smx®

User’s Guide

Version 5.2

February 2024

by Ralph Moore

Micro Digital

© Copyright 1988-2024

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

All rights reserved.

smx is a Registered Trademark of Micro Digital, Inc.

smx is protected by patents listed at www.smxrtos.com/patents.htm and patents pending.

Table of Contents

FOREWORD 1

SECTION I: INTRODUCTION 3
- *references* 3

CHAPTER 1 UNDER THE HOOD 5
- *three-level structure* 5
- *smx services* 5
- *dynamic control blocks* 5
- *dynamically allocated memory regions* 5
- *messaging* 5
- *accurate vs. precise time* 6
- *performance factors* 6
- *error handling* 6
- *safety* 6
- *handle pointer parameters (hp)* 7
- *priorities* 7
- *two ways to get object handles* 7
- *abbreviated names* 8
- *configuration constants* 8
- *programming languages* 8
- *smxBase and smxBSP* 9
- *Protosystem* 9
- *esmx* 9

CHAPTER 2 INTRODUCTION TO OBJECTS 11
- *system objects* 11
- *control blocks* 12
- *handles* 13
- *naming system objects* 14

CHAPTER 3 INTRODUCTION TO TASKS 15
- *what is a task?* 15
- *task types* 15
- *scheduling tasks* 15
- *round-robin* 16
- *time slicing* 16
- *preemption* 16
- *how does smx handle tasks?* 17
- *task states* 18
- *state transitions* 18
- *task main function* 19
- *creating tasks* 19
- *starting tasks* 20
- *task example 1* 20
- *task example 2* 21
- *summary* 23

SECTION II: OBJECTS & SERVICES 25

CHAPTER 4 MEMORY MANAGEMENT 27
introduction 27
DARs 27
base block pools 28
smx block pools 28
creating and deleting smx block pools 29
smx blocks 29
getting and releasing smx blocks 30
making and unmaking smx blocks 32
Peeking 33
block handle pointer parameters 35
message block pools 35
summary 35

CHAPTER 5 HEAPS 37
introduction 37
using xheap 37
heaps vs. block pools 37
xheap vs. compiler heap 38
xheap vs. eheap 38
xheap structure 39
setup 39
initialization 40
operation 41
heap debugging 43
heap optimization 46
reliability 50

CHAPTER 6 STACKS 55
system stack (SS) 55
permanent task stacks 55
shared task stacks 56
task stack control 56
task stack sizes 57
task stack pads 57
finalizing task stack sizes 58
creating and filling task stacks 58
task stack overflow detection 58
task stack overflow handling 59
foreign task stacks 59
stack scanning 59
permanent stack scanning 60
shared stack scanning 60
stack scanning details 60
out of shared stacks 61
system stack fill and scan 61

CHAPTER 7 INTERTASK COMMUNICATION 63
introduction 63
stop SSRs 63
using ITC mechanisms 64

CHAPTER 8 SEMAPHORES 65
introduction 65
resource semaphore 65

event semaphore *68*
threshold semaphore *70*
gate semaphore *71*
other semaphore services *72*
summary *73*

CHAPTER 9 MUTEXES 75
introduction *75*
comparison to binary resource semaphores *75*
creating and deleting mutexes *76*
priority inheritance *76*
ceiling protocol *77*
getting and releasing mutexes *77*
freeing and clearing mutexes *78*
impact upon other smx functions *78*
library function example *79*
summary *80*

CHAPTER 10 EVENT GROUPS 81
introduction *81*
terminology *81*
naming event flags *82*
creating and deleting event groups *83*
testing flags *83*
setting flags *85*
inverse flags *86*
AND/OR testing *87*
other event group services *88*
advice on practical usage *89*
maintaining atomic operation *89*
pros and cons of event groups *90*
state machine example *90*
summary *92*

CHAPTER 11 EVENT QUEUES 93
introduction *93*
creation, deletion, and clearing *94*
counting and signaling *94*
enqueueing a task *95*
accurate timing *96*
summary *96*

CHAPTER 12 PIPES 97
introduction *97*
pipe I/O *97*
intertask communication *100*
LSR to task communication *102*
other pipe services *102*
multiple waiting tasks *103*
packet size *103*
pipe buffers *103*
safe operation notes *103*

CHAPTER 13 EXCHANGE MESSAGING 105
kinds of messaging *105*
exchange messaging *105*
exchange API *106*

message API *107*
messages *107*
getting and releasing messages *108*
sending and receiving messages *110*
making and unmaking messages *112*
peeking at messages and exchanges *115*
message owner *116*
message handle pointers *116*
using the reply field *117*
client/server example *117*
broadcasting messages *119*
proxy messages and multicasting *121*
distributed message assembly *122*
other message services *124*
summary *124*

CHAPTER 14 TASKS 127
introduction *127*
task priority *127*
changing a task's priority *128*
idle task *128*
task flags *128*
ready queue *129*
current task (ct) *131*
inter-task operations *131*
starting and stopping tasks *131*
resuming and suspending tasks *132*
deleting tasks *132*
locking and unlocking tasks *133*
lock breaking *134*
lock nesting *134*
uses for task locking *135*
making tasks act as expected *135*
task local storage (TLS) *136*
task callback functions *136*
task initialization and deletion *137*
normal task entry and exit *138*
one-shot task start and stop *138*
task callback notes *139*
saving coprocessor context *139*
second task main function type *140*

CHAPTER 15 ONE-SHOT TASKS 143
introduction *143*
stack RAM usage *143*
one-shot tasks offer a solution *144*
stack pool size *144*
examples of one-shot tasks *145*
writing one-shot tasks *146*
examples *146*
I/O tasks *150*
a further example *150*
return to self example *151*

CHAPTER 16 SERVICE ROUTINES 153
introduction *153*
system service routines (SSRs) *153*

interrupt service routines (ISRs) ... *153*
link service routines (LSRs) ... *154*
rules of behavior ... *154*
background vs. foreground ... *155*
two ISR types ... *155*
interrupt handling ... *156*
smx ISR and SS operation ... *156*
writing smx ISRs ... *157*
more on LSRs ... *158*
smx calls from LSRs ... *158*
limited SSRs ... *160*
example ... *161*
why do LSRs not have priorities? ... *162*
tips on writing ISRs and LSRs ... *162*
how LSRs work ... *163*
custom SSRs ... *164*
inner SSRs ... *165*

CHAPTER 17 TIMING ... 167
etime and stime ... *167*
tick rate ... *167*
timeouts ... *168*
timeouts for safety ... *168*
timeouts for timing ... *169*
timouts in milliseconds ... *169*
handling actual timeouts ... *170*
timeout priority ... *170*
time delay options ... *171*
very fast delays ... *171*
fast delays ... *171*
medium delays ... *171*
date-time delays ... *172*
using a hardware timer for a fast timeout ... *172*

CHAPTER 18 TIMERS ... 175
introduction ... *175*
reasons to use software timers ... *175*
software vs. hardware timers ... *176*
starting a timer ... *176*
absolute start ... *177*
stopping a timer ... *178*
cyclic timer example ... *178*
duplicating a timer ... *179*
resetting a timer ... *180*
timer LSR control ... *180*
pulse timer control ... *181*
pulse width modulation (PWM) ... *182*
pulse period modulation (PPM) ... *182*
frequency modulation (FM) ... *183*
timer peek ... *183*
timers vs. tasks ... *183*
more precise timers ... *184*

SECTION III DEVELOPMENT 185

CHAPTER 19 STRUCTURE 187
function vs. structure 187
creating subsystems and tasks 187
creating flow diagrams 188
more tasks are better 189
guidelines for defining tasks 189
benefits of multitasking 190
getting started 190

CHAPTER 20 METHODOLOGY 191
system framework development 191
system framework example 191
framework system development 193
evolutionary development 194
summary 194

CHAPTER 21 CODING 195
design techniques 195
keep it simple 197
keep tasks small 197
simpler task loops 198
C statements are not atomic 199
invalid handles 199
task arrays 200
priority inversion 201
deadlocks 202
choosing stack sizes 203
configuring smx 203
user access to smx objects 203
naming objects 204

CHAPTER 22 DEBUGGING 205
debug tools & features 205
important smx variables 206
looking at smx objects 207
debug tips 207

SECTION IV SYSTEM RESOURCES 211

CHAPTER 23 ERROR MANAGEMENT 213
introduction 213
smx error detection 213
smxBase error detection 213
stack overflow detection 213
stack switching for smx_EM() 214
smx_errno vs. task->err and lsr->err 214
central error processing 214
error buffer (EB) 215
event buffer (EVB) 215
error manager hook 215
error manager exit hook 215
standard error handling 216
local error handling 216
deciding which to use 217

CHAPTER 24 RESOURCE MANAGEMENT..219
access conflicts between tasks 219
same priority tasks..219
semaphores...219
mutexes 219
exchanges...220
task locking..221
server tasks 221
server LSRs..223
data hiding...224
foreground conflicts...224
foreground/background access conflicts 224

CHAPTER 25 EVENT LOGGING...227
event logging..227
event buffer 227
selective logging 228
time stamps 228
logging user events 229

CHAPTER 26 PRECISE PROFILING 231
introduction 231
RTC macros 231
task profiling..231
LSR profiling..232
ISR profiling 232
overhead logging 232
RTC accuracy 232
profile samples...232
easy profile viewing 233
remote profile monitoring...233
coarse profiling...234
edge-effect errors...234
runtime limits...234

CHAPTER 27 POWER MANAGEMENT...235
introduction 235
processor power down...235
tick recovery 235
sb_PowerDown() 236
profiling 236

CHAPTER 28 USING OTHER LIBRARIES..237
task reentrancy...237
server tasks 237
OS dependency...238
alternative C library routines 238

CHAPTER 29 SAFETY, SECURITY, & RELIABILITY...239
introduction 239
background...239
RTOS advantages...239
proper design method 239
error prone functions..240
the need for error checking after shipment...240
other ways to reduce memory 241
timeouts...241

tips for reliable RTOS code *241*
high security *244*

CHAPTER 30 OTHER TOPICS 245
multiwait *245*
handle table *246*
encapsulating foreign ISRs *247*
porting smx to other processors *247*

INDEX 249

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 references. 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 35 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 a couple of orders of magnitude in these years.

Such large changes have ramifications on RTOS architecture and have resulted in major changes to smx over the years. The biggest shift has been from wringing out maximum performance to increased functionality, ease of use, safety, and now security, which has become especially important. 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.

The major emphasis of v5.2 is security. New security features are so major, that we have developed a new RTOS incorporating them, SecureSMX, which is built upon smx. Supporting these new security features has resulted in many changes to smx. In addition, in order to make SecureSMX more widely applicable, we have developed FRPort and TXPort to ease porting FreeRTOS and ThreadX applications to smx. See the SecureSMX User’s Guide to learn more about it.

Due to modern high-speed processors, kernel calls can be treated almost like C statements, without regard for their execution times. Hence, using a rich, well-constructed kernel can lead to reduced coding and debug times and stronger systems. v5.2 further supports this goal and adds strong security to your systems. 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, resulting in *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. However all enabled control block regions are statically allocated, which is generally required for safety. See **control blocks** in the next chapter for more discussion.

dynamically allocated memory regions

Dynamically allocated regions (DARs) are available for applications, though not used by smx. See the smxBase User’s Guide for more information.

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 between LSRs and tasks. 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. Control block pools are cache-line aligned, if external memory is being used.

error handling

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

```
err = task->err;
```

reveals the last error for task, and

```
err = lsr->err;
```

reveals the last error for lsr. The global error number, smx_errno may not apply to the current task or lsr. If err == SMXE_TMO, 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 provides callback functions for error recovery. See Chapter 23 Error Management for more information.

safety

Safety is a big concern for smx. 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 29 Safety, Security, & Reliability for more information.

handle pointer parameters (hp)

Object create, get, make, and receive calls have an optional *handle pointer parameter* that can be used to pass the address of the variable that will hold the handle of the object being created. This allows not only assigning the handle via parameter rather than return value, but also checking that the handle is not already initialized. The latter prevents doubly-creating an object or recreating it without first deleting it. If the handle pointed to is not NULL or &smx_nullcb, the call is aborted and SMXE_INV_OP is reported. An hp parameter defaults to NULL, in which case the feature isn't used.

priorities

Task and message priorities can be any value from 0 to 126. 127 (0xFF) is reserved for SMX_PRI_NOCHG, which is used in some smx services when a priority change is not desired.

SMX_CFG_PRI_UP in xcfg.h controls the direction of task priorities. If set to:

1	Priorities are 0 (lowest) to 126 (highest)— preferred / default
0	Priorities are 126 (lowest) to 0 (highest) — compatibility

This option is offered only for compatibility with certain other RTOSs and is not normally used for smx applications. Priorities increasing numerically is most intuitive and is the default for smx. Setting SMX_CFG_PRI_UP to 0 is not recommended beyond the initial port from another RTOS to smx. It is possible to proceed with an inverted priority system, but it is likely to be error prone. All discussions in the smx manuals assume SMX_CFG_PRI_UP is 1.

In order to avoid confusion in smx code the terms *above* or *below* are used to compare priorities rather than *greater* or *smaller.* The following macros are defined in xcfg.h to handle priority comparisons:

```
SMX_PRI_A(a, b)        /* priority a above b */
SMX_PRI_AE(a, b)       /* priority a above or equal b */
SMX_PRI_B(a, b)        /* priority a below b */
SMX_PRI_BE(a, b)       /* priority a below or equal b */
SMX_PRI_MAX(a)         /* priority is maximum */
SMX_PRI_MIN(a)         /* priority is minimum */
```

These macros are used in smx instead of numerical comparisons. Since SMX_CFG_PRI_UP == 1 is normal for smx, in this manual and the smx Reference Manual, the terms *greater* or *smaller* are generally used to compare priorities and ++ increases priority; -- decreases priority.

two ways to get object handles

smx offers two ways to get handles from calls the produce smx objects. The first is via the return value, for example:

```
TCB_PTR  taskA;

if (taskA = smx_TaskCreate(…))
     /* use taskA */
else
     /* handle timeout or error */
```

This is the traditional method for smx. The second method uses a handle pointer parameter, for example:

```
TCB_PTR  taskA;

if (smx_TaskCreate(…, &taskA))
     /* use taskA */
else
     /* handle timeout or error */
```

The handle pointer is always the last parameter of a call that produces an smx object. It defaults to NULL if not specified, as in the first example. If specified, the handle is automatically loaded as in the second example. However, the return value is still accurate and if 0, there was a timeout or error. Handle pointers were added to smx primarily to support SecureSMX tokens. However, they are also useful for ports from other RTOSs, most of which use handle pointer parameters, and for those who prefer this method.

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. 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_ct).

configuration constants

Application configuration constants such as SMX_CFG_NUM_TASKS are defined in acfg.h. We recommend that you set these constants larger than you expect to need, then reduce them when the project is done. smx configuration constants such as enabling profiling, stack scanning, and event buffer recording are present in xcfg.h. In general, these configuration constants enable or disable corresponding code to be included in smx and thus affect its size and performance.

programming languages

smx is written in C. However, it uses default parameters, which make function calling simpler. See xapi.h. In some places it also supports function overloading. Both are C++ features, hence a C++ compilers is required. smx is interoperable with C++ code. In addition, we offer smx++, which provides a C++ API for smx.

Use of assembly language has been minimized. It is used in the early startup code, prescheduler, scheduler porting macros, and ISR enter and exit code for some processor architectures. Application code can be written in assembly, but nothing is provided to assist with this.

smxBase and smxBSP

smxBase provides basic objects and functions for smx and smx middleware. It also may be of use for applications. 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 functions 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 has been provided in every smx evaluation kit and smx delivery. It includes necessary BSP / HAL code for the target processor, usually supplied by the processor vendor. Its purpose is to allow getting started more quickly. However, things change. We are now working toward using application framework tools offered by semiconductor vendors.

esmx

esmx is a set of example code for smx, in the ESMX directory, that links into the Protosystem. To use it, add an ESMX group to the App project with all of the .c files, and enable SMX_ESMX in the master preinclude file (e.g. iararm.h, and disable other SMX modules and demos. DebugTour.pdf guides you to understand the code as you step through it.

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 them 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 omitted.

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:

- tasks
- block pools
- blocks
- link service routines (LSRs)
- messages
- exchanges
- semaphores
- mutexes
- timers
- event queues
- event groups
- pipes
- heap chunk blocks
- task stacks
- interrupt service routines (ISRs)

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 and ISRs which have no control blocks. They are, in order:

TCB	task control block
PCB	pool control block
BCB	block control block
LCB	LSR control block
MCB	message control block
XCB	exchange control block
SCB	semaphore control block
MUCB	mutex control block
TMRCB	timer control block
EQCB	event queue control block
EGCB	event group control block
PICB	pipe control block
CCB	chunk control block in heap
CDCB	chunk debug control block in heap

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. The number of each system object, and hence the size of its control block pool, is user-defined in acfg.h by constants such as SMX_CFG_NUM_TASKS.

Control blocks store information about their corresponding system objects. Kernel services use the information in control blocks to perform their functions. Most other RTOSs and 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);
```

It is necessary for the user only to define handles. As noted above, all control blocks of the same type are grouped together in a pool. 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 not 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 an object's control block. For example:

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

Here, amsg is declared to be a pointer to an MCB. amsg can be either a global or a static 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 used in all subsequent smx calls, e.g.:

```
smx_MsgSend(amsg, xchg, PR0, NULL);
```

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. An smx Create service first allocates a control block from the appropriate pool, then initializes its fields. An smx Delete service clears the control block, releases it back to its pool, then loads smx_nullcb into its handle so that it cannot be used again. smx_nullcb has 0 in all fields, which is convenient for debugging. If a handle is set to NULL, the fields are non-zero, which can be mistaken for an active object, when debugging. For this reason, it is recommended that handles be set to smx_nullcb when first defined:

```
TCB_PTR task = smx_nullcb;
```

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

naming system objects

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

```
TCB_PTR  send_data;
send_data = smx_TaskCreate (send_data_main, 3, 0, SMX_FL_NONE, "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. Hence it is desirable that the task name indicate what the task does and be distinctive, while being as short as possible.

Chapter 3 Introduction to Tasks

Tasks are workers

```
TCB_PTR     smx_TaskCreate(FUN_PTR code, u8 pri, u32 tlssz_ssz, u32 fl_hn, const char* name,
                                                          u8* bp, TCB_PTR* thp)
BOOLEAN     smx_TaskStart(TCB_PTR task, u32 par)
```

what is a task?

A task is usually thought of as some amount of work which is started and finished within some 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 types of task: *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, SMX_PRI_NOCHG);
```

or

```
smx_TaskYield();
```

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 by means of runtime limiting — see **task example 2** below.

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 or attempts to suspend, or stop itself, at which time the top task will take over.

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

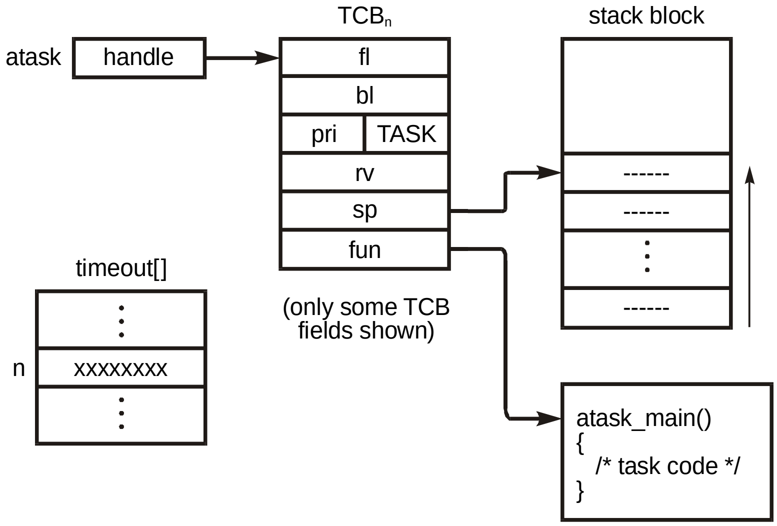
(1) an SSR called from a task

(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 above. 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).

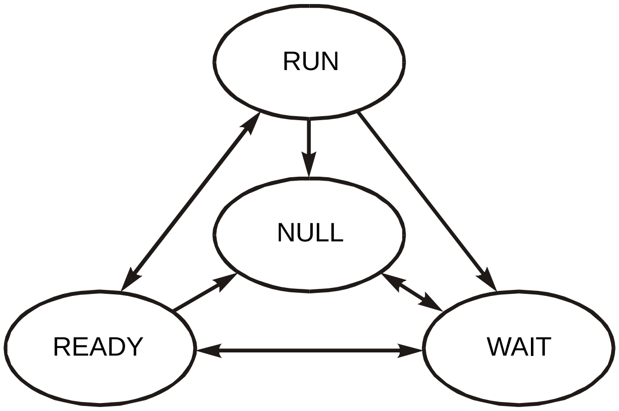
If the task has been assigned a stack, stp and sbp will point to the top and bottom of the stack, respectively. (For simplicity, these have been omitted in the above diagram.) The TCB stores the processor stack pointer in the sp field when a task is suspended. If the task has not yet run or has been stopped, sp is NULL. sbp + 1 points to the register save area (RSA), 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 Chapter 17 Timing for more on timeouts.

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

task states

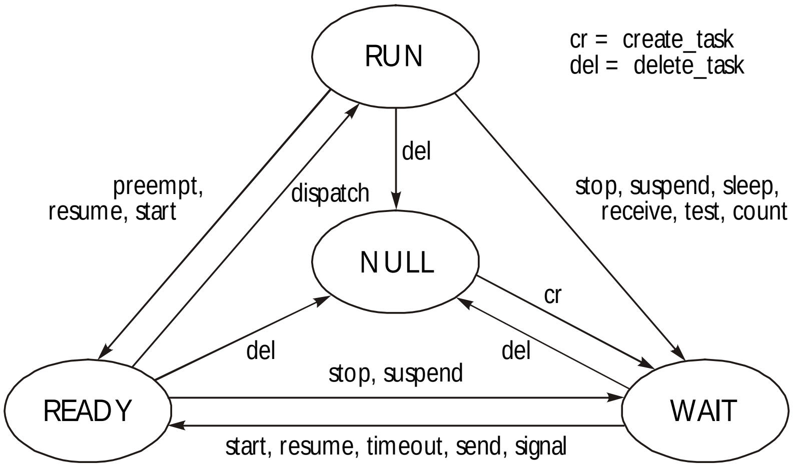
A task may be in one of four states:


RUN
NULL
READY
WAIT

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:


RUN
cr = create_task
del = delete_task
del
preempt,
resume, start
dispatch
stop, suspend, sleep,
receive, test, count
NULL
cr
del
del
stop, suspend
READY
WAIT
start, resume, timeout, send, signal

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 or by a timeout.

Task state transitions may seem complex, but 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(u32 par);
```

This allows passing in a parameter to initialize the task or for other purposes. 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(u32 par)
{
      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(u32 par)
{
      //...
}
```

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

```
smx_TaskStartNew(t2a, par, pri, t2a_run);
```

creating tasks

Creating a task is done as follows:

```
TCB_PTR   t2a;
void  appl_init(void)
{
      t2a = smx_TaskCreate(t2a_main, P2, 200, SMX_FL_NONE, "t2a", NULL, &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 by 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, P2 as its priority, a stack of 200 bytes is allocated from the heap, no task flags are set, and the task is named “t2a”.

The last parameter is the address of the task handle. t2a’s TCB address is loaded into this location. Hence, if you prefer, the above example can be shortened to:

```
TCB_PTR   t2a;
void  appl_init(void)
{
      smx_TaskCreate(t2a_main, P2, 200, SMX_FL_NONE, "t2a", NULL, &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, par);
```

This puts t2a into the ready queue, but it does not actually run it. That is done by the scheduler when t2a becomes the top task. The task main function is as follows:

```
void t2a(u32 par)
```

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, SMX_FL_NONE, "t2a");
      t2b = smx_TaskCreate(t2b_main, P2, 100, SMX_FL_NONE, "t2b");
      t3a = smx_TaskCreate(t3a_main, P3, 100, SMX_FL_NONE, "t3a");
      smx_TaskStart(t2b);
      smx_TaskStart(t2a);
}

void  t2a_main(u32 par)
{
      smx_TaskStart(t3a);
      fun2a();
}
```

```
void  t2b_main(u32 par)
{
      /* do task initialization here */
      while (1)   /* endless loop */
      {
            fun2b();
            smx_TaskSuspend(self, 2);  /* wait 2 ticks */
      }
}

void  t3a_main(u32 par)
{
      fun3a();
}
```

This very simple example, which does nothing useful. The appl_init() function is 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 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 runs first at that level.

(4) Tasks can autostop and become dormant. If fun2a() were to start t3a again, t3a would start from its beginning and run again. We call this a *one-shot task*.

task example 2

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

```
TCB_PTR  tts[T],  trr[R],  t3a;

#define MSEC 200000

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

      for (n = 0;  n < T; n++)
      {
            tts[n] = smx_TaskCreate(tts_main, P1, 100, SMX_FL_NONE);
            smx_TaskSet(tts[n], SMX_ST_RTLIM, x*MSEC);
            smx_TaskStart(tts[n], n);
      }
```

```
        for (n = 0;  n < R; n++)
        {
            trr[n] = smx_TaskCreate(trr_main, P2, 100, SMX_FL_NONE);
            smx_TaskStart(trr[n], n);
        }
        t3a = smx_TaskCreate(t2a_main, P3, 100, SMX_FL_NONE);
        smx_TaskStart(t3a);
    }

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

    void  trr_main(u32 n)           /* round robin tasks */
    {
        /* do task n initialization here */
        while (1)
        {
            /* wait for event then process it */
            smx_TaskBump(self, NO_PRI_CHG);  /* bump self to end of rq level 2 */
        }
    }

    void  t3a_main(u32 n)           /* preemptive task */
    {
        /* do task initialization here */
        while (1)
        {
            fun3a();
            smx_TaskSuspend(self, 5); /* wait 5 ticks */
        }
    }
```

The initialization code creates an array of T time-sliced tasks, tts[n], all of which use the tts_main() function and have priority P1. Initialization then creates a similar array of R round-robin tasks, trr[n] and starts each one. These are at priority P2. Finally initialization creates and starts t3a at priority P3.

Since t3a 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 P2. This creates *round-robin* scheduling.

While the round-robin tasks are waiting for events and t3a is suspended, the time-sliced tasks can run. Each runs for x milliseconds, then waits at the smx_rtlsem gate semaphore. When all time-sliced tasks have run, the idle task at priority P0 runs. After it has run SMX_CFG_IDLE_RTLIM times the runtime frame ends. Then smx_rtlsem is signaled and all waiting tasks are resumed with counts restored. For more information, see Adaptive Timeslicing in the SecureSMX User's Guide. Note that SMX_CFG_RTLIM must be TRUE in order to use runtime limits.

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

(1) Tasks can share identical code. smx_TaskStart() allows passing in a parameter which identifies to the code which task is running. This is a prime illustration that a task is not the code that it runs.

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

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 Chapter 14 Tasks and Chapter 15 One-Shot Tasks for more information on tasks.

SECTION II: OBJECTS & SERVICES

This section covers smx objects and related services. Each chapter deals with a different functional area of smx. Basic services are presented first, followed by advanced services. 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. 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. Using dynamic, rather than static memory allocation, 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 can be used for one-time allocations of blocks, stacks, heaps, etc. They have been replaced by heaps in smx, but are still available for application use.

Block pools provide simple, fast memory management. Since the blocks are all the same size, block pools are not subject to fragmentation, as are heaps. They also provide 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 heaps are best for unpredictable numbers of random size blocks. DARs, bare 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)
```

Unlike a heap, blocks cannot be returned, except for the last one allocated. Hence DARs are best used for one-time allocations during initialization. As noted previously, smx does not use DARs. However, 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 buffers 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). See the smxBase User's Guide for more information.

base block pools

BOOLEAN	sb_BlockPoolCreate(u8* dp, PCB_PTR pool, u16 num, u16 sz, const char* name)
BOOLEAN	sb_BlockPoolCreateDAR(SB_DCB* dar, PCB_PTR pool, u8 num, u16 sz, u16 align, const char* name)
u8*	sb_BlockPoolDelete(PCB_PTR pool)
u32	sb_BlockPoolPeek(PCB_PTR pool, SB_PK_PAR par)
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 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. See the smxBase User's Guide for more information.

smx block pools

BCB_PTR	smx_BlockGet(PCB_PTR pool, u8** bpp, u32 clrsz, BCB_PTR* bhp)
BCB_PTR	smx_BlockMake(PCB_PTR pool, u8* bp, BCB_PTR* bhp)
u32	smx_BlockPeek(BCB_PTR blk, SMX_PK_PAR par)
PCB_PTR	smx_BlockPoolCreate(u8* p, u8 num, u16 size, const char* name , PCB_PTR* php)
u8*	smx_BlockPoolDelete(PCB_PTR* php)
u32	smx_BlockPoolPeek(PCB_PTR pool, SMX_PK_PAR 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, rather than base blocks, for the following reasons:

(1) smx block pool services are preemption-safe.

(2) An smx block is automatically released to the correct pool, given its handle.

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

(4) smx block pool information can be obtained via a block's handle.

These features make working with smx blocks less error-prone than base blocks. As can be seen from the above API, the smx block pool API is similar to the base block pool API, with a few additional services. The main difference is that smx blocks have both a handle and a data pointer. Manipulating blocks by their handles provides safer, more automatic operation than manipulating blocks by their pointers.

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.

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 PCBs dynamically allocated 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 block pools created by smx, a PCB is automatically allocated from the PCB pool when the block pool is created, then its handle (PCB_PTR) is used. 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.

smx blocks

An *smx block* consists of two parts: (1) its data block and (2) its block control block (BCB), which contains information used by smx to manage the smx block. The data block comes from a data block pool and the BCB comes from the BCB pool, *smx_bcbs*. 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.

The BCB pool is a base block pool, which contains free BCBs. It is controlled by a static PCB called smx_bcbs and it is created during initialization if SMX_CFG_NUM_BLOCKS > 1:

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

where SMX_CFG_NUM_BLOCKS is defined in acfg.h. The above operation is performed automatically by smx_CB_PoolsCreate() during startup prior to calling C++ initializers in case it is needed by them.

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

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

PCB_PTR poolA;

u8 pa[NUM*SIZE]   - -OR--   u8* p = (u8*)smx_HeapMalloc(NUM*SIZE);
poolA = smx_BlockPoolCreate(&pa or p, NUM, SIZE, "poolA”);
```

This creates a pool of 100 20-byte blocks starting at &pa or 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 and SIZE must be a non-zero multiple of SB_DATA_ALIGN, which is 4 for ARM and ARM-M, and NUM must be non-zero. 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. As with base block pools, the user is responsible to make sure that the data block is aligned, as desired, and that there is sufficient free memory for the pool.

A pool created by the above function 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:

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

Delete fails if the pool handle is invalid. 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 pool a different name or to reuse the memory space.

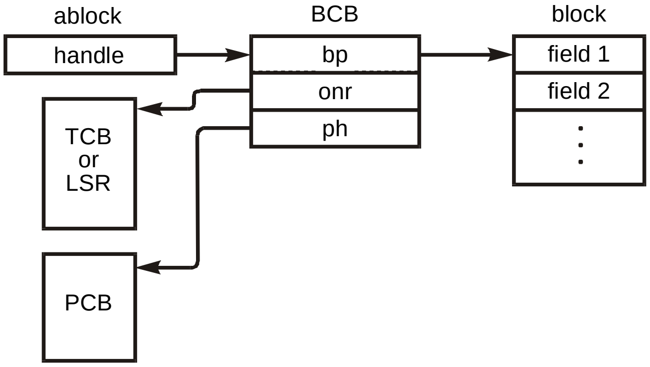
getting and releasing smx blocks

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

```
BCB_PTR blk;
u8* bp;

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

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 16 bytes. Hence, its overhead on a typical data block of 100 bytes size is small. 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.

smx_BlockGet() can be called from a task or an LSR. The handle of the requesting task (smx_ct) or of the LSR (smx_clsr) is stored in the onr field of the BCB. This indicates that the block is in use. 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. 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, which is NULL for a free block.

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:

```
u8              pa[NUM*SIZE]
PCB_PTR         poolA;
SCB_PTR         sr;
TCB_PTR         t2a;

poolA = smx_BlockPoolCreate (&pa, NUM, SIZE, "poolA");
sr = smx_SemCreate(SMX_SEM_RSRC, NUM, "sr");

void t2a_main(u32 par)
{
    BCB_PTR  blk;
    u8* bp;

    smx_SemTest(sr, SMX_TMO_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 at sr to get the block from poolA.

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 */
```

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 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, sb_BlockGet() can be 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 with smx_BlockMake(). The smx block can then be passed onto a task for processing.

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

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

blk 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 it could contain a table used by tasks.

smx_BlockUnmake() converts an smx block to a base 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 base block by an LSR, and passed to an ISR. The block unmake loads a global pointer, bp1, for use by the ISR. 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.

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.

smx_BlockPeek() 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. For available peek parameters see smx_BlockPeek() in the smx Reference Manual.

smx_BlockPoolPeek() for available block pool peek parameters see smx_BlockPoolPeek() in the smx Reference Manual.

