SecureSMX™

User's Guide

Version 5.4.0

July 2025

by Ralph Moore
and
David Moore

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

Table of Contents

Table of Figures

Preface

Adding security to an embedded system increases its complexity and may increase its development time. However, adding isolated partitions with limits has some offsetting advantages such as easier system integration and easier debugging, resulting in:

1. Little or no increase in actual development time.
2. Better products.
3. Reduced future security problems.

SecureSMX can be viewed as a different way to develop microcontroller-based applications. It provides a methodology in which software is modularized, then modules are placed into isolated partitions, and those are moved into unprivileged mode (*umode*). Partitions are allowed to access only a restricted set of system services and tokens are required to access smx objects in order to perform kernel services. Token control is strictly within SecureSMX and cannot be compromised from umode. Communication between partitions is restricted to standardized *portals*. All limitations are hardware-enforced.

SecureSMX provides a toolbox containing many tools to deal with security problems. It is not a one-size-fits-all solution. In fact, it is specifically aimed at not requiring that trusted code be modified. Instead, it is aimed at partitioning untrusted and vulnerable software into isolated partitions and then imposing limitations so that malware in those partitions cannot harm the rest of the system. Whereas designing security in from the start is the best approach, SecureSMX also supports incremental security improvement of existing products.

SecureSMX currently supports MCUs with Memory Protection Units based upon the Cortex-M v7 and v8 architectures. These account for about 80% of all MCUs currently being produced.

Chapter 1 Introduction

1.1 How to Use This Manual

The next three chapters of this manual cover the theory of the SecureSMX methodology. Chapter 4 covers the features of SecureSMX. Chapter 5 presents portals, which you may not need until late in your project. Chapter 6 presents advanced theory, which you may not need at all.

We recommend reading the introductory material, scanning the theoretical material to see what is there, then going on to Chapter 7. Chapter 7 is based upon a series of demos, which can be downloaded from www.smxrtos.com/securesmx, and which are designed to get you going as quickly as possible. You can refer back to the theory chapters when you need more details.

Chapter 8 provides much needed debug help. Appendix A is the API reference for SecureSMX, Appendix B is a complete linker command file, and Appendix C is a glossary of special terms used in this manual.

Although the SecureSMX methodology is not rocket science, it is probably much different than what you are used to. Hence, studying the theory sections, as needed, will avoid misconceptions about how things actually work.

1.2 Partitioning

SecureSMX adds strong security and reliability to the SMX RTOS. It is a next generation RTOS that provides tools and methods to improve the security of systems using Micro Controller Units (MCUs) with Memory Protection Units (MPUs). SecureSMX currently supports ARM Cortex-M MCUs. It facilitates dividing embedded software into isolated partitions through the use of the processor’s memory protection unit (MPU) and its privilege levels.

Figure 1.1 shows typical embedded system software. Everything is jumbled together. If a hacker gains access anywhere, he has access everywhere. Likewise, a bug anywhere in the system jeopardizes the whole system.

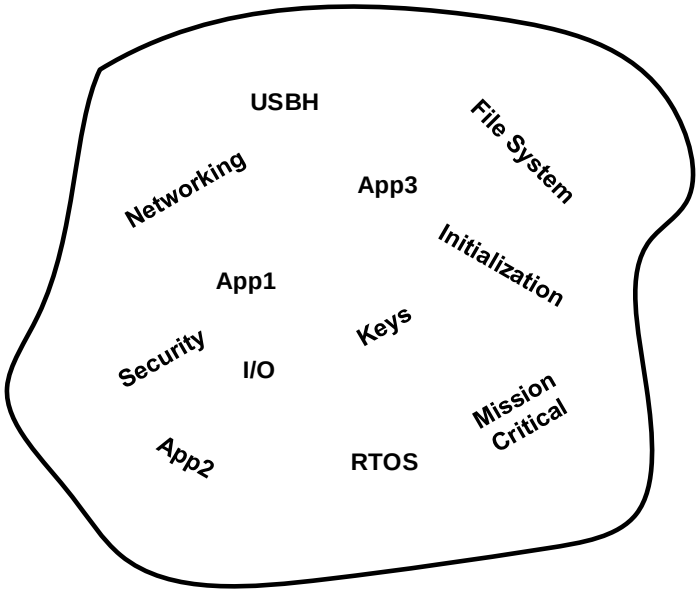

Figure 1.1 Non-Partitioned Software

Figure 1.2 shows the same embedded system software after partitioning.

