater than the top bin number or fnum or bnum are 0, SMXE_INV_PARM is reported and smx_HeapBinScan() returns with TRUE. If the parameters are valid, bin scanning begins. If the bin is empty, its free back link, fbl, is checked and fixed, if necessary, and TRUE is returned. Otherwise, the bin free forward link, ffl, is checked. If it is out of heap range, both bin links are set to NULL, the bin's bmap bit is set to 0, SMXE_HEAP_BRKN is reported and TRUE is returned. The bin is now effectively empty and the chunks that were in its free list cannot be allocated. However, normal operation can continue and, if cmerge is ON, the lost free chunks may eventually be recovered through merging with other chunks.

Barring the foregoing, a bin scan starts at the beginning of the bin free list and checks fnum chunks. As with a heap scan, a global pointer, *smx_bsp* (bin scan pointer) maintains the place to resume the next run. Runs continue until the end of the bin free list reached, then TRUE is returned. Broken fbls are fixed as encountered. If a broken ffl is found, smx_HeapBinScan() scans backward to fix it and *smx_bfp* (bin fix pointer) maintains the place to resume the next run.

Like eheap scan, double breaks are bridged. In that case, the bridged chunks are no longer available to be allocated, but heap operation can continue normally, if cmerge is OFF. Bridging is a partial solution, but SMXE_HEAP_BRKN is still reported. If cmerge is ON, free() may fail when it attempts to merge a bridged chunk with a broken pointer, but. in some cases, the merge will proceed ok.

If a preempting malloc() uses a bin being scanned, the forward or backward bin scan in progress is aborted and forward scan is restarted from the start of the bin.

Because it is expected to run frequently, smx_HeapScan() makes no entries in the event buffer, other than those due to reported errors or fixes. Whenever a fix is made, SMXE_HEAP_FIXED is reported. This can be used to monitor how often problems are being found and fixed.

broken heap

If SMXE_HEAP_BRKN is reported, the heap is pretty much kaput. It can be dealt with by stopping all tasks using the heap, reinitializing the heap, then restarting all tasks that use the heap. Naturally, this will cause a large hiccup for functions requiring the heap. However, if high-priority, mission-critical tasks use only block pools, this can be a workable, worst-case solution – valuable data and processing might be lost, but the system continues performing its critical mission. By taking action before malloc() or free() encounter broken links the system is saved from going into the weeds and failing.

The amount of time and code that should be devoted to ruggedizing the heap is application dependent. It may not be of much importance for applications in protected environments that can be conveniently rebooted. It may be of extreme importance in harsh, remote environments where rebooting has serious consequences. The foregoing services provide some basic tools, but additional tools may be needed.

Note that as object-oriented languages make further inroads into embedded systems, ruggedized heaps will become more important, because these languages are heavy heap users.

Chapter 24 Error Management

introduction

All smx services return FALSE or NULL, if an error or timeout has occurred. The cause can be determined by checking smx_ct->err or by using the macro SMX_ERR. This enables local error handling. smx also implements central error handling via the smx_EM() error manager. Both types of error handling are discussed below, followed by a discussion of which to use.

error detection

smx detects about 70 types of errors — see xdef.h for a complete list. The errors detected for each smx service are specified in the smx Calls section of the Reference Manual. They are described fully in the Glossary of the Reference Manual. Errors fall into the following categories:

(1) Invalid parameter.

(2) Out of a resource.

(3) Miscellaneous.

All smx service parameters are checked. Handles are tested to be in the correct range and have the correct control block type. This avoids accepting NULL handles and handles for wrong objects (e.g. a semaphore instead of a mutex). Non-handle parameters are checked primarily for zero and in some cases for being too large.

Out of resources is a common problem during development — particularly out of control blocks of a particular type. These errors are easily resolved by allocating more of that resource. Miscellaneous errors include stack overflow, other overflows, broken queues, and procedural problems such as attempting to wait in an LSR, wrong mode, wrong type pool, excess unlocks, etc.

When an error is detected in an smx service, the service is aborted and there are no side effects.

stack overflow detection

Stack overflow is a particularly pernicious problem in multitasking systems because there are so many stacks — one stack per active task. In addition, stack overflows are hard to diagnose — they may cause another task to misbehave or cause some completely unrelated function to fail. smx provides three methods to deal with stack overflow:

(1) Overflow detection.

(2) Stack scanning.

(3) Stack pads.

Overflow detection applies to task stacks. It is performed in the scheduler when a task is suspended or stopped. Stack scanning applies to the system stack (SS) as well as to task stacks. Stack pads are placed above stacks and enable a system to continue running, despite stack overflows. Thus it is easier to find and fix them. This is useful during debug. See the Stacks Chapter for detailed discussions.

stack switching

When an error is detected, smx_EM() is automatically called, and the error number is passed to it, as a parameter. For the debug version of the application (SMX_BT_DEBUG defined in the project file preprocessor tab), smx_ERROR() calls smx_EM() using the current task stack. This allows using the call stack window of the debugger to see where the error originated. SMX_BT_DEBUG is passed to smx, at run time, via smx_cf.app_debug = 1.

For the release version, smx_EM() is called after switching to the system stack (SS). In this case, SMX_BT_DEBUG is not defined and smx_cf.app_debug = 0. Using SS saves about 60 bytes per task stack. More importantly, it avoids a situation where smx_EM(), itself, causes a stack overflow error — i.e. one error spawns another. After smx_EM() finishes, a switch is made back to the task stack.

Note: Switching to SS uses the SVC exception for the ARM-M architecture.

error number, errno

Each error type has a unique error number which is recorded in smx_errno and in the err field of the TCB of the task that caused the error. smx_errno stores the last error that occurred in the system, whereas tcb.err stores the result of the last smx service called by the task. It can be accessed with the SMX_ERR macro. If SMX_ERR == SMXE_OK (0), no error has occurred and the return value from the smx service is valid. If SMX_ERR == SMXE_TMO (1), a timeout has occurred. Otherwise, an error has occurred and SMX_ERR is the error number. See the SMX_ERRNO enum in xdef.h for error names. Use these names rather than numbers, because numbers are likely to change in future releases.

central error processing overview

As noted above, the global, smx_errno, is the last smx error that has occurred in the system. The global 32-bit error counter, smx_errctr, is incremented for every error detected. Each error also has its own counter in the smx_errctrs[] array. To minimize RAM usage, each of these counters is only 8 bits. If a counter registers more than a few errors, the exact number probably does not matter. If it is important, the sum of all error counters can be compared to smx_errctr to determine if any have overflowed and, if so, by how much. Each error is also recorded in the error buffer, EB, and the event buffer, EVB, if they are enabled. These are discussed below.

After the above operations, an error message is output to the console. These messages are contained in xem.c. They are short in order to save ROM. Note that console output is buffered and will not display until the idle task runs or sb_MsgDisplay() is called. Hence, when stepping through code, it is more reliable to watch smx_errno in the globals watch window of the debugger in order to see errors. Following this, the Error Manager hook routine, smx_EMHook() is called, if present. Depending upon the severity of the error, smx_EM() returns to the point of call or to the scheduler, or it calls smx_EMExitHook(), which normally reboots. These are discussed, below.

smxBase has a similar, but simpler, error manager. It stores the most recent error number in sb_errno, increments the error counter sb_errctr, and outputs an error message to the console. It does not have an error buffer and does not store error information in TCBs. Keep in mind that since smx services use smxBase services, the smxBase service may be the actual cause of failure. If an smx service calls an smxBase service, which has an error, smx_ERROR() is called to register that an smx error has occurred. As a consequence, SMX_ERR will be greater than SMXE_TMO (=1).