Example usages:

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

This gets the total number of blocks in poolA and subtracts 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 */
    }
}
```

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.

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.)

block handle pointer parameters

For BlockGet() and BlockMake() services, if the block handle pointer parameter, bhp, is set to the address of the block handle, then the block handle will be loaded by these services and need not be loaded from their return values. The BlockRel() and BlockUnmake() services load smx_nullcb into the block handle so it can no longer be used to access the BCB. In addition, if the block handle is not initially NULL or smx_nullcb, a BlockGet() or BlockMake() service will be aborted and SMXE_INV_OP reported. This forces releasing or unmaking a block before getting or making another block for the same handle, thus avoiding block and BCB leaks. The bhp parameter defaults to NULL, thus disabling the foregoing, if not desired.

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. Base blocks are used in smx and they are 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 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.

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

Chapter 5 Heaps

introduction

The smx 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 services are implemented via smx porting shell functions in xheap.c that call corresponding services in *eheap.c*, which is a generic, high-performance, configurable, self-healing heap, with enhanced safety and debug features. It supports multiple heaps, aligned allocations, MPU region allocations, and small, integrated block pools. eheap services meet the ANSI C Standard for malloc(), free(), realloc(), and calloc() and offer many additional services. It is assumed that the reader is familiar with these four functions, so usage is not discussed here. See the smx Reference Manual smx_Heap call descriptions for details.

This chapter provides an appropriate level of detail about the heap to use it. For full details, see the eheap User's Guide.

using xheap

smx heap services are used like the common ANSI C heap functions, except that alignment and heap number parameters can be passed. They are not SSRs but are protected using a mutex, so operations can be performed simultaneously in different heaps. Since heap functions normally do not have timeouts, no timeout parameter has been added. However, in a multi-heap, multitasking situation it may be desirable to have some timeout so that the system runs smoothly. Consequently a single timeout, smx_htmo, is provided in xheap.c. This can be set to any value from 0 to SMX_TMO_INF and applies to all heap operations in all heaps.

heaps vs. block pools

Although eheap has been designed to be fast and deterministic and to minimize fragmentation, block pools still offer a faster, more deterministic solution. However, block pools suffer from *internal fragmentation* due to pools having more blocks than are usually needed and blocks being larger than are 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 these adjustments, 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. Nor do block pools adjust easily to software changes, as do heaps.

One useful strategy is to use block pools for mission-critical tasks and heaps for other tasks such as data processing, user interfaces, and communications. The limitations of block pools may be acceptable for the first kind of tasks. Also, consider using one-shot tasks there, since they get their stacks from the stack pool, rather than from the heap. Under this approach, if a heap fails, then the tasks using it can be stopped, the heap can be reinitialized, and the tasks using it can be restarted. Typically, these tasks will all be in a single partition, making it easier to start and stop the tasks. During this process, high-priority, mission-critical tasks will function normally because they are using block pools.

As embedded systems requirements keep increasing, they are called upon to do more sophisticated processing. Often this software is written in C++, which uses heaps heavily. Hence heaps are becoming more of a necessity in embedded systems.

xheap vs. compiler heap

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

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

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

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

In both cases, alignment and heap are set to 0, so these will access the main heap with 8-byte alignment.

xheap vs. eheap

As previously noted, xheap is a port of eheap to smx. The smx porting functions in xheap.c provide the following:

(1) smx API.

(2) Access protection via a mutex per heap.

(3) Optional event logging in the smx Event Buffer.

(4) Optional conversion from eheap errors to smx errors followed by calling smx_EM().

(5) C++ support.

Unlike other smx services, xheap services are not SSRs. This is necessary to permit multiple heaps to be accessed simultaneously by different tasks. If xheap services were SSRs, then only one heap could be accessed at a time. In order to protect against reentrancy into the same heap, a mutex per heap is used. Reentrancy protection is normally required in a multitasking environment. It can be removed if heap services cannot be preempted by other heap services for the same heap. For systems without multiple heaps, the heap number, hn, defaults to 0.

Using a mutex per heap for access protection is safest since it avoids unbounded priority inversion, and it works best during debugging, but it does add significant overhead to heap calls. The mutex can be replaced with a binary resource semaphore to reduce overhead. However, then owner ID and priority promotion is lost. For a small partition heap, the best approach may be to limit heap access to one task or to a group of tasks having the same priority. Then no protection mechanism is necessary for that heap.

The following provides guidance for setting up and using an xheap. See the eheap User's Guide for detailed information concerning features and usage and see the smx Reference Manual for detailed descriptions of xheap services.

xheap structure

The **physical structure** of xheap consists of *free, inuse, and debug* chunks. These are doubly linked together via forward and backward links in the first two words of each chunk. These links require 8 bytes; the rest of an inuse chunk is available for data. The free and debug Chunk Control Blocks (CCBs) each require 24 bytes, which defines the smallest chunk size. Thus, the smallest data block in an inuse chunk is 16 bytes. All chunks in the heap are at least 8-byte aligned.

The **logical structure** of xheap consists of the heap plus free chunk bins. Bins speed access by enabling xheap to pluck a chunk from anywhere in the heap with little or no searching. Every heap has its own bin structure. The smallest bin has a size of 24 bytes. Normally a heap bin structure starts with a *Small Bin Array (SBA)* of bins spaced 8 bytes apart, e.g. 24, 32, 40, … Each small bin holds only one size and can be quickly accessed via its block size. The rest of the bin structure is called the *Upper Bin Array (UBA)*; it consists of a mixture of large and small bins. A large bin covers a range of chunk sizes, e.g. 48 to 504. This bin would have 58 chunk sizes: 48, 56, 64, … 496, and 504. Each large bin acts like a miniature heap.

The bin structure of a heap is determined by its *bin size array*, consisting of the minimum chunk size of each bin and ending in -1. For example: {24, 32, 40, 48, 512, 1024, 1536, 2048, -1}, defines a heap with a 3-bin SBA (24, 32, and 40) and a 5-bin UBA (48, 512, 1024, 1536, and 2048). The last bin, 2048, is called the ***top bin***; it handles sizes from 2048 bytes on up. A free chunk is linked into a bin via the free forward link and free backward link in its CCB.

setup

The following is an example of how to define a heap:

```
u32 const binsz[] =
/* bin  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, \
/* bin  13   14   15   16   17   18   19   20    21    22    23 */
      128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, \
/* bin  24    25    26    27    28       end */
      1536, 1664, 1792, 1920, 2048, -1};

#if defined(__IAR_SYSTEMS_ICC__)
#pragma data_alignment = SB_CACHE_LINE /* cache align in RAM */
#endif

HBCB  bin[(sizeof(binsz)/4)-1];   /* main heap bins */
EHV  hv;  /* heap variable array */
```

```
#if EH_STATS
u32   bnum[(sizeof(binsz)/4)-1];  /* number of chunks per bin */
u32   bsum[(sizeof(binsz)/4)-1];  /* sum of chunk sizes per bin */
#endif

#if EH_BP
BPCB bpcb[2];  /* two block pools */
#endif
```

binsz[] defines the bin structure for the heap. In the above example, 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. This is the standard bin configuration shipped with SMX, which is intended to provide a good starting heap for most systems.

Bins can easily be added, removed, or resized simply by changing the constants in binsz[] and recompiling. Then 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 binsz[]. This makes it easy to experiment with different bin configurations to see which produces the best performance. Bin arrays of 5 bins, even 1 bin are practical for small heaps.

Note in the above example that binsz[] is put into ROM. The latter is for improved reliability, so it cannot be accidentally changed. However, if available ROM is too slow, binsz[] may be put into SRAM.

bin[] is the actual bin array. Its size is determined from binsz[]. Each entry consists of a *free forward link, ffl,* and a *free backward link, fbl. hv* is the *heap variable structure*. It contains all variables needed to manage the heap. These are defined in EHV in eheap.h. It requires about 128 bytes, depending upon options. *bnum[]* and *bsum[]* are required if heap statistics are enabled. *bpcb[2]* is an array of two pool control blocks which is required if block pools are enabled. Note: These names are used in discussion that follows. Obviously, each heap requires unique names since all heaps and partitions are linked together.

initialization

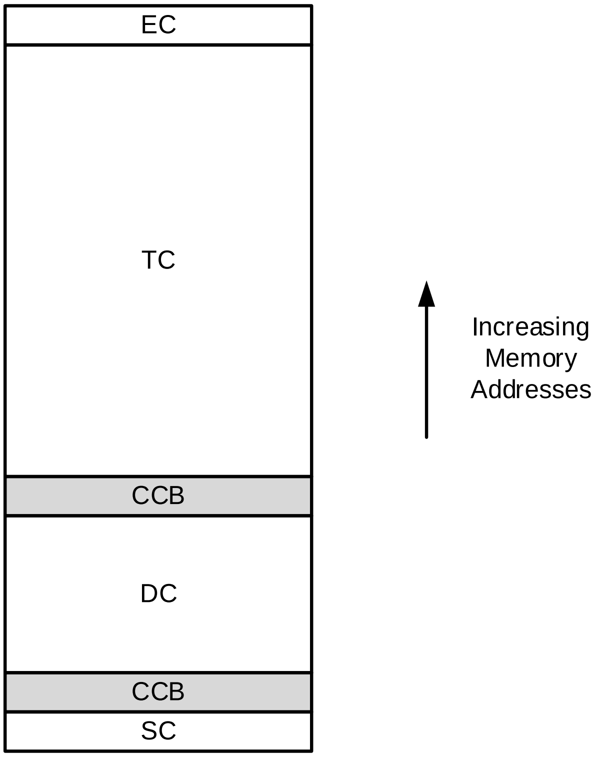
The initialization code for each partition that requires a heap should contain the above plus code such as the following:

```
memset((void*)&hv, 0, sizeof(EHV));
hv.bszap = (u32*)binsz;
hv.binp  = (HBCB*)bin;
smx_HeapInit(hsz, dcsz, haddr, &hv, (EH_CM | EH_FILL | EH_EDA | EH_EM | EH_PRE));
```

This code clears the hv structure, initializes the bzsap and binp fields, sets fill mode to on, then calls smx_HeapInit(), which initializes the heap and returns the heap number. hn. hsz is the size of the heap in bytes, dcsz is the size of the donor chunk in bytes, haddr is the starting address of the heap, and &hv is the address of the hv structure. It then enables chunk merge, fill, error display all, error manager, and preemption. Other mode flags are turned off. EH_PRE means that the heap is protected from preemption by some mechanism outside of eheap. (See the multitasking section of the eheap User's Guide for more information on this.) Space for the main heap is likely to be allocated by the linker command file. Space for smaller, dedicated heaps may be allocated from the main heap or by the linker command file.

eheap maintains an array of hv pointers, hvp[EH_NUM_HEAPS], and hn is the index into it. Hence the hv structure is accessed as hvp[hn]. See smx_HeapInit() description in the smx Reference Manual for more information.

Immediately after initialization, all free heap space is in the *donor chunk*, DC, the *top chunk,* TC, and in the 8-byte and 12-byte block pools, if enabled. The following figure shows a heap without block pools following initialization:



SC is the *start chunk*, EC is the *end chunk*, and CCB is a *chunk control block*. Until bins begin filling up due to smx_HeapFree() operations, SBA-size chunks will come from DC, and larger chunks will come from TC. This results in small chunks being in lower heap and large chunks being in upper heap. This helps to reduce fragmentation caused by small, inuse chunks getting between larger free chunks, thus blocking them from being merged.

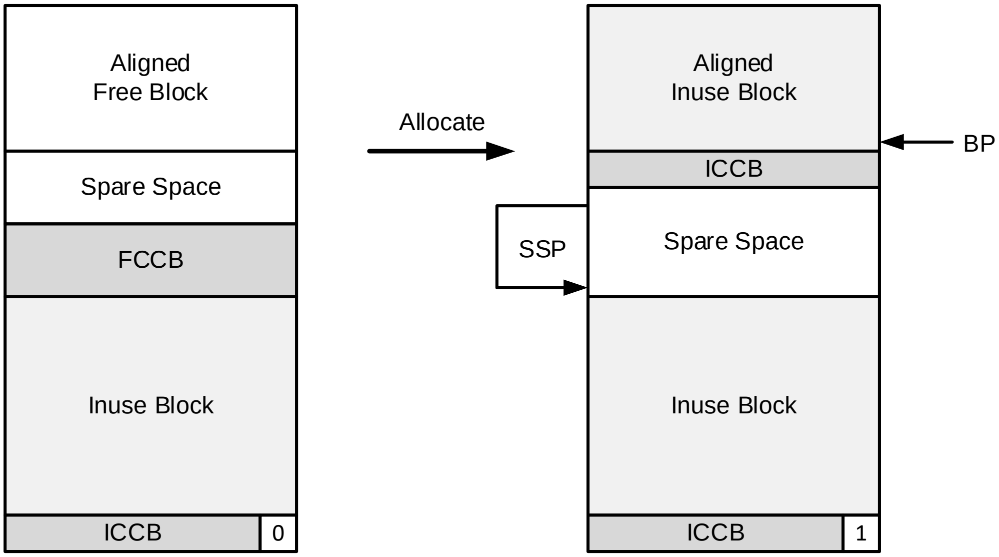
operation

In the following discussion, the chunk size needed is determined by adding CHK_OVH to the requested block size. CHK_OVH is 8 bytes for an inuse chunk and 24 bytes for a debug chunk (see **debugging**, below). In what follows, a *small chunk* is a chunk of SBA size, and a *large chunk* is larger than SBA size.

Normal block allocation for small chunks proceeds 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 heap allocation possible with eheap.

Normal block allocation for large chunks proceeds as follows: selected upper bin -> larger bin -> TC. The selected bin is found with a binary search on binsz[]. If the selected upper bin is occupied, the first big-enough chunk is taken. If the bin is a small bin[1], this will be the first chunk in it. If the bin is a large bin, the bin's free list is searched until a big-enough chunk is found. 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.

When a chunk is allocated from a large bin, it is likely to be larger than necessary. The unused space following the data block or fences above is called *spare space*. If the spare space is large enough, it is split off into a new free chunk. Otherwise, the Spare Space Pointer flag, EH_SSP, is set in blf (bit 2) and the spare space pointer, ssp, is loaded into the last word of the chunk, as shown below (Note: in this and following diagrams memory addresses are increasing upwards):



Aligned block allocations are enabled by EH_ALIGN. Only power-of-two alignments are implemented. In order to prevent errors, the alignment parameter is the alignment number, **an**, as follows:

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

For example: an = 5 corresponds to 2^5 = 32 bytes. an = 0 to 3 result in an 8-byte aligned block, which is the minimum alignment for the heap However, for a block from the 12-byte block pool, the block may be 4-byte aligned, unless an == 3.

For an > 3, an aligned block search is performed. The aligned search is like a normal block search, except that for each candidate chunk, the distance, d, to the next 2^an boundary is added to sz in order to determine if a large-enough chunk has been found. If the block in the chunk is already aligned, then d == 0. When a big-enough chunk is found, its CCB is moved up under the new aligned block and the spare space below is merged into the prechunk (preceding chunk) or split off as a new free chunk. Aligned allocation has been extended to region block alignment, which allocates MPU regions for Cortex v7M processors. See the eheap User's Guide for more information.

---

[1] The UBA may contain small bins.

Block free is performed by smx_HeapFree(). If merging is enabled (mode.fl.cmerge == ON), smx_HeapFree() merges the chunk with free prechunks and postchunks, if any. Merged chunks are removed from bins, unless they are DC or TC, and the final merged chunk is put into a bin, unless it is DC or TC. Only upward merges into DC or TC are permitted (i.e. the chunk is below DC or TC). Spare space in an inuse prechunk will be merged into the freed chunk, if EH_SS_MERGE.

For a small bin, the freed chunk is put at the start of the bin's free chunk list. For a large bin, if the freed chunk is smaller or equal to the first chunk in the bin, it is put ahead of it. Otherwise it is put at the end of the bin, which helps the large bin sort algorithm.

Deferred merging of freed chunks is possible due to double linking of chunks in eheap. xheap merging is controlled by *mode.fl.cmerge*, which is set by:

```
smx_HeapSet(EH_ST_MERGE, ON/OFF);
```

Merging free chunks undermines the effectiveness of bins. For example, a 24-byte chunk is freed to bin 0 and a physically adjacent 48-byte free chunk resides in bin 3. If mode.fl.cmerge is ON, these chunks will be merged into a 72-byte chunk, which will be put into bin 6. Bins 0 and 3 thus have leaked chunks to bin 7. This hurts performance if the application needs 24 and 48-byte chunks and not 72-byte chunks. What is needed is an algorithm to maximize bin populations while avoiding fragmentation failure – see **heap optimization** below.

Two integrated block pools have been added to eheap to provide 8- and 12-byte blocks, primarily for C++ applications, which tend to allocate very large numbers of very small blocks for objects. Block pools are enabled by EH_BP in eheap.h. If EH_BP is 0, all block pool code is omitted. Block pools are very fast and very memory efficient. The advantage of integrated block pools is that if a pool is empty, the block is taken from the heap. Thus, the block pool can be sized to meet normal demands with the heap backing it up for peak or unexpected demands.

Block pools are initialized by smx_HeapInit(), depending upon their *PCB.num_blks* fields. If 0 the block pool is not created, else it is created with num_blks. Blocks are allocated by smx_HeapMalloc() prior to searching the heap. If the pool is empty, sz is increased to 16, and allocation comes from the heap. Blocks are freed by smx_HeapFree(), which frees a block to either pool or the heap depending upon where bp points.

heap debugging

Debug mode is intended to help find problems such as buffer overflows and memory leaks. When mode.fl.debug is ON, a heap is in *debug mode*. In this mode, all chunks allocated are debug chunks rather than inuse chunks. This includes not only smx_HeapMalloc() and smx_HeapCalloc(), but also smx_HeapRealloc(), even if the initial chunk was an inuse chunk. xheap can have any mixture of free, inuse, and debug chunks.

A debug chunk is a special form of an allocated chunk (in fact, its INUSE flag, blf bit 0, is set). Its format is as follows:

fl	physical forward link
blf	physical backward link + flags
sz	chunk size, in bytes
time	time of allocation (etime)
onr	task or LSR that allocated this chunk
fence	first fence
fences	pre-block fences
data block	
fences	post-block fences

The part above the pre-block fences is the *Chunk Debug Control Block, CDCB*. fl and blf in it are common to all chunks, except that its DEBUG flag, blf bit 1, is set. sz is the size of the chunk, not of the block. time is the time when the chunk was allocated. onr is the task or LSR that allocated the chunk. The final field in the CDCB is the first fence. A fence is a one-word pattern, EH_FENCE_FILL (0xaaaaaaa3) defined in eheap.h. Any pattern may be used, but bits 1 and 0 in the pattern must be 1. These flags are necessary to determine the chunk type.

Clearly a debug chunk can be much larger than an inuse chunk, which could be a problem for a tight heap. Hence, it is possible to turn debug ON at the start of a suspected function and OFF when it ends. It can also be turned on only while a suspected task is running. This can be done by turning it ON in the *enter routine* hooked to the task and OFF in the *exit routine* hooked to the task. This permits using debug chunks only for new code being debugged and inuse chunks for already debugged code.

The debug mode can be safely set or reset directly in hooked routines, since they cannot be preempted. For example:

```
hvp[hn]->mode.fl.debug = ON/OFF;
```

The number of pre-block and post-block *fences* is determined by EH_NUM_FENCES in eheap.h. Fences are useful to:

(1) Detect block and stack overflows and block underflows.

(2) Show the overflow footprint, in order to help identify its source.

(3) Protect CCBs so a system can continue running if the overflow does not exceed the extent of the fences.

The time and onr fields in the CDCB are useful to track down memory leaks. For example, the following chunks would be suspect:

(1) An old chunk, unless it has a permanently allocated block.

(2) A chunk allocated by a task that has been deleted or stopped.

(3) An old ISR or LSR chunk since it should have been passed on to a task and freed by the task after it processed the data.

Even if the above are valid inuse chunks, they may indicate poor coding practices that should be corrected.

Debug chunks can be freely intermixed with inuse and free chunks. Their main difference is their size. This may cause some different behavior during debugging:

(1) A debug chunk may come from a higher bin than the corresponding inuse chunk.

(2) Allocation of debug chunks is slower than inuse chunks, due to the need to fill in fences and extra fields.

(3) Allocation could be slower due to higher bins not being populated or coming from a large bin vs. coming from a small bin for the inuse chunk.

(4) If an inuse chunk is mistaken for a debug chunk (when viewed in a debugger), the fields after blf will be garbage. Be sure to check the DEBUG flag, blf bit 1.

These are not show-stoppers, but they may cause confusion when looking at bins via a debugger window. On the positive side, the heap fences stand out in a debugger memory window and clearly delineate data blocks.

Fill mode is another debug aid. mode.fl.fill 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. The heap fill patterns are defined in eheap.h:

```
EH_DATA_FILL     0xDDDDDDDD
EH_FREE_FILL     0xEEEEEEEE
EH_FENCE_FILL    0xAAAAAAA3
EH_DTC_FILL      0xCCCCCCCC
```

If EH_BT_DEBUG, smx_HeapInit() fills DC and TC with the EH_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 is equivalent to filling the entire heap and can increase boot time significantly.

When fill mode is ON, the data blocks of inuse and debug chunks, are filled with EH_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. Spare space above the data block (except ssp) is filled with EH_FREE_FILL. These are helpful to see what chunks are allocated, how much of each data block is being used, how much spare space is in the chunk, and whether or not data has overflowed the spare space. If a chunk is freed with mode.fl.fill mode ON, EH_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 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 is helpful for debugging heap problems in new code without impacting other parts of a system.

Error detection is 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. eheap has a 3-level error reporting system controlled by mode.fl.ed_en =

0	none
1	all but allocation and free
2	all

During debugging, the error reporting level should be set at 2. It is convenient to load hvp[hn] into the debugger watch window and pay attention to the errno field. Seeing an error pop up can save a great deal of debug time chasing the wrong problem. Even better, put a breakpoint at the start of smx_EM() to catch heap errors the moment they occur. Then, using the call stack window it is easy to pinpoint the cause of the error. A typical error might be smx_HeapFree(bp) where bp -> garbage. This will cause an EH_INV_PAR error.

Using built-in error detection can save wasted debug time chasing what appear to be serious errors, which in fact are just uninitialized pointers, wrong sizes, etc. Error handling is discussed more in the **reliability** section below.

Heap information is another heap debug tool. 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 *smx_HeapChunkPeek(vp, par, hn)* and *smx_HeapBinPeek(binno, par, hn)***.** The first, given a block pointer, will return the chunk pointer and using it information such as chunk type, size, binno, previous chunk, next chunk, etc. may be obtained. The second returns information about the bin such as number of chunks in it, first chunk, last chunk, etc.

See the eheap User's Guide for discussion of heap debugging problems and techniques

heap optimization

Tunability is a necessary characteristic for embedded system heaps. Theoretically, it is not possible to design one heap 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 heap, such as dlmalloc, is good at satisfying the needs of most applications. It has the advantage of a large amount of memory and it can usually get more memory, if needed.

This situation is different for most embedded systems. Typically, memory is in short supply, which is exacerbated by the need for multiple heaps to achieve security requirements. Although there is a wide variation of characteristics from one embedded system or partition to the next, a given embedded system or partition typically has 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.

Optimizing bin arrays is a good starting point. The bin structure of xheap is adjustable to suit a wide range of requirements. The best plan is to initially run with the standard bin configuration in order to get the application software running in its final form. Then record the sizes being used and optimize the bin structure accordingly.

For a system using a large variety of chunk sizes, an evenly spaced bin array, such as the standard bin configuration, is probably 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 standard heap bins could be optimized, as follows:

```
/* bin  0   1   2    3   4    5   6   7    8    9   10   11 */
    {24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, \
/* bin  12   13   14   15   16    17   18   19    20    21    22    23 */
     120, 128, 208, 408, 512, 640, 768, 808, 1024, 1152, 1208, 1408, \
/* bin  24     25     26     27     28   end */
     1536, 1664, 1792, 1920, 2048, -1};
```

Bold numbers indicate the bin sizes that have been changed (remember that bin sizes are chunk sizes). Due to the way that free() works, chunks of these sizes will always be put in the front 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 the last-freed chunk first, cache hits for accesses to the blocks in these chunks also improve.

However, if there are a large number of 808-byte chunks in bin 19, for example, then search times for larger chunks in the bin (e.g. 816, 1016, etc.) may become too long. This can be solved by making bin 19 a small bin containing only 808-byte chunks and starting bin 20 at 816. If this is too many chunk sizes for bin 20, bin 21 could be started lower, thus reducing the number chunk sizes in bin 19. An alternative solution would be to turn cmerge on while excessive 808 chunks were being freed. This should merge them into larger chunks that are put elsewhere.

Smaller bin arrays can be used for partitions requiring dedicated heaps if they require only a small number of block sizes and do not require high performance. In these heaps, simple bin arrays should be adequate, such as the following:

```
/* bin  0    1      2      3      4   end */
    {24, 512, 1024, 1536, 2048, -1}
```

This covers the same range, as the standard array above, with only 5 bins. Note that these are all large bins and that there is no SBA. Also, DC 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 4 act like five 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 frequently used chunk sizes, then combined with bin sorting, this heap should be much faster and more deterministic than a simple linear heap.

To go even smaller, consider:

```
/* bin  0  end */
    {24, -1}
```

This defines a one bin heap. bin 0 handles all chunk sizes from 24 bytes and up. This heap would be appropriate for a very small partition, with a tiny heap space, such as 10 KB. Like the previous heap, there is no SBA nor is DC used, but TC still exists. A one-bin heap has many of the xheap 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 much faster allocations.

(2) TC provides a fast start up for allocations and is the source of last resort if a desired chunk is not in the bin.

(3) Bin sorting and deferred merging ensure 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 longer allocation times for large blocks do not necessarily reduce system performance because processing large blocks takes longer than processing small blocks. smx_HeapFree() puts very small chunks at the start and all other chunks at the end of the bin free list. Then, smx_BinSort() will quickly move small- and medium-size chunks back to where they belong. Hence, operation of the one bin heap could be pretty good.

Free chunk merging is controlled by *mode.fl.cmerge*, which can be turned ON or OFF via smx_HeapSet(). For some heaps it may be best to leave cmerge continuously ON. In other heaps, bin leakage due to free chunk merging may hurt performance. Unfortunately running with cmerge OFF increases the risk of allocation failures due to excessive external fragmentation. As noted previously, some bin leakage is caused by chunk splitting, even if cmerge is ON. Increasing EH_MIN_FRAG helps to reduce this, but increases internal fragmentation, which may also cause allocation failures. Hence, avoiding excessive bin leakage while also avoiding heap failure due to fragmentation is not easy.

xheap implements an auto-merge control algorithm, which is enabled if *mode.fl.amerge* is ON. It is implemented as follows:

```
if(hvp[hn]->mode.fl.amerge == ON)
{
    tblcp = bin[hvp[hn]->top_bin].fbl;
    tblcsz = (tblcp == NULL ? 0 : tblcp->sz);
    tcsz = hvp[hn]->tcp->sz;

    if ((hvp[hn]->hused > SMX_CFG_HEAP_USE_MAX) ||
       (tblcsz < SMX_CFG_HEAP_CSZ_MAX &&  tcsz < SMX_CFG_HEAP_CSZ_MAX))
          hvp[hn]->mode.fl.cmerge = ON;

    if ((hvp[hn]->hused <= SMX_CFG_HEAP_USE_MIN) &&
       (tblcsz >= SMX_CFG_HEAP_CSZ_MAX ||  tcsz >= SMX_CFG_HEAP_CSZ_MAX))
          hvp[hn]->mode.fl.cmerge = OFF;
}
```

This code is part of the smx_HeapManager(), which is called from the idle task, so it runs only during idle time. If mode.fl.cmerge is ON, merging of adjacent free chunks by smx_HeapFree() is enabled. If it is OFF, merging is disabled. hused is the current number of heap bytes allocated. It is increased, by the chunk size when a chunk is allocated and it is decreased, by the chunk size, when a chunk is freed.

In the above code, cmerge is turned ON if the heap in use exceeds the maximum value, SMX_CFG_HEAP_USE_MAX. In addition, it may be turned ON if a maximum size chunk, SMX_CFG_HEAP_CSZ_MAX, cannot be allocated either from the top chunk or from the top bin. (The top bin is assumed to be sorted so only its last chunk is tested.) After a period of time, free chunk mergers should reduce the amount of heap in use. Thus, when hused drops below or equal to SMX_CFG_HEAP_USE_MIN, cmerge is turned back on provided that the maximum size chunk can now be allocated. SMX_CFG_HEAP_USE_MAX, SMX_CFG_HEAP_CSZ_MAX, and SMX_CFG_HEAP_USE_MIN defined in acfg.h are configuration constants that can be adjusted for best operation.

It is important to define maximum chunk size, SMX_CFG_HEAP_CSZ_MAX, to help ensure that the largest needed chunk can be allocated at all times. If there is frequent 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.

Dynamic merge control can also be implemented at the task level. For example, cmerge can be turned OFF by tasks, which are heavy heap users. Tasks written in object-oriented languages are likely to be such tasks. Inhibiting merging while these tasks run can improve performance by avoiding leaky bins, particularly in the SBA. cmerge can be turned ON in an **exit routine** hooked to the task and OFF in an **enter routine** also 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 the small chunks it allocated will be merged and become available for larger allocations by other tasks.

Allocation failure is not necessarily catastrophic – see discussion of recovery under **reliability**, below.

Bin seeding provides an alternative to deferred merging. The *smx_HeapBinSeed(num, bsz, hn)* 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 the debug mode. This service allocates a big-enough chunk from higher up, splits it into num chunks the right size for blocks of bsz, and frees the chunks to their correct bin. While in operation, cmerge is ignored.

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

```
void FillBins(u32 hn)
{
    u32 bn, bsz;
      for (bn = 0; bn <= hvp[hn]->sba_top; bn++)
      {
          if (smx_HeapBinPeek(bn, SMX_PK_COUNT, hn) < 2)
          {
                bsz = smx_HeapBinPeek(bn, SMX_PK_SIZE, hn);
                smx_HeapBinSeed(2, bsz, hn);
          }
    }
}
```

FillBins() could be called from the idle task in order to keep 1 to 2 chunks in each SBA bin. This would ensure fast allocations for small chunks. Since the new chunks are not separated by inuse chunks they may soon leak out and unite with other free chunks if cmerge is ON. For this reason, it is probably best to only seed a few chunks at a time and to do so frequently. Two is a good number because the second chunk will be allocated first and will provide an inuse barrier to protect the first chunk.

Debug mode must be OFF for the above code to function as expected. Otherwise chunks will be put into higher bins than expected. Furthermore, the last chunk may be larger than the other chunks due to spare space left in it, and thus it may be put into a higher bin than them.

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 is done in order of increasing size in large bins to improve their performance. (Bin sorting is not necessary for small bins, since they have only one chunk size.) If a large bin is fully sorted, since a large-bin allocation takes the first big-enough chunk, it will be the *best-fit chunk* available in the bin. As a consequence, there will be less splitting of chunks, which improves performance and reduces fragmentation.

Embedded applications must have significant average idle time to deal with the peak loads caused by simultaneous asynchronous events. Also, they must be designed with spare processing time in order to handle future added features. eheap takes advantage of idle time to sort large bins. If there is enough idle time, large bins should almost always be well-sorted. Though poorly sorted bins may cause suboptimal performance, they do not cause heap failures and thus should be acceptable during periods of peak loading, when there is no idle time.

Bin sorting is done by calling *smx_HeapBinSort(binno, fnum, hn)* to sort the bins. This is normally done from a heap manager, which is called periodically by the idle task. A run consists of testing fnum chunks and moving those ahead that are smaller. Dividing a sort into runs is necessary to permit higher-priority tasks that need access to the heap to run, without missing their deadlines. The *binno* parameter in smx_HeapBinSort() specifies the bin to sort. If it is not a valid bin number all bins are sorted that need to be. smx_HeapBinSort() returns TRUE when a bin has been sorted, or if no bins needed to be sorted. Since smx_HeapFree() puts larger-than-first chunks at the end of the bin free list, and smx_HeapBinSort() moves them to where they belong on the first pass, single-pass sorting is possible if done frequently.

reliability

Most embedded systems are expected to run indefinitely without supervision. In addition, some are placed in hostile environments subject to extreme temperatures, voltage transients, large fluxes of high energy particles, etc. And then there is hacking and malware, which is an increasing problem. eheap provides several features to minimize damage from errors and to achieve self-healing.

Error handling: xheap reports the following errors:

SMXE_HEAP_ALRDY_INIT
SMXE_HEAP_BRKN
SMXE_HEAP_FIXED
SMXE_HEAP_ERROR
SMXE_HEAP_FENCE_BRKN
SMXE_INSUFF_HEAP
SMXE_INV_CCB
SMXE_INV_PAR
SMXE_HEAP_RECOVER
SMXE_TOO_MANY_HEAPS
SMXE_WRONG_HEAP

See the smx_Heap call descriptions in the smx Reference Manual for a description of what each error means relative to each heap service. These errors are detected by eheap and mapped to smx errors by the shell functions in xheap.c.

eheap does extensive error detection and reporting, and xheap does extensive error reporting, recording, and handling. This is great during debugging and is highly recommended during normal operation to detect and record latent bugs and to help detect and thwart hacking. However, these features may hurt performance of code that does intensive block allocations and frees, such as object-oriented code. eheap and xheap provide methods to deal with this problem.

eheap has a 3-level error reporting system controlled by mode.fl.ed_en in the heap EHV:

0	none
1	all but allocation and free
2	all

For released software, level 2 is recommended for best reliability and security. However, if performance is more important, level 1 skips error reporting for allocation and free operations and level 0 skips all error reporting. It is important to note that errors are still detected and appropriate actions taken. For example, in smx_HeapMalloc() if sz = 0, NULL is returned. This is necessary to protect heap integrity and to avoid spurious MMFs (Memory Manage Faults – see the SecureSMX User's Guide) and other faults.

In addition, mode.fl.em_en == ON is used in xheap shell functions to enable calling smx_EM() when eheap reports an error. smx_EM() records errors and may perform other functions, depending upon the error. If mode.fl.em_en == OFF, smx_EM() is not called. This allows application code to do its own heap error handling, which may be little or nothing. If mode.fl.ed_en == 2, it is possible to determine the error from hv.errno.

Fragmentation becomes a problem when many small inuse chunks are interleaved with medium-size chunks, thus preventing them 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 at a lower priority and try again, or less important tasks could be stopped and their heap blocks freed with merging enabled to build up larger chunks.

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 seldom used. When freed, these chunks will be put into large bins. If such a chunk is larger than the chunks normally allocated from that bin, it may sit in the bin for a long time until the bin runs out of smaller sizes.

one-shot tasks may help where insufficient heap failures are due to insufficient RAM for the heap. 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 blocks 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.

self-healing is becoming more important with increasing IoT deployment. 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 may exacerbate these problems.

In addition to bit flips, heaps are vulnerable because control information (i.e. CCBs) is sandwiched between data blocks. Data block overflows damage heap control information, again sending the system into the weeds. This kind of damage usually results 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.

eheap implements heap and bin scanning to help overcome these problems.

smx_HeapScan(cp, fnum, bnum, hn) 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 can be called once per idle task pass, 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. It fixes broken forward links either by using the size in free chunks or going to the end of the heap and scanning backwards. Each time called, it will test fnum chunks, if testing, or bnum chunks, if fixing.

In the debug version (EH_BT_DEBUG defined), the scan stops on a broken fence so that the fence can be studied for clues to what happened. In the release version, the fence is fixed. Fixes are reported so they can be saved for 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 MMFs and data abort exceptions, or equivalent, if they are broken.

smx_HeapBinScan(binno, fnum, bnum, hn) scans the bin free lists. It is similar to smx_HeapScan(): it incremental, scans doubly-linked chunk lists, and fixes broken links, when it can. It has four parameters: binno, the bin number, fnum the forward run limit and bnum the backward run limit, and hn. Like heap scan it returns FALSE until it is done with the bin or an unfixable error is encountered. Whenever a fix is made, EH_HEAP_FIXED is reported. This can be used to monitor how often problems are being found and fixed. A high rate could be indicative of a hacker attack or memory about to fail.

MTBF improvement: A modest heap of 10,000 chunks has 2 pointers per inuse chunk and 4 pointers and 1 size per free chunk. Assuming 75% of the heap is inuse there are 7500*2 + 2500*5 = 27,500 control words in the heap. Ignoring chunk splitting, assume an average malloc searches for 2 chunks and then dequeues the chunk found in a bin. This requires 4 pointer accesses. Ignoring chunk merging, an average free needs to access one pointer and a size to compare the chunk, then to access either 1 more pointer or 2 more pointers to enqueue the chunk, each half the time = 3.5 control words, average. If 100 mallocs and 100 frees occur in the time needed for one scan, 4*100 + 3.5*100 = 750 control word accesses are required. Hence the probability of a bad pointer or size access is 100*750/27500 = 2.7%. Thus, broken control words will be fixed before they are used 36 out of 37 times that a bit flip occurs – clearly a big MTBF improvement!

Assuming the 100 mallocs and 100 frees take 20 seconds, it is necessary to scan 500 chunks per second in the heap or 2 per tick and 125 chunks per second in the bins or about 1 per tick, at the default tick rate.

smx_HeapRecover(sz, num, an, hn), is provided to deal with heap failure due to excessive fragmentation. 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. smx_HeapRecover() starts the scan from the start of the heap at SC for small chunks or from DC for large chunks. All scans go to the end of the heap at EC before quitting. If a big-enough chunk can be formed by merging adjacent free chunks, it removes the free chunks (except DC and TC) from their bins and merges them. If the merged chunk is not DC nor TC, it puts the merged chunk into its proper bin, else it updates dcp or tcp, then returns TRUE.

Recovery will be done automatically if mode.fl.auto_rec in EHV is ON. Assuming a big-enough chunk is produced, recovery will be transparent to the application, except for the time taken. If mode.fl.auto_rec is OFF, smx_HeapRecover() can be called directly, or some other recovery method can be used.

smx_HeapExtend(xsz, xp, hn) permits adding more memory to the heap, if all else fails. 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 DRAM, in which case access to chunks in the extension would be slower, but this is preferable to a system failure. If this happens, it is likely that there would be only a few slow chunks, so performance might suffer for only a few unlucky tasks. Note that doing cache-line-aligned accesses could greatly improve data block access times if in DRAM.

Chapter 6 Stacks

smx supports a system stack and task stacks.

system stack (SS)

SS is used for initialization, Interrupt Service Routines (ISRs), trusted Link Service Routines (tLSRs), the scheduler, the smx Error Manager (smx_EM), and exception handlers such as the ARM-M SVC and PendSV handlers. By handling these requirements, SS allows task stacks to be much smaller. Otherwise, the total SS requirement would need to be added to each task stack. For ARM-M, SS is also known as *Main Stack, MS*.

During operation, the SS size requirement is relatively small (on the order of 500 bytes). Its size is determined by the requirement for the PendSV handler + the scheduler + the largest LSR + the largest SSR + smx_EM() + the amount of stack needed for ISRs and exception handlers since they can interrupt. If ISR nesting is permitted, enough stack must be added to allow for maximum ISR nesting. Usually this means the sum of all ISR stack requirements. In order to help determine a good size for SS, SS is filled with SB_STK_FILL_VAL, during initialization. Thus after a long run, looking at SS will determine if a reasonable safety margin exists.

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 are typically controlled by the linker command file, where it is referred to as SVC_STACK for ARM and CSTACK for ARM-M. It is recommended that SS be located in on-chip SRAM to achieve the best performance for ISRs, LSRs, and smx. 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 task stacks

smx supports two types of task stacks: *permanent* stacks and *shared* stacks. Permanent, or *bound*, stacks are the conventional task stacks provided by most RTOSs. A permanent stack may be preallocated or may be obtained from a heap during task create and is permanently bound to the task. The stack size is specified as a parameter in smx_TaskCreate() and may be any size up to 64KB. See smx_TaskCreate() in the smx Reference Manual for more information.

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 small stacks.

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

(3) Each stack is isolated by heap mechanisms.

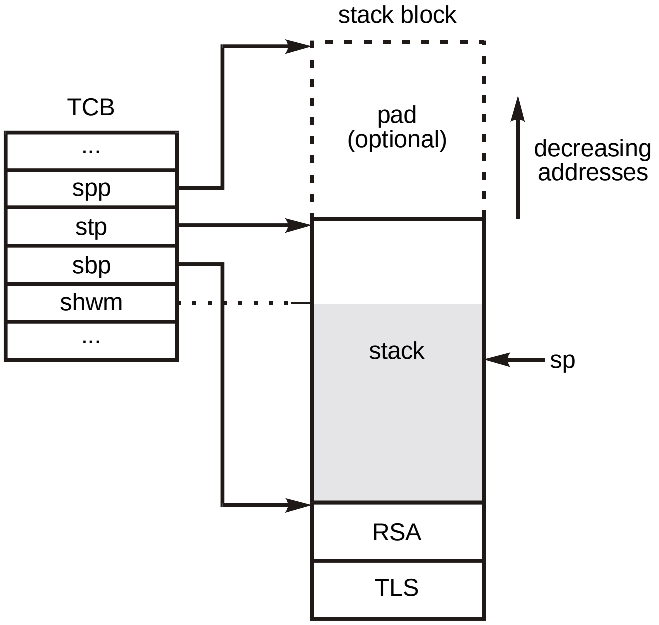
shared task stacks

Shared, or *unbound*, stacks are unique to smx. They are intended to reduce RAM usage by allowing stacks to be shared between tasks. A shared stack is obtained from the stack pool when a task is dispatched by the scheduler. All shared stacks are the same size. The stack pool is allocated from the main heap the first time a task is created. The configuration parameters SMX_CFG_NUM_STACKS and STACK_SIZE, defined in acfg.h, determine the number and size of the stack blocks. If either is 0, no stack pool is created.

Tasks using shared stacks are called *one-shot* tasks. Shared stacks are good for small tasks which seldom run. Several such tasks can share a few stacks from the stack pool. If a stack is not available when a task is ready to run, the task is skipped over until a stack is available. For more information see Chapter 15 One-Shot Tasks.

task stack control

Task 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 and "bottom" means the highest address of the stack. 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 optional stack pad, which is above the stack.

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. xarmm.h for ARM-M). 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 and tool chain (e.g. xarmm_iar.inc).

The optional *Task Local Storage (TLS)* area size is specified by the upper 16 bits of the tlssz_ssz parameter of smx_TaskCreate(). It can be accessed by sbp + SMX_RSA_SIZE and is useful to store task-specific data, such as a structure. For more information, see the TLS discussion in the SecureSMX User's Guide.

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.

task stack sizes

task stack block size = SMX_CFG_STACK_PAD_SIZE + size of task stack + SMX_RSA_SIZE + TLS size. SMX_CFG_STACK_PAD_SIZE is defined in acfg.h. It normally is set to a value such as 40 during debugging and 0 for release. Register Save Area size, SMX_RSA_SIZE, is 32 bytes for ARM-M. Task Local Storage size, TLS, is optional. A task stack must be large enough for maximum function nesting within the task. Functions use the stack for parameters and auto variables. Interrupts and exceptions use it for stack frames. Complex functions can use 100 bytes of stack, and if nested 5 deep, the task could easily need a 500 byte stack.

Stack blocks allocated from a heap are automatically sized according to the foregoing. Preallocated and stack pool stack sizes must be pre-calculated. In addition the stack must be 8-byte aligned, as required by Arm AAPCS. If there is no TLS, for a preallocated stack this is met by aligning the stack block on an 8-byte boundary. The stack block pool is aligned on an 8-byte boundary. Hence, if SMX_CFG_STACK_SIZE is a multiple of 8, then all shared stacks will be properly aligned on 8-byte boundaries.

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-M, 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.

Minimizing stack space will pay off if task 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, because following a task switch there may be many cache misses for stack accesses.

task stack pads

Stacks can be padded by setting SMX_CFG_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 after 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 can 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 task stack sizes

To finalize stack sizes, it is best to allow the system to run for an extended period of time using moderate-size stack 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 may use more or less stack space. Because stack scanning records the exact maximum stack usage, it provides 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 version — as little as 2 words is helpful. The advantage of this is that although smx will detect and report any stack overflows with or without stack pads, having stack pads may 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 crash.

creating and filling task stacks

If SMX_CFG_STACK_SCAN in xcfg.h, is 1, task stacks are filled with the 32-bit SB_STK_FILL_VAL, defined in the smxBase processor-architecture header file (e.g. barmm.h). This can be set to any desired pattern. One stack is scanned for each pass of smx_IdleMain(). Unbound stacks are given priority in order to speed up one-shot task dispatching. (A one-shot task may be waiting for a stack.) The stack is scanned from task->spp down to the end of the pattern and task->shwm is updated, unless the task has been restarted with another stack. If no unbound stacks are waiting to be scanned, the next bound stack is scanned and task->shwm is updated.

task 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 flag 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 task->sp < task->stp or task->shwm >= task->ssz, an overflow has occurred and smx_EM() is called and SMXE_STK_OVFL is reported.

task stack overflow handling

Since a stack overflow is considered to be an irrecoverable error, smx_EM() calls the smx_EMExitHook() callback routine in smxmain.c to allow you to decide what to do. Currently, smx_EMExitHook() does a few reasonable operations and does a DEBUGTRAP (breakpoint) if in debug mode or stops and issues a "TASK STOPPED" warning, if not in debug mode. task->flags.stk_chk is reset to avoid repeating the error reporting, in case the task is allowed to continue, as may be the case during debugging. This does not affect other task stack checking.

Obviously task stack overflow is a serious problem. Structures above, such as a heap chunk control block may have been damaged, rendering further operation problematic. If there is a stack pad and it has not been exceeded, then it may be possible to just restart a one-shot task with a new stack or to delete, recreate, and restart a normal task. However, it may be safer to reboot the whole system or partition .

foreign task stacks

Some third-party libraries switch to *foreign* stacks elsewhere in memory. 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_TaskSet (smx_ct, SMX_ST_STK_CK, FALSE);
/* call to function that switches stacks */
smx_TaskSet (smx_ct, SMX_ST_STK_CK, TRUE);
```