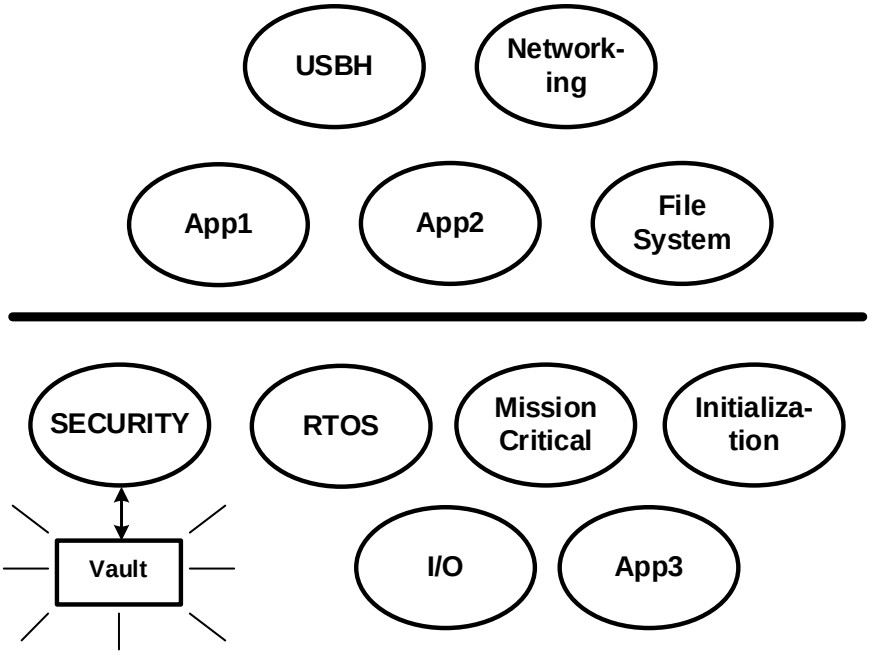

Figure 1.2 Partitioned Software

Because the above partitions are fully isolated from each other, by hardware, if a hacker gains access to one partition, he cannot gain access to other partitions. Hence he can only disable the partition that he has penetrated and he cannot obtain keys and critical information contained in

the Vault and other partitions. Likewise, a bug in one partition can damage only the operation of that partition. In both cases, system monitoring software can be alerted and it can take corrective action as well as notifying the operator or a monitoring center. In the above figure, App3 may be in pmode because it needs direct access to system services or needs maximum performance.

1.3 Advantages of Isolated Partitions

Dividing embedded system software into isolated partitions has the following benefits:

1. Much better security from hackers.
2. Higher reliability and safety.
3. Isolation of low-quality or unknown-quality software.
4. Better plug-in modularity.
5. More disciplined design.
6. Immediate detection of null and wild pointers and stack and buffer overflows.
7. Easier incorporation of legacy software, due to enforced modularity.
8. Partition-only reboot rather than system reboot for recovery.
9. Support for partition-only updates.

Security and Reliability are two sides of the same coin. In the former, hacks are deliberate; in the latter, bugs and malfunctions are accidental. However, both damage system operation, and measures that improve one tend to improve the other. Partitions are often subsystems that perform specific functions, e.g. file systems, networking systems, etc. Hardware enforcement of full isolation enforces modular designs and better module reusability in future systems. Hardware enforcement of better design practices is also desirable. Partial reboots and partial updates save time. These are all good reasons for partitioning.

1.4 Hardware

SecureSMX utilizes the following security features of the Cortex-M ARMM7 and ARMM8 architectures:

1. Memory Protection Unit (MPU).
2. Privileged and Non-privileged processor levels.
3. SVC Exception.

The methodology presented in this manual is the same for both processor architectures. Hence it can be applied to families of products that use both architectures and it permits smooth migration from ARMM7 to ARMM8. The only difference is that the ARMM7 MPU is more difficult to support, however SecureSMX provides methods to overcome its problems. SecureSMX does not require ARMM8 TrustZone in order to provide high security and protection from attacks and

software bugs. For ARMM8, it runs in the *non-secure state*, but it could run in the *secure state*, if preferred.

In the code and in this manual, ARMM7 represents ARMv7-M and ARMM8 represents ARMv8-M.

1.5 Methodology

As evidenced by the breadth and depth of this manual, SecureSMX presents a comprehensive methodology for the design of high-security embedded systems. This new methodology is quite different from doing things the old way. Be prepared to learn some new tricks. The theoretical aspects of the new SecureSMX methodology are explained in chapters 2 through 6.

Full partition isolation requires the following:

1. Limiting code, data, and I/O region access via the MPU.
2. Restricting access to system services via the SVC exception.
3. Dedicated heap for each partition that requires a heap.
4. Portals for communication between partitions.
5. Runtime, service, and object access limitations.

Chapters 7 and 8 cover the design and debug techniques necessary to partition embedded system software. Refer to the Glossary for unfamiliar terms.