Tip: Add smx_errno and sb_errno to the watch window in the debugger to be aware if an smx or smxBase error has occurred. Alternatively, uncomment sb_DEBUGTRAP() in smx_EMHook() in main.c and in sb_EM() in bbase.c to make the debugger breakpoint on an error. The call stack can be inspected to see the path leading to the error.

error buffer (EB)

EB, is a global data structure, which stores the following information for each error:

(1) Time of occurrence (etime)

(2) Error number

(3) Task or LSR for which the error occurred or 1 if a system error

EB is allocated from SDAR, during initialization. The number of records in EB is determined by EB_SIZE in acfg.h. If EB_SIZE is zero, EB is not allocated and smx_EM() is inhibited from storing error records. EB can be viewed via smxAware. In the debugger, smx_ebn points to the most recent error record. During debug, it may be desirable to set EB to a large size. For release, a much smaller size of 10 or less records is probably adequate. The record size is 16 bytes.

event buffer (EVB)

Event logging is enabled by SMX_CFG_EVB in xcfg.h. Logging of errors must also be enabled by the SMX_EVB_EN_ERR flag in smx_evben. The information logged in EVB is the same as EB, with the addition of precise time. See the discussion of event logging in the Debug Chapter and also see the smxAware User's Guide. The advantage of storing error records in EVB is that there is a clear relationship relative to other events. In smxAware, errors show up as red dots on timelines. Since EB contains only error records, it is easier to get a quick view of what errors have been occurring.

error manager hook

smx_EMHook() is called from smx_EM() to allow user code to be included in central error management. It is called after the above variables have been set and error information has been loaded into the error buffer (EB) and the event buffer (EVB). smx_EMHook() is located in main.c. As shipped, it serves just to allow setting a breakpoint in the error manager. See discussion in the Debug Chapter.

This is a place where error-specific handling, based upon the errnum and handle parameters, can be introduced. Since this code runs under smx_EM(), it is non-preemptible like smx_EM(). As a consequence, it is task-safe and LSR-safe, but it also can cause priority inversions — i.e. high priority tasks blocked from running by low-priority task errors. Hence, it should be kept safe.

An apparent solution to this problem would be to load the error information into a message and send it to a pass exchange where an error task waited. This task could perform more extensive error analysis and reporting, without impacting higher-priority tasks. However, a flaw in this plan is that the send service, itself, might encounter an error. This would result in reentering smx_EM() and smx_EMHook(), which is likely to cause a system failure. Hence, a solution not using smx services should be found.

error manager exit hook

smx_EMExitHook() is called from smx_EM() if an irrecoverable error is detected. It should be used to shut the system down and reboot. Since the system is in a damaged state, it is inadvisable to make smx service calls, since these may result in putting the system into an infinite loop if they, themselves, experience errors.

It is probably best to assume that smx is damaged beyond use and to invoke and immediate hardware shutdown. sb_MsgConstDisplay() might be used to output error messages to the console.

standard error handling

Errors are grouped into three levels, by severity:

(1) Record and report, only.

(2) Restart current task.

(3) Reboot.

For the first category, no further processing is needed. The second category occurs when a stack overflow has not damaged another stack. The third category occurs when a stack overflow has damaged another stack or some other non-recoverable error has occurred. For this category smx_EMExitHook() is called and it is up to the user to decide what to do – e.g. reboot the system. the system is always rebooted. Rebooting consists of calling smx_exit(), which can be mapped to an appropriate function. In the Protosystem, it is mapped to aexit(), which attempts to shut the system down cleanly, then calls sb_Exit(), which calls sb_Reboot(). The latter loops in most BSPs. This is intended for debug and should be changed to cause an actual reboot for released systems.

local error handling

All, but a few, smx calls return 0, FALSE, or NULL when the expected result does not occur. This can be due to a detected error or to a timeout. This smx feature permits point-of-call error handling to be implemented, as in the following example:

```
XCB_PTR port_in;

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

    // message processing loop
    while (1) {
        if (msg = smx_MsgReceive (port_in, &mbp, TMO)) {
            // process msg using mbp
            if (!smx_MsgRel(msg, 0))
            break;
        }
        else {
            if (SMX_TMO)
                // handle timeout -- break if too many timeouts.
            else break;
        }
    }

    // error handling code
    switch (taskA->err)     {
        case SMXE_INV_XCB:
            // exchange handle is out of range
            ...
            break;
        case SMXE_INV_PRI:
            // msg priority is not in range
            ...
            break;
        case SMXE_INV_MCB:
            // msg handle is not valid
```

```
                smx_TaskStart(self);
            case SMXE_TMO:
                // too many timeouts
                ...
                break;
            default:
                // unknown error
        }
    }
```

In this example, the while (1) loop does the main processing. It waits for a message at the port_in exchange. If a message is received within TMO ticks, it is processed, then released, and control goes back to message receive. If a message is not received, due to an error or timeout, control goes to the else statement. If a timeout (SMXE_TMO) has occurred, it is handled, and control goes back to message receive. Otherwise control breaks out of the while (1) loop and goes to the switch statement. Here there are cases for all smx_MsgReceive() error types, as well as a case for too many timeouts.

After error processing (shown by “...”), taskA_Main() returns to the scheduler, which stops it. This insures that taskA will not cause further damage until the problem can be fixed or diagnosed, if debugging. When the problem has been fixed, taskA can be restarted and it will go back into its main processing loop.

taskA can also be restarted, without stopping, as shown for the invalid MCB case. In this case, the message is discarded and taskA goes back to its main processing loop. There is no “...” because restarting taskA is the error processing. Also, there is no break after task restart, because no statements after it, will execute.

This example shows how to distinguish a timeout from an error and how to distinguish error types. Note that timeout handling, which probably requires notification then retry, stays in the while loop, whereas error handling is performed outside of the while loop. This avoids cluttering the main code, thus making it easier to understand. The error handling code might be fleshed out later, or possibly just used to set breakpoints during debug.

Another example of incorporating point of call error handling code:

```
MCB_PTR  net_msg;
PCB_PTR  network_msgs;
u8 * mp;

if ((net_msg = smx_MsgGet(network_msgs, &mp, 0)) != NULL)
    /* fill net_msg and send it on */
else
    sb_MsgOutConst(SB_MSG_ERR, “Send failed because no free network messages”);
```

sb_MsgOutConst() simply enqueues a pointer to the message in sb_omq for later output and thus adds very little overhead. Alternatively, use sb_MsgConstDisplay() for immediate output. This application-specific message could be helpful in tracking complicated protocol stack problems. Later it may become apparent what corrective action is necessary:

```
while (1)
{
    if ((net_msg = smx_MsgGet(network_msgs, &mp, 0)) != NULL)
        /* fill net_msg and send it on */
    else
        /* delay n ticks */
}
```

Even though the exact reason why the network_msgs pool becomes empty is not understood, the above code might enable the application to recover and to run reliably.

deciding what to use

If properly used[8], many system paths will flow through smx, and thus smx is in a good position to detect errors and unexpected conditions. The smx error manager adds negligible code and execution overhead to a typical system. In addition, a reasonable amount of point-of-call exception handling code can be added.

Point-of-call error management tends to add complexity to the main code and to make it larger and slower. However, for example, networking software running out of free messages might be a common occurrence. Hence, dealing with it at the point of call is necessary. Even if it is a rare occurrence, the point-of-call error reporting might be a life-saver when tracking down an infrequent networking problem.

For other types of software, once debugging is done, most errors should never occur. Timeouts might be the only frequent occurrence to deal with, and they can be handled rather simply. In this case, relying on central error management may be best because it does not add much overhead and does not complicate the main code. Yet, it is possible to see what, if any, errors are occurring in released systems and how frequently they are occurring. It is also possible to see what task or LSR is involved.

Using smx_EMHook() provides a middle ground, where additional information can be gathered for specific errors and error-specific or task-specific recoveries can be initiated.

[8] Dividing an application into many tasks ensures that smx will be used extensively for intertask communication and coordination. This enables smx to detect more system faults.

Chapter 25 Resource Management

access conflicts between tasks

Although there is only one physical processor, preemption of one task by another means that tasks may compete for the same resource. This causes a problem if resources are non-sharable. Resource management is a means of preventing access conflicts to a resource while granting access to the resource. smx provides several methods to do this:

(1) same priority tasks

(2) semaphores

(3) mutexes

(4) exchanges

(5) task locking

(6) server tasks

(7) server LSRs

(8) data hiding

(9) non-preemptible tasks

The material that follows has been presented in previous chapters, but is useful to bring it all together in one place since resource conflicts can cause serious problems that may be difficult to find.

same priority tasks

The simplest and lowest overhead method of avoiding resource conflicts is to give all tasks using a particular resource the same priority. Also no task may suspend or stop itself while using the resource. Surprisingly often, this is a practical solution. However, it will fail if a priority is later changed.

semaphores

What might be called the textbook approach to resource management is to use a binary resource semaphore per resource. For example:

```
SCB_PTR  print_ok;

void  init_function(void)
{
    print_ok = smx_SemCreate(SMX_SEM_RSRC, 1,  "print_ok");  /* binary resource semaphore */
    smx_SemSignal (print_ok);
}

void  client_function(void)
{
    smx_SemTest (print_ok, INF)
    /* put bytes to print_out pipe */
    smx_SemSignal (print_ok);
}
```

The above code (in client_function()) appears in every task writing bytes to print_out pipe. Hence only one task at a time can write to this pipe. Note that the print_ok semaphore is a binary semaphore. This ensures that it cannot count above 1. If a limit greater than one were used, it would be necessary to ensure that only one smx_SemSignal() could occur per smx_SemTest(). This rule is crucial, because if an extra signal were sent to print_ok, the next smx_SemTest() would pass even if the resource is not available. A binary semaphore avoids this problem, since extra signals are ignored.