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

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. Since nested subroutine calls can occur at other times, a task's *stack high water mark*, *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 it with a pattern before the task starts, then scanning periodically to see how far the pattern goes down. The address of the 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.

Except for SS scan following startup, stack scanning must be enabled by setting SMX_CFG_STACK_SCAN, in xcfg.h. This causes the stack scan code to be included. 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.

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.

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 by smx_TaskCreate(). Scanning is performed by smx_StackScanB() (B = bound) and is from task->stp 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, 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 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 to 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 smx_StackScan() calls smx_StackScanB(), and the stack of the next bound task is processed. 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 a task can even 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.

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 during idle time, it should not interfere with more important tasks, unless waiting for a stack. However, such tasks normally do not have tight deadlines. If they do, they should be given permanent stacks.

out of shared stacks

When the smx scheduler is ready to dispatch an unbound task, 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 the waiting task. While in smx_StackScanU(), smx_inssu is set. If idle is preempted while smx_inssu is set, the scheduler performs an *idleup* operation, which runs idle in place of the preempting task in order to complete running StackScanU(). When it is done, smx_inssu is reset, idle drops back to priority level 0, and restarts the scheduler, which now is able to dispatch the preempting task. 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

Detecting and protecting against SS overflow is important to system integrity, since ISRs, tLSRs, the scheduler, and even the error manager, smx_EM(), depend upon correct operation of it. 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. 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 smx_main() in smxmain.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 necessary.

At the start of application initialization (ainit() in smxmain.c), SS is again filled with the SB_STK_FILL_VAL all the way to the bottom (which is possible because SS is not being used at that time). SS is scanned by smx_StackScan() in idle, when it sees that smx_sshwmv is 0 (i.e. its high-water mark is no longer valid). 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. This is an irrecoverable error, and smx_EMExitHook() is called to handle it.

Chapter 7 Intertask Communication

introduction

Once a system has been divided into tasks, communication between the tasks is necessary 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 (ITC):

(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. The following chapters are intended to introduce you to above mechanisms and show you how to use them effectively.

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 1 tick.

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 smx_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)
```

where ptype is the parameter type, such as u32, BOOLEAN, MCB_PTR, etc. See the smx Reference Manual for details. Stop SSRs are primarily used by one-shot tasks. See Chapter 15 One-Shot Tasks, for more information.

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.

Chapter 8 Semaphores

Semaphores are used for control

```
BOOLEAN     smx_SemClear(SCB_PTR sem)
SCB_PTR     smx_SemCreate(SMX_SEM_MODE mode, u32 lim, const char* name, SCB_PTR* shp)
BOOLEAN     smx_SemDelete(SCB_PTR* shp)
u32         smx_SemPeek(SCB_PTR sem, SMX_PK_PAR par)
BOOLEAN     smx_SemSet(SCB_PTR sem, SMX_ST_PAR par, u32 v1, u32 v2)
BOOLEAN     smx_SemSignal(SCB_PTR sem)
BOOLEAN     smx_SemTest(SCB_PTR sem, u32 timeout)
void        smx_SemTestStop(SCB_PTR sem, u32 timeout)
```

introduction

Semaphores are used primarily used for signaling events and for resource control. They should not be used for mutual exclusion (see Chapter 9 Mutexes).

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(SMX_SEM_RSRC, 1, "sbr");
```

creates a binary resource semaphore named sbr. It is recommended that the name chosen for a semaphore indicate its operating mode, in some manner. (For example, “br” for binary resource,) This will help to avoid misuse. Semaphore modes are discussed in the sections that follow.

resource semaphore

Resource semaphores are used to regulate access to resources, such as block pool blocks or IO devices. The above binary resource semaphore could be used as follows:

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

```
void t1a_main(u32 par)
{
    sbr = smx_SemCreate(SMX_SEM_RSRC, 1, "sbr");
    t2a = smx_TaskCreate(t2a_main, P2, 500, SMX_FL_NONE, "t2a");  /* 500 = stack size */
    t3a = smx_TaskCreate(t3a_main, P3, 500, SMX_FL_NONE, "t3a");
    smx_TaskStart(t2a);
}

void t2a_main(u32 par)
{
    smx_SemTest(sbr, TMO)
    smx_TaskStart(t3a);
    /* use resource here */
    smx_SemSignal(sbr);
}

void t3a_main(u32 par)
{
    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 top 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, if any, are ignored, but there should be no additional signals — tests and signals should always be paired, as shown in the example. Note that a resource semaphore starts with its count already set.

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

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

Where lim is the number of resources.

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

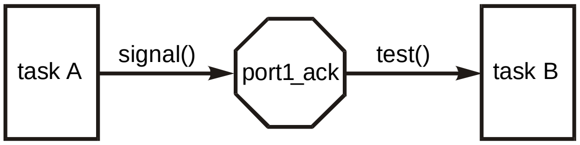
```
PCB_PTR  blk_pool;
SCB_PTR  sr;

void init(void)
{
    u8* p;

    p = (u8*)smx_HeapMalloc(1000);
    blk_pool = smx_BlockPoolCreate(p, NUM, SIZE, "blk_pool");
    sr = smx_SemCreate(SMX_SEM_RSRC, NUM, "sr");
}

void t2a_main(u32 par)
{
    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 block is returned to the caller. If not, the current task is suspended on sr. RelBlock() releases the block back to blk_pool and signals sr. If tasks are waiting for blocks, the top waiting task will get the block and resume. If no task is waiting, sr’s count is incremented so the count equals the number of available blocks. The maximum possible count is lim. Any signals after count == lim are ignored, but there should be none.

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 each other. 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 == 0 for a multiple event semaphore and 1 for a binary event semaphore. In both cases, the semaphore's internal count starts at 0 — i.e. there are no events to report.

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. Hence, it is good for situations where every event counts and missing events could have bad consequences.

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(SMX_SEM_EVENT, 0, "sme");

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

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

void t2a_main(u32 par)
{
    while (smx_SemTest(sme, TMO)) {
        /* process event A */
    }
    /* or handle timeout or error */
}
```

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

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 events or malfunctions.

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

```
SCB_PTR sbe;
sbe = smx_SemCreate(SMX_SEM_EVENT, 1, "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 produced, rather than testing the semaphore for each item produced, which is less efficient. 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. A binary event semaphore does not have the potential count overflow problem of a multiple event semaphore, because its count is limited to 1.

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(SMX_SEM_EVENT, B, "sbe");

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

```
void es4_t3a_main(u32 par)
{
    u8  ch;
    do
    {
        ch = *inport;
        smx_PipePut8(inpipe, ch);
        smx_SemSignal(sbe);
    }  while (ch);
}

void  es4_t2a_main(u32 par)
{
    u8  ch;
    char*  dp;

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

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

In the above example, t3a gets characters from inport and puts them into inpipe, until it gets a 0 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. If a multiple event were used, t2a would run once for every character put into inpipe, which would obviously be wasteful.

threshold semaphore

smx has a unique semaphore called a *threshold semaphore*. It 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. A threshold semaphore is created as follows:

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

where T is the threshold, e.g. 10. The initial count is set to 0, as with other event semaphores. The motivation for using a threshold semaphore is similar to that of using a binary event semaphore — namely to avoid unnecessary task runs. However, in this case, events are not discarded; they are all counted. T must be > 0 or SMXE_INV_PAR error is reported.

The example above serves to illustrate the threshold semaphore, with “ste” substituted for “sbe”. 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 top task waiting at st (t2a) is resumed and the signal counter is reduced by T. If no task is waiting, the count continues, but it is not allowed to be incremented over 255. It this occurs, the count stays at 255 and SMXE_SEM_CTR_OVFL error is reported.

gate semaphore

Another unique smx semaphore is the *gate semaphore*. It resumes all waiting tasks simultaneously — one signal opens the gate and all waiting tasks resume in the order suspended. To create a gate semaphore:

```
SCB_PTR  sg;
sg = smx_SemCreate(SMX_SEM_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 is 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(SMX_SEM_GATE, 1, "sg");
    st = smx_SemCreate(SMX_SEM_THRES, 2, "st");

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

/* slave task 1 */
void t2s1_main(u32 par)
{
    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(u32 par)
{
	while(1)
	{
		smx_SemTest(sg, INF);
		/* do process 2 here */
		smx_SemSignal(st);
	}
}

/* master task */
void t2m_main(u32 par)
{
	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, and 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.

other semaphore services

smx_SemTestStop(sem, tmo) is the stop variant of smx_SemTest(). It is 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 Chapter 15 One-Shot Tasks 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(*semp) 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.

smx_SemSet(sem, SMX_ST_CBFUN, cbfun) is used to load a pointer to the semaphore signal callback function, cbfun, in sem. If sem->cbfun != NULL, cbfun is called every time sem is signaled. This can be used for monitoring system operation, multiple waits, etc. See the smx_SemSignal() discussion in the Reference Manual for more information.

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

summary

The smx semaphore operates in one of 6 modes: The binary resource semaphore can be used to regulate access to one resource. The multiple resource semaphore offers the unique capability to regulate access to multiple equivalent resources, such as blocks from a block pool. The binary event semaphore is useful when only the first of a series of events need 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. The gate semaphore serves to start all waiting tasks on one signal, such as a master task starting all slaves for the next cycle.

Chapter 9 Mutexes

```
BOOLEAN     smx_MutexClear(MUCB_PTR mtx)
MUCB_PTR    smx_MutexCreate(u8 pi, u8 ceiling, const char* name , MUCB_PTR* muhp)
BOOLEAN     smx_MutexDelete(MUCB_PTR* muhp)
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 should not be used for mutual exclusion because:

(1) A task will stop running if it tests the semaphore twice without first signaling it.

(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.

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, and 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 is 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 can cause other tasks to become permanently blocked at semaphores held by the deleted task.

For the above reasons, mutexes are recommended over binary resource semaphores if memory space is available. Adding basic mutex calls to a design that is already using semaphores adds less than 1000 bytes of code. A mutex control block is 32 bytes vs. 24 for a semaphore. The overhead to signal a semaphore and the overhead to get a mutex are comparable. Mutexes are well worth the safety they add.

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. If non-zero, pi enables priority inheritance. If non-zero, ceil is the ceiling priority of the mutex. These two forms of priority promotion (ceiling and inheritance) each have advantages and disadvantages, which are discussed below.

To delete a mutex:

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

priority inheritance

Priority inheritance means that when a higher priority task waits on an owned mutex, the owner’s priority is temporarily promoted to the higher priority. If waiting in another queue, the owner is moved to the proper position in that queue for its new priority. If the queue is for a mutex and the mutex has priority inheritance enabled, then priority inheritence can repeat with that mutex. This process can propagate through as many such linked mutexes as necessary. Priority inheritance does not require knowing the top priority of tasks using the mutex. Hence, it is good for complex systems where task priorities change dynamically or are being changed to fine-tune system operation.

ceiling protocol

If the ceiling is non-zero, a task is immediately promoted to the mutex ceiling priority when it becomes the owner of the mutex. Ceiling is simpler than priority promotion and results in the fewest task switches. Also, it prevents task deadlocks. However, it requires knowing the top priority among the tasks sharing the resource and it blocks the same or lower priority tasks that do not need the resource. For example:

```
MUCB_PTR  print_ok;    /* mutex */

void  init_function(void)
{
      print_ok = smx_MutexCreate(NO_PI, P3, "print_ok");
}

void  client_function(void)
{
      smx_MutexGet (print_ok, INF)
      /* put bytes to print_out pipe */
      smx_MutexRel (print_ok);
}
```

The ceiling has been set to 3. because no print tasks have a priority greater than 3. Hence, all printing is done at this priority level.

Ceiling protocol is rather interesting in that it causes all tasks to run at the ceiling priority when they own the mutex. This makes sense if the resource is important because it will always be accessed at the ceiling priority and its users cannot be preempted by equal or lower priority tasks.

In the case where two tasks must each own two mutexes, in order to run, and both mutexes have the same ceilings, if one of the tasks owns one of the mutexes, the other task cannot run because its priority is less than or equal to the ceiling priority. Hence the other task cannot get the second mutex and cause a deadlock, thus the first task is guaranteed to get the second mutex. Therefore, the usual requirement that mutexes always be obtained in the same order to avoid deadlocks is not necessary.

The smx mutex allows a combination of ceiling and inheritance. The ceiling can be set to a medium priority level that satisfies most users, and inheritance can be enabled to take care of tasks above that level.

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, if it is higher. If an error is encountered or a timeout occurs, smx_MutexGet() returns FALSE. If called from an LSR, TRUE is returned if the mutex has no owner, else FALSE is returned. This allows an LSR to *borrow* a free mutex. (Note that an LSR cannot be preempted by a task, nor another LSR.)

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 attempts to release mtx, or it is already free, an error is reported. If an LSR calls smx_MutexRel(), TRUE is returned.

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 use when 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 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).

smx_MsgReceive(PXCHG), smx_MsgReceiveStop(PXCHG), and smx_MsgSend(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.

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, SMX_FL_NONE, "lo_task");
    hi_task = smx_TaskCreate(hi_task_main, PRI_HI, 0, SMX_FL_NONE, "hi_task");
    smx_TaskStart(lo_task);
    smx_TaskStart(hi_task);
}

void  lo_task_main(u32 par)
{
    smx_MutexGet(in_clib, INF);
    print_msg("This is lo_task.");
    smx_MutexRel(in_clib);
}

void  hi_task_main(u32 par)
{
    smx_TaskSuspend(smx_ct, 2);
    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, 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, and they can cause stack overflows that go undetected because the stack pointer jumps a large amount, possibly past the pad. 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 and discussed using mutex services correctly.

Chapter 10 Event Groups

BOOLEAN	smx_EventGroupClear(EGCB_PTR eg, u16 init_mask)
EGCB_PTR	smx_EventGroupCreate(u32 init_mask, const char* name, EGCB_PTR* eghp)
BOOLEAN	smx_EventGroupDelete(EGCB_PTR* eghp)
u32	smx_EventGroupPeek(EGCB_PTR eg, SMX_PK_PAR par)
BOOLEAN	smx_EventGroupSet(EGCB_PTR eg, SMX_ST_PAR par, u32 v1, u32 v2);
BOOLEAN	smx_EventFlagsPulse(EGCB_PTR eg, u16 pulse_mask)
BOOLEAN	smx_EventFlagsSet(EGCB_PTR eg, u16 set_mask, u16 pre_clear_mask)
u32	smx_EventFlagsTest(EGCB_PTR eg, u32 test_mask, u16 post_clear_mask, u32 timeout)
void	smx_EventFlagsTestStop(EGCB_PTR eg, u32 test_mask, u16 post_clear_mask, u32 timeout)

introduction

Event groups 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 mutexes) do not permit a task to wait on more than one event at a time. However, doing so is necessary in many situations. For example, it might be desirable to qualify waiting on event X in mode M. This is logically represented by 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. Event groups can handle this kind of logic. smx extends flag testing to AND/OR logic. For example, it might 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 (M && X) || (!M && Y).

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 cleared immediately after causing a task to resume or start. A **mask** represents flags to set, clear, or initialize. Flags are represented by bits in 16-bit words.

Bits 16 and 17 in the event group *test mask*. specify the logic to be applied to the flag bits stored in the event group when testing it:

```
#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 flags. With AND logic, the testing will only pass when **all** of the bits specified in the test mask are set in the event group flags. AND/OR logic triggers a more complicated testing mechanism that is explained below in the AND/OR Testing section.

naming event flags

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. Symbolic 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 the “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.

Symbolic 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 EGCB 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 EGCB pool, and its handle, eg, is cleared so it cannot be used again.

testing flags

Testing flags is performed with smx_FlagsTest(eg, test_mask, post_clear_mask, tmo) or smx_FlagsTestStop(eg, test_mask, post_clear_mask, tmo). For example:

```
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 and return M+F1. M is not cleared because it is the current mode of the system and is controlled by other code. The if() statement tests for a non-zero return and finding it, processes whatever F1 represents, such as an event.

If the test condition is not met, smx_ct is suspended on the eg wait queue. Bit 16 of *test_mask* is stored in the *ef_and* flag of the task's TCB and bit 17 is stored in the *ef_andor* flag. The *in_eq* flag is also set and *test_mask* and *post_clear_mask* are stored in task->sv, since each is 16 bits. These are used during subsequent smx_EventFlagsSet() operations until one resumes the task. If both M and F1 become set before the tmo timeout expires, the task will be resumed and M+F1 will be returned. Otherwise, smx_EventFlagsTest() fails and returns 0.

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 must occur before the next Test() or an error will occur. Manual reset is useful if the current task is to ignore additional F3 events until it finishes its processing. For this to be successful smx_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 smx_ct to wait until F3 is set again. It operates, as follows: smx_ct is locked, so it cannot be preempted. It processes F3, then clears the F3 flag in eg. Since smx_ct is locked, F3 cannot be set by another task. When smx_ct again calls smx_EventFlagsTest(), the task lock is automatically cleared, but smx_EventFlagsTest(), itself, cannot be preempted until it has suspended smx_ct on eg. Then, another task can set F3, resulting in the above operation being repeated.

Rather than two tasks testing flags in one event group, it might seem better for each task to have its own event group. An extra EGCB costs only 24 bytes. Of course, smx_EventFlagsSet() would need to be called twice, costing extra time. However, the main downside is that setting two flags at different times is less atomic than setting one flag at one time, introducing a subtle error. Of course, the two flag sets could be done with the current task locked. However, that interferes with higher priority tasks running. Either approach can be taken.

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

setting flags

Flags are set 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 permits mutually exclusive flags, for example:

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

ensures that only F1 will be set, even if F2 was already set.

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

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

smx_EventGroupClear() clears all flags, but it also resumes all waiting tasks. The above does not.

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 waiting task’s test_mask 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 starts running 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 M && A and the post_clear_mask is A, then the 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*.

If there are multiple tasks waiting, the above procedure is repeated for each. Obviously this takes time and therefore, long task wait queues are not recommended. When all tasks have been processed, their flag clear masks are ORed, 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, if 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 permitted.

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 */
```

It could be that t2a is not clearing F3, so that t2b can see that it is set. However, since t2a and t2b have the same priority, either could run first, so t2a could miss F3 set. 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 flags.

inverse flags

Sometimes one really wants to use the inverse of a flag, such as ~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 M && X and the second test passes for !M && Y. 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, or:

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

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

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

ensures starting correctly in !M.

AND/OR testing

There are times when it is necessary to test AND/OR combinations of flags. One case is where a system operates in different modes and should test different flags in different modes, for example:

```
#define M       0x10
#define E       0x8
#define nM      0x2
#define F       0x1
#define ME      (M+E)
#define nMF     (nM+F)

void eeg4(void)    /* priority 1 */
{
        eg = smx_EventGroupCreate(nM, "eg");
        t2a = smx_TaskCreate(eeg4_t2a_main, P2, ESS, SMX_FL_NONE, "t2a");
        smx_TaskStart(t2a);

        smx_EventFlagsSet(eg, E+F, 0);  /* set flags E and F, resume t2a for ME match */

        smx_EventFlagsSet(eg, M, nM); /* change to mode M, resume t2a for nMF match */

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

void eeg4_t2a_main(u32 par)    /* priority 2 */
{
        u32 flags;

        while (flags = smx_EventFlagsTest(eg, ANDOR+ME+nMF, E+F, 10))
         {
                switch (flags)
                {
                        case nMF:
                                processF();
                                break;
                        case ME:
                                processE();
                 }
        }
}
```

In this example, eeg4() creates eg with nM set, creates t2a, and starts it. t2a tests for ME + nMF — i.e. if in mode M it tests for event E and if not in mode M it tests for event F. Since neither flag is set, t2a waits. eeg4() runs again and, to simplify this example, sets flags E and F together. t2a resumes for nMF and calls processF(). Then it waits again (F was reset by the test, but E was not). Now eeg4() changes to mode M and t2a resumes for ME and calls processE().

In the smx_EventFlagsTest(), ANDOR (bit 17) is the and/or flag. The AND/OR test requires that flags in each AND term be adjacent and that there be a 0 flag between AND terms (which represents the OR). As shown above, the ME term is bits 4 and 3, bit 2 is 0, and the nMF term is bits 1 and 0. For good performance, the terms should be as close to the least significant end, as possible.

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 its 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. However, the form

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

is not supported.

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_EventGroupSet(eg, par, v1, v2) is used to load a pointer to the event flags set callback function, cbfun, in the event group control block. If cbfun != NULL, cbfun is called every time eg flags are set. This can be used for monitoring system operation, multiple waits, etc.

smx_EventFlagsPulse(eg, pulse_mask) is like smx_EventFlagsSet() except that it does not leave the flags in pulse_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. This might be useful to resume a task if it is idle, but otherwise leave it alone. 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_EventFlagsTest(), except that it stops and restarts the current task, rather than suspending and resuming it. 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 Chapter 15 One-Shot Tasks 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.

advice on practical usage

smx event groups have been designed to meet a wide variety of requirements in 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 overly complicated quickly. This is not the way that smx event groups are intended to be used. Excess complexity is not desirable for reliability of the system nor 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 simpler. 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 condition is met. Or it may be desirable to pick a task to run based upon what flag was set due to what event occurred.

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.

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. Then, t1a will resume and cause t3a to resume. Since t3a has higher priority, you would want it to run ahead of t2a when flag F1 is set. To make that happen, lock the Set() operations:

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

This makes them atomic. Now t3a will run ahead of t2a, as expected.

pros and cons of event groups

Event groups are useful for handling multiple events and modes of operation. 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 multiple event semaphores may be a better choice, because they do not lose events. An exception to this is if lost events are not important.

state machine example

A state machine is a good example for an event group and for using one-shot tasks. Only one state should be true, at a time, in a state machine. One-shot tasks that share a single stack can represent states, since only one can run at a time.

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 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 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.

```
static void eeg8_t2a(u32 par);
static void eeg8_t2b(u32 par);
static void eeg8_t2c(u32 par);
static void eeg8_lsr_main(u32 par);
static u32 actr;
static u32 bctr;
static u32 cctr;
static TMRCB_PTR tmr;

void eeg8(void)       /* priority P1 */
{
        actr = bctr = cctr = 0;

        /* create objects */
        eg = smx_EventGroupCreate(0, "eg");
        t2a = smx_TaskCreate(eeg8_t2a, P2, 0, SMX_FL_NONE, "t2a");
        t2b = smx_TaskCreate(eeg8_t2b, P2, 0, SMX_FL_NONE, "t2b");
        t2c = smx_TaskCreate(eeg8_t2c, P2, 0, SMX_FL_NONE, "t2c");
        eeg8_lsr = smx_LSRCreate(eeg8_lsr_main, NULL, 100, 0, “eeg8_lsr”);

        /* get tasks waiting and start the timer */
        smx_TaskStart(t2a, 0);
        smx_TaskStart(t2b, 0);
        smx_TaskStart(t2c, 0);
        smx_TimerStart(&tmr, 100, 0, eeg8_lsr, 0, "tmr");

        /* 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);
}

void eeg8_t2a(u32 par)   /* state A */
{
        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);
}
```

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

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

/* stop the machine */
void eeg8_lsr_main(u32 par)
{
        smx_TaskStop(t2a, INF);
}
```

In the above, t2a, t2b, and t2c preempt eeg8() and start immediately. Each is started with par = 0, which causes it to skip the if statement and wait on eg for its flag. Then setting flag F1 in eeg8() starts t2a and the state machine begins running. 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 eeg8_lsr, which stops t2a, and thus stops the machine.

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 eeg8_lsr_main(), 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 check which events occurred and perform the appropriate operation. 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.

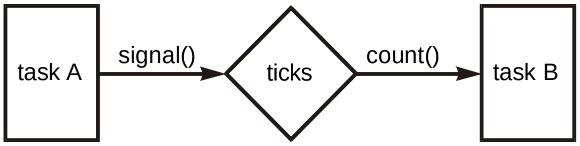
Multiple tasks may test and wait upon different combinations of flags at the same event group. Tasks wait in FIFO order and all tasks whose test masks match the event group flags are resumed together.

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 , EQCB_PTR* eqhp)
BOOLEAN     smx_EventQueueDelete(EQCB_PTR* eqhp)
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 by LSRs:



An example of using of an event queue is to count cans passing a photocell on a conveyer belt. Each passing can would trigger a count. TaskQC might wait for every 100th object at the event queue, then shuttle it off to the quality test station. TaskBox might switch to a different boxing machine after every 12th can. In this case, two tasks wait at the same event queue with different counts.

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 event semaphore allows one task to keep precise track of events. The event queue allows multiple tasks to keep track of events. There are many 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 loses 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, one per type of 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(eq);
```

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 over the monitoring of an event.

counting and signaling

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

```
ECB_PTR  events;

void  atask_main(u32 par)
{
     while (smx_EventQueueCount (events, 10, 100))
     {
          ProcessEvents();
     }
}
```

To work reliably, ProcessEvents() should require less time than the minimum time between events, else an event could be lost while ProcessEvents() is running. Another problem is that atask might be preempted while ProcessEvents() is running, thus causing an event to be lost. This could be overcome by either locking atask while ProcessEvents() is running or promoting it to top priority, as follows:

```
void  atask_main(u32 par)
{
     u32 pri = smx_TaskPeek(atask, SMX_PK_PRI);

     while (smx_EventQueueCount(events, 10, 100))
     {
          smx_TaskBump(atask, PRI_MAX);
          ProcessEvents();
          smx_TaskBump(atask, pri);
```

```
        }
}
```

This is a little cumbersome but may be necessary for system reliability.

Event queue count stop is used as follows:

```
ECB_PTR  events;

void  atask_main(u32 par)
{
        if (par)
        {
                ProcessEvents();
        }
        smx_EventQueueCountStop(events, 10, 100);
}
```

atask is a one-shot task. It is started with par == 0, so that it just waits at events the first time. Thereafter par == TRUE when atask is restarted.

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

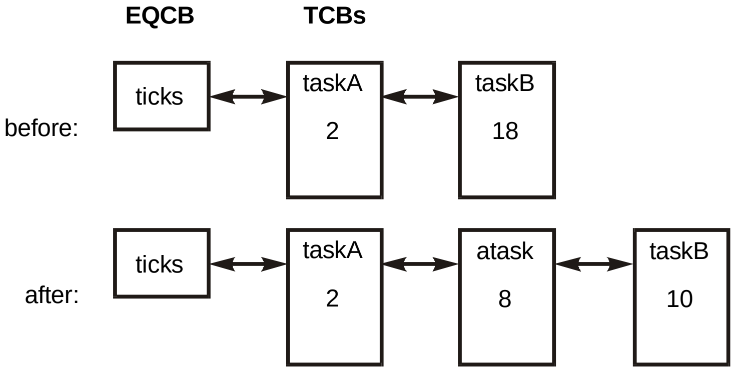
```
smx_EventQueueSignal(events);
```

This increments sv of the first task – see next section.

enqueueing a task

```
smx_EventQueueCount(events, 10, 100);
```

enqueues the current task in 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:



(numbers are counts)

Each subsequent smx_EventQueueSignal() will decrement the count in taskA->sv. When it reaches 0, that task will be resumed and atask will become the first task.

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 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 are put into rq in FIFO order, at their respective priority levels.

accurate timing

Timeouts are accurate to a tick. Thus, to delay the current task, for 10 ticks:

```
smx_TaskSuspend(smx_ct, 10);
```

For convenience, a macro allows specifying delays in milliseconds:

```
smx_TaskSuspend(smx_ct, smx_ConvMsecToTicks(10)+1);  or

smx_DelayMsec(10));
```

converts 10 msec to ticks, rounded up to the nearest tick.

Tick rates of 1000 Hz are practical for most embedded processors. Hence, accuracies of 1 millisecond are possible. (Note that accuracy applies only to putting the task into the ready queue. The amount of time until it runs depends on what is ahead of it.

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 will miss signals when not in the queue.