1.6 Security

1.6.1 The Increasing Need for Security

Most embedded systems have little or no security built in, yet they are being connected into the Internet of Things (IoT) at a rapid pace. As a consequence, once isolated, defenseless embedded systems are becoming accessible via the Internet. This makes it much easier for hackers to gain access and to do damage, not only to the devices, but also, through them, to entire systems.

1.6.2 Protection Goals

The **primary goal** of protection is to protect trusted, critical software and data from less-trusted, non-critical software, which has become infected with malware or is buggy. Examples of trusted software are: the RTOS kernel, exception handlers, security software (e.g. crypto, authentication, secure boot, and secure update), and mission-critical software. Examples of less-trusted software are: code vulnerable to malware attacks such as: protocol stacks, device drivers, software of unknown pedigree (SOUP), and insufficiently tested new code.

The **secondary goal** is to detect intrusions and bugs and shut them down so that critical system operation is not imperiled, and sensitive data is not stolen. Dealing with intrusions and bugs may be handled by stopping and restarting a penetrated partition or may require stopping and rebooting the entire system.

The **tertiary goal** is to minimize the amount of trusted code that must be written, since it is more difficult to write trusted code. Code that is less trusted can be run in unprivileged mode (umode) partitions, which are strongly isolated from trusted code partitions. This insures that failures or hacking of less-trusted code will not impair the critical function of the system – e.g. to keep a patient alive or to control a dangerous machine.

The degree of protection that must be implemented depends upon the security and safety requirements of a specific system and the threats to which it may be exposed. SecureSMX provides a range of security tools that enable achieving a level of protection appropriate for a given system. And security can be steadily improved, in future releases, as a system becomes more widely distributed and therefore more likely to be attacked. SecureSMX is structured to foster progressive security improvement.

1.6.3 What You Need

SecureSMX is not a complete security solution. You also need:

1. Secure boot.
2. Secure update.
3. System monitoring.

SMX middleware products do provide crypto and authentication software.

1.7 SecureSMX Snapshot

The main things SecureSMX does to achieve better security and reliability in a multitasking system are as follows:

1. Allow defining different MPU regions for each task.
2. Perform MPU region switching during task switches.
3. Automatically generate regions from linker command files.
4. Provide a Supervisor Call (SVC) API to allow unprivileged code (ucode) to call system services, as well as to limit which services can be called from ucode[1].
5. Allow allocation of protected blocks and messages.
6. Run mission critical code trusted code in privileged mode (pmode) tasks and partitions.
7. Run middleware and application code in unprivileged mode (umode) tasks and partitions.
8. Run the SMX RTOS kernel APIs, LSRs, ISRs, scheduler, error manager, and other system services in hmode.

[1] See Appendix D: SMX API Limitations.

9. Provide protection for both ptasks and utasks.
10. Individually protect task stacks.
11. Provide portals for communication and operations between fully isolated partitions.
12. Provide runtime limiting.
13. Control system object accesses via tokens.
14. Allow tasks to have privilege levels.
15. Provide a method to move most ISR code into a umode partition.

SecureSMX has had these principal design goals:

1. To enable developers to achieve high security for their systems.
2. To support incremental security improvement for existing systems as well as to provide a security base for new systems.
3. To provide a flexible solution that permits achieving the right level of security for a given system.

1.8 SecureSMX Licensing

SecureSMX is made available under the Apache License, Version 2.0. It is protected by several U.S. patents, which are listed in smx.h, and patents pending. A patent license is granted according to the Apache License for the use of SecureSMX in OEM products but not to remove and integrate code with another OS, RTOS, or kernel. This is only a summary of intent; please see the actual license terms in license.txt and apache2.txt in the release and the comment block at the top of each source file.

Support and contracting services are available from Micro Digital at support@smxrtos.com.

Chapter 2 Background

2.1 MMUs vs. MPUs

Memory Management Units (MMUs) are used with full Operating Systems (OSs), such as Linux and Windows to provide *isolated virtual memories* for *processes*. Processes are independently compiled and linked and then individually loaded and run by the OSs. Process to process isolation is good enough that if a process becomes infected by malware or starts malfunctioning, for any reason, the OS can usually shut it down with little or no damage to other processes nor to the system, itself. Hence, security is good, with regard to infected processes. The use of MMUs is well-studied and well-understood. A downside to using MMUs is that they typically require high-performance processors and very large memories.

High-performance, power-hungry, expensive processors and very large memories are not compatible with the requirements for most embedded systems. In addition, full OSs do not have the response times needed for many real-time applications. Most embedded systems use low-cost, low-power, moderate-performance Micro Controller Units (MCUs) controlled by Real Time Operating Systems (RTOSs). Embedded systems usually have meager memories and processors compared to full OS systems. For security, many MCUs offer Memory Protection Units (MPUs); these are faster than MMUs but not capable of creating virtual address spaces. Instead, they allow dividing memory into isolated regions within a single address space.