This textbook solution may not always be the best solution: It results in increased task switching and it is susceptible to deadlocks and unbounded priority inversion.

mutexes

Mutexes accomplish the above in a similar, but safer manner.

```
MUCB_PTR  print_ok;      /* mutex */

void  init_function(void)
{
    print_ok = smx_MutexCreate(PI, P0, "print_ok");
}

void  client_function(void)
{
    smx_MutexGet (print_ok, INF)
    /* put bytes to print_out pipe */
    smx_MutexRel (print_ok);
}
```

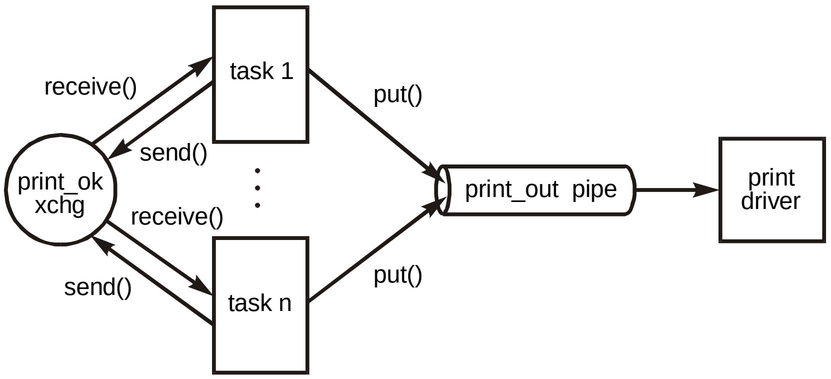
In this example, priority inheritance is enabled. Hence, if a low priority task is currently printing and a high priority task wants to print, the low priority task’s priority will be promoted to high priority until it releases the print_ok mutex. This eliminates unbounded priority inversion. The ceiling has been set to 0. If it were known that no print tasks had a priority greater than 3, then the ceiling could have been set to 3, and all printing would be done at this priority level.

```
    print_ok = smx_MutexCreate(NO_PI, P3, "print_ok");
```

An advantage of this is to avoid deadlocks because the current owner cannot be preempted by any other task, which also wants this resource.

Mutexes have higher overhead, but avoid most of the problems of semaphores. See the Mutexes Chapter for more discussion.

exchanges



The use of exchanges for resource management involves assigning a token message per resource and an exchange to wait for a message. An advantage of this method is that the token message can contain information needed to use the resource — e.g. port number, speed, maximum packet size, type of error control, etc. The message can also accumulate usage information — e.g. number of sends, number of receives, average message length, average retries, etc.

When a task requires access to a resource, it waits at its exchange for a token message. When the task currently using a resource finishes, it sends its token back to the exchange. The waiting task is resumed with the token and can now access the resource:

```
XCB_PTR  print_ok;
PICB_PTR  print_out;

void  print(void)
{
    MCB_PTR  ptoken;
    u8 *mp:

    if (ptoken = smx_MsgReceive (print_ok, &mp, INF))
        /* use mp to access printer information */
        /* put bytes to print_out pipe */
    smx_MsgSend(ptoken, print_ok);
}
```

If there were multiple printers, there would be multiple messages. Each message would identify which pipe to fill. The message might also have other useful information such as printer characteristics, margins, control characters, etc. Normally, there would be only one printer in a system. However, even so, using token messages allows changing to a printer with different characteristics merely by changing the message. (This assumes, of course, that print() has been written to handle different printers.)

The printer exchange could be a normal exchange or a pass exchange. The use of a pass exchange offers the interesting possibility of assigning priorities to resources. (i.e. The client task would assume the priority of the printer which is carried in its message.)

task locking

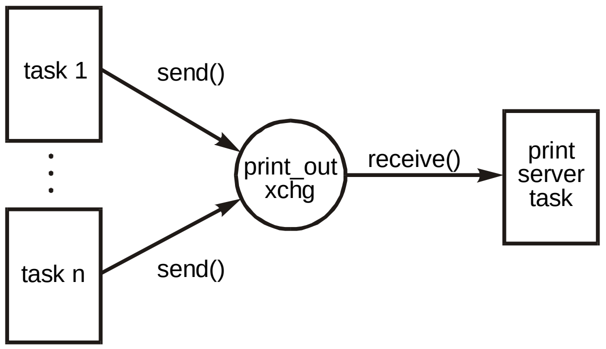
is a simple technique for protecting a critical section of code or using a resource:

```
smx_TaskLock();
/* use resource */
smx_TaskUnlock();
```

This is simpler than using a semaphore, a mutex, or an exchange and is also faster. The disadvantage is that it is non-specific — i.e. it blocks all higher priority tasks even if they are not trying to access the resource. As a consequence, task locking is best used for short critical sections of code, not for resource management. This is because resources are normally used for relatively long periods of time.

Be careful not to do any calls in a critical section which could suspend the current task, because the lock counter will be reset and the critical section will no longer be protected, when the task resumes. See the Tasks Chapter for more on task locking.

server tasks



In the fourth approach, a single server task is assigned to a resource. To access the resource, a client task must prepare a message and send it to an exchange, where the server task waits. Typically, the exchange is a pass exchange. The server task waits at the exchange for a message. When it receives a message, the server task accesses the resource and performs whatever is required by the message. For example:

```
PCB_PTR  pmsgs      /* print message pool */
XCB_PTR  px;        /* printer pass exchange */
SCB_PTR  Bgo;       /* binary event semaphore */

void  clientA_main(void)
{
    MCB_PTR  rpt;
    u8  *mp;
    u32  next_hour;

    rpt = smx_MsgGet (pmsgs, &mp, 0);
    /* load report A into rpt using mp */
    smx_MsgSendPR (rpt, px, LOW, NO_REPLY);
    next_hour = (smx_StimeGet()/3600 + 1) * 3600;
    smx_TaskSleepStop(next_hour);
}

void  clientB_main(void)
{
    MCB_PTR  rpt;
    u8  *mp;

    do
    {
        rpt = smx_MsgGet (pmsgs, &mp, 0);
        /* load report B into rpt using mp */
        smx_MsgSendPR(rpt, px, HI, Bgo);
    } while (smx_SemTest (Bgo, 10));
    /* handle timeout */
}

void  print_rpt_main(void)       /* server task */
{
    MCB_PTR rpt;
    u8  *mp;
    SCB_PTR  rsem;
```

```
            u32 sz;

            while ((rpt = smx_MsgReceive(px, &mp, INF)) != NULL)
            {
                /* print msg using mp */
                if ((rsem = (SCB_PTR)smx_MsgPeek(rpt, SMX_PK_REPLY)) != NULL);
                    smx_SemSignal(rsem);
                sz = smx_PeekMsg(rpt, SMX_PK_SIZE);
                smx_MsgRel(rpt, sz);
            }
        }
```

This example shows two different clients sharing the print_rpt server. print_rpt waits at the px pass exchange. (A normal exchange would be used if priorities were not important.)

Once an hour, clientA wakes up. It gets a message from the pmsgs pool, loads report A into it, and sends it to the px exchange, with LOW priority and no reply expected. It then sleep stops for an hour. During this time, it consumes no stack. These hourly reports are not urgent and a minimum of resources is thus allocated to them.

clientB is different. It also gets a message from the pmsgs pool, loads report B into it, and sends it to px. However, in this case, it gives HI priority to the message and requests a reply in the form of a signal to Bgo. It then waits at Bgo for the signal and resumes when it is received. As shown in this example, clientB will run as fast as the printer until it runs out of report B's to print.