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

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

Chapter 12 Pipes

introduction

Pipes are composed of *cells*, which hold *packets*. The cell size equals the pipe width, which can be from 1 byte to 255 bytes. The pipe length is the number of cells that it contains. Pipe width and length are specified when a pipe is created and cannot be changed during 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 usually put into the input end and gets get from the output end. Puts can also put into the output end, which is called a *put-to-front*.

As packets are put into a pipe, the pipe's internal write pointer is advanced until it is equal to its internal read pointer. At that point the pipe is full and pipe->flags.full is set; a further put will result in failure or in the task waiting on the pipe, depending upon the put service and the timeout value. 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 the task waiting on the pipe, depending upon the get service and the timeout value.

smx Pipes serve two purposes:

(1) I/O

(2) Intertask communication

In both cases, the pipe structure is identical. The function of the pipe is determined solely by the services used for it. These are discussed next.

pipe I/O

```
BOOLEAN     smx_PipeGet8(PICB_PTR pipe, u8* bp)
u32         smx_PipeGet8M(PICB_PTR pipe, u8* bp, u32 lim)
BOOLEAN     smx_PipeGetPkt(PICB_PTR pipe, void* pdst);
BOOLEAN     smx_PipePut8(PICB_PTR pipe, u8 b)
u32         smx_PipePut8M(PICB_PTR pipe, u8* bp, u32 lim)
BOOLEAN     smx_PipePutPkt(PICB_PTR pipe, u8* psrc)
```

Pipe I/O is intended for low-speed, asynchronous, serial i/o such as input from keypads or output to character printers. The above are not SSRs, but rather ordinary functions. Generally, one end of the pipe is served by an ISR and the other end by a task. Since ISRs cannot call SSRs, smx has the above special put and get functions for use from ISRs. They can also be used from LSRs and tasks, if desired. These functions are interrupt-safe, but they are not preemption-safe, if used from tasks. However, they can be used safely from LSRs, which are non-preemptible. Also they do minimal parameter testing. A put function fails if the pipe is full and a get function fails if the pipe is empty. In these cases data will be lost unless the caller takes remedial action.

Three types of get and put functions are provided: The Get8() and Put8() functions are byte-by-byte transfer functions intended to deal with byte streams. The Get8M() and Put8M() functions transfer lim bytes, byte-by-byte from or to buffers. They are intended to speed up byte operations or can be used for wider pipes. The GetPkt() and PutPkt() functions, do actual packet transfers: word by word if packet and pointer are word-aligned, half-word by half-word if packet and pointer are half-word aligned, and byte by byte otherwise.

These functions cannot wake up a task waiting at the other end of the pipe. There are three ways to deal with that:

(1) When a task operation is required, the ISR invokes an LSR to call smx_PipeResume(pipe).

(2) A software timer regularly invokes an LSR that calls smx_PipeResume().

(3) A task regularly checks the pipe each time it times out.

In the first two cases, if a task is waiting and the put or get operation can be completed, it is completed and the task is resumed[2] with TRUE. Otherwise, nothing happens. In the third case, when the task times out, it attempts to do the put or get operation and returns with TRUE or FALSE.

The following example illustrates pipe I/O using pipe functions and SSRs:

```
u32  cc = 0;              /* character counter */
u16 fill = 0;             /* fill characters */
PICB_PTR pkt_pipe;        /* pipe */
TCB_PTR  pkt_task;        /* pkt processing task */
u8    pkt_buf[18];        /* received packet buffer */
void pkt_task_main(u32 par);

void pkt_Init(void)
{
    void* pbp;

    pkt_task = smx_TaskCreate(pkt_task_main, PRI_NORM, 200, 0, "pkt_task");
    pbp = smx_HeapMalloc(20*4);
    pkt_pipe = smx_PipeCreate(pbp, 20, 4, "pkt_pipe");   /* pipe width = 20, pipe length = 4 */
    smx_TaskStart(pkt_task);
}

void pkt_ISR(void)
{
    u8 ch = UartHandle->RDR;
    cc++;
    smx_PipePut8(pkt_pipe, ch);
    if (cc == 18)
    {
        smx_PipePut8M(pkt_pipe, &fill, 2);
        smx_LSR_INVOKE(pkt_LSR, 0);
        cc = 0;
    }
}
```

---

[2] Note: if a task has been stopped on a pipe, resuming it has the same effect as starting it. Hence, here and in other discussions, the term *resume* should be interpreted as *restart*, for a stopped task.

```
void pkt_LSR_main(u32)
{
    smx_PipeResume(pkt_pipe);
}

void pkt_task_main(u32 par)
{
    while (smx_PipeGetPktWait(pkt_pipe, &pkt_buf, 100))
    {
        ProcessPkt(pkt_buf);
    }
    /* report pkt not received in 100 ticks */
}
```

In the above example, pkt_Init() creates pkt_task of normal priority and pkt_pipe of width 20 bytes and length 4 cells. The pipe buffer is allocated from the main heap. As a consequence, it is at least 4-byte (word) aligned. Note that the packet size (see pkt_buf) is 18 bytes. However, the pipe width has been set at 20 bytes so that all packet transfers will be done word-by-word to achieve best performance.

Then pkt_Init() starts pkt_task, which calls smx_PipeGetPktWait(). pkt_task suspends on pkt_pipe since it is empty. If pkt_task times out, it reports no packets received in 100 ticks.

When a character in interrupt occurs, pkt_ISR() runs. It gets the character, puts it into pkt_pipe, and increments cc. This process continues every interrupt until cc reaches 18 meaning that a full packet has been received. Then smx_PipePut8M() is called to fill the rest of the cell with two 0 bytes, pkt_LSR is invoked, and cc is cleared so that pkt_ISR() is ready to receive the next character.

pkt_LSR_main() calls smx_PipeResume(pkt_pipe). This wakes up pkt_task and causes it to get the packet that has been put into the pipe by pkt_ISR and put it into pkt_buf. pkt_task then calls ProcessPkt() to process the packed. When this is complete, it goes back to waiting for the next packet.

This example shows how packets can be loaded by an ISR byte-by-byte into a packet pipe, then removed work-by-word by a task. Hence, a buffer between the input port and the pipe to convert the byte stream to packets is not needed. If pkt_task is not waiting on pkt_pipe when a full packet has been received, pkt_ISR can continue loading up to 3 more packets into pkt_pipe. When pkt_task does call smx_PipeGetPktWait(), it will get all full packets in the pipe, but not the partial packet currently being loaded, if any.

The pipe buffer and pkt_buf above are word-aligned to get best performance. The pipe width must also be a multiple of 4, 20 in this case, even though the packet size is 18. If pipe width were 18, then packet transfers would alternate between word and half-word aligned, so there would be a performance hit. However, in a particular case this may be ok and it saves a fill operation as well as reducing memory needed for the pipe.

intertask communication

BOOLEAN	smx_PipeGetPktWait(PICB_PTR pipe, void* pdst, u32 tmo)
void	smx_PipeGetPktWaitStop(PICB_PTR pipe, void* pdst, u32 tmo=0)
BOOLEAN	smx_PipePutPktWait(PICB_PTR pipe, void* psrc, u32 tmo, u32 mode)
void	smx_PipePutPktWaitStop(PICB_PTR pipe, void* psrc, u32 tmo, u32 mode)

The second main use for pipes is intertask communication. In this case pipes do the same function as what are called *message queues* in other RTOSs. In smx we use the term *packets* instead of *messages*, because the term messages is reserved for *exchange messages* (see Chapter 13 Exchange Messaging). Also, we use the term *pipes* instead of *queues* because the latter term is reserved for conventional queues, such as task wait queues.

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 a task at the other end of the pipe. Hence, pipes provide synchronization between tasks as well as communication. Pipes also permit multiple tasks to wait at either end. Hence they can provide *data gathering* or *data distribution* functions.

Two put services are provided: smx_PipePutPktWait() and smx_PipePutPktWaitStop(). These are SSRs. smx_PipePutPktWait() puts a packet from the buffer pointed @ psrc of size less than or equal to the pipe width. If the pipe is not full, the packet is put at the input end or the back of a pipe, if mode == 0 or at the output end or front of the pipe if mode == 1. This allows the new packet to bypass all packets already in the pipe. If successful, TRUE is returned. If the pipe is full and if the timeout, tmo, is not zero, the task is suspended on the pipe wait queue. If tmo is zero or if the task times out, the SSR returns with FALSE.

smx_PipePutPktWaitStop() is like smx_PipePutPktWait(), except that the task is stopped instead of suspended, and it is restarted instead of resumed. When the task is restarted, TRUE or FALSE is passed in as the task main function parameter. If the pipe is full and if the timeout, tmo, is not zero, the task is stopped put into the pipe wait queue. If tmo is zero or if the task times out, the task is restarted with par = FALSE.

Two get services are provided: smx_PipeGetPktWait() and smx_PipeGetPktWaitStop(). These are SSRs. The first gets a packet of size equal to the pipe width and puts it into the buffer at pdst. The packet is taken from the front of the pipe if the pipe is not empty. If the pipe is empty and if the timeout, tmo, is not zero, the task is suspended on the pipe wait queue. If tmo is zero or if the task times out, the SSR returns with FALSE.

smx_PipeGetPktWaitStop() is like smx_PipeGetPktWait(), except that the task is stopped instead of suspended and it is restarted instead of resumed. When the task is restarted, TRUE or FALSE is passed in as the task main function parameter. If the pipe is empty and tmo == 0, the task is stopped, then restarted with par = FALSE.

If a task is waiting to put a packet into a full pipe and another task does a get and the put mode == 0, the second task gets the packet at the front of the pipe, then the packet of the waiting task is put at the back of the pipe. Both tasks resume with TRUE. If the put mode == 1, the second task gets the packet of the waiting task and both tasks resume with TRUE. If a task is waiting to get a packet on an empty pipe and another task does a put, its packet goes directly to the waiting task, and both tasks resume with TRUE.

The following example illustrates intertask communication using pipe SSRs:

```
PICB_PTR xfr_pipe;          /* transfer pipe */
TCB_PTR  pro_task           /* pkt producing task */
TCB_PTR  con_task;          /* pkt consuming task */
u8          pro_buf[20];    /* producer buffer */
u8          con_buf[20];    /* consumer buffer */

void pro_task_main(u32 par);
void con_task_main(u32 par);

void pkt_xfer_init(void)
{
    void* pbp;

    pro_task = smx_TaskCreate(pro_task_main, PRI_NORM, 200, 0, "pro_task");
    con_task = smx_TaskCreate(con_task_main, PRI_NORM, 200, 0, "con_task");
    pbp = smx_HeapMalloc(20*4);
    xfr_pipe = smx_PipeCreate(pbp, 20, 4, "xfr_pipe");  /* pipe width = 20, pipe length = 4 */

    smx_TaskStart(pro_task);
    smx_TaskStart(con_task);
}

void pro_task_main(u32 par)   /* producer task */
{
    for (u8 j = 0; j < 4; j++)  /* fill xfr_pipe with data pkts 0 to 3 */
    {
        for (u8 i = 0; i < 20; i++)
            pro_buf[i] = j+i;
        smx_PipePutPktWait(xfr_pipe, &pro_buf, 0, 0);
    }

    for (u8 i = 0; i < 20; i++)  /* put pkt 4 to front */
        pro_buf[i] = 4+i;
    smx_PipePutPktWait(xfr_pipe, &pro_buf, 100, 1);
}

void con_task_main(u32 par)  /* consumer task */
{
    static u32 j = 0;

    /* get all pkts from xfr_pipe and from pro_task, then wait for more */
    while (smx_PipeGetPktWait(xfr_pipe, &con_buf, 100))
    {
        /* verify put-to-front pkt received first and others follow in order */
         if (((j == 0) && (con_buf[0] != 4)) || ((j > 0) && (con_buf[0] != (j-1))))
            /* report error */
        j++;
    }
}
```

In the above example pkt_xfer_init() creates the pro_task and con_task of normal priority and xfr_pipe of width 20 bytes and length 4 cells. The pipe buffer is allocated from the main heap and is at least 4-byte aligned. con_buf and pro_buf are also 4-byte aligned and packet width is a multiple of 4 bytes. These are done to insure maximum performance by transferring packets word-by-word.

pro_task is started and creates 4 packets, consisting of sequential numbers and puts them into xfr_pipe. It creates a 5th packet and attempts to put it into the front of xfr_pipe, but xfr_pipe is full so pro_task suspends on xfr_pipe. Thus con_task runs.

con_task gets packet 4 from pro_task, then gets packets 0 – 3 from xfr_pipe. It verifies that each packet is correct by checking the first byte, then it attempts to get a sixth packet and suspends on xfr_pipe for 100 ticks. This allows pro_task to resume and autostop, then con_task times out and autostops.

LSR to task communication

smx_PipeGetPktWait() and smx_PipePutPktWait() can also be called from LSRs with no wait timeouts since LSRs cannot wait. Stop SSRs cannot be called from LSRs since LSRs cannot stop. Nonetheless, pipes can be used to send and receive packets from LSRs.

An example of using an LSR, in place of a task, is its use to process a high-speed data stream 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.

other pipe services

```
BOOLEAN     smx_PipeClear(PICB_PTR pipe)
PICB_PTR    smx_PipeCreate(void* ppb, u8 width, u16 length, const char* name , PICB_PTR* php)
void*       smx_PipeDelete(PICB_PTR* php)
u32         smx_PipePeek (PICB_PTR pipe, SMX_PK_PAR par)
BOOLEAN     smx_PipeResume(PICB_PTR pipe)
BOOLEAN     smx_PipeSet(PICB_PTR pipe, SMX_ST_PAR par, u32 v1, u32 v2)
```

smx_PipeCreate() and smx_PipeDelete() are used to create and delete pipes. smx_PipeClear() clears all packets from a pipe and resumes all tasks waiting on the pipe. smx_PipePeek() allows getting pipe information.

smx_PipeResume() resumes the first task waiting on a pipe if its put or get operation can be completed and returns TRUE, else it leaves the task waiting on the pipe and returns FALSE. Both smx_TaskResume() and smx_TaskStart() restart a task waiting on a pipe without attempting to complete the put or get operation.

PipeSet() loads a callback function, cbfun, into the pipe control block. This function is called whenever a pipe put wait occurs. PipeStatus() returns the number of packets in the pipe and other pipe status information. See the pipe section of the Reference Manual for details on these functions.

multiple waiting tasks

Multiple tasks can wait on a pipe in priority order. Normally, a pipe is 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.

packet size

Pipes provide a means to pass short messages efficiently. However, such messages must be copied into and out of a pipe. At a certain size, it becomes more efficient to put packets into blocks and put pointers to the blocks in the pipe. This avoids the copying operations. However, 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 block pointers.

When used in this manner, there is no message control block. Thus 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. See Chapter 13 Exchange Messaging for more information.

pipe buffers

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.

safe operation notes

(1) Pipe functions are not protected from preemption and thus should not be used in tasks unless preemption is blocked.

(2) Pipe put and get functions could be used on a pipe between two tasks of equal priority in order to achieve faster operation, but neither task could wait on the pipe. Thus synchronization would need to be provided by some other mechanism, such as a semaphore or mutex.

(3) Same pipe gets from one ISR are safe from puts by another ISR and vice versa. But same pipe gets from one ISR are not safe from gets by another ISR, nor are puts.

(4) 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.

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

(6) SSRs may not be used in ISRs.

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 most RTOSs. The term *queue* or *message queue* is commonly used by them, instead of *pipe*. We think pipe is a more accurate description for this kind of object. As discussed above, pipes permit data to be exchanged between tasks, LSRs, and ISRs. 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 RTOS to smx. See Chapter 12 Pipes for more information on this kind of messaging.

Some RTOSs 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, or neither depending upon which is waiting for which or if nothing is waiting. It was named for 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 from which exchange it receives and when.

(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 Chapter 24 Resource Management.)

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

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

(8) Broadcasting, multicasting, and *distributed message assembly* are supported.

(9) smx messages are the basis for protected messages (pmsgs) in SecureSMX.

exchange API

```
BOOLEAN      smx_MsgXchgClear(XCB_PTR xchg)
XCB_PTR      smx_MsgXchgCreate(SMX_XCHG_MODE mode, const char* name, XCB_PTR* xhp)
BOOLEAN      smx_MsgXchgDelete(XCB_PTR* xhp)
u32          smx_MsgXchgPeek(XCB_PTR xchg, SMX_PK_PAR par)
BOOLEAN      smx_MsgXchgSet(XCB_PTR xchg, SMX_ST_PAR par, u32 v1, u32 v2)
```

Exchanges are capable of operating in one of three *exchange 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. Any high priority message sent to the exchange will be put at the start of the queue and will be processed first. This allows sending an urgent message, which might result in aborting an operation and discarding any waiting messages. A normal exchange is created as follows:

```
XCB_PTR  nx1;
nx1 = smx_MsgXchgCreate (SMX_XCHG_NORM, “nx1”);
```

nx1 should be a global variable so this exchange can be accessed by other tasks. An exchange is headed by an exchange control block (XCB), which is allocated from the XCB pool. The size of this pool is controlled by SMX_CFG_NUM_XCBS in acfg.h; it statically allocated, at startup, ahead of C++ global object constructors.

Tasks are always enqueued in priority order. The highest priority task enqueued is the first to receive the next message. Multiple tasks servicing a single exchange is likely to be a case where the messages are used as tokens to control when the tasks run, such as when resources are being shared. For example, there might be multiple printers and each token message would contain a printer port number and other information needed to use that printer.

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 (SMX_XCHG_PASS, “px1”)
```

The messages waiting at px1 are enqueued in priority order. The message priority has significant impact upon when the server runs. If it is higher than the client, the server will preempt and run immediately. This is most like a direct function call. If the message priority is equal to the client priority, the server will run when the client is suspended or stopped. Hence the client is allowed to finish more important work, then the server runs. If the message priority is less than the client priority, the server will not run until some time in the future. This would be appropriate for logging the client operation, for example.

Pass exchanges are the basis for partition portals in SecureSMX.

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 thus to the message block pointer in the MCB, 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.

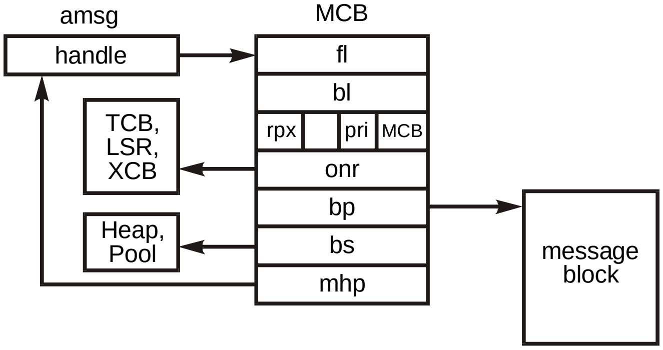
message API

```
BOOLEAN    smx_MsgBump(MCB_PTR msg, u8 pri)
MCB_PTR    smx_MsgGet(PCB_PTR pool, u8** mbpp, u16 clrsz, MCB_PTR* mhp)
MCB_PTR    smx_MsgMake(u8* bp, PCB_PTR pool, u8 pri, MCB_PTR* mhp)
u32        smx_MsgPeek(MCB_PTR msg, SMX_PK_PAR par)
MCB_PTR    smx_MsgReceive(XCB_PTR xchg,  u8** mbpp, u32 timeout, mhp)
void       smx_MsgReceiveStop(XCB_PTR xchg,  u8** mbpp, u32 timeout, mhp)
BOOLEAN    smx_MsgRel(MCB_PTR msg, u16 clrsz)
u32        smx_MsgRelAll(TCB_PTR task)
BOOLEAN    smx_MsgSend(MCB_PTR msg, XCB_PTR xchg, u8 pri, void* reply)
u8*        smx_MsgUnmake(MCB_PTR msg, PCB_PTR* pool)
```

messages

An smx message consists of a data block, called its *message block* and a *message control block (MCB)*, which smx uses to handle the message. Message blocks 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 Chapter 4 Memory Management for information on these functions. The only difference is that smx messages use MCBs and smx blocks use BCBs.

The structure of an smx message is as follows:



Note that an MCB has 28 bytes[3] vs. 16 bytes for a BCB. Hence, the overhead per message is higher, which tends to favor using smx blocks for small packets and smx messages for larger messages. However, in most cases, the benefits of exchange messaging overrule the overhead consideration — even for small packets.

The onr field of the MCB contains the handle of the task, LSR, or exchange that currently owns the message. If it is non-zero the message is in use. This is the most reliable way to determine if a message is in use or free, in which case onr == NULL. Whether in use or free, an MCB is still physically in the MCB pool.

The message block pointer, bp, and the block source, bs, are also stored in the MCB. If bs == -1 the message block came from a free block, otherwise bs points to the heap or block pool from which the message block came. An MCB has forward and backward links, which allow it to be linked into an exchange queue. It also has a cbtype, priority, and a reply index. The cbtype and the handle range are checked by all message services to verify that the handle is a valid message handle. Priority and reply index are discussed below.

getting and releasing messages

smx_MsgGet(pool, mbpp, clrsz, mhp) can be called from a task or an LSR. A data block is obtained from the block pool to use as the message block, an MCB is obtained from the MCB pool, and the MCB is linked to the data block and initialized. The MCB address is returned in 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 block are cleared and the message block starting address is loaded into the message block pointer, pointed to by bp. mbp is intended to used by the application to load the message block with data. For example:

```
void t2a_main(u32 par)
{
      u8* mbp;

      if (msg = smx_MsgGet(poolA, &mbp, 4))
            LoadData(mbp);
      else
            /* report failure */
}
```

---

[3] Excluding fields for SecureSMX.

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. It might be 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(u32 par)
{
      MSGA*       mp;
      MCB_PTR  msg;

      if (msg = smx_MsgGet(poolA, (u8**)&mp, 4))
      {
            ProcessMsg(mp);
            smx_MsgRel(msg, sizeof(MSGA));
      }
      else
            /* report failure */
}
```

Then the message block can be accessed via mp->time and mp->data[i].

smx_MsgGet() aborts and returns NULL, if the pool handle is invalid, if the data block pool is empty, or the MCB pool is empty. 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, or the task can retry later.

smx_MsgRel(msg, clrsz) is used to release a message, using its handle, msg: The msg block is released to the block pool or to the heap that it came from, if it is not a standalone block. If the block came from a block pool, it is cleared clrsz bytes from byte 4 but not past the end of the block. 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. smx_MsgGet() and smx_MsgRel() are interrupt-safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.

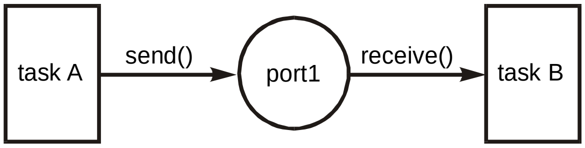
smx_MsgRelAll(task) releases all messages owned by a task and returns the number released. To do this it searches the MCB pool for an MCB whose owner is the task, then calls smx_MsgRel() to release that message. This process is repeated until all MCBs have been checked:

```
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 task-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 need to know 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(SMX_XCHG_NORM, "port1");

void  taskA_main(u32 par)
{
      u8* mbp;
      MCB_PTR  msg;

      msg = smx_MsgGet(msg_pool, &mbp, 0);
       LoadMsg(mbp);
      smx_MsgSend(msg, port1, 0, NULL);
}
```

A message is obtained from msg_pool, it is loaded, 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.

To receive a message from an exchange:

```
void  taskB_main(u32 par)
{
      u8* mbp;
      MCB_PTR  msg;

      while (msg = smx_MsgReceive (port1, &mbp, 100))
      {
            ProcessMsg(mbp);
            smx_MsgRel(msg, 0);
      }
      /* deal with timeout or error */
}
```

taskB waits at port1 until it receives a message or 100 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 block is loaded into mbp. The latter is used to process the message. Then taskB releases the message 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 100 ticks.

taskB will be resumed when a message is received or 100 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 100, taskB would wait forever for a message. If NOWAIT is specified, taskB would return immediately with or without a message.

If port1 were a pass exchange, then the priority of taskB would depend upon the priority of each message it received. This ensures 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. Usually only one task waits at a priority exchange. Such a task is usually a *server task*.

The following example shows the use of a resource semaphore to regulate getting and releasing messages between two tasks.

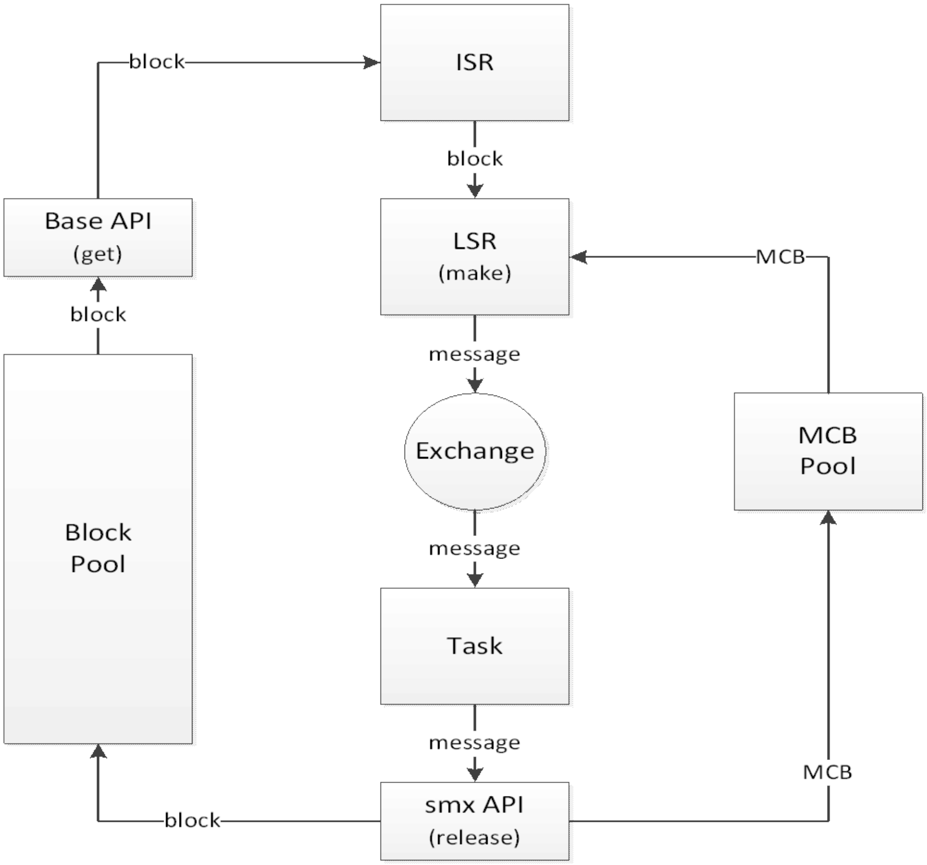
```
PCB_PTR  msg_pool;
u8          pa[NUM*SIZE]
SCB_PTR  sr;
TCB_PTR  t2a, t2b;
XCB_PTR  xa;

msg_pool = smx_BlockPoolCreate (&pa, NUM, SIZE, "msg_pool");
sr = smx_SemCreate(SMX_XCHG_RSRC, NUM, "sr");
xa = smx_MsgXchgCreate(SMX_XCHG_NORM, "xa");

void t2a_main(u32 par)
{
    u8*  mbp;
    MCB_PTR  msg;

    while (smx_SemTest(sr, INF))
    {
        msg = smx_MsgGet(msg_pool, &mbp, 4);
        FillMsg(mbp);
        smx_MsgSend(msg, xa);
    }
}

void t2b_main(u32 par)
{
    u8*  mbp;
    MCB_PTR  msg;

    while (msg = smx_MsgReceive(xa, &mbp, INF))
    {
        ProcessMsg(mbp);
        smx_MsgRel(msg, 0);
        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 is greater than 0, so the test passes, and the internal count is decremented. If no message is available, t2a is suspended and waits at sr for a message to become available. When t2a 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 exhausts 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 finished.

This illustrates the importance of task priorities. If, for example, t2b were t3b (i.e. priority increased to 3), then every time t2a sent a message, t3b would preempt, process the message, and free it — a big difference in the sequence of events due to a small difference in priorities! It is important to be aware of priorities, when designing multitasking systems, since the first sequence is probably not what was wanted.

making and unmaking messages



As shown above, smxBase provides interrupt-safe block pools for use from ISRs, so that an ISR can obtain an input block from a block 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 might be partially processed by the first task, then sent 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 block 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 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.

Free-standing message blocks could just as well be used in the above, as block pools. However, heaps cannot be used because ISRs cannot access heaps. Normally, if there is a small range of messages sizes and a limited number of each size, required at a time, block pools work just fine. However, if there is a large range of sizes and numbers needed, it may work best for receiving ISRs to divide large messages into smaller message blocks. The messages will arrive at an exchange in order and can be combined into a large message by the receiving task. Similarly for outgoing messages.

smx_MsgMake(bp, bs, mhp) converts a *bare block* (i.e. one with no MCB nor BCB) to an smx message. For example:

```
u8*  mbp;
MCB_PTR  msg:
PCB  poolA;

mbp = sb_BlockGet(&poolA);
msg = smx_MsgMake(mbp, &poolA);
```

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

```
msg = smx_MsgGet(&poolA, &mbp);
```

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, blksz);
```

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 and output.

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

```
u8  smsg[NUM];
msg = smx_MsgMake(smsg, NULL);
```

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

```
smx_MsgRel(msg, blksz);
```

releases the MCB and clears the block, but does not attempt to release it to a pool.

Note that a free standing block could be located in ROM, in which case, it would be a read-only message, and the blksz parameter should be 0, when releasing it (i.e. do not clear). A ROM message could contain a table of information for use by a task:

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

msg = smx_MsgReceive(xchg, (u8**)&tp, 100);
```

The task could use this table to tell it how to process data messages that it is receiving from other sources. In this way, one task might control the operation of another task by sending it fixed messages telling it what to do.

smx_MsgUnmake(msg, bsp) converts an smx message block to a bare block. It does so by releasing its MCB back to the MCB pool and loading the message block source into the bsp location. Unmake() can be used as follows:

```
u8*  bpi;
PCB_PTR  poolA;
XCB_PTR  xa;
PCB_PTR  ppi;

void taskA_main(u32 par)
{
       u8*  mbp;
       MCB_PTR  msg;

       msg = smx_MsgGet(poolA, &mbp, 4);
       FillMsg(mbp);
       smx_LSRInvoke(LSRA, (u32)msg);
}

void LSRA(u32 par)
{
       MCB_PTR  msg = (MCB_PTR)par;

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

```
void outISR(void)
{
    if (*bpi != 0)
    {
        Output(*bpi++);
    }
    else
    {
        sb_BlockRel(ppi, bpi, 0);
    }
}
```

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. smx_MsgUnmake() loads the globals, ppi and bpi, for use by outISR() and releases the MCB back to its pool. The ISR uses bpi to access the message block a byte at a time. When all bytes have been sent, the ISR uses ppi to return the block to poolA.

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(pool, par)**. Although it is possible to read MCB and XCB fields directly, it is preferrable to use peeks, rather than direct field accesses, because they are task-safe, and smx object field names may change. In addition, SecureSMX() utilizes a software interrupt (SWI) API in order to run application code in user mode and smx in privileged mode. Thus, smx objects are not accessible from application code.

See smx_MsgPeek() in the smx Reference Manual for a full list of message peek parameters. This function can be used only on messages that are in use. Otherwise, 0 is returned.

See smx_MsgXchgPeek() in the smx Reference Manual for a full list of message exchange peek parameters.

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().

To timestamp all messages in use:

```
MCB_PTR  max, msg;
u32* mbp;
u32  ts = smx_etime;  /* timestamp */

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.[4]) When an owned MCB is found, its message block pointer is obtained and the timestamp into the first word of the message block.

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 some method, such as locking tasks 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 from an exchange, other than 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. LSRs can perform these operations because this is necessary for no-copy I/O. In the case of a broadcast exchange, the sender retains ownership. See broadcast messages, below.

message handle pointers

For the smx_MsgGet(), smx_MsgMake(), and smx_MsgReceive() services, if the message handle pointer parameter, mhp, is set to the address of the message handle, then the message handle will be loaded by these services and need not be loaded from their return values. The smx_MsgRel() and smx_MsgSend() services load smx_nullcb into the message handle so it can no longer be used to access the message. In addition, if the message handle is not initially NULL or smx_nullcb, an smx_MsgGet(), smx_MsgMake(), or smx_MsgReceive() service is aborted and SMXE_INV_OP is reported. This forces releasing or sending a message before getting or receiving another message for the same handle, thus avoiding message block and MCB leaks. The mhp parameter defaults to NULL, thus disabling the foregoing, if not desired.

---

[4] MCBs are cleared when returned to the MCB pool. Hence, a zero owner field indicates a free MCB.

using the reply field

The sender can tell the receiver where to reply simply by passing a fourth parameter, reply, in smx_MsgSend(). This parameter is stored in mcb.rpx, which is an 8-bit index into the XCB pool. The receiving task can peek at the reply field of the MCB, 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(SMX_XCHG_NORM, "ack_xchg");
data_xchg = smx_MsgXchgCreate(SMX_XCHG_PASS, "data_xchg");

void clientA_main(u32 par)
{
    u8*  mbp;
    MCB_PTR  msg;

    u8  pri = (u8)smx_TaskPeek(smx_ct, SMX_PK_PRI);

    if  (msg = smx_MsgGet(msg_pool, &mbp, 0))
    {
        *mbp = ACK;    /* get started */

        do
        {
            if ((BOOLEAN)*mbp == ACK)
                load_new_msg(mbp);
            else
                load_old_msg(mbp);
            smx_MsgSend(msg, data_xchg, pri, ack_xchg);
        }  while (msg = smx_MsgReceive(ack_xchg, &mbp, 100));
    }
    /* take corrective action */
}
```

```
void serverX_main(u32 par)
{
    u8*    mbp;
    MCB_PTR  msg;
    u32  reply, type;

    while (msg = smx_MsgReceive(data_xchg, &mbp, INF))
    {
        BOOLEAN pass;

        *mbp = ProcessMsg(mbp);

        reply = smx_MsgPeek(msg, SMX_PK_REPLY);
        smx_MsgSend(msg, (XCB_PTR)reply);
    }
}
```

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 at the reply exchange. The data exchange is a pass exchange, so serverX will run at the priority of the message, which is that of the client, while it processes the message. serverX then loads ACK if successful or NAK if not successful, in the first byte of the message block. It obtains the reply handle from the message, and sends the reply message to the client’s reply exchange.

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 the reply exchange. It has no information of where things are coming from or going to. 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, 100 ticks is specified and if a reply has not been received, within this time, the do while 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 and report SMXE_OUT_OF_MCBS.

broadcasting messages

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 the message handle and a pointer to its message block. Furthermore, the task queue is a FIFO queue. When a message is sent to a broadcast exchange, all waiting tasks are resumed with the message handle and message block pointer. Any subsequent task receiving from the broadcast exchange will also get the message handle and message block pointer and it will not wait at the exchange. A message is removed from the exchange with smx_MsgRel(), when it is no longer needed, so a new message can be sent to it. 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 block. This is the essence of a broadcast. Normally, slave tasks only read broadcast messages and do not modify them. However, each slave could be assigned a section of the message block that it is permitted to alter. This fosters *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 block and resend it to the broadcast exchange, effectively resulting in a new message being broadcast.