In most embedded systems, we deal with *partitions* instead of *processes*. The idea is basically the same – partitions include one or more tasks and perform specific functions for a system. Just like processes, it is desirable to isolate partitions from each other so that if one partition is infected with malware or begins malfunctioning, it can be stopped with minimal damage to the rest of the system. Unfortunately, this is not as easy to accomplish with MPUs as it is with MMUs.

The biggest challenge is that all MCU software is compiled and linked into a single executable that runs in a *single address space*. Also, MPUs impose limitations of their own. The use of MPUs for security is not well-studied, nor well-understood, and there are many complex tradeoffs involved, especially due to the above limitations of embedded systems and the need for deterministic real-time performance.

An advantage of MPUs over MMUs is that switching from one partition or process to another is faster. In MMU systems it may be difficult to meet response time requirements. Such systems are usually referred to as *soft real-time systems* as opposed to *hard real-time systems* implemented with MCUs.

2.2 Cortex Micro Controller Units (MCUs)

2.2.1 Cortex-M

The Cortex-M processor architecture, which includes ARMM7 and ARMM8, offers the following security features:

1. Privileged and Unprivileged levels of processor operation.
2. Supervisor Call (SVC) Instruction.
3. Memory Protection Unit (MPU).

The first is implemented via the three modes of processor operation:

1. Handler Mode: Privileged mode for ISRs, fault handlers, the SVC handler, and the PendSV handler. This mode can be entered only via an interrupt or an exception. We refer to handler mode and any non-task, privileged code that uses the main stack as being in *hmode*.
2. Privileged Thread Mode: Privileged tasks (ptasks) run in this mode. It can be entered only from handler mode, by setting CONTROL.nPRIV = 0. We refer to this as *pmode*.
3. Unprivileged Thread Mode: Unprivileged tasks (utasks) run in this mode. It can be entered from either of the above two modes, by setting CONTROL.nPRIV = 1. We refer this as *umode*.

The p prefix can be interpreted either as *privileged* or *protected* and the u prefix can be interpreted as either *unprivileged* or *user*. Code and tasks that run in pmode are called *pcode* and *ptasks*; code and tasks that run in umode are called *ucode* and *utasks*.

An important aspect of the Cortex-M architecture is that pmode can be entered only via an interrupt or an exception. The SVC N instruction causes an exception. It can be executed from umode with an 8-bit argument, N. This allows making a system call from umode where N specifies the function to call. The called function executes in pmode and returns its result to the umode caller transparently. The SVC N instruction can also execute from pmode, which is helpful for migrating a partition from pmode to umode.

The MPU provides N *slots* for N *regions*. Each region has a starting address, a size, and access parameters, such as Read-Only (RO), Read/Write (RW), eXecute Never (XN), etc. If a memory access is not permitted by a region in the MPU, a Memory Manage Fault (MMF) is generated. The MMF is an exception that causes the MMF Handler to run. It normally stops or deletes the faulting task and initiates recovery.