The print_rpt server task resumes when it receives a message from px. For an A report, it assumes LOW priority; for a B report, its assumes HI priority. It prints the report then tests if a reply is required. For an A report, there is no reply, but for a B report, rsem must be signaled. print_rpt obtains the message size, then clears and releases the message back to the pmsgs pool (determining the pool from rpt is automatic). It is important to do all operations on rpt (peeks in this case) before releasing it, otherwise incorrect values may result. To assure this, it is a good practice to do the release at the very end. Also combining operations — e.g.:

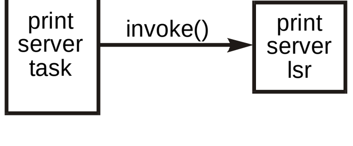
```
smx_MsgRel(rpt, smx_PeekMsg(rpt, SIZE));
```

is not recommended, for the same reason.

Task action is a little tricky in the above example. If an A report is waiting at the px when print_rpt finishes printing a B report, print_rpt will still have HI priority when it receives the A report. However, its priority will then drop to LOW and ClientB will preempt to prepare another B report and send it to px. ClientB will then suspend on Bgo and print_rpt will print the A report, then the next B report. This illustrates that actions may not occur as expected, unless one pays close attention to priorities.

Server tasks are a nice way to manage major resources such as printers, graphics displays, and others. Since the server task is the only task which runs the code to operate the resource, the code need not be reentrant. The exchange serves as a priority work queue (which is a nicely organized way of going about doing things). The client tasks need not be concerned with the details of handling the resource — they just create messages in the expected format. This is nice because changing of the resource (e.g. a printer) can be handled merely by activating a different server task. All in all, this is a good structure for many uses.

server LSRs



To access a resource controlled by an LSR the LSR must be invoked. This can be done by a task, ISR, or LSR. A handle can be passed to the LSR as in the following example (In this case, the SSR version of LSR invoke is used since report_n() runs as a task.):

```
PCB_PTR  free_msgs;

void  make_report_main(void)      /* client task */
{
    u32  next_hour;
    MCB_PTR  report;
    u8*  mp;

    report = smx_MsgGet(msg_pool, &mp, 0);
    /* load reporthere, using mp */
    smx_LSRInvoke(printLSR, (u32)report);
    next_hour = (smx_SysStimeGet()/3600 + 1) * 3600;
    smx_TaskSleepStop(next_hour)
}

void  printLSR(u32  m)              /* server LSR */
{
    u8* mp;
    MCB_PTR msg = (MCB_PTR)m;

    mp = (u8*)smx_MsgPeek(msg, SMX_PK_BP);
    message_print(mp);
    smx_MsgRel(msg, 0);
    }
```

The print LSR will execute as soon as it is invoked — even if the report task is locked. It also effectively has top priority over all other tasks. Unlike a function call, printLSR cannot be preempted. Even if it is interrupted, it will complete executing before another LSR, which was invoked by the interrupt, begins running. These statements would not be true for a subroutine called by report.

A server task usually runs after its client tasks. A server LSR, on the other hand, runs immediately, if invoked from a task, but not if invoked from an ISR or an LSR. It is simpler to use, but its main attraction is that it resolves foreground/background access conflicts — i.e. those due to a resource being used from both foreground and background. Like a server task, a work queue can build up for a server LSR, in the form of LSRs waiting in lq. This could happen if the LSR were interrupted and invoked again by one or more ISRs. Each time, the above printLSR would have a different message parameter.

In the above example, hopefully printing is a fast process, since the report task must wait until printLSR is done. If printing is not fast, then it would be better for printLSR to schedule print jobs and for a low-priority task to control the actual printing.

non-preemptible tasks

This is a good approach for short one-shot tasks. Note that all tasks accessing the resource must be non-preemptible or locked while accessing the resource.

data hiding

This can be accomplished by putting data into messages or pipes rather than global variables. Doing so forces sequential access to data by tasks because each task must receive a message or packet before it can access the data in it. However, be careful to do clean handoffs — i.e. do not continue to access a message or packet after it has been sent.

foreground conflicts

ISR to ISR conflicts are best handled by structuring so that ISRs do not share data and there is a clean handoff of data between ISRs and the LSRs they invoke. Using pipe and messaging services helps in this regard. Note that LSR/LSR conflicts are not possible because LSRs cannot be nested.

foreground/background access conflicts

are best handled by making sure that common data never occurs between ISRs and tasks — always use LSRs as intermediaries. This reduces the problem to LSR/task conflicts. Since an interrupt can occur at any time, an LSR can be invoked at any time. Hence, there is no way to predict what the current task will be or what it may be doing when an LSR runs. Therefore, the problem becomes one of protecting against conflict between any LSR and any task.

Many of the task to task techniques do not work here. It is not possible, for example, for an LSR to wait on a semaphore. Techniques which do work are:

(1) server LSR — a server task obviously will not work to prevent LSR/task conflicts. However, a server LSR will. It can be invoked from a task, from an ISR, or from another LSR. Only the server LSR is allowed access to the resource. This is the best approach for peripherals which may be accessed from either foreground or background.

(2) data hiding — use messages or pipes between foreground and background, rather than common data areas. This is an especially important technique for preventing LSR/task conflicts.

(3) semaphore — can be used to block an LSR even though an LSR cannot wait at a semaphore:

```
if (smx_SemTest (oktodoit, NO_WAIT))
    /* do it */
else
    /* don’t do it */
```

(4) disable interrupts in critical sections of tasks (bad choice, but sometimes the only way).

Chapter 26 Event Logging

smx and user events are selectively logged into the event buffer (EVB) as they occur, then uploaded to smxAware when the processor stops (usually at a breakpoint). This is the basis for the event timeline display and event buffer which can be used during debug via smxAware and during normal operation via smxAware Live. The smxAware User's Guide should be studied in conjunction with this chapter.

event logging

The following types of events can be logged:

(1) task (start, stop, suspend, resume)
(2) SSR calls (ID, parameters, and return value)
(3) LSR entry, exit, and invoke
(4) ISR entry and exit
(5) errors
(6) print-ring writes
(7) user events

Event logging is enabled by setting SMX_CFG_EVB to 1 in xcfg.h. Also, a non-zero size must be assigned to EVB_SIZE in acfg.h. Each print ring write (see debugging tools & features in Debug) adds a record to the event buffer whenever it adds an entry to the print ring buffer.

event buffer

The event buffer (EVB) consists of error records loaded by logging functions and macros. Event records vary in length from 3 to 10 words. A variable length format is used to achieve maximum records per fixed-size EVB. EVB is a cyclic buffer, so the oldest records are overwritten as new records are logged. smxAware uploads the entire buffer, then figures out where it starts, based upon timestamps.

smx_evbi, smx_evbx, and smx_evbn point to the first word, the last word, and the next word to be written into EVB. All error records start with 0x5555000n, where n is the record length. The next word is the time stamp, then the current task or LSR. After this, fields depend upon what is being logged. Aided by the 0x5555 start-of-record mark, it is possible (but painful) to look through EVB via the debugger memory window to find records of interest. However, it is much easier to use the timeline display in smxAware.

Task, SSR, and error events are logged automatically by smx. LSR, ISR, and user events require that macros be added to the code. See smx_TickISR() in main.c and smx_KeepTimeLSR() in xtime.c for examples. In addition, see the smx_EVB_LOG descriptions in the Reference Manual. User event logging is discussed, in detail, below.

selective logging

Generally speaking, it is best to make EVB as large as possible, especially during debugging, so that smxAware can show long timelines. However, this is not always possible. To effectively use smaller buffers, events can be selectively disabled. This is also allows weeding out excess information in order to focus on events of interest.

For example, clearing the SMX_EVB_EN_ISR flag in smx_evben disables logging ISRs. This would be useful if an ISR were occurring frequently and it was not of interest. SSRs can be put into one of 8 groups. SSRs in SG0 will never be logged. Those in SG1-8 will be logged if selected by smx_evben. Since SSRs occur frequently, this can help to reduce background noise by putting the few SSRs of interest into SGn and enabling only it.

The selective logging flags for smx_evbn are defined in xevb.h. For example, if the SMX_EVB_EN_TASK flag is set in smx_evbn, then all task events will be logged, if the SMX_EVB_EN_SSRn (n = 1 to 8) flag is set, then the SGn group of SSRs will be logged.

smx_evbn is usually set during initialization. For example:

```
#if SMX_CFG_EVB
 smx_EVBInit(SMX_EVB_EN_ALL);
#endif
```

This enables all events to be logged. smxAware permits easily changing the events to be logged via its Options Dialog window. It does so, by modifying smx_evbn.