The following example illustrates a simple message broadcast:

```
SCB_PTR semG, semT;

void em10(void)
{
      u8*  mbp;
      MCB_PTR  msg;

      t2a = smx_TaskCreate(em10_t2a_main, P2, 500, SMX_FL_NONE, "t2a");
      t3a = smx_TaskCreate(em10_t3a_main, P3, 500, SMX_FL_NONE, "t3a");
      xbd = smx_MsgXchgCreate(SMX_XCHG_BCST, "xbd");
      semG = smx_SemCreate(SMX_SEM_GATE, 1, “semG”);
      semT = smx_semCreate(SMX_SEM_THRES, 2, “semT”);
      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_SemSignal(semG);
```

```
        if (smx_SemTest(semT, 100);
        {
            smx_MsgRel(msg, 0);
        }
    }

    void em10_t2a_main(u32 par)
    {
        u8*  mbp;
        MCB_PTR  msg;
        TCB_PTR  onr;

        while (smx_SemTest(semG, INF))
        {
            msg = smx_MsgReceive(xbd, &mbp, INF);
            ProcessMsg(mbp);
            smx_SemSignal(semT);
        }
    }

    void em10_t3a_main(u32 par)
    {
        u8*  mbp;
        MCB_PTR  msg;
        TCB_PTR  onr;

        while (smx_SemTest(semG, INF))
        {
            msg = smx_MsgReceive(xbd, &mbp, INF);
            ProcessMsg(mbp);
            smx_SemSignal(semT);
        }
    }
```

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 ensure that all intended tasks receive a broadcast message and that none receive it more than once. After sending the message, the master signals the gate semaphore at which all of the slaves wait, so they can process the message. It then waits at the threshold semaphore, semT, for all slaves to complete, before it releases the message and autostops. The threshold is set to the number of slave readers.

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 rather than the broadcast exchange, which normally would be the owner. Hence an error would occur.

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. This can be done 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(bp, NULL);
```

smx_MsgGet() is used to get a real message, rm; MsgFill() loads data into the message block; 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 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 has better control over which tasks receive its messages.

Receiving task n would typically look like:

```
void tn_main(u32 par)
{
    u8* mbp;
    MCB_PTR  pxmsg;

    while (pxmsg = smx_MsgReceive(xchg[n], &mbp, 100))
    {
        ProcessMsg(mbp);
        smx_MsgRel(pxmsg, 0);
    }
}
```

The above is similar to receiving a normal message, except that the message block of pxmsg is not actually released by tn. This is because pxmsg->bs == -1, indicating that the message block is a standalone block. The message block must be released by the master task:

```
smx_MsgRel(rm, 0);
```

It might be clearer in tn to use:

```
smx_MsgUnmake(pxmsg, NULL);
```

which does the same thing as smx_MsgRel().

distributed message assembly

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 block 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 tasks using proxy messaging:

```
SCB_PTR  st2;  /* threshold 2 semaphore */
TCB_PTR  t3m, t2h, t2d;
XCB_PTR  xa, xb, xout;
st2 = smx_SemCreate(THRES, 2, "st2");
t3m = smx_TaskCreate(t3m_main, P3, ESS, SMX_FL_NONE, "t3m");
t2h = smx_TaskCreate(t2h_main, P2, ESS, SMX_FL_NONE, "t2h");
t2d = smx_TaskCreate(t2d_main, P2, ESS, SMX_FL_NONE, "t2d");

void t3m_main(u32 par)
{
    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(hp, NULL);
    smx_MsgSend(pma, xa);

    /* make proxy msg b using the data section pointer and send to xb */
    pmb = smx_MsgMake(dp, NULL);
    smx_MsgSend(pmb, xb);

    /* start msg builder tasks and wait for them to finish */
    smx_TaskStart(t2h);
```

```
        smx_TaskStart(t2d);
        smx_SemTest(st2, 100);

        /* Forward the assembled message */
        smx_MsgSend(msg, xout);
    }
    void t2h_main(u32 par)
    {
        MCB_PTR  pma;
        u8*  hp;

        /* get  proxy msg a and load header */
        pma = smx_MsgReceive(xa, &hp,  10);
        LoadHeader(hp);

        /* release proxy msg and signal done */
        smx_MsgRel(pma, 0);
        smx_SemSignal(st2);
    }
    void t2d_main(u32 par)
    {
        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(st2);
    }
```

After t3a creates the real message and two proxy messages and sends the latter to tasks t2h and t2d, it waits at threshold semaphore st2 for the tasks to complete their work. It then sends the completed message to the xout exchange. The advantage of this approach is that t3a need not know what protocol is being used. t2h and t2d are protocol-specific and can be created with appropriate main functions for a particular installation.

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 block pointers. The importance of this is that it provides a means to distribute information to multiple viewers without duplicating the information, which takes time and memory. It also allows assembling information from multiple servers into one message and then passing the information 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 block.

other message services

smx_MsgReceiveStop(xchg, **bpp, tmo, mhp) 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 Chapter 15 One-Shot Tasks for discussion of how to use one-shot tasks and stop services.

smx_MsgXchgClear(xchg) resumes all waiting tasks with NULL or releases all waiting messages and clears xchg 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_MsgXchgSet(xchg, par, v1, v2) permits controlling a message exchange. Currently only par = SMX_ST_CBFUN is implemented, which loads v1 = cbfun into the xchg control block, where cbfun is the callback function called whenever a message is received by xchg. This can be used for multiwait implementation or monitoring operation.

smx_MsgBump(msg, pri) bumps msg to the position after 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, by allowing the message's priority to be increased. It might be used by a task that has timed out waiting for a reply to the message.

summary

smx offers both pipe (AKA "message 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.

A *message block* can come from a block pool or a heap, or it can be a standalone block. The MsgGet() function gets a message block from a pool and combines it 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 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 block 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 fun, u8 pri, u32 tlssz_ssz, u32 fl_hn, const char* name, u8* bp, TCB_PTR* thp)
TCB_PTR	smx_TaskCurrent(void)
BOOLEAN	smx_TaskDelete(TCB_PTR* thp)
void*	smx_TaskLocate(const TCB_PTR task)
BOOLEAN	smx_TaskLock(void)
BOOLEAN	smx_TaskLockClear(void)
u32	smx_TaskPeek(TCB_PTR task, SMX_PK_PAR par)
BOOLEAN	smx_TaskResume(TCB_PTR task)
BOOLEAN	smx_TaskSet(TCB_PTR task, SMX_ST_PAR par, u32 v1, u32 v2=0)
BOOLEAN	smx_TaskSleep(u32 stime)
void	smx_TaskSleepStop(u32 stime)
BOOLEAN	smx_TaskStart(TCB_PTR task, u32 par)
BOOLEAN	smx_TaskStartNew(TCB_PTR task, u32 par, u8 pri, FUN_PTR fun)
BOOLEAN	smx_TaskStop(TCB_PTR task, u32 timeout)
BOOLEAN	smx_TaskSuspend(TCB_PTR task, u32 timeout)
BOOLEAN	smx_TaskUnlock(void)
BOOLEAN	smx_TaskUnlockQuick(void)
BOOLEAN	smx_TaskYield(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

(See the priorities section in Chapter 1 Under the Hood for the definition of smx priority.)

A large number of priorities in a system is not usually desirable. 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.

PRI_MIN, is used by the idle task, smx_Idle. See the discussion, below. PRI_MAX is the highest priority for application tasks. PRI_SYS is the highest defined priority and is reserved for special system tasks.

changing a task's priority

To change a task's priority, use:

```
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.

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, SMX_PK_RTC);
```

TaskStartNew() can also be used to change a task's priority:

```
smx_TaskStartNew(task, 0, NEW_PRIORITY, t2a_main);
```

This would normally be done as part of changing the task's main function.

idle task

smx_Idle has priority PRI_MIN. It is the only task required by smx and is created by smx_Go(). smx_Idle runs only when no other task is ready[5]. 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 PRI_MIN so other PRI_MIN tasks can run in a round-robin manner. It is recommended that low priority 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 a callback function to automatically perform certain operations when the task starts, resumes, stops, suspends, and is deleted. For this to work the callback function address must have been loaded into task->cbfun. Set hookd to 0 if there is no task callback function.
(3) strt_lockd	Starts the task locked.
(4) umode	Task is running in umode. (See SecureSMX User's Guide for discussion.)

---

[5] The smx scheduler is capable of looping internally if there is no task to run.

The stk_chk flag is set when a task is created. It may be set or cleared by:

```
smx_TaskSet(task, SMX_ST_STK_CK);
```

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_TaskSet(task, SMX_ST_CBFUN, cbfun, 1);
```

It is cleared if cbfun == NULL. The strt_lockd flag is set by using SMX_FL_LOCK at the time the task is created:

```
t2a = smx_TaskCreate(t2a_main, P2, 200, SMX_FL_LOCK, "t2a");
```

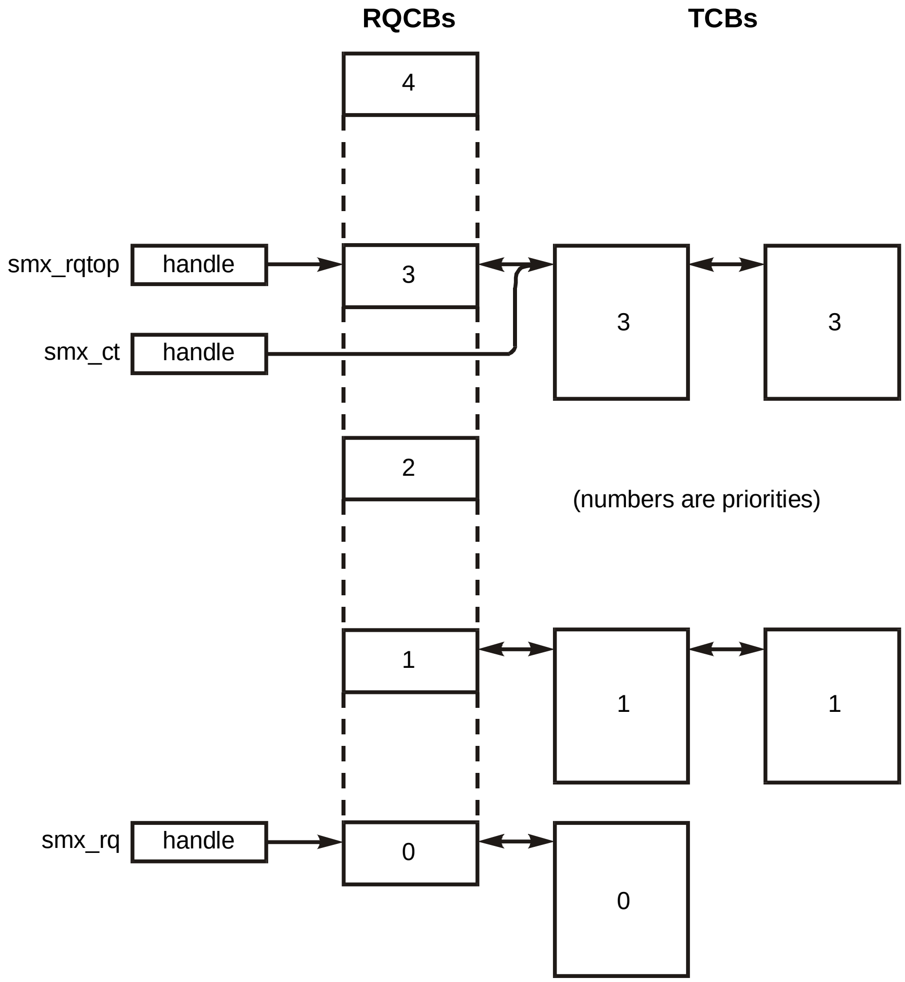
or by:

```
smx_TaskSet(task, SMX_ST_STRT_LOCKD);
```

This is similar to starting an ISR with interrupts disabled. Normally it is used to prevent preemption during the task initialization code. The flag must be set before the task is started. When the task is ready for normal operation, use smx_TaskUnlock(task) to unlock it, else other tasks will not be able to run until it suspends or stops itself.

ready queue

The *ready queue*, *smx_rq*, contains all tasks that are ready to run. It is designed to support large numbers of tasks with good performance. The ready queue is defined as a static array in xglob.c of ready queue control blocks (RQCBs). There is one rq level per priority number, including 0, so the size of the array is SMX_PRI_NUM. The levels are in increasing priority number order. Each level has an RQCB which heads the sub queue of tasks at that level's priority.



In the above diagram, smx_rq points to the lowest priority number level of the ready queue. If SMX_CFG_PRI_UP == 1, smx_rqtop is the highest occupied priority number, as shown in the above figure. If SMX_CFG_PRI_UP == 0, smx_rqtop is the lowest occupied priority number. Hence smx_rq is the same either way, only smx_rqtop differs. smx_ct points to the current task's TCB, which is the first task in the smx_rqtop level. For more discussion of priority direction, see Chapter 14 Tasks, task priority section.

Even though the current task is running, its TCB remains in the ready queue. Also, its TCB is typically the first at the level smx_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 that 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 *smx_rqtop* pointer is used. It points to the highest occupied level in rq. The top task is the first task in this level. smx_rqtop is increased or decreased as the top occupied level of rq changes.

If smx_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 smx_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 becomes the current task. If the current task is *unlocked* and another task becomes the top task, the current task is suspended and new top task runs. 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 smx_ct, unless the current smx_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 can be used to abort event waits that have taken too long.

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 until another task starts it with:

```
TCB_PTR atask;
smx_TaskStart(atask, par);
```

This puts atask into the ready queue. It does not actually run the task. That is done by the scheduler. A variant of smx_TaskStart() is:

```
smx_TaskStartNew(atask, par, priority, new_main);
```

This allows changing the priority and main function and passing in a new parameter. It is used primarily after a task has initialized itself in order to put it into its normal run mode

A task can also be directly stopped by another task:

```
smx_TaskStop(atask, timeout);
```

This removes atask from any queue it may be in and sets its timeout as follows:

```
smx_timeout[tn] = smx_etime + timeout;
```

where tn is atask's index into the smx_timeout array, and smx_etime is the current elapsed time. tn = atask->indx. Three special values of timeout are:

(1) SMX_TMO_INF: smx_timeout[tn] is made inactive and atask will wait forever.

(2) SMX_TMO_NOCHG: atask is stopped, but its timeout is not changed.

(3) 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, par) 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 tmo 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.

The current task may start or stop itself. Starting itself results in moving itself to the end of its rq level. If it is the only task in this level, it will immediately restart from the beginning of its main function. If the current task stops itself, it is removed from smx_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 delete 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 that came from a heap, it is freed to the heap. Also all owned blocks, messages, mutexes, and timers are automatically released. Finally, the TCB is released and the atask handle is set to smx_nullcb so that it cannot be mistakenly reused. smx_nullcb has all 0 fields so it is not mistaken for an active TCB.

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 to perform maintenance or updates. Task deletion is also used for partition recovery under SecureSMX

Deleted tasks cannot wait for an event and cannot be seen in smxAware. They are in the NULL or non-existent state. If this is 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 92 to 116 bytes of TCB and timer space for reuse.

Task deletion is hazardous and not recommended for frequent operation. There is a danger of memory and object leaks and other problems. Task callbacks are intended to reduce this problem. See **task callback functions** below.

locking and unlocking tasks

```
BOOLEAN    smx_TaskLock(void)
BOOLEAN    smx_TaskLockClear(void)
BOOLEAN    smx_TaskUnlock(void)
BOOLEAN    smx_TaskUnlockQuick(void)
```

smx_TaskLock() 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. Using smx_lockctr permits a task to lock itself multiple times, which can happen if nested subroutines each lock the current task -- see discussion below.

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.

smx_TaskUnlock() decrements the lock counter down to 0, each time it is called. It must be called as many times as smx_TaskLock() was called. If called too many times, FALSE is returned and SMXE_EXCESS_UNLOCKS is reported. When smx_lockctr == 1, smx_TaskLockClear() is called.

smx_TaskLockClear() clears smx_LockCtr() and checks to see if a higher priority task has become ready to run while the current task was locked. If so, the higher priority task will preempt immediately. This allows relatively long lock periods to be used. If smx_LockCtr > 1, FALSE is returned and SMXE_INSUFF_UNLOCKS is reported.

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();
```

lock breaking

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 cause 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 = SMX_TMO_NOWAIT, but not otherwise. smx_SemTest(semA, tmo) for tmo > 0 will break the lock, even if no wait actually occurs. This is because the same call cannot break the lock one time and not another. To do so would cause unpredictable behavior. 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.

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();
    /* operationA() calls funB() */          /* operationB() */
    smx_TaskUnlock();                        smx_TaskUnlock();
}                                        }
```

If locking were implemented with a simple lock flag, operationA would not be protected because the task unlock in funB() cleared the lock before operationA completed. 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 ensure 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 xcfg.h).

The current value of the lock counter is returned by smx_TaskPeek(task, SMX_PK_LOCK).

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 Chapter 15 One-Shot Tasks.)

Another important use for task locking is to avoid unnecessary task switches. In the following example, task t3a is waiting on semA. When task t2a signals semA, t3a will preempt immediately and run:

```
void t3a_main(u32 par)
{
    while (1)
    {
        smx_SemTest(semA, TMO);
        funA();
    }
}
```

```
void t2a_main(u32 par)
{
    while (1)
    {
        funB();
        smx_SemSignal(semA);
        smx_SemTest(semB, INF);
    }
}
```

When t3a finishes it will wait on semA again, and t2a will resume, only to wait on the semB. Hence a wasted task switch has occurred, which increases overhead. A more efficient way to code t2a is as follows:

```
void t2a_main(u32 par)
{
    while (1)
    {
        funB();
        smx_TaskLock();
        smx_SemSignal(semA);
        smx_SemTest(semB, INF);
    {
}
```

Now t3a cannot preempt until t2a tests semB. The lock is automatically cleared on the semB test, whether it passes or not, so it is unnecessary to call smx_Unlock().

making tasks act as expected

Code incorporating kernel services may not behave as you expect. For example, in the following code, which task runs first — higher priority t3a or t2a?

```
void  t1a_main(u32 par)
{
	while (smx_SemTest(sema, tmo))
	{
		smx_TaskStart(t2a);
		smx_TaskStart(t3a);
	}
}
```

The answer is t2a, because 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 also note that two hidden task switches (t2a -> t1a, and t3a -> t1a) occur, which wastes processor time. To fix these:

```
void  t1a_main(u32 par)
{
	while (smx_SemTest(sema, tmo))
	{
		smx_TaskLock();
		smx_TaskStart(t2a);
		smx_TaskStart(t3a);
	}
}
```

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, as expected. Another way to achieve the same result is for t1a to have priority 3.

task local storage (TLS)

When a task is created the tlssz_ssz parameter specifies the stack size (low 16 bits) and the TLS size (high 16 bits). If TLS size is > 0, and if the stack block is allocated from the heap (i.e. ssz > 0), additional space is allocated in the stack for Task Local Storage (TLS) below the Register Save Area. For information about using TLS, see the SecureSMX User's Guide.

task callback functions

Task callback functions provide powerful extensions to smx, as described in the following sections. They are useful for tasks that have extended states and in SecureSMX systems for partition recovery.

To hook a callback function to taskA, the following is used, after taskA has been created:

```
smx_TaskSet(taskA, SMX_ST_CBFUN, cbfunA, h);
```

This loads a pointer to cbfunA() into taskA->cbfun. In addition, taskA->flags.hookd is set if h > 0. The hookd flag controls whether cbfunA() is called for each of the cases: SMX_CBF_ENTER, SMX_CBF_EXIT, SMX_CBF_START, and SMX_CBF_STOP. If cbfunA != NULL, cbfunA() is called for each of the cases: SMX_CBF_INIT and SMX_CBF_DELETE regardless of the value of h.

To minimize impact upon interrupt latency, cbfun is called with interrupts enabled. Interrupts can be disabled, if necessary to prevent ISRs from running. Task callbacks are task-safe, so there is no need to be concerned about other tasks, LSRs, or SSRs running. If a task does not require a callback function, the above set function is not called, and there is no callback function overhead for that task. Callback functions can be shared between tasks.

task initialization and deletion

In the following example, case SMX_CBF_INIT occurs when a task, is being started the first time. Thus it provides an opportunity to get all of the resources that the task needs and to perform all necessary initialization needed to start the task. This is especially helpful for one-shot tasks by avoiding the need to call smx_TaskStartNew() to restart the task with its main function after its initialization function has run. It is also helpful in the case where a task runs in pmode for initialization, then changes to run in umode.

Case SMX_CBF_DELETE occurs when a task is being deleted. Thus it provides an opportunity to release all resources and to perform all necessary cleanup functions in order to cleanly delete the task. For systems using SecureSMX, this is important for partition deletion and recovery in order to recover from malware. Having the operations and counter operations present within a single function helps to make sure that nothing is missed and that a task can be shut down with no resource leaks.

The typical task callback function structure for this is:

```
void cbfun(u32 mode, u32 task)
{
    switch (mode)
    {
        case SMX_CBF_INIT:
            /* get resources and do initialization for task */
            break;
        case SMX_CBF_DELETE:
            /* release resources and do cleanup for task */
            break;
        …
    }
}
```

In the above example, task->cbfun(SMX_CBF_INIT, task) is called by smx_TaskStart() or smx_TaskStartNew() if task->cbfun != NULL. Hence cbfun() runs within an SSR. It can call other services. For example, it could create a semaphore and get a block from a heap. Since the callback runs within an SSR, it cannot be preempted, unless it waits, which should not happen.

task->cbfun(SMX_CBF_DELETE, task) is called by smx_TaskDelete() if task->cbfun != NULL. Hence cbfun() runs within an SSR. It can call other services. For example, it could delete the semaphore and release the heap block obtained by SMX_CBF_INIT. Putting this case after the SMX_CBF_INIT case in cbfun() makes it easy to see the operations to reverse and helps to ensure that there will be no object or memory leaks when task is deleted.

For another example, see the Custom SSTs section of the SecureSMX User’s Guide and see smx_TaskDelete() in the smx Reference Manual for more information.

normal task entry and exit

Some tasks have extended contexts 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. If taskA->flags.hookd is set, cbfun(SMX_CBF_EXIT, taskA) is called by the scheduler whenever taskA is suspended, and cbfun(SMX_CBF_ENTRY, taskA) is called by the scheduler whenever taskA is resumed. As shown in the example below, these can be used to transparently preserve and restore extended task states and to perform other exit and entry functions on a task-specific basis.

The typical task callback function structure is:

```
void cbfun(u32 mode, u32 task)
{
    switch (mode)
    {
        …
        case SMX_CBF_EXIT:
            /* save extended state */
            break
        case SMX_CBF_ENTER:
            /* restore extended state */
            break;
    }
}
```

The task parameter and the return value are not used in these cases.

one-shot task start and stop

Because one-shot tasks release their task stacks when they stop, there is normally no carryover of registers or other information from one run to the next. Normally, this is acceptable. However, in some cases, it may be desirable to carry over some information such as a counter or a flag. If taskA->flags.hookd is set, cbfun(SMX_CBF_STOP, 0) is called by the scheduler whenever taskA stops, due to a stop service or due to autostop, and cbfun(SMX_CBF_START, 0) is called by the scheduler whenever taskA starts, due to an awaited event, an event wait timeout, or smx_TaskStop(). As shown in the example below, these can be used to transparently preserve and restore task contexts and to perform other exit and entry functions for one-shot tasks.

The typical task callback function structure for a one-shot task is:

```
void cbfun(u32 mode, u32 task)
{
	switch (mode)
	{
		…
		case SMX_CBF_STOP:
			/* save context */
			break;
		case SMX_CBF_START:
			/* restore context if not first start */
			break;
	}
}
```

The above example is similar to the normal task example. A small difference is that there is no context to restore on the first start. Hence, it is necessary for the START case to ignore the first start. The task parameter and the return value are not used in these cases.

task callback notes

Scheduler-initiated task callbacks operate as extensions to the scheduler. Hence their execution times directly add to task suspend and resume or start and stop times. Therefore, they should be as fast as possible.

If a resume task operation is aborted after its entry case has run, the exit routine will run, even though the task, itself, did not run. Hence, entry operations will be reversed, as though the task had run. This situation can occur because most of the scheduler runs with interrupts enabled. Hence, an ISR can invoke an LSR, which, in turn, can call an SSR, resulting in a higher-priority task being made ready to run. Then an LSR flyback occurs in the scheduler, which causes the higher priority task to be dispatched, instead of the preempted task. These actions should be transparent to these cases, but might not be in all cases. For example doing something in an entry or start case that cannot be reversed must be avoided. Task start and stop are handled similarly.

If necessary, a callback function can be hooked when needed and unhooked when not needed:

```
smx_TaskSet(smx_ct, SMX_ST_CBFUN, cbfun, 1);
/* callback active */
smx_TaskSet(smx_ct, SMX_ST_CBFUN, cbfun, 0);
```

Since the latter function only resets smx_ct->flags.hookd, the callback function will still be called for the SMX_CBF_INIT or SMX_CBF_DELETE cases.

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 callback function cases into each such tasks, as discussed above. Then the coprocessor registers can be saved and restored automatically when each task is suspended and resumed. We have implemented this for Cortex-M4 and M7 FPU in our fpudemo.c. See the SMX Target Guide section ARM-M/ Architectural Notes/ Floating Point for details.

In some cases, a coprocessor may have a large number of registers thus greatly increasing 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 the coprocessor.

Another way to deal with this is to bracket just the coprocessor operations,

```
smx_TaskSet(taskA, SMX_ST_CBFUN, (u32)fpfun, 1); /* enable fpfun() */
/* coprocessor operations */
smx_TaskSet(taskA, SMX_ST_CBFUN, NULL, 0); /* disable fpfun() */
```

Then, the extra coprocessor overhead is encountered only in certain regions of taskA, which may seldom run. Another alternative is:

```
smx_TaskLock();
/* coprocessor operations */
smx_TaskUnlock();
```

to avoid saving the coprocessor registers, but this may cause unacceptable task latency causing a high-priority task to miss its deadline.

second task main function type

The second task main function supported by smx has u32 return type, 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(et7_taskA_main, P2, ESS, SMX_FL_NONE, "taskA");
smx_TaskStart(taskA, ENTER);

/* subsequent start */
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, EXIT is passed into taskA_main(), which causes it to release owned resources and prepare taskA to become dormant. Hence, the error manager, in this example, does not need to know how to shut down taskA.

Chapter 15 One-Shot Tasks

introduction

Creating a system with a few complex tasks is not making the best use of an RTOS. Instead, a system should consist of many relatively simple tasks. It is much easier to write and debug a simple task, and the system will make greater use of RTOS services rather than requiring new user-developed code.

The downside of the above is that each task requires a TCB and a stack and these should come from on-chip SRAM for best performance. The smx TCB is about 100 bytes, so a 20 task system would need 2 KB for TCBs, which is not too bad. However task stacks are usually in the 500 to 2000 byte range, which requires 10 to 40 KB. Processors are being produced with more on-chip SRAM than in the past, but this is still a pretty big hit.

One-shot tasks help to reduce RAM usage by sharing stacks between tasks. Since they 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 on-chip SRAM. This is especially true if auto variables are used extensively to achieve reentrancy.

The System Stack frees task stacks of ISR, LSR, scheduler, exception handler, 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 in the current stack. Hence, a function of moderate complexity can require significant stack space. Coupled with deep function nesting this can result in large stack requirements in the thousands of bytes.

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 these operations, but also they are hidden from debug tools such as smxAware.

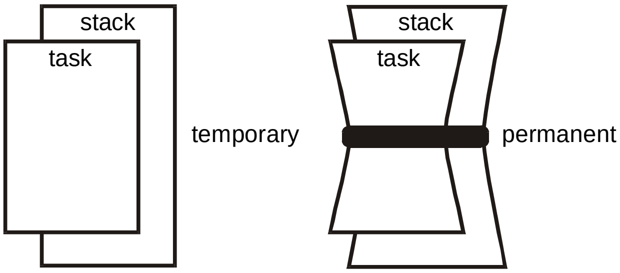
Deleting tasks when not needed is another way to reduce stack RAM usage. However, such tasks cannot wait for events. Recreating such tasks and getting them restarted, when needed, is time consuming. Also, when in the NULL state, they drop off the radar as far as the debugger and smxAware are concerned.

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. A good example of such a task is a server task which serves many clients. While waiting for the next job from a client, it has no need to store information from the previous client. Hence, it does not need a stack when stopped. 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 that normal tasks suspend to wait for. For example, a one-shot task can stop at the same exchange where a normal task has suspended, both waiting for a message. The difference is that the one-shot task releases its stack, whereas the normal task retains it 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, which is why it is called a “one-shot task”. There is little performance penalty for starting a task with a new stack, versus resuming a task that already has a stack, because the time for getting a new stack is balanced by not needing to restore registers. The one-shot task need not save registers since it restarts from the beginning of its main function.

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 before the one-shot task is done. This causes no harm, but it may necessitate a larger stack pool.

It is often the case that some one-shot tasks are mutually exclusive. Hence, only a subset of the one-shot tasks can run simultaneously. If there are enough stacks in the stack pool for the maximum 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 stalled one-shot top tasks. Eventually, a stack will become available and the task will run. If fast SRAM is at a premium and the foregoing performance hit is acceptable, this would be an acceptable way to design a system. Otherwise some restructuring may be required.

An interesting observation is: if we make a rule that one-shot tasks may not suspend themselves but must 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 fast SRAM 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 it might be sufficient for a smaller system.

Very few RTOSs permit stack sharing. OSEK is one, but it shares a single stack between tasks. As a consequence, tasks must stop running in the inverse order that they were started. That is not the case here.

examples of 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? A server task that performs a complex, seldom-needed calculation, such as an orbit correction is a good example. Having done its job and reported its results, the task has no history to remember and thus can give up its stack.

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. In such a case, the steps are mutually exclusive and can be implemented as one-shot tasks sharing a single stack. In addition, this approach provides flexibility. 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, can 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 these are high-priority tasks. They can wait at event semaphores, do their job when signaled, then quit. This kind of operation might be done by LSRs, except that LSRs cannot wait.

writing one-shot tasks

A one-shot task is created by passing a 0 stack size parameter and a NULL preallocated stack block pointer (bp) to smx_TaskCreate(). It can be stopped and give up its stack by calling any Stop SSR, such as smx_MsgReceiveStop(). smx_TaskStop() and smx_TaskStart() also stop a task, as will autostop — running through the end of the task main function.

Like a normal task, when a one-shot task is started by smx_TaskStart(task, par), par is passed into its main function. However, when a one-shot task waits on an smx object via calling an SSR and then is restarted, the return value of the SSR is passed in as par. For example if the task called smx_SemTestStop(), then the parameter would be TRUE if the semaphore test passed and FALSE otherwise. This is logical, but may be conceptually difficult, at first — it takes some getting used to.

One-shot tasks have a different code structure than normal tasks, for example:

```
void  atask_main(u32 par)
{
      /* process par */
      /* calculate new par */
      return (par);
}
```

Note that there is no infinite loop, as with a normal task. The one-shot task simply runs once and, in this case autostops.

To create a one-shot task:

```
atask = smx_TaskCreate(atask_main, PRI_NORM, 0, SMX_FL_NONE, "atask", NULL);
```

In this case, stack size is 0 and there is no preallocated stack.

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(u32 par);

void ug_main(u32 par)
{
      taskA = smx_TaskCreate(taskA_main, P2, 500, SMX_FL_NONE, "taskA");
      smx_TaskStart(taskA, 0);
}

void   taskA_main(u32 par)
{
      u8*  dp;
      MCB_PTR  msg;

      /* initialize taskA here */
```

```
        while (msg = smx_MsgReceive(xa, &dp, 100))
        {
            if (msg)
            {
                ProcessMsg(dp);
                smx_MsgRel(msg, 0);
            }
            else
            {
                /* 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(u32 par)
    {
        taskB = smx_TaskCreate(taskB_init, P2, 0, SMX_FL_NONE, "taskB", NULL);
        smx_TaskStart(taskB, 0);
    }

    void   taskB_init(void)
    {
        /* initialize taskB here */
        smx_TaskSet(taskB, SMX_ST_FUN, taskB_main);
        smx_MsgReceiveStop(xa, &dp, 100);
    }

    void  taskB_main(MCB_PTR msg)
    {
        if (msg)
        {
            ProcessMsg(dp);
            smx_MsgRel(msg, 0);
        }
        else
        {
            /* handle timeout, or error */
        }
        smx_MsgReceiveStop(xa, &dp, 100);
    }
```

Note that taskB is created with no stack and with function taskB_init(). This code runs the first time taskB runs and it initializes taskB, as is done before the while loop for taskA. When taskB initialization is complete, taskB->fun is changed to taskB_main. This changes taskB->fun, not the current function pointer. Then taskB does a receive stop on exchange xa. When taskB restarts, its parameter is the received message handle; 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 by the else 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(u32 par)
{
    taskB = smx_TaskCreate(taskB_main, PRI_NORM, 0, SMX_FL_NONE, "taskB");
    smx_TaskStart(taskB, INIT);
}

void  taskB_main(MCB_PTR msg)
{
    if  (msg)
    {
        if  (msg == (MCB_PTR)INIT)
         {
            /* initialize taskB here */
        }
        else
        {
            ProcessMsg(dp);
            smx_MsgRel(msg, 0);
        }
    }
    else
    {
        /* handle timeout or error */
    }
    smx_MsgReceiveStop(xa, &dp, TMO);
}
```

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(u32 par)
{
    t2a = smx_TaskCreate(t2a_main, P2, 0, SMX_FL_LOCK, "t2a");
    t2b = smx_TaskCreate(t2b_main, P2, 0, SMX_FL_NONE, "t2b");
    t3a = smx_TaskCreate(t3a_main, P3, 0, SMX_FL_NONE, "t3a");
    xa = smx_MsgXchgCreate(SMX_XCHG_NORM, "xa");
    smx_TaskStart(t2a, INIT);
}

void t2a_main(MCB_PTR msg)
{
    if  (msg)
    {
        if  (msg == (MCB_PTR)INIT)
         {
            /* initialize t2a here */
        }
        else
        {
            smx_TaskStart(t2b);
            smx_TaskStart(t3a);
        }
    }
    else
    {
        /* handle timeout or error */
    }
    smx_MsgReceiveStop(xa, &dp, SMX_TMO_NOWAIT);
}

void t2b_main(u32 par)
{
    /* do t2b operation */
}
void t3a_main(u32 par)
{
    /* do t3a operation */
}
```

Note that SMX_FL_LOCK is passed to smx_TaskCreate() so it sets t2a->flags.strt_lockd. Thus, t2a always starts locked and remains locked until it stops for the next message. Between smx_MsgReceiveStop() and restart, a task of higher priority can run, even though NOWAIT 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, such as t2b, 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. In this case, t2b would run. 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 non-preemptible 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, allocating one task per channel may be 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, if all have the same priority, 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.

a further example

The following example illustrates 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:

```
u8 bp;

taskA = smx_TaskCreate(taskA_main, PRI_NORM, 0, SMX_FL_NONE, "taskA");
taskB = smx_TaskCreate(taskB_main, PRI_NORM, 0, SMX_FL_NONE, "taskB");
mxchgA = smx_MsgXchgCreate(SMX_XCHG_NORM, "mxchgA");
mxchgB = smx_MsgXchgCreate(SMX_XCHG_NORM, "mxchgB");

void taskA_main(MCB_PTR msg)
{
    if (msg)
    {
        ProcessMsg(bp);
        smx_MsgReceiveStop(mxchgA, &bp, 100);
    }
    /* handle timeout or error */
}

void taskB_main (MCB_PTR msg)
{
    if (msg)
    {
        ProcessMsg(bp);
        smx_MsgReceiveStop(mxchgB, &bp, 100);
    }
    /* handle timeout or error */
}
```

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 task code shown above and a TCB.

Note that the timeout and error handling in this example is different than in the previous examples. In the latter, once the timeout or error was handled the tasks waited for the next message. In this example, the tasks autostop and must be restarted.

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 way to implement multiple waits.

For further discussion of advantages and usage of one-shot tasks see our white paper *One-Shot Tasks Reduce RAM Usage*.

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(u32 par)
{
    t2a = smx_TaskCreate(t2a, P2, 0, SMX_FL_NONE, "t2a");
    smx_TaskStart(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 */
    }
}
```

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 autostops. t1a starts t2a again; t2a runs with par == 1, from its previous run. It performs operation 1, passes par == 2 to itself, and autostops. t1a starts t2a again; t2a runs with par == 2 from its previous run. It performs operation 2 and autostops. This example illustrates how a one-shot task can pass information, such as mode control, to itself.