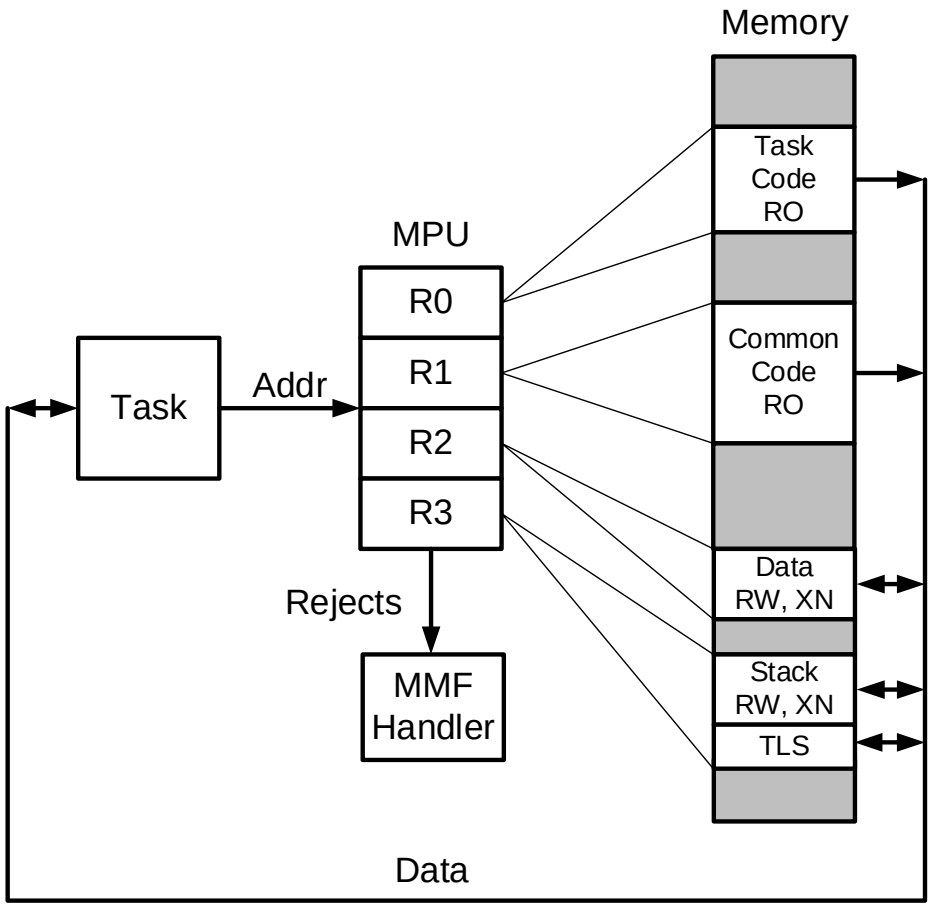

Figure 2.1 MPU Operation

Figure 2.1 shows a simplified MPU with only 4 slots. Most MPUs have 8 slots, some have 16. The region stored in R0 is a Read Only (RO) region for task code. R1 is an RO region for code that is common with other tasks. R2 is a Read/Write (RW) and eXecute Never (XN) region for data. R3 is the task stack, which is also RW and XN. Also shown also is optional Task Local Storage (TLS) that can be part of the task stack region and which can be used for local task data, such as a buffer.

Probably, the biggest drawback of the Cortex-M MPU is insufficient slots. Nearly all Cortex-M processors have 8-slot MPUs. This would seem to be enough, but actually 10 or 12 would be better. SecureSMX provides several methods to deal with this problem

2.2.2 Cortex-M ARMM7

The ARM Cortex-M ARMM7 processor architecture was introduced in 2005 and was intended for medium-size embedded systems. Since then, thousands of different Cortex-M ARMM7 based Micro Controller Units (MCUs) have been developed by the semiconductor industry; they are used in tens of thousands of products developed by device manufacturers; and billions of chips have been shipped to date. It is by far the most dominant MCU architecture (70% market share) and hence the one we have supported first.

A serious limitation of the ARMM7 MPU is that region sizes must be powers-of-two and regions must be aligned on their size boundaries. This tends to result in substantial memory waste, which is probably why this MPU has been very unpopular and seldom used in embedded systems. However, we have found several methods to deal with this problem and to reduce memory waste to acceptable levels, thus opening up this MPU to serious use for security improvement.

Each region is divided into 8 subregions and each subregion can be individually disabled. This permits overlapping the unused portion of one region with another region and is one method used by SecureSMX to reduce memory waste.

2.2.3 Cortex-M ARMM8

The Cortex-M ARMM8 architecture was announced several years ago. It requires that region sizes and alignments only be multiples of 32 bytes. This effectively eliminates the memory waste problem of the ARMM7 architecture. However, the number of MPU slots is unchanged, and a new problem has been introduced: region overlap is not permitted. This is a serious and unnecessary break with the ARMM7 architecture. It creates an obstacle to porting software from one to the other and causes increased complexity, as is discussed in later sections.

The Cortex-M ARMM8 architecture also offers a new feature called *TrustZone*®, which permits secure and non-secure states. TrustZone secure state may good for storage of keys and other private data. However, running code in secure state may not be worth the extra code complexity and debug difficulty. In addition, we believe that most device manufactures will want a security solution that works with both ARMM7 and ARMM8 processors. Therefore SecureSMX does not require nor support TrustZone.

Chapter 3 Getting Started

SecureSMX supports both upgrading legacy code security and providing a secure foundation for new code. We start with legacy code.

3.1 Legacy Code

Often, in an existing system, there are one or more portions of the code that present security problems. It might be, for example, a networking stack that was recently added to the system or a third-party software package that is of unknown quality.

The first step is to get the entire system running in pmode with the MPU on. This process is described, in detail, in section 7.2 Creating an Isolated Umode Partition Demo. Briefly, it consists of using default regions that permit access to all memory and I/O for all tasks. The MPU is not actually doing anything at this point, but the entire application has been put into what we call the *main* partition and it will continue to run normally.

The next step is to define a partition for the problem code, its data, and I/O. This partition must have at least one task. Then define code, data, and I/O regions for the partition that are limited to the partition. These regions are loaded into the MPU whenever a task in the new partition runs. In addition, by including the xapiu.h file, all system service calls are routed through the SVC exception API. Now the MPU will prevent these tasks from accessing code, data, and I/O in the main partition, thus protecting it. The final step is to move the partition into umode, thus putting it into a secure *sandbox*.