In order to change SSR group assignments, it is necessary to modify the SSR function IDs, which are defined in xdef.h. Function IDs have the format: 0xMMSSPPIII, where MM = module (01 for smx), SS = SG0 - 8, PP = number of parameters, and III = function ID. Initially all SSRs are in SG1. You can change group assignments by modifying the SS fields, as desired. Then, you can then enable or disable SSRs by enabling or disabling the groups they are in.

time stamps

Time stamping uses the smx precise time feature. Time stamp resolution varies from 1 to about 50 processor clocks, depending upon the processor. For three typical processors:

(1) LM3S2965 50 MHz Cortex-M3: 1 clock * 20 ns = 20 ns.

(2) MCF5208 167 MHz ColdFire: 32 clocks * 6 ns = 192 ns.

(3) AT91SAM9G20 396 MHz ARM9: 48 clocks * 2.53 ns = 121 ns.

This enables zooming in to sub-microsecond resolution on a timeline display in smxAware and to make accurate time measurements — see Duration of an Event in smxAware UG.

The tick timer clock rate (sb_ticktmr_clkhz) and the tick timer counts per tick (sb_ticktmr_cntpt) must be defined in the processor architecture BSP file (e.g. \BSP\ARM\AT91\SAM9\bsp.c). These are used by smxAware.

logging user events

User event logging permits defining logging your own events. They will appear, with system events, in the Error Buffer window in smxAware. This is helpful because it allows relating user events to system operations, such as ISRs, LSRs, and task switches. Also the events are precisely time-stamped, which may be even more important.

Six user event logging macros are provided:

```
smx_EVB_LOG_USER0(p)
...
smx_EVB_LOG_USER6(p, par1, par2, par3, par4, par5, par6)
```

where p is a void pointer; it can be a function pointer, or used for other purposes. Parameters 1 through 6 can be used for anything, you wish. If SMX_CFG_EVB is set, the above macros equate to corresponding logging functions, else they equate to nothing. This permits them to be easily removed when not needed.

Example usage:

```
void funA(void)
{
    smx_EVB_LOG_USER4((void*)funA, temp1, temp2, press1, press2);
    ...
}
```

logs two temperatures and pressures, whenever funA() starts running. Another example:

```
u32 funB(u32 par1, u32 par2)
{
    u32  A;
    smx_EVB_LOG_USER2((void*)funB, par1, par2);
    ...
    smx_EVB_LOG_USER1((void*)funB, A);
    return A;
}
```

logs when funB starts and with what parameters. After funB runs, it logs the end of funB and logs the return value. Since both events are time-stamped, it is easy to determine how long funB took to execute. If all other logging were turned off, then the text error buffer would contain nothing but the above information, which could be quite a long run if EVB is large enough. The text can be saved to a file for later viewing.

A better example for performance data gathering is:

```
static void* test_log = smx_SysPseudoHandleCreate();
smx_HTAdd(test_log, "TEST LOG");

void  hourly_record(void)
{
    u32  next_hour;

    smx_EVB_LOG_USER4(test_log, temp1, temp2, press1, press2);
    next_hour = ((smx_SysStimeGet()/3600)+1)*3600;
    smx_TaskSleepStop(next_hour);
}
```

This function records two temperatures and pressures every hour. Each record requires 40 bytes, so a 40,000-byte EVB would be good for 1000 hours — about 40 days, assuming nothing else is logged. smxAware Live could be used remotely to download the data once a month and store it in a file. (See clock/calendar time in Tasks for more explanation of this example.) In this particular example, each sample will be identified by TEST LOG, when viewed with smxAware. Note that hourly_record is a one-shot task, which sleeps for one hour between samples without consuming a stack.

In the above examples, note the exact match between the version of smx_EVB_LOG_USERn() and the number of parameters or values (n) to be recorded. This conserves EVB space and results in quicker uploads.

Chapter 27   Precise Profiling

Profiling helps to improve performance.

introduction

Precise profiling is a valuable tool which allows you to see exactly how long each task is running. It also allows you to see how much processor time is consumed by ISRs, LSRs, and overhead. Idle time is indicated by the run-time count of the idle task.

Profiling is performed by accumulating run-time counts for tasks, total LSR, total ISR, and overhead. The count resolution is equal to the clock rate used for the tick timer. If SMX_CFG_PROFILE, in xcfg.h, is true, profiling is built into the smx library; otherwise profiling code and its overhead are not present.

RTC macros

The following profiling macros are provided to capture run-time counts:

```
smx_RTC_ISR_START()
smx_RTC_ISR_END()
smx_RTC_LSR_START()
smx_RTC_LSR_END()
smx_RTC_TASK_START()
smx_RTC_TASK_END()
```

These equate to corresponding profiling functions if SMX_CFG_PROFILE is true, otherwise to nothing. Hence profiling code can easily be added or removed from smx and the application. The above macros are defined in xapi.h and the corresponding RTC functions are defined in xprof.c

RTC START() and END() macros are positioned as optimally as possible to capture the desired run times. They are written efficiently, in C, to minimize overhead added by their own code. (Of course, only a hardware mechanism could be perfect at this.) Since the tick counter resolution for a typical processor is 16 processor clocks, RTC function overhead should be acceptable in most cases.

task logging

Profiling is implemented by RTC task macros built into smx that record the timer count when a task starts running and the timer count when it stops running. The differential count is then added into the rtc field of the task’s TCB (which is the current task, smx_ct). This code is part of smx and should not be modified.

LSR logging

The RTC LSR macros are to be used to record when an LSR starts running and stops running. These are built into the LSR scheduler and should not be modified. All LSR differential counts are accumulated in a global variable called smx_lsr_rtc. (It is not possible to accumulate individual LSR RTCs because LSRs do not have control blocks.)

Note: SSR run times are included in task or LSR run times, whichever called them. Hence they are being treated like any other subroutine call and are not considered part of smx overhead.

ISR logging

RTC ISR macros are included in smx_ISR_ENTER() and smx_ISR_EXIT() and thus need not be included in smx ISR code. ISR differential counts for all smx ISRs are accumulated in smx_isr_rtc. Counts for non-smx ISRs get included in RTCs for interrupted tasks or LSRs or in overhead, if nothing was running. Hence it would not be possible to determine if such an ISR were consuming too much processor time. If this is a problem, contact us for ISR functions that can be used in non-smx ISRs.

overhead logging

The overhead run time count is calculated rather than being accumulated — it equals whatever is not accounted for in a profile period (called a *frame*) by the sum of all other RTCs. This is more accurate than trying to accumulate it. It also reduces profiling overhead.

RTC accuracy

The method of accumulating run times is very accurate vs. using ticks. For example, for a processor with a 100 MHz main clock and a tick counter clock that is 1/16 of it (6.25 MHz), profiling is accurate to 16 instruction times. As delivered, the tick rate for smx is 100 ticks/sec, which is 62,500 instruction times.

profile samples

Actual profiling requires accumulating run time counts for a specified period and performing percentage-of-total calculations. Profiling is enabled by setting PROFILE in acfg.h and defining RTC_FRAME and RTCB_SIZE, also in acfg.h.

RTC_FRAME is the period, or frame, defined in ticks; it is 100 ticks, as delivered, which is 1 second at the default tick rate. This can be changed to whatever frame size is desired. The maximum for any run time count is sb_ticktmr_cntpt (cntpt = counts per tick). As discussed above, this is 62,500 for the example processor. Each RTC cell is a 32-bit word. So the frame size could be up to 2^32 / 62,500 = about 68,000 ticks, or 680 seconds. Such a large frame is unlikely to be useful, but a 60 second frame might be good for some systems to operation logs.

A profile sample consists of all RTCs accumulated during a profile frame. At the end of every frame, smx_ProfileLSR() records all RTCs in smx_rtcb, which is a two dimensional array of (NUM_TASKS + 5) RTCs per sample, times RTCB_SIZE samples. In a typical case, perhaps there are 30 tasks and you want to store 10 samples. Then smx_rtcb would require 1200 bytes of RAM. This space is dynamically allocated from the heap. Samples are loaded in cyclic fashion — i.e. the new sample overwrites the oldest sample.

Profile samples are loaded into smx_rtcb by smx_ProfileLSR() in xprof.c. It is invoked at the end of each profile frame by smx_KeepTimeLSR() in xtime.c. All RTCs are cleared on the very first frame and no entries are made into rtcb. On subsequent cycles RTCs are stored in rtcb and then cleared for the next frame. To avoid errors, interrupts are disabled while smx_isr_rtc is read and cleared. Accessing the task and LSR RTCs does not require protection because they cannot be updated until smx_ProfileLSR finishes running.