Chapter 16 Service Routines

ISRs are first responders, LSRs are helpers, and SSRs are workers

introduction

Service routines are task-like objects which operate at a higher priority level than tasks. Service routines 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

These are discussed in what follows.

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 with internal details of SSRs. Hence, it is necessary only to be aware of their existence.

smx_SSR_ENTERn(call_id, p1, p2, ... pn) 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 SSR completes and to prevent interruption by another SSR or LSR.

CAUTION: SSRs enable interrupts**.**

interrupt service routines (ISRs)

Interrupt service routines handle interrupts. They run in *handler mode, hmode*. 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 after the enter macro and before the exit macros, respectively — see Chapter 25 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. LSRs have LSR Control Blocks, LCBs, which are obtained from smx_lcbs pool and initialized by smx_LSRCreate(). There are three types of LSRs:

(1) Trusted LSRs

(2) Safe LSRs for umode

(3) Safe LSRs for pmode

The safe LSRs are available only with SecureSMX and are documented in the Safe LSRs chapter in the SecureSMX User's Guide. Trusted LSRs are used with smx and they use very little of the LCB. They run in hmode like ISRs. They are referred to herein as just "LSRs" and work the same as LSRs always have in smx. LSR start and end are logged in the Event Buffer by the scheduler.

rules of behavior

Service routines adhere to the following rules of behavior:

(1) ISRs may not call SSRs, but can invoke LSRs using smx_LSR_INVOKE() function, to do so.

(2) LSRs must be invoked, not called directly.

(3) LSRs 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. The 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 in a few other places. They are not disabled in system services.

(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 operate like top-priority, non-blockable mini-tasks, and 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 background objects — switching from one task to another is generally too slow for foreground servicing. smx provides a powerful, two-level structuring of foreground via ISRs and LSRs.

two ISR types

The two types of ISRs are *smx ISRs* and *non-smx ISRs*.

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.

For non-ARM-M processors, smx_ISR_ENTER() increments smx_srnest, and smx_ISR_EXIT() decrements smx_srnest. 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 putting the LSR handle into the *lsr queue, smx_lq,* followed by *par,* and in incrementing the *lsr counter, smx_lqctr*. smx_ISR_EXIT() passes control to the smx scheduler when smx_srnest == 0 and smx_lqctr > 0. The first part of the smx scheduler is the LSR scheduler, which runs the LSR and passes par to it. The LSR may call one or more SSRs, which may result in scheduler flags being set to indicate a task reschedule is or may be necessary. When all LSRs have run, control passes to the task scheduler, which determines the top task and dispatches it

For ARM-M processors, smx_srnest is not used for ISR nesting. Instead, the RETTOBASE register flag controls ISR nesting. When 0 there is nesting; when 1, there is no nesting. If smx_lqctr > 0 and RETTOBASE == 1, smx_ISR_EXIT() triggers the *PENDSV Handler*, which calls the smx scheduler. Operation then proceeds, as above.

non-smx ISRs are also supported. These do not use the above macros and therefore do not directly link to the background. They generally perform small foreground functions. Such ISRs must meet one of two criteria:

(1) Higher priority than all smx ISRs.

(2) Never enable interrupts.

If neither is met, then a non-smx ISR may be interrupted by an smx ISR. If the latter invoked an LSR, control would go to the scheduler at the end of it, and the non-smx ISR would be left dangling for an unpredictable period, because processor control would be diverted to smx_PreSched, which might run the LSR and task schedulers, which might switch tasks, rather than return immediately to the interrupted 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() to enable ISRs.

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.

Traditional ARM is an example of the first type. For this 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. ARM-M and ColdFire are examples. There are two processor variants of this type:

(1) Those that enable interrupts after the first machine instruction.

(2) Those that do not.

ColdFire 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.

ARM-M is an example of the second variant. For this kind of processor, 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.

For more information, see the SMX Target Guide for your specific processor architecture and tools.

smx ISR and SS operation

As noted above, the structure of an smx ISR depends upon the processor being used. In general, ISR code must be sandwiched between smx_ISR_ENTER() and smx_ISR_EXIT() macros.

For non-ARM-M processors, in addition to incrementing smx_srnest, smx_ISR_ENTER() saves volatile registers and switches to the system stack (SS) from a task stack. 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 (because they don't use smx_ISR_ENTER()). For this reason, they should use minimal stack, and it is best if they do not nest.

ARM-M processors save volatile registers and switch to SS (aka main stack (MS)) from a task stack, in hardware. To return to the point of interrupt, this process is reversed, in hardware. As noted above, if smx_lqctr > 0 and RETTOBASE == 1, smx_ISR_EXIT() triggers the PENDSV Handler, which calls the smx scheduler.

writing smx ISRs

The way to write ISRs varies for different processor architectures, which is documented for each in the SMX Target Guide. Some require an assembly shell (ColdFire) or dispatcher (ARM), which save and restore volatile registers and do smx_ISR_ENTER() and smx_ISR_EXIT(). Some can be written in C using an interrupt keyword (x86) that saves volatile registers, and each ISR has smx_ISR_ENTER/EXIT() macros. The easiest case is ARM-M which allows an ISR to be written as a normal C function since the processor saves and restores volatile registers, and each ISR has smx_ISR_ENTER/EXIT() macros and looks like this:

```
void MyISR(void)
{
    smx_ISR_ENTER();
    /* ISR code */
    smx_LSR_INVOKE(lsr, par);   /* if needed */
    /* final ISR code */
    smx_ISR_EXIT();
}
```

Since the ISR is a C function, the compiler will save any non-volatile registers used. Hence, nothing special needs to be done. For all cases, if a task switch is made, the smx scheduler saves all registers in the register save area (RSA).

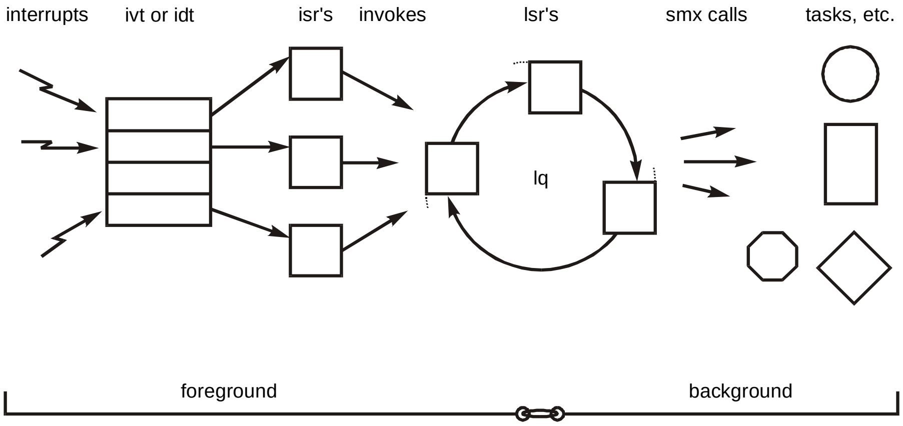
In addition to the above macros, ISRs may use only bare functions such as: smx_PipeGetPkt() and smx_PipePutPkt(), including the 8 and M variants, and sb_BlockGet() and sb_BlockRel(). SSRs may not be called from ISRs.

more on LSRs

LSRs serve four purposes:

1. Minimize interrupt latency.
2. Deferred interrupt processing.
3. High-priority, non-preemptible processing.
4. Timer processing.

Conceptually, LSRs fit as shown:



Link service routines are so named because they link the foreground to the background. They can be invoked from ISRs with:

```
smx_LSR_INVOKE (send_LSR, par);
```

The handle 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 with:

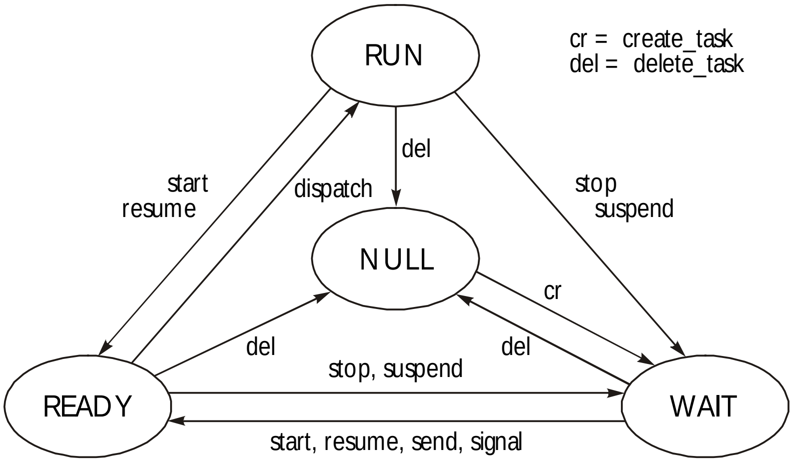
```
smx_LSRInvoke (send_LSR, par);
```

This can be useful to start an interrupt-driven process or to 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 on smx objects. LSRs are designed to be fast so they can be used for foreground operations. In practice, the fact that LSRs have no priority and execute in FIFO order has little system impact because all LSRs execute ahead of all tasks. In addition, since LSRs execute in the order invoked, the order they run will mirror the order that ISRs ran. For interrupts that occur simultaneously, ISR order is determined by ISR priorities, so LSRs will also run in ISR priority order.

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 Chapter 3 Introduction to Tasks), it is apparent that the only restriction on LSR calls is in causing the current task to leave the run state due to calling a *limited LSR* – see discussion, below.

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, the call fails and SMXE_WAIT_NOT_ALLOWED is reported. Consider the following example:

```
LCB_PTR  send_LSR;
SCB_PTR  done;
XCB_PTR  data_out;

void  send_LSR_main(u32 arg)
{
    MCB_PTR  msg;
    u8* mbp;

    if (msg = smx_MsgReceive (data_out, &mbp, SMX_TMO_NOWAIT)
        ProcessMsg(mbp);
}
```

In this example, if a message is available at the exchange, the LSR will receive and process it. Otherwise nothing happens until the LSR is invoked again. This is an example of using an LSR to implement polling.

Heap calls have an automatic mutex timeout, which would cause an error for a heap call from an LSR if the heap were not available. To avoid this, if the mutex is not owned, the LSR is allowed access to the heap by borrowing the mutex. This is ok because no task nor other LSR can interrupt the current LSR. However, if the mutex is owned, the call fails without generating an error.

limited SSRs

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. For example:

```
void  receive_LSR_main(MCB_PTR msg)
{
    /* process msg */
    msg = smx_MsgReceiveStop(data_out, &mbp, SMX_TMO_NOWAIT);
}
```

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_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 could be any task that was running when the LSR runs.

example

```
typedef  enum {INIT, START, STOP, ERROR}  mt;

tmode = smx_TaskCreate(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_main(u32 par)
{
    smx_TaskStart(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(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_main(), 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 an error has occurred. This is a good use for a one-shot task, since it obviously runs very seldom, it performs a simple function, and then it autostops and releases its stack. Note that tmode runs locked, so that no other task can run while it is changing the system mode.

why do LSRs not have priorities?

The main reasons are:

1. Simplicity.
2. Performance.
3. Temporal integrity.
4. Multiple invocations.

If LSRs had priorities, things would be more complicated for smx and for the user. For example, we might have LSRs preempting each other. Performance is obviously reduced if an LSR priority queue were to be implemented. It is important to realize that as far as tasks are concerned, LSRs run all at once.

Whereas it is generally desirable to prioritize interrupts, it is also desirable to keep events in order of occurrence at the LSR level. We call this *temporal integrity*. The ability to invoke an LSR multiple times allows absorbing high event loads without the system breaking. Hence LSRs act as shock absorbers for a system. This would be hard to do if LSRs, like tasks, had priorities.

If priority processing is more important, then it is recommended to use LSRs just to resume or restart tasks that do the work. Task priority levels can mirror the priorities of the interrupts that they service. Hence the LSRs can be just a thin separation layer between ISRs and tasks. On the other hand, as noted above, when ISRs occur simultaneously, the LSRs invoked will tend to be in ISR-priority order.

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 of the ISR. This helps to avoid access conflicts and simplifies debugging.
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, the LSRs will run in the order they were invoked, thus maintaining temporal integrity. The importance of this is that system may become sluggish, but not break, thus enabling 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 because smx_ct could be any task at the time the LSR runs.

(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_main(u32 par)
{
      TCB_PTR task = (TCB_PTR)par;
      smx_TaskStart(task);
}
```

how LSRs work

If an SSR is interrupted and an LSR is invoked by the ISR, the LSR will not run until:

(1) All nested ISRs have completed.

(2) Any SSR that was interrupted has completed.

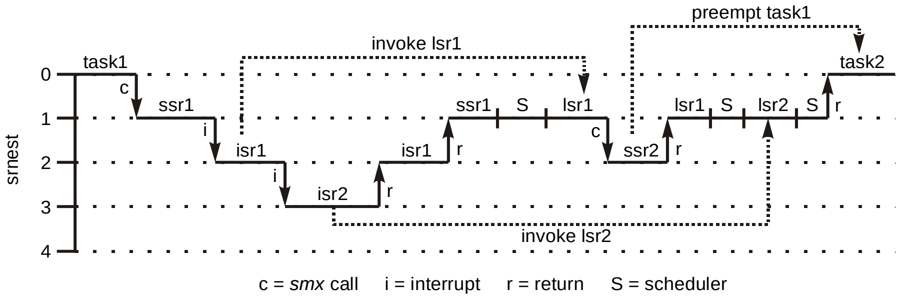
(3) Any LSR that was interrupted has completed.

(4) The scheduler, if interrupted, has completed.

(5) All LSRs ahead of it in the LSR queue have run.

LSRs cannot become nested. 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. srnest is the service routine nesting level:



At time 0, task1 is running 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 at 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 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 runs.

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 would not be correct.

Careful examination of the above figure reveals that neither SSRs[6] 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 Chapter 21 Coding.

Although LSRs cannot cause access conflicts for smx objects, there can be access conflicts between LSRs and the current task for system resources such as global variables or peripherals. To avoid this, call smx_LSRsOff() in the task prior to accessing such a resource and call smx_LSRsOn() after. See also **foreground/background access conflicts** in Chapter 24 Resource Management.

custom SSRs

To convert a function to a system service routine (SSR), start it with smx_SSR_ENTERn() and end it with smx_SSR_EXIT(). For example:

```
ret_type  NewService(p1, ... pn)
{
      smx_SSR_ENTERn(SMX_ID_NEW_SERVICE, p1, ... pn);
      /* new service code */
      return (smx_SSR_EXIT(ret_val));
}
```

smx_SSR_ENTERn() is a macro that increments smx_srnest. If logging is enabled (SMX_CFG_EVB is 1), it also causes the NewService ID and its n parameters to be logged into the event buffer, EVB, whenever NewService() runs. The ID can be defined in the SMX_ID list at the end of xdef.h. This allows smxAware to recognize NewService and to show it by number when it occurs. Alternatively, you can simply ignore this feature, by using smx_SSR_ENTER0(0).

---

[6] An SSR can call another SSR, which would result in SSR nesting. We are ignoring that case here to make the point that SSRs cannot be simultaneously called from tasks and/or LSRs.

The maximum size of the return type from an SSR is 32 bits. 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:

```
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:

```
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 copying the smx SSR and giving it a different name, and then modifying it as you wish. Custom SSRs can also be used to add new functionality to smx. The advantage of using custom SSRs instead of functions is that they are task-safe and LSR-safe.

inner SSRs

An *inner SSR* is an SSR called from another SSR, called the *outer SSR*. An important limitation of inner SSRs is that they cannot wait. If no timeout is specified, there is no problem. However, if a timeout is specified, problems can occur. For example, if smx_MutexGet(mutex, tmo) is an inner SSR, tmo > 0, and the mutex is not free, the current task will be enqueued in the task queue of the mutex, and the scheduler control flag will be set to SUSP. However, since smx_srnest > 1 (due to being in an inner SSR) the scheduler will not be called until the outer SSR exits. Then the current task will be suspended; it will be resumed when it becomes the mutex owner. This causes three problems:

(1) The outer SSR will have continued executing, when it should have waited on the mutex. Thus reentrancy protection has been lost.

(2) The task will return after the point of call of the outer SSR, not after the inner SSR.

(3) The task will now be the permanent owner of the mutex, since it did not release it. Hence all other tasks will be locked out from the critical section protected by this mutex.

Chapter 17 Timing

Timing, of one sort or another, plays a vital role in most real-time systems, and smx supports a variety of timing requirements.

etime and stime

Within smx, two times are maintained:

(1) smx_etime elapsed time
(2) smx_stime system time

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 primarily for task timeouts.

smx_stime is the elapsed time, in seconds, since either the last cold start or since some time origin such as 00:00:00 1/1/2001 GMT. It is a 32-bit counter and has a range of over 100 years. smx_stime is used by smx_TaskSleep() and smx_TaskSleepStop(); it may also be used for time stamping files in a file system and similar purposes.

smx_stime is initialized by sb_StimeSet(), which is called from main(). As delivered, if SB_CC_TIME_FUNCS is TRUE, smx_stime is initialized to 00:00:00 1/1/2001 GMT; if FALSE it is initialized to 0. 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 smx_stime 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 smx_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 SMX_CFG_TICKS_PER_SEC in acfg.h. sb_TickInit(), in bspm.c, 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, in ticks. When such an SSR is executed, the task's timeout is loaded with:

```
smx_timeout[tn] = smx_etime + timeout;
```

where tn is the task index = task->indx. timeout has a maximum value of (2exp31-1). Timeouts are handled by LSRs, which are invoked from smx_TickISR(). This provides more precise operation than handling timeouts by tasks.

smx_TimeoutLSR, in xtime.c, compares the smallest task timeout to smx_etime. If it is less than or equal to smx_etime, its task is resumed or restarted. smx_TimeoutLSR then searches the timeout array to find the next smallest timeout and invokes itself to allow other LSRs to run. It continues this process until the smallest timeout is greater than smx_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 by smx_KeepTimeLSR every tick.

timeouts for safety

Timeouts are intended primarily as a safety feature to ensure that tasks do not permanently stop running. For reliable operation, all waits should have finite timeouts. For this reason SMX_TMO_DFLT has been defined in xdef.h. It is the default timeout for all SSRs that have timeouts. SMX_TMO_DFLT is set to 10 seconds, which is an arbitrary value that should be changed to suit the requirements of each system. The intention is that it be longer than any expected normal wait time. Example usage is:

```
msg = smx_MsgReceive (port1, &bp);
```

Since no timeout is specified, the default timeout applies.

In cases where an event is expected to seldom or never occur, SMX_TMO_INF is an appropriate value. An example of this is an error manager waiting at a semaphore. Errors are not expected duirng normal operation, so an infinite timeout is appropriate. For example:

```
msg = smx_MsgReceive (port1, &bp, SMX_TMO_INF);
```

This causes the timeout for smx_ct to be made inactive, even though the task is waiting for a message at port1.

For services called from LSRs, timeouts must be set to SMX_TMO_NOWAIT. Otherwise an SMXE_WAIT_NOT_ALLOWED error will occur because LSRs cannot wait. For example:

```
msg = smx_MsgReceive (port1, &bp, SMX_TMO_NOWAIT);
```

can be used from an LSR. If no message is available, NULL is returned, and the LSR does nothing until the next time it is invoked. This is one way to implement a polling operation wherein the LSR is periodically invoked.

For tasks, SMX_TMO_NOWAIT creates what is called a *non-blocking call*. Non-blocking calls are used where there is other useful work for a task to do rather than waiting idly. Tasks can use non-blocking calls to poll for an event, like LSRs. 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 which is used when it is not desired to change a task's timeout. For example:

```
if (smx_TaskPeek(taskA, SMX_PK_STATE) == SMX_TASK_WAIT)
{
	smx_TaskStop(taskA, SMX_TMO_NOCHG);
}
```

could be used to remove taskA from a queue where it is waiting and stop it, without changing its timeout. When the timeout occurs, taskA will restart with its parameter == TRUE.

timeouts for timing

It is also possible to use the timeout mechanism for accurate (+0/-1 tick) timing. This can be useful in protocols and control operations and often makes the code simpler. In this case, a timeout would spur a retry or other normal operation. For example:

```
set_speed();
smx_TaskSuspend(self, 50);
test_speed();
```

waits 50 ticks after setting a speed before testing it.

Using timeouts like 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. Note that because the next tick may occur immediately after the SSR, a timeout of 1 may produce no timeout at all. Hence, the timeout should always be 2 or greater.

timouts in milliseconds

Timeouts may be specified in milliseconds rather than ticks by ORing with SMX_FL_MSEC, for example:

```
smx_SemTest(sema, 15 | SMX_FL_MSEC);
```

In this case, if the tick rate is 100, then 15 msec = 1.5 ticks, which is rounded up to 2. Hence, the actual wait is 2 ticks == 20 msec. If you want true msec timeouts, increase the tick rate to 1000, and then specifying ticks is the same as specifying msec and SMX_FL_MSEC is unnecessary.

handling actual timeouts

All smx SSRs return 0 or FALSE 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_ct->err == SMXE_TMO)
        /* handle timeout */
    else
        /* handle error */
}
```

Note that smx_ct->err is tested rather than smx_errno. This is because some other task could have preempted and experienced an error. smx_errno is the most recent system error, whereas task->err is the last error made by this task. Timeouts often need to be dealt with at the point of call, whereas errors may be dealt with centrally by the error manager, smx_EM() -- see Chapter 23 Error Management.

timeout priority

When a task timeout occurs, the task priority is set to task->pritmo. This allows the timeout to be processed at a higher priority, if necessary. When a task is created, task->pritmo = pri, in which case there is no difference in task priority after a timeout. After a task is created, its pritmo can be changed as follows:

```
u32   old_pri;

smx_TaskSet(task, SMX_ST_TMO_PRI, new_pri, (u32)&old_pri);
```

The effect of this is to raise task->pritmo to new_pri and to save task->prinorm in old_pri. After timeout processing is done, the previous pritmo can be restored as follows:

```
smx_TaskSet(task, SMX_ST_TMO_PRI, old_pri);
```

Timeout priority bump-up is particularly useful in protocols where timeouts occur due to unexpected events, and messages must be resent or other corrective actions taken. This requires gaining a higher priority than the receiving task in order to preempt it and to perform the necessary corrective actions.

Since timeouts usually mean that something has not gone as planned, timeout priority bump-up is likely to be useful in other contexts. The advantage is that the corrective action can be handled within the code of the task experiencing the timeout, thus keeping it local, rather than invoking another task to perform the corrective action, which is more complicated.

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) 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 timer counter and offers speed and precision equivalent to the main clock rate. 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 more 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 the tick rate is set to this or ticks are derived from it (e.g. call smx_TickISR() every 10th interrupt for a 100 Hz tick rate). 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 Chapter 18 Timers, for discussion.

smx timeouts can also provide tick-accurate timing, such as:

```
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.

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, 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.

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(u32 par)
{
    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 ensures that there is no cumulative error.

smx_TaskSleep() or smx_TaskSleepStop() merely computes an equivalent smx_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. For sleeps, usually the most important thing is that there be no cumulative timing error, which can be ensured by careful programming, as shown above.

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 msec 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(u32 par)
{
    //...
    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_main(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 stop_hdw_timer() is able to stop 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(TMRCB_PTR* tmrbp, TMRCB_PTR tmr, const char* name)
u32        smx_TimerPeek(TMRCB_PTR task, SMX_PK_PAR par)
BOOLEAN    smx_TimerReset(TMRCB_PTR tmr, u32* tlp)
BOOLEAN    smx_TimerSetLSR(TMRCB_PTR tmr, LCB_PTR lsr, SMX_TMR_OPT opt, u32 par)
BOOLEAN    smx_TimerSetPulse(TMRCB_PTR tmr, u32 period, u32 width)
BOOLEAN    smx_TimerStart(TMRCB_PTR* tmhp, u32 delay, u32 period, LCB_PTR lsr, const char* name)
BOOLEAN    smx_TimerStartAbs(TMRCB_PTR* tmhp, u32 time, u32 period, LCB_PTR lsr, const char* name)
BOOLEAN    smx_TimerStop(TMRCB_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 objects. The *timer control block, TMRCB*, exists only when the timer is running. If it times out, the timer's TMRCB is released back to the TMRCB pool. A timer is created and started by smx_TimerStart() or smx_TimerStartAbs().

reasons to use software timers

Software timers are appropriate for:

1. one-shot timeouts
2. cyclic timing requirements
3. pulse timing requirements
4. tick-accurate timing

Timers 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.

(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. 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:

```
TMRCB_PTR   TimerA;

smx_TimerStart(&TimerA, delay, period, LsrA, “TimerA”);
```

This creates TimerA and enqueues it in the timer queue (tq) at (smx_etime + delay). smx_etime is the elapsed time since the system was started. To create a timer, smx gets a timer control block (TMRCB) from the timer control block pool (smx_tmcbs) and initializes its fields. In particular, it loads the above parameters into corresponding TMRCB fields and default parameters into the other fields. 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 TMRCB. 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 (TMRCB) is linked between the TMRCB with less-than-or-equal expiration time and the TMRCB with greater expiration time. Each TMRCB stores a differential count in its diffcnt field. Only the diffcnt of the first TMRCB 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 (smx_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 TMRCB 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[7], 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 current time. 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 smx_etime, the timer is not started and SMXE_INV_PAR is reported.

---

[7] If it does, SMXE_LQ_OVFL error is reported.

stopping a timer

Timers can be stopped as follows:

```
TMRCB_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 TMRCB 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_TMRCB 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

```
TMRCB_PTR  SampleTimer;
TCB_PTR  SampleTask;
LCB_PTR  SampleLSR
#define RATE 100;

void  ainit(void)
{
    smx_TimerStart(&SampleTimer, RATE, RATE, SampleLSR,  "SampleTimer");
    SampleTask = smx_TaskCreate(SampleTaskMain, P2, 0, SMX_FL_NONE, "SampleTask");
}

void  SampleLSRMain(u32 par)
{
    u32 s;

    s = TakeSample();
    smx_TaskStart(SampleTask, s);
}

void  SampleTaskMain(u32 s)
{
    ProcessSample(s);
}
```

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:

```
TMRCB_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_TMRCB is reported. If &TimerB == NULL, or TimerB already exists (!= NULL), SMXE_INV_PAR is reported. If the TMRCB pool is empty, SMXE_OUT_OF_TMRCBS is reported. In all of these cases, the operation fails and 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:

```
TMRCB_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_main(u32 n) { }

void LsrP_main(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:

```
TMRCB_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_TMRCB 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 can 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	smx_etime at timeout
SMX_TMR_COUNT	number of timeouts since start

These options add convenience for timer LSR design.

The smx_TimerSetLSR() 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_main(TMRCB_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 have happened.

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 ensure 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_main(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, (LCB_PTR)LsrB,  “TimerB”);

void LsrB_main(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_PAR will be reported.

pulse period modulation (PPM)

Alternatively, pulse-period modulation could be implemented as follows:

```
smx_TimerStart(&TimerC, 1, PERIOD, (LCB_PTR)LsrC,  “TimerC”);

void LsrC_main(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, pulse 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_PAR 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_main(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 smx_TimerPeek() in the Reference Manual for the complete list of parameters.

If the timer handle is invalid, SMX_INV_TMRCB is reported. If par is not recognized, SMXE_INV_PAR is reported. In all three cases, 0 is returned.

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 smx_TaskPeek() in the smx Reference Manual. This could be used to log what the current task is doing, take 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 by a higher-frequency periodic interrupt from a hardware timer. Then, profiling and task timeouts would still be driven the slower tick, but timers would operate at the higher rate.

SECTION III DEVELOPMENT

The previous sections have explained how smx works, but not how to use it. This section explains how to structure an application for multitasking, how to develop it, how write the code, and how to debug it. We hope this section will get you off to a good start.

If your experience has been with superloop designs, you will find that a multitasking kernel provides a significantly different environment, which takes some getting used to. If you are accustomed to multitasking, you are not likely to find any surprises in what follows, but you may find some good ideas.

Chapter 19 Structure

function vs. structure

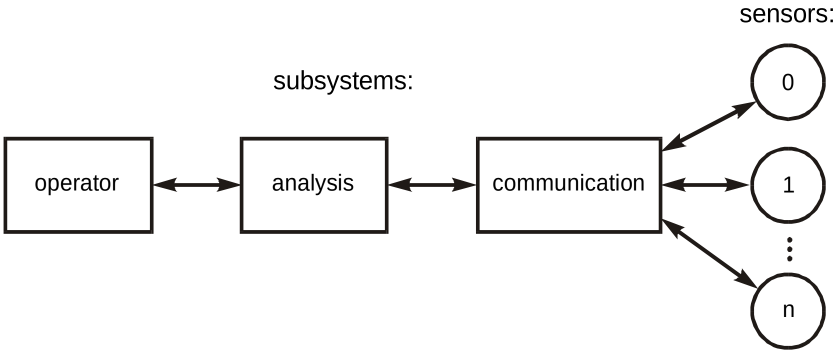
There are two aspects to a system: The functional aspect is concerned with what the code does and how it does it (e.g., the algorithms used). The structural aspect is concerned with how the code operates.

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 using objects such as tasks, messages, and exchanges. Using smx causes the system to be structured in a way that 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 the system.

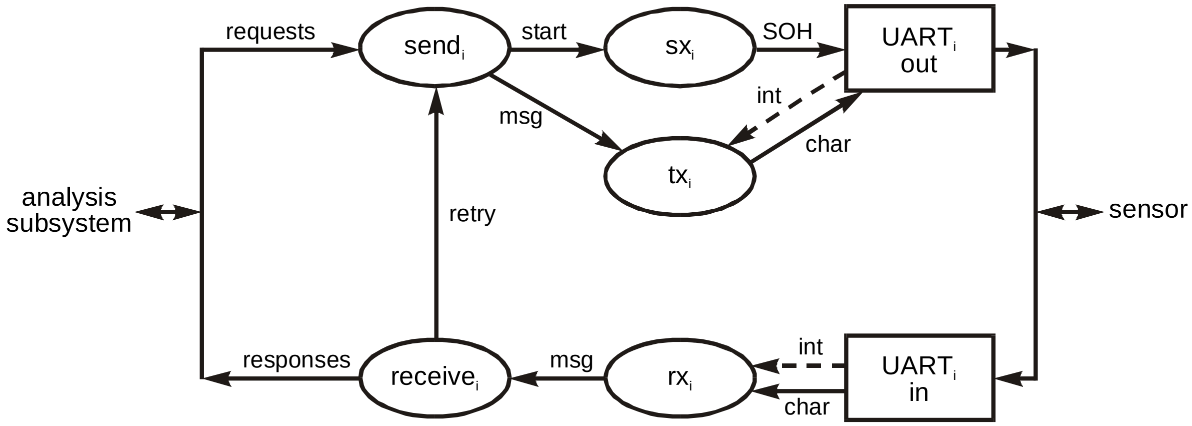
creating subsystems and tasks

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 sensors, as below:



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 a query message and initiates transmission.

(2) sx which initializes the UART send channel and causes it to begin transmission with the SOH (Start Of Header) byte.

(3) tx which outputs a byte to the UART each time a send interrupt occurs. This is an ISR.

(4) rx which accepts a byte from the UART receive channel each time a receive interrupt occurs. This also is an ISR.

(5) A receive task which accepts a complete or partial message, checks it, and forwards the message to the analysis subsystem if ok, else requests a retry. Note, there must be an LSR between rx and the receive task.

creating flow diagrams

The preceding diagram is a data flow diagram showing the flow of data and control between system objects. The lines can represent either data or control or both. Ovals represent software and rectangles represent hardware. 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 object 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.

The important thing is to start this way rather than immediately diving into the code.

more tasks are better

It is desirable to have many small tasks rather than just a few large tasks, for the following reasons:

(1) Simpler code.
(2) Greater use of smx services.
(3) More explicit structure.
(4) Better defined interfaces.
(5) smx error detection helps debugging.
(6) smxAware helps debugging.
(7) Increased modularity.

More tasks results in simpler code because each task does less, and more is done by smx services. The latter means less green code to develop and debug, and use of well-documented methods for future maintenance programmers. Structure is more explicit because there is a closer relationship between tasks and functions, and interactions are external rather than internal to tasks. External interactions use well-documented smx services rather than complex internal code that may have been patched and repatched so many times that it is incomprehensible.

Projects typically overrun schedules during the debugging phase, so anything that helps to debug projects faster is of great importance. Simpler tasks are, of course easier to debug. Better defined interfaces means less difficulty during integration of code from different programmers. More explicit structure also helps with this. The extensive error detection performed by smx is a big help during debugging, and smxAware is also a big help because it allows seeing how tasks are interacting plus making it easier to view smx objects.

Of couse modularity is greater with smaller tasks, since each does less. This makes it easier to modify systems by replacing tasks with tasks that operate differently. It also allows tasks to be more easily used in other systems.

guidelines for defining tasks

The criterion of whether a task is needed or not is largely based upon how you want to structure your system. There are no hard rules, but the following guidelines may help:

(1) Functions which are asynchronous, with respect to each other, should be in separate tasks. This makes better use of smx facilities and is simpler.
(2) Functions which are unrelated should be put into separate tasks.
(3) Distinctive processing such as floating point calculations, encryption, and Fourier transforms merit separate tasks — especially if they utilize special hardware, such as coprocessors.
(4) Generally, each peripheral will require at least one task. Often, two tasks are required, one for input and one for output functions of the peripheral.
(5) If there are multiple sensors, ports, actuators, etc., of the same type, each should have its own task. Then, while some tasks are waiting on their devices, other tasks can be handling their devices.

(6) Sometimes it simplifies design if tasks might mirror external objects, such as devices being assembled on an assembly line.

(7) It is better to define too many tasks, at first, and later combine them when it becomes apparent that there is no need for so many tasks, rather than defining too few tasks, at first. Defining too few large tasks may obscure subtle interactions that are better handled with multiple tasks. An example of this is #5 above, where using multiple tasks might greatly improve performance.

benefits of multitasking

Division of an application into many independent domains is one of the main benefits of multitasking. It helps to minimize the propagation of changes within the code. 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.

getting started

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 our programming mistakes!

Chapter 20 Methodology

The methodology is the means by which you progress from flow diagrams to code. 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 *system framework*.

system framework development

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 framework actually runs, and allows you to work with your system before most of the code has been written. This helps to avoid rewriting large amounts of code, and is therefore highly recommended.

It is possible to experiment with different structures supported by smx to determine which works best. A framework 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 framework to determine their impact. If not favorable, this may lead to picking a different processor, reduction of scope, or other alternatives.

Framework development is especially helpful in multi-programmer projects because it permits focusing on interfaces between subsystems, rather than code, and defining how the subsystems interact, before hundreds 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.

The framework 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 it, but also to determine its impact upon your plan. If you are attempting to use third-party middleware, you may find that integration problems are more difficult than expected or it is unreliable or too slow. Better to find these out sooner than later.

system framework example

Starting from your system flow diagrams and object descriptions, first create tasks:

```
TCB_PTR  Op, Anlyz, Comm;   /* main tasks */

void ainit(void)
{
     Op = smx_TaskCreate(Op_main, P3, 200, SMX_FL_NONE, "Op");
     Anlyz = smx_TaskCreate(Anlyz_main, P3, 200, SMX_FL_NONE, "Anlyz");
     Comm = smx_TaskCreate(Comm_main, P3, 200, SMX_FL_NONE, "Comm");
     smx_Start(Op);
     smx_Start(Anlyz);
     smx_Start(Comm);
}
```