Of course, the actual process it a bit more complicated. It is detailed in Chapter 7 Partition Demos pd0 to pd4 and accompanied with demos for each step. “pd” means “partition demo”. However, the process is not overly difficult, as is shown by the demos. The main partition is largely unchanged and runs as it always has. Putting vulnerable code into an isolated partition also requires very little code change nor understanding of it.

In order for the new partition and the main partition to communicate and call services in each other, it may be necessary to implement a *portal* between them. See Chapter 5 Partition Portals. SecureSMX enables this to be done with little change to the code in either partition.

Once the above baseline has been achieved, it may be desirable to divide the main partition into smaller partitions and possibly move some of these into umode. These changes will improve security, safety, and reliability of the system. SecureSMX fosters doing security changes in an incremental manner and it also fosters migrating security changes into the next generation system. SecureSMX has been designed so that the same techniques are generally applicable when migrating from a ARMM7 processor to a ARMM8 processor. Thus it is possible to develop a single security plan covering both architectures for both old and new systems.

3.2 New Code

For new code, all partitions should be identified from the outset as well as determining which mode they are to run in. Although the approach can be taken of getting new code running in pmode then converting it to umode, we recommend writing the code for the intended mode from the beginning. This is no harder and saves many steps. Of course, there may be some legacy code, which may need to be moved to umode.

With new code, there is much more freedom to partition the code in a manner that makes sense for security, safety, and reliability. Generally, the approach is to define partitions along functional lines – i.e. each partition performs a subfunction of the system. Usually a single task is defined per partition. This task may become the parent task for child tasks that perform functions for the parent.

Due to limitations such as the SVC API for system services and the need for portals, utasks run more slowly than ptasks. Hence, mission-critical, high performance code may be left in pmode. Also security code, as illustrated in Figure 1.2, is left in pmode. Middleware, stacks, device drivers, and vulnerable code should run in umode, where it can be best isolated from other partitions. In cases where the inherent process is slow (e.g. serial communication, file IO, etc.) the overhead of umode is often negligible compared to the operation time.

The next step is to identify portals. SecureSMX supports *tunnel* portals for large data transfers (e.g. file I/O) and *free message* portals for commands and small data transfers. As part of defining portals, some partitions are identified as *servers* and others as *clients*. Portals can cross the umode/pmode boundary (see Figure 1.2) in either direction, so that is not a concern. They utilize *protected messages, pmsgs*, which are regions in themselves.

Not previously mentioned are heaps. A heap cannot be shared between isolated partitions. Hence, each partition needing a heap must have its own heap. SecureSMX includes *eheap*, which is designed for embedded systems and which supports multiple heaps in a simple manner. eheap can be configured to support a large range of heaps from simple and small to large and complex.

3.3 References

The following references should be on hand when reading this manual and when converting to SecureSMX (1 – 5 are Micro Digital manuals):

1. smx Reference Manual.
2. smx User’s Guide.
3. smxBase User’s Guide.
4. smxAware User’s Guide.
5. Middleware manuals, as needed.
6. The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Memory Protection Unit chapter, by Joseph Yiu, Elsevier Inc, 2014.
7. ARMv7-M Architecture Reference Manual, ARM Ltd. or
   ARMv8-M Architecture Reference Manual, ARM Ltd.

8. ARM Platform Security Architecture Overview, ARM Ltd. 2017, for security term definitions.
9. IAR EWARM Manuals.

Chapter 4 Basic Theory

4.1 Partitions and Tasks

4.1.1 What are Partitions?

Unlike specific code and data objects, which reside at specific locations in memory, a partition is an abstract object consisting of the union all MPU regions that the partition's tasks are allowed to access. These are normally contained in a single *partition template* and are assigned to the partition's tasks as needed. The reason for this is the 8-slot limitation of most MPUs. A partition may need more than 8 regions. This can be handled by defining tasks that each perform a portion of the partition's job and thus can operate within 8 regions.