For best performance, smx works with pointers into smx_rtcb, but smx_rtcb is defined as a two dimensional array in main.c:

```
#if PROFILE
u32       smx_rtcb[RTCB_SIZE][NUM_TASKS + 5];
#endif
```

and in smx_Go():

```
#if PROFILE
smx_rtcbi = &smx_rtcb[0][0];
#endif
```

This allows convenient viewing as a two dimensional array in the debugger's watch window. It appears as a one dimensional array per sample consisting of the following entries: etime, ISR total, LSR total, idle, all other tasks, task total, and total overhead. The values are run time counts per frame. Tasks appear in the same order as TCBs appear in the TCB pool, which is not necessarily the order in which they were created. To determine what task corresponds to an smx_rtcb entry, look at smx_tcbi[n-3], where n is the smx_rtcb entry number.

easy profile viewing

When the processor is stopped, smxAware uploads the information from smx_rtcb, calculates percentages, and displays them graphically in its Profile window. This permits easily seeing where processor cycles are being used. Each sample, or frame, can be viewed individually. The Next Frame and Prev Frame buttons allow quickly scrolling back and forth to see how profiles dynamically change. The All Frames button shows the average of all frames. The more frames the better picture you can get of dynamic system operation.

remote profile monitoring

smxAware Live can be used to periodically upload smx_rtcb to a central host in order to maintain an operating record for a system. For such use, RTCB_SIZE and RTC_FRAME might be quite large. If both were 100, then 100 seconds of operation could be covered, per upload. Comparing 1 second profiles might suffice to spot an abnormal behavior. Then it might be possible to use 100 shorter frames, such as 0.1 seconds, to zero in on the problem.

Note: smx uses smx_cf->rtc_frame, not RTC_FRAME, which can be dynamically changed, provided that smx_cf is in RAM. In addition, smx_rtc_frame_ctr can be set to a very high value, then dropped to 0 to start recording RTC samples. When smx_rtcb is full, smx_rtc_frame_ctr can be returned to a very high value to stop recording.

coarse profiling

If PROFILE is true, smx_ProfileDisplay(), in xprof.c, is called from idle, once per second. It takes the accumulated RTCs for idle and work (work = ISRs + LSRs + all other tasks) and computes %idle, %work, and %overhead in tenths of a percent. The results are output to the bottom line on console display. This information could also be saved in a log. (See logging user events in Event Logging).

Coarse profiling can be used, as development progresses, to see how much more capacity the processor has for work (i.e. % idle), to spot excessive overhead, or to spot abnormally low work levels. Since the numbers are derived directly from precise run time counts, it is possible to immediately delve into the details of what is wrong by looking at smx_rtcb.

Once the system is deployed, the coarse profile readings can serve as gauges of how the system is behaving, similar to a tachometer on a car.

edge effects

It is not possible to accumulate perfect run-time counts with software. Edge effects can cause counts in the current frame to be wrongly attributed to the previous frame. This occurs because the end of each frame is defined by smx_TickISR() and it is possible that LSRs invoked in the previous frame run between smx_TickISR() and smx_ProfileLSR(). smx_KeepTimeLSR() also runs in between. To avoid the counts for these LSRs being attributed to the previous frame, smx_lsr_rtc and smx_isr_rtc are captured at the start of smx_TickISR(), with interrupts disabled. Thus the counts for these intervening LSRs will be correctly attributed to the current frame. However, that is not sufficient. Anything that delays smx_TickISR(), such as an ISR that has already started or interrupts being disabled, effectively stretches the previous frame and results in some counts being attributed to the wrong frame. Note that this situation is aggravated if smx_TickISR() is not the highest priority smx ISR.

Unless ISR activity is especially high or interrupts are being delayed for especially long times, edge-effect errors will typically be less than 1% for one-tick frames. This percentage can be reduced by using longer frames or by averaging over many short frames. Unfortunately for short frames, if overhead is low, it can actually go negative due to the way it is calculated. In order to avoid possible problems that this might cause, overhead RTC is limited to 0, minimum. This may result in its average being slightly too large. In this case, the error can be reduced by using longer frames.

run-time limiting

Task RTCs can be used to implement run-time limiting. This is planned for smx, in the future. In the interim you can implement your own RTL as follows: Each time a task resumes, use a hook entry routine to compare its RTC to a table of run-time limits per task. If the task has exceeded its run-time limit per frame, set a flag to cause it to suspend itself on the frameGate semaphore. This will allow potentially starved tasks to run. Modify smx_ProfileLSR() to signal frameGate at the end of the frame, which will resume all tasks that exceeded their run-time limits. Also signal frameGate and start a new frame whenever idle runs, so the processor is not needlessly idle when there is work to do. This is the general idea; it may have a few rough spots.

Chapter 28   Power Management

```
BOOLEAN     smx_SysPowerDown(u32 sleep_mode)
u32         sb_PowerDown(u32 sleep_mode)
```

introduction

The purpose of power management is to reduce power usage when there is no useful work to be done. This is primarily of importance for products which are powered by batteries or energy harvesting. smx supports processor power down with tick recovery.

processor power down

smx_SysPowerDown(sleep_mode) puts the processor into the selected sleep mode. The sleep mode is processor-specific. For example, Cortex-M processors offer SLEEP and DEEP_SLEEP modes. If the sleep mode is 0, the function returns immediately with FALSE. Otherwise, the SSR is entered and sb_PowerDown(sleep_mode) is called. This is a user-implemented function, which should do the following:

(1) Save the tick timer count.

(2) Start an external timer or record external real-time-clock time.

(3) Put the processor into the desired sleep mode.

When power is restored, control returns to sb_PowerDown() and it must:

(4) Read the external reference to determine tick timer counts lost and add it to the saved tick timer count.

(5) Divide the sum by the clocks per tick, load the remainder into the tick timer, and return the quotient as ticks lost.

(6) Return to smx_SysPowerDown().

If the resolution of the external timing device is comparable to that of the tick timer, this should result in very little cumulative time error due to power cycling. If not, then there may be a cumulative time error, over a period of time.

tick recovery

When sb_PowerDown() returns to it, smx_SysPowerDown() performs tick recovery in a manner that preserves the proper order of timer, task, and TicksEQ timeouts. It also updates etime and stime. The time to perform tick recovery is generally very short. It depends only upon the number of timeouts and not upon the length of time that power was off. Ticks occurring during tick recovery are saved and applied when normal operation resumes. Operation is transparent to the application, if sb_PowerDown() is able to accurately determine the time lost.

Following tick recovery, smx_SysPowerDown() exits, and LSRs then tasks execute in the order they were invoked or resumed. Since interrupts are enabled during tick recovery, smx_TickISR() can run and can invoke smx_KeepTimeLSR(), which will run after other LSRs have run. Thus, new ticks will not be lost and LSRs will run in their correct order.

smx_SysPowerDown() is normally called at the end of the idle task loop, after all idle functions (e.g. stack and heap scanning, profiling, etc.) have finished, as follows:

```
void smx_IdleMain(void)
{
    while(1)
    {
        ...
        if (idle_done)
            smx_SysPowerDown(SLEEP);
    }
}
```

At this point there is no useful work left to do, hence the processor can be put into sleep mode. Of course, once the processor is put into sleep mode, it is then dependent upon an interrupt or event to wake it up.

sb_PowerDown()

sb_PowerDown() is dependent upon the characteristics of the target processor and upon the requirements of the application. Even within a specific processor architecture, there may be variations in power-down features. Also, some MCUs or SoCs may have an external timer or real-time clock and others may not. If there is no such circuitry available, then tick recovery is not possible and sleeps should be less than one tick period, unless tick recovery is not important.

This function may be simple or it may be complex. It may need to be written in assembly language and it may need to disable interrupts. As delivered, it is just a stub.

profiling

The profile frame is not changed by tick recovery, because it is most accurate to leave it alone. This is because run time counters are not being updated when the processor is not running, so to alter the profile frame would actually introduce errors. Therefore, the profile frame is completed after power is restored, as though nothing happened. Note, however, that the tick timer most likely will be loaded with a different value than it had at power down. Hence, the profile frame may be up to one tick shorter or longer, than normal. However the run time counters will be proportionately less or greater, so profile jitter caused by power down should be small.