```
void Op_main(u32 par)
{
    open_delay();
}

void Anlyz_main(u32 par)
{
    anlyz_delay();
}

void Comm_main(u32 par)
{
    comm_delay();
}
```

This code should compile and run. Of course, it does nothing. The delays are just estimates to make operation more realistic. We have identified the main tasks for each subsystem, created and started them. Now let’s expand upon the Comm task:

```
#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"]}

void Comm_main(u32 par)
{
    u32  i;

    for (i = 0; i < N; i++)
    {
        rec[i] = smx_TaskCreate(rec_main, P2, 300, SMX_FL_NONE, rtname[i]);
        send[i] = smx_TaskCreate(send_main, P2, 300, SMX_FL_NONE, stname[i]);
        smx_TaskStart(rec[i], i);
        smx_TaskStart(send[i], i);
        rx[i] = smx_MsgXchgCreate (SMX_XCHG_NORM, rxname[i]);
        sx[i] = smx_MsgXchgCreate (SMX_XCHG_NORM, sxname[i]);
     }
}
```

In the previous chapter, it looked like having separate receive and send tasks for each port was a good idea, so let’s implement that idea here. If it does not work, there is not much code to change. 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 at P2 priority. Note that the port number, i, is passed to each receive and send task. Continuing:

```
PCB_PTR  spool[N];
u8* sdp[N];

void send_main(u32 i)
{
    u8* mbp;
    MCB_PTR  msg;

    while ((msg = smx_MsgReceive(sx[i], &mbp, INF)) != NULL)
    {
        sdp[i] = smx_MsgUnmake(msg, &spool[i]);
        start_uart(i);
    }
}
```

The N send tasks share send_main(). The task is identified by the port number, i, passed to it by smx_Start(send[i], i) in Comm_main(). Send task i waits at the sx[i] exchange for a message to send. When it receives a message, it unmakes it and starts the UART for port i. The data block pointer is passed via the sdp array to smx_MsgUnmake(). The code above will compile and you can step through it with your debugger. Not much will happen, but 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 framework while the analysis and operator programmers are working on their frameworks. Then the frameworks can be put together to see if this primitive system operates well.

framework system development

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(). At every step, code will compile and run and smxAware can be used to visualize what is happening. At this early stage, as problems are found that dictate a different approaches, the necessary changes can be made without scrapping or rewriting large amounts of code.

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.

Also, 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.

The capability to run the system framework at any time, as system design progresses, should appeal to management. Being able to see the design unfold via smxAware should inspire their confidence that the project is on track. In addition, late Marketing changes are likely to be accommodated with changes to a single or a few tasks rather than propagating through the whole code and undermining reliability.

In effect, system integration is being performed as application code is being written and debugged, rather than afterward. Thus, when the last code has been debugged, system integration is done and system testing well underway.

evolutionary development

Frequently, projects start with legacy non-multitasking code, which runs ok, but needs to be extended. Typically, the background code can be initially run as a single main task (including the superloop). ISRs will require some minor restructuring as explained in the chapters on ISRs and LSRs.

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

Skeletal system development uses smx resources to develop an operational system that runs prior to detailed code development. The objective is a well-defined place and interface for each block of application 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 to confirm that operation is as expected or to identify problems and fix them before too much code is written.

Since multitasking is complicated and there are many inter-task communication mechanisms and task structures to choose from, it makes sense to get the basic structure right before generating code.

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. 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 framework design. This is a compendium of useful techniques, from previous chapters. The goal is to achieve the best fit of form to function as the framework 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 Chapter 13 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. Then an ISR can complete outputting the pipe contents. See **pipe I/O** in Chapter 12 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 release the smx blocks or messages back to their base pools for reuse by ISRs. See Chapter 12 Pipes and Chapter 13 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. ISRs 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. Then tasks not only represent states, but

also perform the actions of the states. See the **state machine example** in Chapter 10 Event Groups.

(6) Interrupt Events. External events typically trigger interrupts, which cause ISRs to run. ISRs should perform minimal urgent processing. If more processing is needed ISRs can invoke LSRs, which can perform moderate, quick processing. If even more processing is needed, LSRs can start tasks to run. Hence, three levels of processing are available to achieve optimum interrupt responses. See Chapter 16 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. See Chapter 18 Timers.

(8) Resource Sharing. Sharing resources between tasks without conflicts is very important. See Chapter 24 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. See **gate semaphore** in Chapter 8 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 Chapter 8 Semaphores.

(11) Periodic Operations are best implemented using cyclic timers. When such a timer expires, it immediately restarts itself so no ticks are lost and then invokes an LSR to perform the desired operation. Since LSRs cannot be blocked by tasks, this results in low-jitter operation. See Chapter 18 Timers.

(12) Date/Time Operations are best implemented with task sleeps. See **date-time delays** in Chapter 17 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 Chapter 13 Exchange Messaging.

(14) Multicasting is similar to broadcasting. 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 messages and multicasting** in Chapter 13 Exchange Messaging.

(15) 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 Chapter 7 Intertask Communication.

(16) Avoid Global Variables. These are the cause of many problems, especially in multitasking systems. It is best to pass the data that a task needs via a message sent to an exchange or a via block put into a pipe. See Chapter 13 Exchange Messaging and Chapter 12 Pipes.

(17) Group related variables into structures. This makes better use of processor addressing mechanisms and data caching in order to improve performance and reduce code. For example, in ARM-M two LDR instructions are necessary to load a variable – one to load its address, the second to load the variable. If the variables are in a structure, then one LDR loads the structure address and subsequent LDRs load fields by means of offsets.

(18) Aligning Structures on cache lines is even better, if the processor has a data cache:

```
#pragma data_align = SB_CACHE_LINE
struct  x
{
      /* fields */
}
```

(19) Compress Structure Fields to improve performance. For example flags can be combined into a single byte, half word, or word bit field; counters can often be reduced to 8 or 16 bytes; addresses can be reduced to 8 or 16 bit indices. Compressed fields require processing, but processing cycles can be much faster than variable accesses.

keep it simple

With its many calls and system objects, smx may seem overwhelming. 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. Building and running a framework helps to see what works and what does not work for your application. What you do not use is not linked in, so it does not matter. However, if you run into a new problem, you may find that some unused smx feature is now needed.

keep tasks small

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, then waiting for different event, doing a different operation, and so on, you probably have implemented a superloop inside of a task! This is not the right idea — that task needs to be broken up. The advantages of doing so are:

1. (1) Each task will be simpler.
2. (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.

simpler task loops

As discussed in previous chapters, normal tasks have infinite internal loops as follows:

```
void taskA_main(u32 par)
{
    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(u32 par)
{
    /* 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.

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, the same happens. 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(u32 par)
{
    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 it might be necessary to use the following, instead:

```
    while ((msg = smx_MsgReceive(xchgA, &mbp, tmo)) != NULL)
```

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 Chapter 24 Resource Management for discussion of ways to protect application code.

invalid handles

All system objects, except timers, are 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 and report errors (e.g. SMX_INV_TCB).

Generally speaking, code such as the following is not good programming practice:

```
t2a = smx_TaskCreate(...);
smx_TaskStart(t2a);
```

because the t2a handle is not being tested before using it. However, since smx_TaskStart() checks that t2a is a valid task handle, you can get away with this to simplify your code. However, it is not likely to pass static testing.

task arrays

Sometimes it is helpful to create several tasks that do the same thing and use the same code. This situation can be handled by creating an array of tasks as follows:

```
#define N 3

TCB_PTR  tsk[N];
XCB_PTR  xchg[N];
const char* tname[] = {"tsk1", "tsk2", "tsk3", …};

void app_init(void)
{
        for (n = 0; n < N; n++)
      {
            tsk[n] = smx_TaskCreate(task_main, P2, 200, SMX_FL_NONE, tname[n]);
            xchg[n] = smx_MsgXchgCreate(SMX_XCHG_NORM, NULL);
            smx_TaskStart(tsk[n], n);
      }
}

void  task_main(u32 n)
{
      u8* bp;
      MCB_PTR msg;

      /* initialize tskn here */

      while (msg = smx_MsgReceive(xchg[n], &bp, 100))
      {
            ProcessMsg(n, bp);
            smx_MsgRel(msg, 0);
      }
}
```

In this example, N tasks share the same code, task_main(). A task handle array, tsk[N] is defined, and a for loop is used to create and start N tasks. The task main function parameter is the task number so task_main(n) can initialize each task. Then task n enters an infinite loop where it waits for a message from xhg[n] and processes it when received. Each exchange constitutes a different message stream, so each task processes a separate message stream. As indicated, the processing of each stream could be different or could be the same.

The same could be done with a single task processing messages from a single exchange.
The advantages of the multitask approach are:

(1) The number of streams can easily be changed, just by changing N.

(2) The parallel stream structure is more visible to tools such as smxAware.

(3) The parallel stream structure is easier to modify, if requirements change.

The cost for the parallel task structure is low: 84 bytes for the TCB and about 200 bytes for the stack = 2840 bytes for 10 tasks. This could be reduced further by using N one-shot tasks, as follows:

```
#define N 3

TCB_PTR  tsk[N];
XCB_PTR  xchg[N];
const char* tname[] = {"tsk1", "tsk2", "tsk3", …};

void app_init(void)
{
    for (n = 0; n < N; n++)
    {
        tsk[n] = smx_TaskCreate(task_init, P2, 0, SMX_FL_NONE, tname[n]);
        xchg[n] = smx_MsgXchgCreate(SMX_XCHG_NORM, NULL);
        smx_TaskStart(tsk[n], n);
    }
}

void task_init(u32 n)
{
    smx_TaskSet(smx_ct, SMX_ST_FUN, task_main);
    smx_MsgReceiveStop(xchg[n], NULL, 100);
}

void  task_main(u32 msg)
{
    MCB_PTR msg = (MCB_PTR)m;
    XCB_PTR xchg;

    bp = (u8*)smx_MsgPeek(msg, SMX_PK_BP);
    ProcessMsg(n, bp);
    smx_MsgRel(msg, 0);
    xchg = (XCB_PTR)smx_MsgPeek(msg, SMX_PK_REPLY);
    smx_MsgReceiveStop(xchg, NULL, 100);
}
```

In this example, N tasks are created and started with task_init() as their main function. Each is a one-shot task since no stack is allocated. As each task runs, its main function is changed to task_main() and it does a receive stop at exchange xchg[n]. When a message is received at xchg[n], tsk[n] is restarted with task_main() as its main function, and the received message handle is passed into it. The message block pointer is obtained with a peek, and the message is processed and released. The exchange to wait for more messages is obtained with another peek. For this to work, the sending task must specify the xchg[n] as the reply exchange:

```
smx_MsgSend(msg, xchg[n], 0, xchg[n]);
```

Then tsk[n] waits with no stack for another message at xchg[n]. Now the cost is 84*N + 200 = 1040 bytes for 10 tasks. Only one stack is required because all tasks have the same priority so only one can run at a time.

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* is said to occur if one or more mid-priority tasks preempt the low priority task and keep the high priority task waiting for an indeterminant time. 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 determine if the high-priority task will meet its deadline.

The unbounded case cannot be calculated because it is unknown what mid-priority tasks might preempt the low priority task and for how long. Hence, in a system where high-priority tasks have tight deadlines, unbounded priority inversion must be avoided.

Methods to avoid unbounded priority inversion, include:

(1) Use mutex priority promotion.
(2) Use mutex ceiling protocol.
(3) Use servers.
(4) Use LSRs.

For the first two see Chapter 9 Mutexes. Using a server to access a shared resource is a nice solution because it allows the high priority task to go about its business while the server task handles the shared resource. LSRs are better than tasks for deferred interrupt processing because they run ahead of all tasks and thus are not subject to task priority inversion.

Unfortunately, unbounded priority inversion can result in other ways. For example, if a high-priority task is waiting at a resource semaphore and a low-priority task is using the last resource, it could be preempted by medium-priority tasks thus causing unbounded priority inversion. This could be avoided by locking the low-priority task while it has the resource. Then, when it returns the resource, the high-priority task should be first in line for the resource because of its priority.

There are other ways that unbounded priority inversions can occur. In general, it is best to avoid sharing resources between low-priority tasks and high-priority tasks. It is also best to avoid high-priority task dependencies on low-priority tasks. For example, high-priority tasks can send messages to exchanges for processing by low priority tasks, but should not wait for responses from the low priority tasks.

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, finite timeouts should always be specified on waits so tasks can recover and continue operating.

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 usage by ISRs, exception handlers, 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 the 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_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() needs 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 Chapter 22 Debugging and Chapter 6 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 event recording, profiling, stack scanning, handle table, lock nest limit, etc.

Configuration constants that are application-orientated, such as the numbers of various system objects (e.g. SMX_CFG_NUM_TASKS), LSR queue size, stack pool stack size, 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.

A master pre-include for a tool/processor combination, such as iararm.h specifies the products included in a build, such as smxAware and SecureSMX, demos, and the target board being used.

user access to smx objects

smx peek services are the recommended way to obtain information about smx objects rather than accessing smx objects directly. They are currently provided for most smx objects. Peek services are implemented with SSRs and thus are task-safe. For example,

```
first_task = (TCB_PTR)smx_SemPeek(sem, FIRST);
```

will return the 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 to inhibit LSRs while accessing sem->fl, because the C assignment statement is not task-safe and a meaningless value could be loaded into first_task, if a preemption occurred that changed sem->fl. smx_TaskLock() blocks preemption by another task. smx_LSRsOff() blocks preemption by an LSR invoked by an interrupt as well as preemption by another task, which requires an LSR to run. Hence, it is safer, but unnecessary if no LSR is accessing sem.

naming objects

We recommend naming smx handles 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 sectors_pool;
MCB_PTR next_sector;
u8* mp;

next_sector = smx_MsgGet(sectors_pool, &mp, 0);
```

sectors_pool is a message block pool of sector-size message blocks; next_sector is the next message to be used.

All caps are used in smx for manifest (i.e. compile time) constants, C macros, typedefs[8], directives, and keywords. In this manual, a name in all caps is usually either an smx data type or a manifest constant

---

[8] 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.

Chapter 22 Debugging

Because tasks can preempt each other and thus run in random order, multitasking systems are more difficult to debug than non-multitasking systems. This chapter is intended to provide some helpful ideas for multitasking debugging.

debug tools & features

The following smx tools and features are available to help with debugging:

(1) smx error manager detects and reports over 80 error types, which are helpful during debug. See Chapter 23 Error Management.

(2) smxAware is a DLL that adds kernel-aware capabilities to debuggers. When integrated with a debugger like C-SPY, “smxAware” appears in the top menu bar. Clicking on it brings up the following options:

   a. Text shows details for all smx objects.
   b. Graph shows event timelines for each ISR, LSR, and task. Can zoom in and a detail window shows what services were called.
   c. Event Buffer shows the same event information in a table.
   d. Resource Usage shows how many objects of each type are in use.
   e. Memory Map shows everything in RAM, including heap allocations.

   See the smxAware User’s Guide for full details about features and debuggers supported.

(3) profiling. When SMX_CFG_PROFILE in xcfg.h is 1, smx 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*. Chapter 26 Precise Profiling.

(4) event logging. When SMX_CFG_EVB in xcfg.h is 1, 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.

(5) time measurement macros are provided for precise time measurements of up to one tick. See smxBase User’s Guide.

(6) stack checking. Stack usage is automatically monitored by smx to help tune stack sizes and to detect stack overflow. When SMX_CFG_STACK_SCAN in xcfg.h is 1, stack scanning is 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 Chapter 6 Stacks.

(7) heap checking. smx_HeapScan() is called regularly from the idle task to scan a few heap nodes at a time. smx_HeapBinScan() is called regularly to scan a few bin nodes at a time. See Chapter 5 Heaps.

(8) message output. sb_MsgOutConst() and sb_MsgOutVar() are used to output debug messages at various points in the code to see the order in which things are executing. They are used by the smx error manager and middleware and may be used by application code, as well. Messages are buffered and written to the terminal/UART by sb_MsgDisplay() which is called by the idle task. sb_MsgDisplay() also calls sa_Print() to add them to the smxAware print ring, which appears in the Print section of the smxAware Text displays. sa_Print() may be called directly by the application as well. Messages pending in the buffers and not yet displayed to the terminal at a breakpoint or crash are displayed in the smxAware Print display. See the smxBase and smxAware User's Guides for more information about these features.

(9) task suspend/resume location. task->susploc stores the address of the next instruction that will execute for task at the point of SSR call or interrupt, when it is resumed. This is useful to see where and why a task was suspended or to set a breakpoint there to continue stepping through it.

(10) smx_TickISRHook() is useful to emulate an interrupt during early debugging.

(11) handle table allows assigning names to objects that do not have control blocks or which do not have name fields in their control blocks.

important smx variables

It is helpful to keep some or all of the following smx variables in the debugger watch window while stepping through code:

(1) smx_errno and sb_errno – show reported errors.

(2) smx_ct — current task handle.

(3) smx_clsr — current LSR handle, if not 0 nor smx_nullcb.

(4) smx_sched — scheduler flags: CTSTOP, CTSUSP, & CTTEST. Determine what the scheduler will do.

(5) 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.

(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.

looking at smx objects

smx_errno and sb_errno It is important to keep an eye on these, when debugging – if you are seeing something that is confusing it may be because an error occurred.

smx_cf holds configuration values from acfg.h in a structure for smxAware and that the user can view in the watch window.

smx_ct is the current task. Clicking on it shows the full TCB. smx_ct->name is good to watch.

smx_clsr is the current LSR, if not zero nor smx_nullcb. Clicking on it shows the full LCB.

smx_rq is statically defined as an array of RQCBs. Clicking on it shows all rq levels. Clicking on a level shows the RQCB for that level. If the level is occupied, its fl points to the first task at the level. Clicking on the fl shows the TCB with the task's name. Clicking on the task's fl shows the next task, etc. Hence the order of tasks at the rq level can be determined. Note: clicking on Ready Queue in the smxAware Text window makes this much easier.

smx_rqtop points to the top occupied rq level.

LSR queue (LQ) holds LSRs waiting to run. It is easily displayed by clicking LSR Queue in the smxAware Text window.

event buffer (EVB) is defined in the linker command file. smx_evbi, smx_evbx, and smx_evbn point to the first, last, and next to fill error records. They are word pointers and event records are of variable size. Hence, it is only practical to look at the event buffer via smxAware.

error buffer (EB) is defined in the linker command file. smx_ebi, smx_ebx, and smx_ebn point to the first, last, and next to fill error records. To see the record of the last error click on smx_ebn-1. In smxAware, EB is shown in the Text displays under Diagnostics and by the Error button on the Graph display.

debug tips

(1) Watch for errors. Often, debugging difficulty is simply due to not recognizing that an error has occurred. For example, being out of an object such as mutex control blocks, can result in puzzling behavior. To avoid going down a rabbit hole, watch for errors – smx detects and reports a lot of them. When an error is detected, an error message is output to a console, if you have one. This is a bit easier to see than watching smx_errno and sb_errno in the debugger watch window, although that works too.

(2) Task tracing. Unlike superloop tracing, when task tracing, it often is not clear what task will run next. Put a breakpoint at the end of smx_PreSched where it returns to the new task with POP {PC}, BX LR, or similar, and when hit, step into the task.

It is also possible to debug a task from the point it was suspended, since smx stores that address in its TCB. In the smxAware Text window Tasks display, look at the row for the task, and copy the address in its SuspLoc column into the debugger's Disassembly window to show the code there. Set a breakpoint on that assembly statement and run to it. (The address is stored in the susploc field of the task's TCB, but smxAware only displays it when it's valid.)

(3) Confusing results when stepping is a frequent problem. It is easy to forget that C statements are not atomic. Each consists of many machine instructions, and interrupts can occur between these instructions. When you step, interrupts may occur, causing ISRs, LSRs, and higher priority tasks to run. A lot can happen in the blink of an eye, possibly causing confusing behavior when stepping through a task. To make matters worse, hardware peripherals aren't stopped at a breakpoint, changing the sequence in which things run on each step in the debugger. One way to simplify things is to disable interrupts in sections of code that are being stepped through. Another is to lock tasks that are being stepped through. The main thing is to be aware of this problem when stepping through task code.

(4) Finding where an error occurred. Uncomment sb_DEBUGTRAP() in smx_EMHook() in smxmain.c and in sb_EM() in bbase.c to make the debugger breakpoint on an error. You can determine where the error occurred from the call stack window. To enable sb_DEBUGTRAP(), if not already enabled, click on project file Options, C/C++ Compiler, Preprocessor, and set SMX_BT_DEBUG=1 in Defined symbols.

Use taskA->err, rather than smx_errno to determine what error, if any, occurred for taskA. It is the last SSR result that occurred specifically for taskA. smx_errno is the last error that occurred for the system. Hence it could be an error for taskB, if taskB preempted taskA. This seldom happens, but obviously can cause a lot of confusion when it does.

(5) Use stack pads Stack pads are put above the tops of stacks. During debug, it is recommended to allocate stack pads of 100 bytes or larger if memory allows. This allows an application to continue running when a stack overflow occurs. Otherwise, an adjacent stack or heap control block will be damaged causing erratic system behavior that obfuscates finding the actual problem. SMX_CFG_STACK_PAD_SIZE in acfg.h controls the size of stack pads. Stack overflow is detected and reported as normal.

(6) Task stops running can occur if not using timeouts on waits. It is recommended to pick a default timeout and to use it for all waits. Then a timeout will cause smx_EM() to call smx_EMHook(errnum, handle) where you can call sb_DEBUGTRAP() if errnum == SMXE_TMO. The call stack window will reveal were the timeout occurred and you can fix it. This problem can also occur when using a library, which uses binary resource semaphores to protect non-reentrant functions and one such function calls another. In that case, it is best to find another library.

(7) Assigning names to system objects is very helpful. Most smx 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 displayed by smxAware.

(8) Looking at task timeouts. Task timeouts are grouped into the *smx_timeout[]* array. 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 task->indx then enter smx_timeout[indx] into the debugger's Watch 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 left to go. *smx_tmo_min* is the value of the next timeout to expire and *smx_tmo_indx* is its index into smx_timeout[]. You can also use smx_TaskPeek(task, TMO) to get the time to go for task.

(9) Use smxAware extensively. It can make debugging much easier. See above discussion and the smxAware User's Guide.

SECTION IV SYSTEM RESOURCES

Prior sections have presented smx functionality, how to use smx, and how to debug smx projects. This section presents system resource information, such error management, resource management, event logging, precise profiling, and power management.

Chapter 23 Error Management

introduction

All smx services return FALSE or NULL if an error or timeout has occurred. The cause can be determined during debugging by checking smx_errno, smx_ct->err, or smx_clsr->err in the debugger watch window. smx_TaskPeek(task, SMX_PK_ERROR) should be used in code to determine the error for 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 discussion of which to use.

smx error detection

smx detects over 80 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 SSR 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.

smxBase error detection

smx_Base reports 16 types of errors in sb_errno for smx_Base operations. See bdef.h for a list of error types. During debugging, these errors can be monitored with sb_errno in the debugger watch window. If a base function has an error, sb_EM() is called, which increments sb_errctr and outputs and error message to the console. In addition, the base function returns FALSE or NULL, so local error management can be implemented. Tasks seldom call base functions directly. Mostly they are called by smx SSRs, which report appropriate smx errors when base functions fail. See smxBase User’s Guide for more information

stack overflow detection

Stack overflow is a particularly serious 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 Chapter 6 Stacks for detailed discussions.

stack switching for smx_EM()

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.

For a Release build, smx_EM() is called after switching to the system stack (SS). In this case, SMX_BT_DEBUG is not defined. Using SS for smx_EM() saves about 60 bytes per task stack. More importantly, it avoids a situation where smx_EM(), itself, could cause a stack overflow error. After smx_EM() finishes, a switch is made back to the task stack.

smx_errno vs. task->err and lsr->err

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 or in the LCB of the LSR that caused the error. smx_errno stores the last error that occurred for the system, whereas task->err stores the last error that occurred for task, and lsr->err stores the last error that occurred for lsr. task->err can be accessed with smx_TaskPeek(task, SMX_PK_ERROR). If the last smx service returned an error indication, then task->err or lsr->err is the error or timeout that occurred.

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

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. These are 8-bit counters to minimize RAM usage. If the counters are checked regularly, 8-bits should be enough. If a counter has overflowed, the sum of the error counters will be less than smx_errctr. Each error is also recorded in the error buffer, EB, if smx_eben is TRUE, and in the event buffer, EVB, if SMX_CFG_EVB is 1 in xcfg.h and logging errors is enabled by SMX_EVB_EN_ERR in xevb.h.

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 watch window of the debugger in order to see errors. Next, the smx_EMHook() callback function is called. For most errors, smx_EM() returns to the point of call or to the scheduler. For irrecoverable errors, it calls the smx_EMExitHook() callback function, which normally reboots. These are discussed below.

error buffer (EB)

EB, is a global data structure, which stores the following information for each error:

(1) Time of occurrence (smx_etime).

(2) Error number.

(3) Task or LSR for which the error occurred or 1 if a system error

Space for EB is allocated by EB size in the linker command file. The number of records in EB is determined by EB size/sizeof(EREC). 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 should be adequate. In smxAware, EB is shown in the Text displays under Diagnostics and by the Error button on the Graph display.

event buffer (EVB)

Space for EVB is allocated by EVB size in the linker command file. Event logging is enabled by SMX_CFG_EVB in xcfg.h. Logging of errors must also be enabled by the SMX_EVB_EN_ERR in xevb.h. The information logged in EVB is the same as EB, with the addition of the precise time of occurrence. The advantage of storing error records in EVB is that they are in order relative to other events. Also, errors must be logged if EVB is regularly uploaded to check for anomalies that might indicate malware is present. In smxAware, errors show up as red dots in timelines.

error manager hook

smx_EMHook() is called from smx_EM() to allow user code to be included in central error management. smx_EMHook() is located in smxmain.c. As shipped, it serves just to allow setting a breakpoint in the error manager.

This is the place where error-specific handling, based upon the errnum and handle parameters, should be introduced. Since this code runs under smx_EM(), it is non-preemptible as is smx_EM(). As a consequence, it is task-safe and LSR-safe. However, managing errors from low-priority tasks can block high-priority tasks from running and thus miss their deadlines. To avoid this, it is recommended that error information be loaded into a message sent to a pass exchange where a task waits to do error analysis and reporting. The message priority would determine the processing task’s priority. Thus most messages might be processed at a low priority level, but some might merit high-priority processing.

error manager exit hook

smx_EMExitHook() is called from smx_EM() if an irrecoverable error has occurred. What to do requires some consideration. It would seem best to shut the system down cleanly, then to reboot. This way, external damage should be minimized. However if the system is seriously damaged, attempting to do this may result in even more damage and the best option may be to immediately shut it down.

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. shut the system down and reboot.

One way to shut the system down is to call:

```
smx_TaskStartNew(smx_Idle, SMXE_OK, PRI_SYS, (FUN_PTR)aexit);
```

This reverses the way the system was initialized by running the idle task at the highest priority level with ainit() as its code. In this case idle is run at the highest priority with aexit() as its code. aexit attempts to reverse what ainit() did and to shut the system down cleanly. It then calls sb_Exit(), which is intended to reboot the system.

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 an error or to a timeout. This permits point-of-call error handling to be implemented, as in the following example:

```
XCB_PTR port_in;

void  taskA_Main(u32 par)
{
    u32  err;
    u8*  mbp;
    MCB_PTR  msg;
    u32 tmo_ctr;

    while (1)
    {
        if (msg = smx_MsgReceive (port_in, &mbp, TMO))
        {
            ProcessMsg(mbp);
            smx_MsgRel(msg, 0))
        }
        else
        {
            err = smx_TaskPeek(smx_ct, SMX_PK_ERROR)
            if (err == SMXE_TMO)
                tmo_ctr++;
            else
                break;
                }
        }
        /* handle errror */
}
```

In this example, the while (1) loop 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 control goes to the else statement. If a timeout (SMXE_TMO) has occurred, tmo_ctr is incremented and control goes back to message receive. Otherwise control breaks out of the while (1) loop and goes code that handles the error.

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))
        /* fill net_msg and send it out */
else
        sb_MsgOutConst(SB_MSG_ERR, “Send failed because no free network messages”);
```

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 which to use

If properly used, many system paths will flow through smx services, 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. Point-of-call error management adds complexity to the main code and makes it larger and slower. However, some problems are best dealt with at the point of call.

Once debugging is done, most errors should never occur. Timeouts might be the only frequent occurrences to deal with. 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.

smx_EMHook() allows introducing error-specific code instead of point-of-call code.

Chapter 24 Resource Management

access conflicts between tasks

smx provides several methods to prevent task access conflicts to a resources. It is useful to bring them all together in one place in order to pick the best one for a particular case:

(1) same priority tasks
(2) semaphores
(3) mutexes
(4) exchanges
(5) task locking
(6) server tasks
(7) server LSRs
(8) data hiding

same priority tasks

The simplest and lowest overhead method of avoiding resource conflicts is to give all tasks that access a particular resource the same priority and require that no task suspend or stop itself while using the resource. See Chapter 14 Tasks for more information.

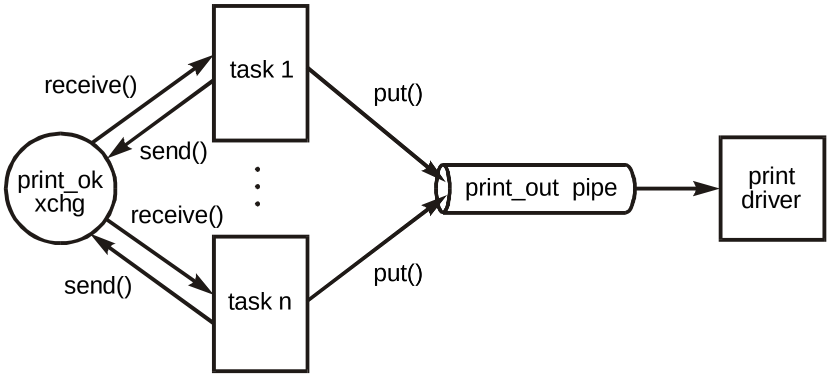
semaphores

Multiple resource semaphores are useful for controlling access to multiple resources of the same type, such as blocks in a block pool. A task can wait at the semaphore for a resource, if none are available. Use of the binary resource semaphore is discouraged – a mutex is usually better. See Chapter 8 Semaphores for more discussion.

mutexes

Mutexes are the preferred method for mutual exclusion. They can be tested multiple times by the same task without causing the task to hang. Only the task that owns the mutex can release it. And mutexes provide priority promotion and ceiling protocols to minimize priority inversion for higher priority tasks. See Chapter 9 Mutexes 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 then 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. In this case, information from the token message might be passed to the print driver so it could handle the different printer. See Chapter 13 Exchange Messaging for more discussion.

task locking

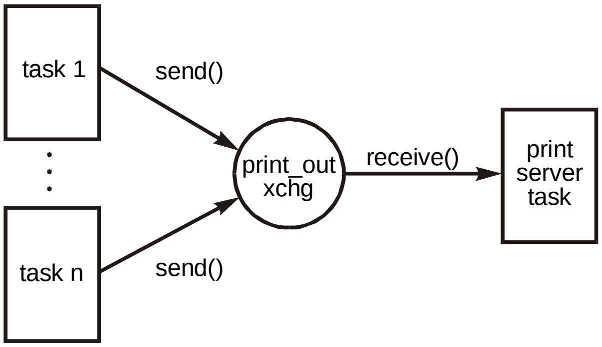
is a simple technique for protecting a critical section of code or using a resource:

```
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 blocks 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 resources that are 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 Chapter 14 Tasks for more on task locking.

server tasks



In this approach, a single server task is assigned to a resource. To access the resource, a client task prepares a message and sends it to an exchange, where the server task waits. Typically, the exchange is a pass exchange so the server runs at the message priority assigned by the client. 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:

```
#define  ACK  1
PCB_PTR       pxmsgs    /* print message pool */
XCB_PTR       px;       /* printer pass exchange */
XCB_PTR       rx;       /* reply exchange */

void  clientA_main(u32 par)
{
    MCB_PTR  msg;
    u8*  mp;
    u32  next_hour;

    msg = smx_MsgGet (pxmsgs, &mp, 0);
    LoadReportA(mp);
    smx_MsgSend (rpt, px, PRI_HI, NULL);
    next_hour = (smx_StimeGet()/3600 + 1) * 3600;
    smx_TaskSleepStop(next_hour);
}
void  clientB_main(u32 par)
{
        MCB_PTR  msg;
        u8* mp;
        msg = smx_MsgGet (pxmsgs, &mp, 0);
        do
        {
                LoadReportB(mp);
                smx_MsgSend(msg, px, PRI_NORM, rx);
                msg = smx_MsgReceive(rx, &mp, 100))
        }  while (msg && *mp == ACK)
}
void  print_rpt_main(u32 par)          /* server task */
{
        MCB_PTR msg;
        u8* mp;
        while (msg = smx_MsgReceive(px, &mp, 100))
        {
                PrintMsg(mp);
                reply = (XCB_PTR)smx_MsgPeek(msg, SMX_PK_REPLY);
                if (reply)
                {
                        *mp = ACK;
                        smx_MsgSend(msg, rx, 0, NULL);
                }
                else
                {
                        smx_MsgRel(msg, 0);
                }
        }
}
```

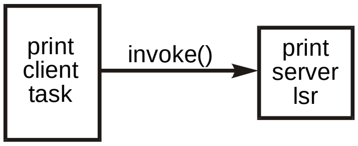
Once an hour, clientA wakes up. It gets a message from the pxmsgs pool, loads report A into it, and sends it to the px exchange, with high priority and no reply expected. It then sleep stops for an hour. During this time, it consumes no stack. When the print server receives the message, it prints it, then releases it back to the pxmsgs pool.

clientB gets one message from the pxmsgs pool, loads report B into it. It gives normal priority to the message, requests a reply to be sent to exchange rx, and sends the message to px.. It then waits at rx for a reply. The print server receives the message and prints it. The server then gets the reply exchange from the message, loads ACK into the message, and sends it to rx. ClientB resumes when a reply is received. If the reply is ACK, client B sends the next report. Note that the same message is being recycled.

As shown in this example, clientB sends a steady stream of messages to the printer. When clientA sends a message, its message will print ahead of a clientB message due to its higher priority.

Server tasks are a good way to manage resources such as printers. Since the server task is the only task that runs the code to operate the resource, the code need not be reentrant. The exchange serves as a priority-based work queue. The client tasks need not be concerned with the details of handling the resource; thus changing a resource can be handled merely by activating a different server task.

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.):

```
LCB_PTR  printLSR;
PCB_PTR  free_msgs;

void  print_client_main(u32 par)
{
    u32  next_hour;
    MCB_PTR  report;
    u8*  mp;

    report = smx_MsgGet(msg_pool, &mp, 0);
    LoadReport(mp)
    smx_LSRInvoke(printLSR, (u32)report);
    next_hour = (smx_SysStimeGet()/3600 + 1) * 3600;
    smx_TaskSleepStop(next_hour)
}

void  printLSR_main(u32 m)   /* server LSR */
{
    u8* mp;
    MCB_PTR msg = (MCB_PTR)m;

    mp = (u8*)smx_MsgPeek(msg, SMX_PK_BP);
    PrintReport(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 an LSR invoked by the ISR begins running.

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 another 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.

data hiding

Data hiding is a good way to avoid access conflicts. It can be accomplished by putting data into messages or pipes rather than into global buffers. Doing so forces sequential access to data by tasks because each task must receive a message or packet before it can access the data it contains. However, be careful to do clean handoffs. For example, 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 preempt other LSRs.

foreground/background access conflicts

are best handled by making sure that common data never exists 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.