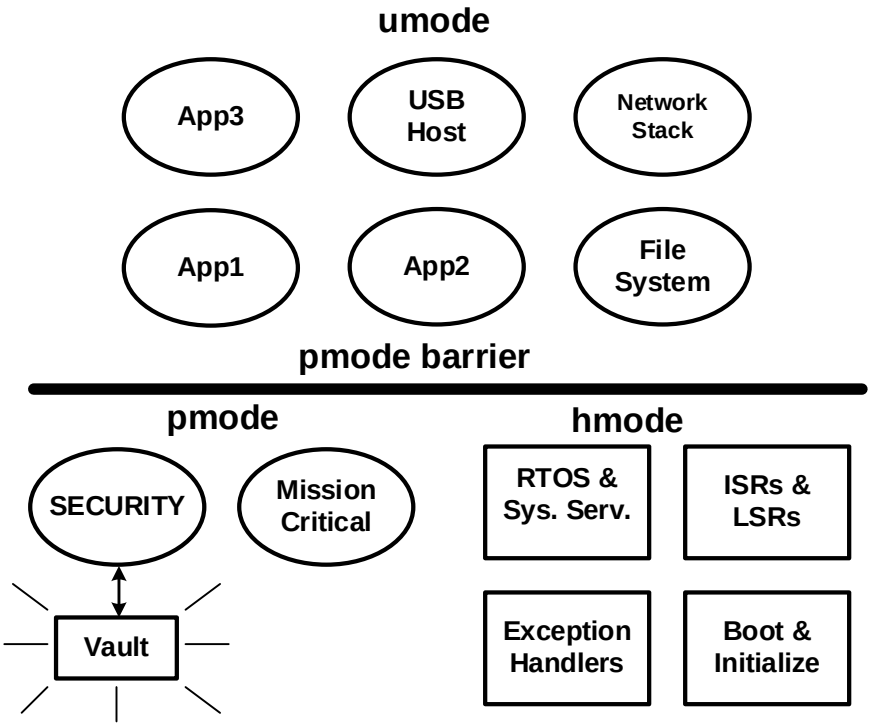

Figure 4.1 Partitions

Figure 4.1 illustrates the software structure we are trying to achieve for security. In this diagram, ovals represent isolated partitions, and rectangles represent just code. The partitions above the heavy line run in umode, and the partitions below the heavy line run in pmode. The pcode runs in hmode. The heavy line represents the boundary between unprivileged operation and privileged operation and is called the *pmode barrier*. This isolation is enforced by the Cortex-M processor.

Above the heavy line are three application partitions and three middleware partition. The goal is to achieve isolation of each partition from all others. Then breaking into a partition does not enable a hacker to access other partitions and thus the breach is contained. Each umode partition contains one or more utasks. The utasks form the basis of isolation from tasks in other partitions, but not from tasks in their own partition. umode partitions are capable of strong isolation. Hence, vulnerable code such as drivers, middleware, and application code should be put into umode partitions.

Below the heavy line are the Mission Critical and Security partitions. The first is likely to consist of multiple tasks which perform the main function of the system, such as controlling a machine or acquiring data. This code is probably field-proven, tightly written code that nobody wants to change. By running in pmode, it is strongly protected from umode code by the pmode barrier and requires minimal change. It is similar for the security partition. Isolation between pmode partitions is not as strong as that between umode partitions[2]. Hence pmode partitions should contain only trusted code.

The hmode code runs using the main stack and does not utilize tasks. Secure Boot runs in a task-free environment following power up or system reset. RTOS & System Services includes smx and other needed system services. Exception handlers run in hmode. Unfortunately, so do ISRs and LSRs.

[2] It is expected that malware cannot breach pmode. If it does, it can access anything in the system by simply disabling the MPU.

4.1.2 Secure Boot

When the system powers up or is reset, the processor comes up in hmode, and it is in the Secure Boot code.