Chapter 29   Using Other Libraries

To work with smx a library function must be both reentrant and non-os dependent. Usually, some functions of a library meet these requirements and can be used as is, and others do not. The standard C library is a good example of this.

reentrancy

Functions which are not reentrant pose a problem in a multitasking environment. They are basically critical sections that need to be protected, and the techniques discussed in the Resource Management chapter are generally applicable. For short functions, locking the current task is effective; longer functions are normally handled via a resource semaphore. The in_clib semaphore in the Protosystem is an example of this. smx_CLibEnter() and smx_CLibExit() macros are provided to test and release it. See discussion in the Reference Manual.

Whether locking or using a semaphore, it is best to create macros with similar names to library functions, except all caps. These macros should lock, call the library function, then unlock, or do the equivalent for the semaphore. Then, only the macros should be used for calling the library functions. This insures that the protection mechanism is always invoked. One could use a single semaphore for all library calls, however that could result in blocking higher priority tasks needlessly. It is recommended to use a semaphore per library or even per group of library calls. However, be careful that functions in one group do not call functions in another group, which could result in a protection breakdown.

The above mechanisms do not provide protection against calls from ISRs or LSRs. The best thing is to not make non-reentrant library calls from ISRs and LSRs. If this is not possible, then it may be necessary to disable interrupts or turn LSRs off when making such calls.

converting to SSRs

Another approach is to convert non-reentrant C library functions to SSRs, as in the following example:

```
u32  LibFunSSR(a, b, c)
{
    u32 val;

    smx_SSR_ENTER3(ID_LIB_FUN, a, b, c);
    val = (u32)lib_fun(a, b, c);
    return(smx_SSR_EXIT(val, ID_LIB_FUN));
}
```

The input parameters can be any types; the return value is an unsigned 32-bit value. If this does not work, pass an address to a location for the return value, in the SSR:

```
void  LibFunSSR(a, b, c, type *val)
{
    smx_SSR_ENTER3(ID_LIB_FUN, a, b, c);
    *val = lib_fun(a, b, c);
    return(smx_SSR_EXIT(0, ID_LIB_FUN));
}
```

The advantage of the above is that the library function is now safe from preemption — provided that the prohibition against calling SSRs from ISRs is obeyed. See custom system services in the Additional Features chapter for more information on creating SSRs.

If source code is available, it may be simplest just to fix the non-reentrant functions you want to use to make them reentrant. Usually changing static variables to auto variables (i.e. those stored in the task's stack) is sufficient. The only problem with this approach is that more complex library functions call dozens of subroutines. Hence, extensive effort may be required. The approaches above avoid this.

Another approach for code which is non-reentrant because it uses global variables is to hook exit and entry routines which save and restore the global variables. This is easily done via task stacks. The data can be stored at the end of the stack that it grows toward. This is found by tcb.stp.

The foregoing are ways to allow multiple tasks to use a library. An alternative is to allow only a single, server task to use it.

server tasks

Many libraries control a resource or are, themselves, a resource. Hence many of the techniques discussed in the Resource Management chapter are applicable. In particular, server tasks often work well.

Consider, for example, a graphics package. Several tasks may need to output messages, icons, graphs, etc. to the display. In fact, each such task might have its own window. Rather than allowing each task to directly use graphics library services, a single graphics server task can be created. It is necessary to develop a message format which can describe all graphics operations needed by the application. Client tasks, then, would compose and send graphics messages to an exchange where the graphics server task would wait for work. This task would interpret each message and call appropriate graphics library services. This solves the non-reentrancy problem of the graphics library. It also unloads the client tasks from graphics duties.

os dependency

Libraries intended for embedded use typically have relatively little os dependency. Usually there will be some functions that are os-dependent, so a standard practice is for the library developer to provide a porting layer that must be implemented by the user. This is a relatively small amount of code that needs to be tailored to the environment under which the library will run. The idea of a porting layer is to consolidate such functions into a small set of files so it is not necessary to make changes throughout the library sources.

Other libraries may have a great deal of os dependency. This is especially true of the run-time libraries supplied with common x86 compilers. These are highly DOS- or Windows-dependent. This is because these compilers are intended for use in producing DOS or Windows applications, not embedded ones.

SMX offers an increasing number of supported libraries, so please review our product literature or contact us to see if we offer a library you need.

alternative functions

Some C library functions, such as printf(), use hundreds of bytes of stack space. It is best to avoid such functions and to use other alternatives, such as the smx print ring buffer (see debug tools & features in Debugging) or user event macros (see logging user events in Event Logging).

rtl.c in \SHARED offers C run-time library replacement routines to correct various other problems in standard C libraries. Also the C compiler you are using may offer alternative C run-time library functions, which are better suited to embedded systems. See xapi.h for utility and other macros and functions, which may be helpful.

Chapter 30 Safety, Security, & Reliability

introduction

safety generally means to avoid damage to people and property; *security* generally means freedom from interference; *reliability* generally means dependability — i.e. consistently producing the same results for the same inputs. smx is not specifically targeted at markets with high safety, security, and reliability requirements. Nonetheless, these are increasingly important requirements for all embedded systems and smx does offer features which can help to achieve them. These features are reviewed below.

background

If there is a concern with safety, as with medical equipment, industrial espionage, international intrigue, or basic nastiness of misanthropes, then error management may be equally important with functionality. Run-time errors can be caused by radiation, RFI, power glitches, malware, bugs, etc.

Competition is driving semiconductor manufactures to ever smaller feature sizes, because there is no profit in yesterday's technology. Yet smaller feature sizes mean less electrons per bit and increased susceptibility to environmental factors. For most embedded systems, processor capabilities and memory sizes already exceed what is needed. Yet we cannot stop technology from advancing.

Malicious attacks are also an increasing concern. In addition, embedded software complexity keeps growing, making bug-free software an increasingly elusive goal. This has led to increased emphasis on design standards and using tools such as static testing. However, all of these can never eliminate human error, nor is it possible to test for all combinations of events that may occur for even a small embedded system.

The OS can be a powerful ally in the struggle against errors. Continual low-level error checking by the OS can often detect problems before serious damage is done and permit them to be corrected. It makes sense to turn the excess capacities of processors and memories to good use in order to counter the very problems that they create as well as all of the other problems leading to reduced reliability.

smx provides strong error detection and handling capacities, which has previously been described in the Error Management Chapter. Properly applying these can help greatly to achieve safe, reliable systems.

an opportunity to do it right

It is possible in a standard kernel, like smx, to incorporate the reliability enhancement schemes for which there is not time in a typical development project. One advantage a standard kernel has over application software is economic — the development cost of reliability enhancement is spread over many projects.

Another advantage arises from the uniformity which a standard kernel imposes upon an application. Control is forced to flow through a small number of mechanisms. As a consequence, the places to look for errors are more limited and more uniform. There are fewer kinds of errors possible. The possible errors can be cataloged and named. This promotes better understanding and more rapid correction of problems.

As a result of the above, it is possible to do a thorough job of error checking without excessive memory or performance penalties.

A third advantage is that using a standard kernel encourages better programming practices to begin with. Rather than burying complex interactions in low-level code, these interactions are easily visible because they are implemented via smx calls and objects. This is especially helpful when a new programmer takes over.

Of course, giving a person a hammer doesn't make him a carpenter. Clearly skills must be developed both to use a hammer well and to use smx well.

proper design method

As discussed in the Development Section, starting at one corner of a design and progressing through it at the detailed level is not the way to achieve a safe, reliable system. Proper use of top-down structuring and development are the way to do it. Creating a skeleton at the start and flushing it out over time provides good project direction and the ability to consider failure and safety scenarios as the design unfolds.

avoiding error prone functions

The following is an example of what is meant by avoiding error-prone features:

Error prone:

```
BCB_PTR blk;
blk = BlockGet(pool);
...
BlockRel(blk, pool);
```

Error safer:

```
BCB_PTR blk;
blk = BlockGet(pool);
...
BlockRel(blk);
```

At first glance, the first version of BlockRel() seems preferable, because it is more general. However, what happens if the user decides to get a block from a different pool, but forgets to change the pool that it is being returned to? This would lead to a bug where one pool would shrink and the other pool would grow. Eventually the system would fail because a vital function could not get a block from the shrunken pool. This might take a long time to happen and therefore be difficult to find. If the new block size were smaller, then smaller blocks would be put into the first pool. This would probably result in block overflows for users of that pool, which could cause strange non-repeatable behaviors.