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. For example, in the LSR:

```
if (smx_SemTest (oktodoit, SMX_TMO_NOWAIT))
        /* do it */
else
        /* invoke self to try again */
```

(4) Disable LSRs in critical sections of tasks using smx_LSRsOff() and reenable them with smx_LSRsOn(), when done. This has less impact than disabling interrupts. While disabled, LSRs will accumulate in the LSR queue. As long as it does not overflow, nothing is lost, but performance might be impacted.

Chapter 25 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 display which can be used during debug via smxAware and during normal operation via smxAware Live. See the smxAware User's Guide for more information.

For shipped systems, EVB should be periodically uploaded and checked for signs of tampering or bugs. In this case the types of events logged should be very selective to avoid EVB overflow.

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 (sa_Print())
(7) user events

sa_Print() adds a record to the event buffer whenever it adds an entry to the print ring buffer for smxAware. Event logging is enabled by setting SMX_CFG_EVB to 1 in xcfg.h. Also, a non-zero size must be assigned to evbsz in the linker command file. If DRAM is available, EVB can be put into it in order to maximize its size, but there may be an impact on performance.

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 EVB. EVB is a cyclic buffer, so the oldest records are overwritten as new records are logged. smxAware uploads the entire EVB, 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 Event Buffer display in smxAware.

Task, SSR, and error events are logged automatically by smx. ISR and user events require that macros be added to the code. See smx_TickISR() in smxmain.c for an example. See the smx_EVB_LOG descriptions in the smx Reference Manual for more information.

selective logging

Generally speaking, it is best to make EVB as large as possible, especially during debugging, so that smxAware can show a long view of timelines. However, this is not always possible. To effectively use smaller EVBs, 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 helpful if ISRs were not of interest, yet were occurring frequently. 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, background noise can be reduced by putting the few SSRs of interest into SG2, for example, and enabling only it.

The selective logging flags for smx_evbn are defined in xevb.h. smx_evbn is set by smx_EVBInit(), which is called by smx_Go(). For example, if the SMX_EVB_EN_TASK flag is set in xevb.h, then it will be set in smx_evbn and all task events will 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 logging SSRs by enabling or disabling the groups they are in.

time stamps

Time stamping of EVB records 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 the smxAware User’s Guide.

The tick timer clock rate (sb_ticktmr_clkhz) and the tick timer counts per tick (sb_ticktmr_cntpt) are defined in the target processor BSP file, e.g. bspm.c. These are used by smxAware and some smxBase functions.

logging user events

User event logging permits logging your own events. They will appear, with system events, in the Error Buffer window and in bars in the Event Timelines graph in smxAware. This is helpful because it allows relating user events to system operations. Also the events are precisely time-stamped, which may be even more important.

Six user event logging macros are provided:

```
smx_EVB_LOG_USER0(h)
...
smx_EVB_LOG_USER6(h, p1, p2, p3, p4, p5, p6)
```

h identifies the record. It will typically be a pseudo handle created by smx_SysPseudoHandleCreate(), or it could be a function, or NULL. Parameters 1 through 6 are values to record and can be used for anything you wish, such as variable or register values. These are saved in the event record and shown in the smxAware displays.

smx_HT_ADD(h, name) can be used to add h to the handle table with an assigned name. smxAware prints this name in the Name column of the text Event Buffer or the Name field of the Details dialog when looking at an event line in a bar in the Event Timelines graph. If unnamed, it shows the value. Instead of a handle, the address of the function could be passed. The value must not match another handle or pseudo handle to avoid confusing smxAware. The same pseudo handle or value can be used for multiple user events that are related, or individual handles or values can be assigned to each.

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 p1, u32 p2)
{
    u32  A;
    smx_EVB_LOG_USER2((void*)funB, p1, p2);
    /* compute A */
    smx_EVB_LOG_USER1((void*)funB, A);
    return A;
}
```

This logs when funB starts and its 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. The text can be saved to a file for later viewing.

Another example for performance data gathering is:

```
static void* test_log = smx_SysPseudoHandleCreate();
smx_HT_ADD(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. In this 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.

Chapter 26 Precise Profiling

introduction

Precise profiling allows determining exactly how long each task is running, and how much processor time is consumed by ISRs, LSRs, and overhead. Idle time is 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.

Profiling is normally not useful during the early stages of debugging and should be turned off. Once a sizeable portion of the system is running, profiling is useful to determine what tasks, ISRs, or LSRs are consuming too much processor time and should therefore be improved. It is also useful to determine how much reserve capacity exists in the system.

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_START_ID()
smx_RTC_TASK_END()
```

These equate to corresponding profiling functions if SMX_CFG_PROFILE in xcfg.h 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. smx must be initialized before these macros are called.

RTC START() and END() macros are positioned as optimally as possible to capture the desired run times. The corresponding functions, e.g. smx_RTC_ISRStart(), are written to minimize overhead, which should be acceptable in most cases.

task profiling

Task profiling is implemented by the above 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 added into the rtc field of the task's TCB, each time the task runs, so it equals the total time the task has run since the start. This code is part of smx and should not be modified.

LSR profiling

The RTC LSR macros are used to record when an LSR starts running and when it 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.

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 profiling

RTC ISR macros are used to record when an ISR starts running and when it stops running. They are built into the smx_ISR_ENTER() and smx_ISR_EXIT() macros and thus need not be included in smx ISR code. All smx ISR differential counts are accumulated in smx_isr_rtc. Counts for non-smx ISRs get included in other RTCs unless the non-smx ISRs are started with smx_RTC_ISRStart() and ended with mx_RTC_ISREnd().

overhead logging

The overhead run time count is calculated rather than being accumulated — it equals whatever is not accounted for in a *profile 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, the *profile frame*, and performing percentage-of-total calculations. Profiling is enabled by setting SMX_CFG_PROFILE in xcfg.h and defining SMX_CFG_RTC_FRAME and SMX_CFG_RTCB_SIZE, also in xcfg.h.

SMX_CFG_RTC_FRAME is the period, or frame during which RTC counts are accumulated, 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.

RTC counts are accumulated in 32-bit RTC counters. The frame size must not be so long that any RTC counters overflow. RTC increments are taken from the tick timer counter. For the above example, the input clock to the tick timer is 6.25 MHz. Assuming that the largest RTC is 10% of the total, then the maximum frame size = 2^32/0.625 MHz = 1717 sec. Frames should normally be a few seconds, for best accuracy and a reasonable update rate.

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. For example if there are 30 tasks and 10 samples, smx_rtcb would require 35*10*4 = 1400 bytes of RAM. This space is dynamically allocated from the heap. Samples are loaded in cyclic fashion such that a new sample overwrites the oldest sample.

smx_ProfileLSR[9] is invoked at the end of each profile frame by smx_KeepTimeLSR. 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 the RTCs are 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 smxmain.c:

```
#if SMX_CFG_PROFILE
u32  smx_rtcb[SMX_CFG_RTCB_SIZE][SMX_CFG_NUM_TASKS + 5];
#endif
```

and in ainit():

```
#if SMX_CFG_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: smx_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, smx_tcbs. If n is a task’s TCB position in smx_tcbs, then its RTC = smx_tcbi[n-3].

easy profile viewing

When the processor is stopped, smxAware uploads the information from smx_rtcb, calculates percentages, and displays them graphically. To see these, select the Graph window, then the Profile button. This shows where processor cycles are being used. Initially, the average of all frames (samples) is shown. Clicking Next Frame allows viewing each sample, starting from the first. This gives an idea of how much variation there is from sample to sample. The more samples, 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. Another possibility is to upload smx_rtcb with EVB for remote monitoring. For this, it might be desirable to increase RTC_FRAME to a large time, such as 1000 (10 seconds), then with RTCB_SIZE = 10, 100 seconds would be covered. These profiles could be examined to spot abnormal behavior.

---

[9] smx_ProfileLSR is the LSR handle. Its code is smx_ProfileLSRMain(). This is also true of the other LSRs mentioned.

coarse profiling

If SMX_CFG_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 the console display. This information could also be saved in a log. (See **logging user events** in Chapter 25 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 a 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-effect errors

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.

runtime limits

Task runtime limits are used to implement time partitioning in SecureSMX. However, runtime limits can be used for other purposes, such as time-slicing or limiting run times for certain tasks. Not all tasks need to have runtime limits. To enable runtime limits, set SMX_CFG_RTLIM to 1, and set SMX_CFG_IDLE_RTLIM to the desired value for idle passes per runtime limit frame. Both are in xcfg.h. See Runtime Limiting in the SecureSMX User’s Guide for more information.

Chapter 27 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 and task timeouts. It also updates smx_etime and smx_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(u32 par)
{
    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.

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 28 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 that.

task 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 mutex. The in_clib mutex can be used for the standard C library.

Whether locking or using a mutex, a good approach is to create macros with similar names to library functions. These macros lock, call the library function, then unlock, or do the equivalent for the mutex. Using these macros insures that the protection mechanism is always invoked. It is recommended to use a mutex per library or even per group of library calls.

Converting non-reentrant C library functions to SSRs is another way to make them reentrant. See **custom SSRs** in Chapter 16 Service Routines for more information on this.

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. A problem with this approach is that complex library functions may call dozens of subroutines, making the approach infeasible

For a function that is non-reentrant because it uses global variables, another approach is to use the task callback function EXIT case to save the global variables and the ENTER case to restore them. See **task callback functions** in Chapter 14 Tasks for more information on this.

The above approaches do not work for ISRs and LSRs. The best thing is to avoid using non-reentrant functions in ISRs and LSRs. Another approach is to use simplified versions of the functions that are reentrant.

server tasks

Sometimes, entire libraries are largely non-reentrant. The best solution in such a case is to create server task which is the only task allowed to use the library. Consider, for example, a graphics library. Several tasks may need to output messages, icons, graphs, etc. to the display. Rather than allowing each task to directly use graphics library services, a single graphics server task is created. A message format is developed that allows specifying all graphics operations needed by the application.

Client tasks, compose and send graphics messages to an exchange where the graphics server task waits for work. Upon receipt, it interprets each message and calls appropriate graphics library services. This solves the non-reentrancy problem of the graphics library and it unloads the client tasks from graphics duties. It also opens the possibility of being able to substitute one graphics package for another by changing the server.

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 to 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.

alternative C library routines

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 sb_MsgOut functions, the smx print ring buffer (see **debug tools & features** in Chapter 22 Debugging) or user event macros (see **logging user events** in Chapter 25 Event Logging).

bcc.c in \XBASE offers C run-time library replacement routines to correct various other problems in standard C library routines. Also the C compiler you are using may offer alternative C run-time library routines, which are better suited to embedded systems. Also see xapi.h for utility and other macros and functions, which may be helpful.

Chapter 29 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

Competition is driving semiconductor manufactures to ever smaller feature sizes, 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. Thus, there is excess capacity that can be turned to improving safety, security, and reliability of embedded systems.

Malicious attacks are also an increasing concern. In addition, embedded software complexity keeps growing, making bug-free software an increasingly elusive goal.

RTOS advantages

An advantage that an RTOS has over application software is the development cost of safety, security, and reliability enhancements is spread over many projects.

Another advantage arises from the uniformity which an RTOS imposes upon an application. Control is forced to flow through a small number of paths. As a consequence, the places to look for errors are more limited and more uniform and there are fewer kinds of errors.

A third advantage is that using a standard RTOS 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 RTOS calls and objects.

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 an RTOS 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 necessary. Creating a framework at the start and flushing it out over time allows considering failure and safety scenarios as the design unfolds.

error prone functions

The following is an example of an error-prone function:

Error prone:

```
BCB_PTR blk;

blk = sb_BlockGet(pool);
...
sb_BlockRel(pool, blk);
```

Less error prone:

```
BCB_PTR blk;

blk = sb_BlockGet(pool);
...
smx_BlockRel(blk);
```

At first glance, the first version of BlockRel() seems preferrable, because it is more general. However, the pool parameter in sb_BlockRel() gives the programmer the opportunity to make the mistake of releasing blk to the wrong pool. In the second example, this mistake is not possible because there is no pool parameter in smx_BlockRel() -- the block is automatically released to the pool that it came from.

At first, it may seem that releasing a block to the wrong pool is not a serious mistake, but it is. As well as one pool shrinking, a worse problem is if blk is smaller than the block size of the pool it was wrongly put into. Then when it is used, overflow is likely to occur because it is too small and this will damage adjacent objects.

This is a good example of how the right RTOS can help to create safer code.

the need for error checking after shipment

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.

transient errors can 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 transient errors, latent bugs, and malware. Rapid detection of errors permits minimizing the damage they cause.

other ways to reduce memory

Rather than removing error checking, if memory is tight, it is better to adjust the following configuration constants,. Setting SMX_CFG_ERROR_MSGS in xcfg.h to 0 results in displaying just error numbers and saves over 1000 bytes of ROM. Setting SB_CON_OUT and SB_CON_IN, in bcfg.h, to 0 saves quite a bit of ROM and RAM. This is appropriate if there is no console, as is likely in shipped systems. Setting SMX_CFG_MACROS in xcfg.h to 0 results in macros being replaced by function calls. SMX_CFG_EB_SIZE and SMX_CFG_EVB_SIZE, in acfg.h, can be reduced. There are several other configuration constants that might also be set to 0 to save memory

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 during debugging. However, when it is time to ship, finite timeouts should be specified wherever possible. Timeouts can keep the system running when unexpected things happen.

tips for reliable RTOS code

There are many coding standards, such as MISRA, intended to achieve reliable code but there are no equivalent RTOS best-usage standards that we know of. The following tips are intended to help remedy this situation. Many of these have been mentioned elsewhere, but it is useful to bring them all together in one place:

(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) Report forbidden states:

```
switch (var)
{
    case 1:
        /* action 1 */
    case 2:
        /* action 2 */
    default:
        /* report error */
}
```

If var is put into a forbidden state, due to noise or a bug, it will be caught and reported. The error should be logged into the event buffer, EVB, so there is a record of it that can be downloaded at a later time. smx provides smx_EVB_LOG_USERn(h, p1, p2, …) macros for this purpose.

(3) If a task is getting too complicated, consider dividing it into two or more tasks, and let smx services help to deal with the complexity. Smaller tasks usually require smaller stacks because subroutine nesting is not as deep. In addition, it might be possible to make one or more of the tasks into one-shot tasks. Hence the increase in memory required may not be much.

(4) Interlocked operations are recommended over open-ended operations – i.e. verify that the operation actually was successful, rather than assuming that it was. For example, wait for an acknowledgement before sending the next message. If not received, resend the previous message. Exchange messaging supports this by providing a reply exchange in each message.

(5) Always check that the values of variables are reasonable before using them, especially inputs from outside of the system. This makes the system more robust against noise and malware.

(6) smx does not clear the block pointers used to access message and smx blocks. This should be done to prevent accidental use of these pointers when the blocks are no longer owned by a task. For example:

```
BCB_PTR blk;
MCB_PTR msg;
u8* mbp, bp;
smx_MsgSend(msg, xchg, PR0, NULL);
mbp = 0;
smx_BlockRel(blk, 0);
bp = 0;
```

(7) Use finite timeouts for all task waits, even for servers waiting for work. When timeouts occur they indicate a problem or a lack of understanding. When no timeout value is obvious, use a default value such as MAX_TMO that you think would never happen, but if it did you would want to know.

(8) Recovery from timeouts must be implemented in order to break deadlocks and to enable the system to recover from missed events.

(9) Avoid preemptions when not necessary. Watch for the case where a task is preempted by a task that must wait for the lower priority task to finish. In this case, control goes back to the lower priority task, which finishes, then the higher priority task runs. For example:

```
void taskA_main(u32)
{
    smx_TaskLock();
    smx_SemSignal(semA):
    smx_MsgReceive(xchgA, &dp, 100);
}

void taskB_main(u32)
{
    smx_SemTest(semA, 100);
    …
}
```

Assuming taskB has higher priority, it could preempt and run when taskA signals semA. Then control would come back to taskA, which might suspend on xchgA. Locking taskA blocks taskB from preempting it and allows taskA to suspend on xchgA before taskB runs. Hence, an unnecessary task switch (A back to B) is avoided. If taskA does not wait on xchgA, smx_MsgReceive() unlocks it anyway, and taskB preempts. Thus little time is lost for taskB.

(10) Short one-shot tasks can be locked when started and kept locked, until done. This reduces task preemptions and can be done without impacting higher priority tasks if the one-shot task run time is nearly the same as task switching time.

(11) Start with a small number of priorities, and add more only as necessary. This reduces preemptions since same-priority tasks cannot preempt each other.

(12) When initializing structures, set every field to its proper initial state even if the field is not used in the current design, to avoid future errors.

(13) When disabling interrupts, it is recommended to save the interrupt state, then restore it later. For example:

```
CPU_FL ps;

ps = sb_IntStateSaveDisable();
/* perform operation with interrupts disabled */
sb_IntStateRestore();
```

Assuming you know the current interrupt state can have dangerous consequences if it changes or you are wrong. Better to be safe.

(14) 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.

(15) 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..

(16) Minimize comments because they invariably get out of step with the code and are thus worse than useless. Let the code speak for itself and write it so it speaks clearly.

(17) Change symbol names as their functions change. Appropriate names improve understanding. Short names are more memorable than long names.

(18) 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, bug-free code.

(19) Desk checking finds more bugs per hour than any other form of testing. However, programmers don’t hand write code nor print it, anymore. So, stepping through code with the debugger is the best way to find bugs.

(20) 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.

(21) Distinguish between true constants and initialized variables. The const keyword should be applied to true constants and is useful to detect programming errors. It also results in the constants being put into ROM.

(22) Use the volatile keyword for peripheral registers and any other global variables which may change due to an interrupt or preemption. Otherwise, the compiler will not reload the variable each time it is accessed.

(23) 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.

(24) It is better to allocate buffers from a heap than to define them statically (e.g. buf[100]). This is because a common problem with buffers is overflow. If allocated from a heap, block overflow will damage a chunk control block. But, this is likely to be caught and fixed before damage occurs, if smx_HeapScan() is being run periodically. In the second case, nearby global variables will be damaged leading to unpredictable behavior.

(25) If a task is locked, be aware that any smx service that might suspend or stop the task will break the lock regardless of whether the task is actually suspended or stopped. The only way to avoid this is to specify SMX_TMO_NOWAIT as the timeout.

(26) Deleting objects is hazardous. smx releases tasks and messages waiting at an object being deleted. It also loads smx_nullcb into the object handle so it will not be reused. However great care is still necessary to avoid problems. It is generally best to delete an object only from the lowest priority task that uses it and never from LSRs. Careful consideration is necessary to determine if deleting the object will cause other parts of the system to malfunction.

(27) See also **design techniques** in Chapter 21 Coding and **tips on writing ISRs and LSRs** in Chapter 16 Service Routines.

high security

SecureSMX is an extension to smx that allows dividing an application into isolated partitions, then imposing strong limits on what each partition can do, in case it gets hacked. See the SecureSMX User’s Guide for more information.

Chapter 30 Other Topics

This chapter contains discussion of additional smx features that may prove helpful for your project.

multiwait

Sometimes there is a need for a task to wait upon more than one event simultaneously.

One way to do this is to use one-shot tasks. One-shot tasks are effective for multiwaits since each task can wait on an object, such as a semaphore, without a stack. Thus the memory overhead for each one-shot task is just its TCB, which is about 100 bytes. Only one stack is required for all of these tasks, because only one task can run at a time. When an event occurs, the waiting one-shot task is started.

If the main task is started, it can determine what happened from the parameter passed to it. Or the one-shot tasks could send messages to an exchange where the main task waits. Another possibility is that each one-shot task sets a unique flag in an event group and the main task waits on the OR, AND, or AND/OR of those flags. The one-shot tasks might do some preprocessing to simplify main task code and to permit greater flexibility by interchanging one-shot tasks.

Another multiwait method employs object callback functions. The following example shows how a single task can wait on two event semaphores, simultaneously:

```
void tmw_main(u32 par);
void fmw(void);

SCB_PTR  sem1;
SCB_PTR  sem2;
TCB_PTR  tmw;

void mw_init(void)
{
    /* create sem1 and sem2 */
    sem1 = smx_SemCreate(SMX_SEM_EVENT, 1, "sem1");
    sem2 = smx_SemCreate(SMX_SEM_EVENT, 1, "sem2");

    /* create and start tmw */
    tmw = smx_TaskCreate(tmw_main, TP2, TS_SSZ, 0, "tmw");
    smx_TaskStart(tmw);
}
```

```
void tmw_main(u32 par)
{
    /* set fmw callback function for sem1 and sem2 */
    smx_SemSet(sem1, SMX_ST_CBFUN, (u32)fmw);
    smx_SemSet(sem2, SMX_ST_CBFUN, (u32)fmw);

    while (smx_TaskSuspend(smx_ct, SMX_TMO_DEF))
    {
        if (smx_SemTest(sem1, SMX_TMO_NOWAIT))
            ProcessEvent(1);
        if (smx_SemTest(sem2, SMX_TMO_NOWAIT))
            ProcessEvent(2);
    }
}

void fmw(void)
{
    smx_TaskResume(tmw);
}
```

In this example, mw_init() creates sem1 and sem2 and creates and starts tmw. tmw_main sets the fmw() callback function in both sem1 and sem2, then suspends itself to wait for an event. It is assumed that event1 causes sem1 to be signaled and that event2 causes sem2 to be signaled. fmw() is called whenever either semaphore is signaled. It resumes tmw, which tests each semaphore and processes the corresponding event if the test passes. Note that one, both, or neither test may pass.

Multiwaits may be performed for any combination of:

- Semaphore signals.
- Event flag sets.
- Pipe wait puts.
- Message sends.
- Task suspends or resumes.

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(). smxAware uses the handle table for names of ISRs and other unnamed objects, such as user objects. Most smx objects have names in their control blocks that are used by smxAware.

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, first create pseudo handles using smx_SysPseudoHandleCreate(). The size of the handle table is controlled by SMX_CFG_HT_SIZE in acfg.h.

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 will branch to the scheduler, which could switch to other tasks, suspending the foreign ISR with the current task. (smx uses smx_srnest and in some cases a processor feature (such as ARM-M RETTOBASE) to ensure the branch to the scheduler is only done by the outermost ISR, to avoid this problem. A foreign outermost ISR will not do this.)

One way to avoid this problem is to assign higher priorities to foreign ISRs so they cannot be interrupted by smx ISRs. If this is undesirable, it is necessary to encapsulate the foreign ISR using interrupt vector N, as follows:

```
ISR_PTR old_ISR();

void  appl_init(void)
{
      ...
      ps = sb_IntStateSaveDisable();
      old_ISR = sb_IntVectGet(N, 0);
      sb_IntVectSet(N, new_ISR);
      sb_IntStateRestore();
      ...
}

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. Now, when interrupt N occurs, new_ISR() runs instead of old_ISR() and new_ISR() executes smx_ISR_ENTER(), calls old_ISR(), then executes smx_ISR_EXIT(). Now, the foreign ISR will behave like an smx ISR with respect to nesting.

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.

porting smx to other processors

smx supports various 32-bit processors. See the smx Porting Guide for directions to port to another processor architecture. To port to a new processor of a supported architecture, see the section for that architecture in the SMX Target Guide.

Index

- abbreviated names 8
- access conflicts 164, 219
  - foreground 224
  - foreground/background 224
  - LSR/task 224
  - peripherals 224
- application
  - benefits of dividing 190
  - objects 11
- asynchronous functions 189
- atomic 199
- background 155, 158
- backward link (bl) 17
- base block pools 28
- block migration 195
- block pools
  - base 28
  - smx 28, 29, 30, 32, 33, 35
- breakpoints 207
- broadcasting 196
- buffers 27
- changing task priority 128
- checkout 240
- client/server 195
- code for task 19
- code readability 243
- coding 195
- coldstart 167
- configuration 203
- configuration constants 8
- console 207
- const 244
- constants 243
- control block 11
  - exchange (XCB) 106
  - ready queue (RQCB) 129
  - task (TCB) 17
- control block pools 12
- control blocks 5, 12
- control variables 241
- coprocessor
  - context saving 139
- ct 132
- current task 131
- cyclic operations 175
- DARs 5, 27
- data hiding 224
- date/time 196
- deadlock 202
- debug 205
  - features 205
  - smx objects 207
  - smx variables 206
  - tips 207
  - tools 205
- delays
  - accurate 175
- delete operator 28
- design guidelines 189
- design techniques 195
- development
  - evolutionary 194
  - framework 191
  - skeletal 239
  - skeletal example 191
  - summary 194
- development method 191
- dormant 20
- elapsed time (etime) 132, 167, 168
- error
  - detection 213
  - reporting 214
  - system 213
- error buffer 215
- error checking
  - after checkout 240
  - minimal RAM 241
- error handling 6, 216
  - local 216
- error management 213
- error manager 205, 214
  - exit hook 215
  - hook 215
- error number 214
- error-prone techniques 240
- errors 208
  - rapid detection of 240
  - transient 240
- etime 167, 168
- event buffer 215, 227

Index

event flags 81
  setting 86
event group 81
  AND 82
  AND/OR 82
  AND/OR testing 87
  API 81
  atomic operation 89
  creating and deleting 83
  inverse flags 86
  naming flags 82
  OR 82
  other services 88
  practical usage 89
  pros and cons 90
  setting and clearing flags 85
  setting flags 86
  state machine example 90
  summary 92
  terminology 81
  testing flags 83
event logging 205, 227
  selective 228
  timestamps 228
  user 229
event queue 93
  accurate timing 96
  considerations 96
  count and signal 94
  create and delete 94
  diagram 95
  enqueue task 95
  signaling 95
  summary 96
  tick rate 96
evolutionary development 194
exchange 220
  advantages of 105
  example diagram 220
  for resource management 220
exit/enter routines 137
floating point
  context saving 139
flow diagram 187
flowcharts 187
foreground 155, 158
forward link (fl) 17
fun field 17
functions
  error-prone 240
gathering 196
gating 196
global variables 20, 197, 224
handle
  invalid 199
handle pointer parameters 7
handle table 206, 246
handles 13
hardware timer 172
heap
  allocation algorithm 41
  checking 206
  chunk information 46
  compiler 38
  macros 38
  search algorithm 41
  translation functions 38
  using 37
  vs. block pools 37
hook routines 137
hp parameters 7
idle task 128
interrupt handling 156
interrupt service routine (ISR) 153
interrupts 196, 243
  disable 225
intertask communication 196
intertask communication (itc) 63
ISR 153, 224
  encapsulating 247
  example 161
  foreign 247
  nesting 164
  operation 156
  tips on writing 162, 244
  types 155
  writing 157
languages 8
libraries 237
  alternative functions 238
  making SSRs 237
  os dependency 238
  server tasks 237
link service routine (LSR) 154
link service routines 158
lq 158
LSR 154, 224
  access conflicts 163
  diagram 158
  example 161
  how it works 163
  interrupts 163
  purposes 158
  server 224
  server 223
  server diagram 223
  smx calls from 158
  smx_ct 160
  tips on writing 162
LSR queue (lq) 158, 162
malware 239
MCB

  - diagram 108
  - onr field 108
- memory 27
  - management 27
- memory management
  - summary 35
- message
  - API 107
  - broadcasting 119
  - client/server example 117
  - get 108
  - make/unmake 112
  - multicasting 121
  - other services 124
  - owner 116
  - peek 115
  - proxy 121, 196
  - receive 110
  - release 107
  - reply 117
  - send 110
  - summary 124
- message exchange
  - API 106
  - broadcast 107
  - control block 106
  - diagram 110
  - modes 106
  - normal 106
  - pass 221
  - pass 106
- message pools
  - smx 35
- messaging 5, 105
  - exchange 105
  - kinds 105
- multicasting 196
- multiwait 245
- mutex 75
  - clear 78
  - create 76
  - delete 76
  - example 79
  - free 78
  - get 77
  - impact on other functions 78
  - nesting 76
  - release 78
  - resource management with 219
  - vs. semaphores 75
- mutual exclusion 75
- names 208
- naming objects 14, 204
- new operator 28
- non-reentrant functions 237
- objects 11
  - accessing directly 203
  - application 11
  - naming 204
  - naming 14
  - system 11
- one-shot task 143, 145
  - examples 150
  - examples 146
  - for I/O 150
  - RAM savings 144
  - writing 146
- onr field 30
- operation interlocks 242
- performance 6
- periodic operations 196
- peripherals 189
- permanent stack 55
- pipe 97
  - buffer 103
  - message queue 100
  - multiple waiting tasks 103
  - operation 97
  - packet size 103
  - restrictions 103
  - safe operation 103
  - uses 97
- pipes 195
- pointers 11
- polling 196
- pool
  - block 27
- porting smx 247
- power management 235
  - power down 235
  - profiling 236
  - sb_PowerDown() 236
  - tick recovery 235
  - user control 236
- preemption 16, 242
- print ring buffer 238
- priority
  - ceiling 76, 77
  - inheritance 76
  - task 127
- priority inversion 76, 201
- priority promotion 76
- processor porting 247
- profile monitoring 233
- profile viewing 233
- profiler 231
- profiling 205, 231
  - accuracy 232
  - coarse 234
  - control 232
  - edge effects 234
  - frame 232

  - ISRs 232
  - LSRs 232
  - overhead 232
  - precise 231
  - RTC macros 231
  - samples 232
  - tasks 231
- Protosystem 9
- Quick Start manual 3
- ready queue (rq) 129
  - diagram 130
- reentrancy 237
- reliability 239
- reliable code
  - techniques for 241
- reply 117
- resource management 219
- return values 216
- round-robin 16
- round-robin scheduling 128
- rq
  - priority
    - level 129
  - task enqueue 130
- runtime limiting 234
- safety 6, 239
- sb_ services 9
- sb_DARInit 27
- scheduler 130, 164
  - hook routines 138
- scheduling 15, 16
- security 239
- semaphore 224
  - binary event 69
  - binary resource 66
  - counting 66
  - diagram 68
  - event 68
  - gate 71
  - multiple resource 66
  - nested tests 75
  - other services 72
  - resource 65
  - resource management with 219
  - summary 73
  - threshold 70
  - uses for 65
- semaphores 65
- server LSR 223, 224
- service routine 153
  - interaction 158
  - interrupt (ISR) 153, 224
  - link (LSR) 154, 224
  - operation 164
  - rules 154
  - system (SSR) 153
- shared stack 56
- smx block pools 28
  - creating and deleting 29
  - getting and releasing blocks 30
  - making and unmaking blocks 32
  - message 35
  - peeking 33
- smx message pools 35
- smx objects 13
  - looking at 207
- smx services 5
- smx variables
  - important 206
- smx_ct 132
- smx_DelayMsec() 96
- smx_EB 207, 215
- smx_EM 214
- smx_EMExitHook() 215
- smx_EMHook() 215
- smx_errctrs 214
- smx_errno 214
- smx_EVB 207, 215, 227
- smx_evbn 228
- smx_EventFlagsTest() 84
- smx_EventQueueCount() 94, 95
- smx_EventQueueCountStop() 95
- smx_EventQueueCreate() 94
- smx_EventQueueSignal() 95
- smx_Idle 128
- smx_MsgBump() 124
- smx_MsgReceive() 110
- smx_MsgReceiveStop() 124
- smx_MsgRel() 109
- smx_MsgRelAll() 109
- smx_MsgXchgClear() 124
- smx_MsgXchgCreate() 106
- smx_ProfileLSR 232
- smx_rq 207
- smx_rqtop 130
- smx_SSR_ENTER() 153, 164
- smx_SSR_EXIT() 153, 164
- smx_TaskBump() 16, 128
- smx_TaskCreate() 19, 20
- smx_TaskDelete() 132
- smx_TaskResume() 132
- smx_TaskSetStackCheck() 59
- smx_TaskSleep() 172
- smx_TaskSleepStop() 172
- smx_TaskStart() 20, 131
- smx_TaskStartNew() 131
- smx_TaskStartPar() 23
- smx_TaskStop() 131
- smx_TaskSuspend() 132
- smx_TaskYield() 16
- smx_TickISRHook() 206
- smx_TimeoutLSR 168

- smx_TimerStart() 176, 177, 181, 182, 183
- smx_TimerStop() 178, 180
- SMX_TMO_INF 132
- SMX_TMO_NOCHG 132
- SMX_TMO_NOWAIT 132
- smx++ 28
- smxAware 60, 205
- smxBase 9
- smxBSP 9
- sp 17
- SSR 153
  - custom 153
  - custom 164
  - from LSR 158
  - groups 228
  - IDs 228
  - interrupts 163
  - limited 160
- stack 55
  - bottom 56
  - checking 205
  - control 56
  - creating and filling 58
  - diagram 56
  - fields in TCB 56
  - foreign 59
  - high water mark 58
  - high-water mark 57
  - out of stacks 61
  - overflow 213
  - overflow detection 58
  - overflow handling 59
  - pads 57, 208
  - permanent 55
  - pool 132
  - RAM usage 143
  - scanning 59, 60
  - scanning, permanent stack 60
  - scanning, shared stack 60
  - shared 56
  - stack pool 56
  - system 55
  - system stack fill and scan 61
  - top 56
  - usage 60
- stack pool size 144
- stack size 203
  - setting 58
- STACK_PAD_SIZE 57
- state machines 195
- states
  - forbidden 241
- stepping 208
- stime 167
- stop SSRs 63
- structure
  - system 5
- structure of system 187, 190
- structures 243
- subsystem 187
- subsystems 187
- symbol names 243
- system
  - function 187
  - structure 187, 190
- system errors 213
- system objects 11, 13
- system service routine (SSR) 153
- system stack 55, 156
  - fill and scan 61
- system structure 5
- system structure guidelines 189
- system time (stime) 167
- task 127
  - API 127
  - arrays 200
  - context
    - extending 137
  - controlling 15
  - creating 19
  - current (ct) 132
  - current (smx_ct) 131
  - deadlock 202
  - delete 132
  - diagram 17
  - dispatching 164
  - dormant 131
  - example 20, 21
  - extending context 137
  - flags 128
  - infinite loop 198
  - lock 133, 135, 221
  - lock counter 133
  - lock nesting 134
  - lock uses 135
  - locked 131
  - main function 19, 140
  - many small 197
  - multiwait 245
  - one-shot 143, 144, 145, 146, 150, 151
  - operations 131
  - preemption 135
  - priority 127
  - priority ceiling 76, 77
  - priority change 128
  - priority inheritance 76
  - priority inversion 201
  - priority inversion 76
  - priority promotion 76
  - resume 132
  - scheduling 15
  - server 221, 224

  server diagram 221
  stack is assigned 17
  stack size 203
  start 131
  starting 20
  state 18, 20
    wait 18, 168, 241
  state transitions 18
  stop 131
  suspend 132
  switching 135
  timeout 168
  top 130, 164
  types 15
  unlock 133
  unlocked 164
task resume location 206
task suspend location 206
tasks 15, 17
  how many 189
  same priority 219
  skeletal 193
TCB 17
tcb.err 214
tick rate 167
time delay
  options 171
time measurement 205
timelines 228
timeout 167, 172, 241
  accurate 169
  for safety 168
  for timing 169
  handling 170
  in milliseconds 169
  priority 170
  task 168
timeout parameter 63
timeouts 131, 208, 242
timer 171, 175
  absolute start 177
  API 175
  duplicate 179
  example 178
  frequency modulation (FM) 183
  hardware 172
  LSR control 180
  peek 183
  precise 184
  pulse 181
  pulse period modulation (PPM) 182
  pulse width modulation (PWM) 182
  reset 180
  software vs. hardware 176
  start 176
  stop 178, 180
  vs tasks 183
  why use? 175
timeslicing 16
timing 167
  date-time 172
timing accuracy and precision 6
variables
  limit checking 242
volatile 244
work areas 27
work queue 223, 224