In the second case, the RTOS remembers what pool the block came from and releases it to that pool, thus avoiding the above problems. When one considers that this is the kind of bug that could be easily be introduced by a last-minute change as the product went out the door, the importance of avoiding it is even more important.

reduced error checking if space is tight

smx calls do various safety checks and report errors via the smx error manager. These checks and the error manager are always present. Removing them is not recommended because error checking is very helpful for finding errors in the lab and in the field. There are a few configuration constants you can adjust if memory is tight. Setting SMX_CFG_ERROR_MSGS to 0 results in displaying just error numbers and saves over 1000 bytes of ROM. Of course, if there is no console, then setting SB_CON_OUT, in bcfg.h, to 0 saves quite a bit of ROM and RAM. Also EB_SIZE and EVB_SIZE, in acfg.h, can be set very small or even to 0. tcb.err, smx_errno, smx_errctr, and smx_errctrs[] will still be active and can be viewed in RAM.

the need for error checking after checkout

smx error detection and correction can be disabled in the final systems being shipped. Reasons for not doing so are as follows:

(1) It is difficult to find transient bugs. In some cases it is difficult to even determine if the problem is of software or hardware origin.

(2) Software is never done. After the first system is shipped, there is usually a steady stream of enhancements, fixes, and customizations.

(3) It often is impractical, even impossible, to connect a debugger to an installed system to find a bug. All too often, the errors of interest show up only once in several days — usually when no programmer is around!

(4) Rapid detection of errors permits minimizing the damage they cause. This is important if safety, expensive machinery, or expensive materials are involved. It is also important if data may be destroyed or rendered questionable.

(5) Errors in handles or control blocks may cause bizarre errors which can be hard to locate because they involve the kernel.

In as much as embedded systems are frequently unsupervised and often perform critical functions, the case can be made that they, more than any other systems, need extensive error checking. In embedded systems, it is not enough to merely perform the function, the function must be performed correctly time after time.

If the hardware dies, there is little that the software can do. However, *transient* errors are another matter. These arise from many sources:

(1) Latent software bugs which seldom appear

(2) Power transients

(3) Soft errors in DRAMs

(4) Environmental noise and stress

(5) Hardware bugs

(6) Synchronization problems

Having robust error checking and recovery present and enabled can make a big difference for system robustness, in the face of the above.

timeouts

smx is designed such that all calls which can put a task into the wait state require a timeout parameter. It is tempting to use SMX_TMO_INF so timeouts won't be a bother. However, it is better to think about what would be a reasonable upper limit and specify that. In addition, timeouts should be reported and delt within a reasonable manner. This can be a major factor in dealing with unexpected situations especially deadlocks.

globals

To the degree that you can avoid using global application variables, your system will be safer and more reliable.

Chapter 31 Other Topics

This chapter contains discussion of additional smx features that may prove helpful for your project.

multiwait

Some RTOS kernels implement the ability for a task to wait for multiple events, such as waiting for a semaphore or a message, simultaneously. We have not implemented a multiwait capability in smx because we have not seen a strong need for it, and multiwait implementations tend to be complicated. If you have a need for a task to wait upon multiple events simultaneously, it may be that the task is too complicated and the application should be restructured. However, if multiwait is the best fit, there are two ways to implement it under smx. The first is via event messages, as shown in the following example:

```
enum type {SIG = 1, MSG};

typedef struct {
    u32   event;
    u8*   ptr;
} *EVT_PTR;

void em13(void)
{
    u8*  dp;
    EVT_PTR  ep;
    MCB_PTR  msg;

    /* do multiwait */
    while (msg = smx_MsgReceive(xa, (u8**)&ep, TMO))
    {
        switch (ep->event)
        {
            case SIG:
                /* do switch action */
                smx_MsgRel(msg, 0);
                break;
            case MSG:
                dp = ep->ptr;
                /* process msg */
                sb_BlockRel(&io_pool, dp, 0);
                smx_MsgRel(msg, 0);
        }
    }
}

/* send signal */
void em13_t2a_main(void)
{
    MCB_PTR  emsg;
    EVT_PTR  ep;
    /* get event msg, set for signal, and send to xa */
    emsg = smx_MsgGet(msg_pool, (u8**)&ep, 0);
    ep->event = SIG;
    smx_MsgSend(emsg, xa);
}

/* send message */
void em13_t2b_main(void)
{
    u8*  dp;
    EVT_PTR  ep;
    MCB_PTR  emsg;

    /* get block and load message into it */
    dp = sb_BlockGet(&io_pool, 0);
    /* load message using dp */

    /* get event msg, set for msg, and send to xa */
    emsg = smx_MsgGet(msg_pool, (u8**)&ep, 0);
    ep->event = MSG;
    ep->ptr = dp;
    smx_MsgSend(emsg, xa);
}
```

The second way to implement multiwait is via one-shot tasks. In this case, a one-shot task is created to wait-stop on each object in the multiwait group. Only one stack is required for the tasks, because only one can run at a time. When an event occurs, the waiting one-shot task will be started. Like an ISR, it can do some preprocessing of its event, then signal an event group at which the main task waits. The main task can determine what happened from the flag, which is set. Additional information such as a message handle can be passed from the one-shot task via globals accessible by the main task.

This solution has more flexibility than the typical multiwait implementation, because the main task can wait on the AND or AND/OR of events as well as the OR of events. Also preprocessing by one-shot tasks might simplify main task code and permit greater flexibility.

handle table

The handle table (HT) is a global data structure that associates object handles with names. It is allocated from the heap by smx_HTInit(), which is called by smx_Go(). See “handle table” in the smx Glossary section of the smx Reference Manual for a diagram. Prior to v4.2, smxAware got all symbolic names for objects from the handle table. Now, it only uses it for names of ISRs and LSRs, since it creates its own local handle table by collecting names from control block name fields.

Entries can be added with smx_HT_ADD() and removed with smx_HT_DELETE(). Any entry added with smx_HT_ADD() must be removed with smx_HT_DELETE() before the corresponding object is deleted. To add entries for ISRs and LSRs, first create pseudo handles using smx_SysPseudoHandleCreate(). Other smx_HT calls can be used by the application, if desired. See the smx_HT call descriptions in the smx Reference Manual. The size of the handle table is controlled by HT_SIZE in acfg.h.

system status

smx provides calls to make it easier to find the status of system objects, such as:

(1) smx_BlockPeek(blk, SMX_PK_POOL) and smx_MsgPeek(msg, SMX_PK_POOL) return the pool handle for a block or message, NULL if no pool.

(2) smx_MsgPeek(msg, SMX_PK_XCHG) returns the exchange where the message is waiting, NULL if none.

These calls are mostly of value for exception handling, error recovery and system monitoring. We are in the process of implementing peek calls for all smx objects in order to eliminate the need to view control block fields directly.

encapsulating foreign ISRs

A foreign ISR is an interrupt service routine for which the source code is not available. If such an ISR enables interrupts and has lower priority than an smx ISR, it can be interrupted by an smx ISR. This can cause system failure because the interrupting smx ISR may branch to the scheduler instead of returning to the foreign ISR If a task switch then occurs, the foreign ISR will be suspended with the task for a potentially long time.

To avoid this problem, it is necessary to encapsulate the foreign ISR as follows:

```
ISR_PTR old_ISR();

void  appl_init(void)
{
    //...
    sb_ISD();
    old_ISR = sb_IntVectGet(N)
    sb_IntVectSet(N, new_ISR);
    sb_IR();
    //...
}

void interrupt new_ISR(void)
{
    smx_ISR_ENTER();
    (*old_ISR) ();
    smx_ISR_EXIT();
}
```

sb_IRQVectGet() saves the old ISR address from vector N into the old_ISR pointer defined above. sb_IRQVectSet() loads the new_ISR() address into vector N. Notice that interrupts must be disabled. Now, when interrupt N occurs, new_ISR() runs instead of old_ISR(). new_ISR() executes smx_ISR_ENTER(), chains to old_ISR() (which causes it to run), then executes smx_ISR_EXIT(). Now, the foreign ISR will behave just like an smx ISR and interrupt nesting will not cause a problem.

Note: We recommend using sb_IRQVectGet() and sb_IRQVectSet() for hardware interrupts since the vector number passed is an IRQ number, which we define to be relative to the first hardware interrupt vector. IRQs are a sub range of the full interrupt vector range. The Int versions were used above only to make the discussion easier.

Chapter 31

porting smx to other processors

smx supports ARM, Cortex, ColdFire, PowerPC, and x86. It can be ported to other 32-bit processors. See the smx Porting Guide for directions to port to another processor family. If you only need to port to another processor within a supported family or to other tools, see the section for the processor family in the SMX Target Guide.

Index