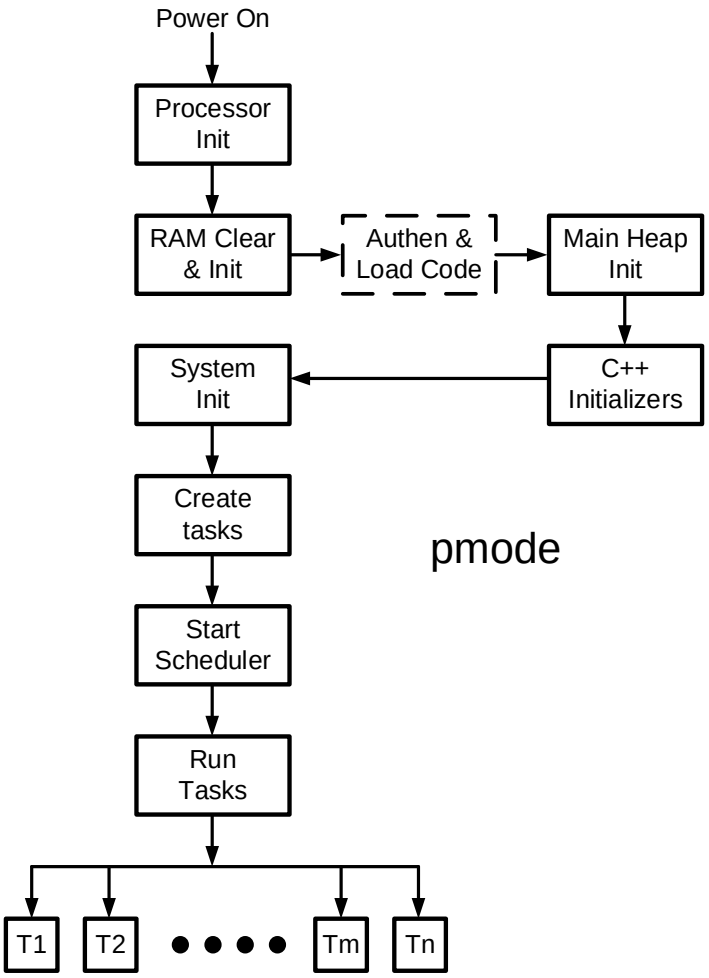

Figure 4.2 Secure Boot and Startup

As illustrated in Figure 4.2, secure boot software does basic hardware and software initialization, and it loads code, if necessary. SMX startup code takes over at this point. It initializes the smx kernel and heaps, creates the tasks necessary to start operation, and then starts the scheduler. Prior to starting the scheduler no tasks are running. After starting the scheduler, the system is running only tasks. Partitions do their own initializations. Both for structural and security reasons, it is best to minimize the secure boot code. Secure boot loaders are available from many sources; SecureSMX does not include a secure boot loader.

4.1.3 RTOS & System Services

RTOS & System Services contains the SMX RTOS kernel and system services, such as error management and error recovery code. This code runs in hmode when called from utasks and in pmode when called from ptasks.

4.1.4 The Vault

The Vault is where we store encryption keys, passwords, authentication codes, certificates, etc. If pmode is breached, the Vault springs open and the Kingdom is lost. Therefore, protecting the Vault is of paramount importance, and thus only the security partition, which contains crypto, authentication, and other security code, is allowed to access the Vault.

4.1.5 Mission Critical

The mission critical partition(s) contains application ptasks which perform the main functions of the system. As explained in the legacy code section, this code has probably been used in prior systems and represents many years of development and testing. Typically this code is not the security problem and thus requires very little modification – mostly getting it running with the MPU.

Unfortunately, ISRs and LSRs, which are part of the mission critical code, run in hmode using the main stack. If hacked, the hacker can easily turn off the MPU and access whatever he wants. If the ISRs and LSRs have been carefully written and employ minimal code, they may not be a problem. See section 4.8.3 Interrupts for more discussion.

4.1.6 utasks

utasks provide most secure isolation. This is primarily because they cannot access the MPU nor change the CPU privilege level. The MPU is loaded with the regions that a utask is allowed to access, including access permissions (e.g. read-only, execute never, etc.) for each region. This is done by the smx scheduler when the utask is dispatched. Generally speaking, it is preferrable for utasks to interface to the outside world since they are more strongly isolated than ptasks.

4.1.7 ptasks

As with utasks, the MPU is loaded with regions that the ptask is allowed to access when it runs. Unfortunately, security provided by ptasks is weak compared to utasks. This is because if a ptask is breached, only one instruction is required for malware to either turn off the MPU or to turn on its Background Region (BR). Unfortunately, the MPU is defenseless in pmode.

However, ptasks may help to thwart an attack by catching certain hacking techniques (e.g. stack or buffer overflow, attempted execution from a stack or buffer, etc.) and thus trigger an MMF before a hacker gains control. The MMF handler can then stop or delete the ptask under attack, then restart it — hopefully with only a small hiccup in system operation.

ptasks are just as capable as utasks to catch bugs and to prevent them from causing harm. Hence, ptasks are helpful for increasing system safety and reliability.

4.1.8 Parent and Child Tasks

A hacker could cause a great deal of trouble if he could create, delete, start, and stop tasks from a umode partition that he had penetrated. Hence, it would appear that task services should not be permitted in umode, only in pmode.

Unfortunately, this does not always work well, especially if converting legacy code. To require that all tasks be created during pmode initialization can result in significant restructuring of software, which we want to avoid. Embedded systems are dynamic, and in many situations tasks

must be created as they are needed in order to deal with real world events as they occur. For example, tasks may be created as USB devices are plugged in, and the tasks may be deleted when the USB devices are unplugged. As another example, some USB controllers can be switched between host and device modes, thus requiring one USB stack to be disabled and the other to be enabled. In order to minimize the use of resources, this may be implemented by deleting one set of tasks and creating another.

Normally, a partition has one main task, which is created in pmode and which initially runs in pmode to perform certain partition initializations, including spawning child tasks. The task then switches itself into umode, where it starts its child utasks and possibly creates and starts others. Figure 4.3 illustrates a *task family*. Note that child tasks can create other child tasks, thus becoming parents of those children. This provides the flexibility needed for dynamic system configuration and control.

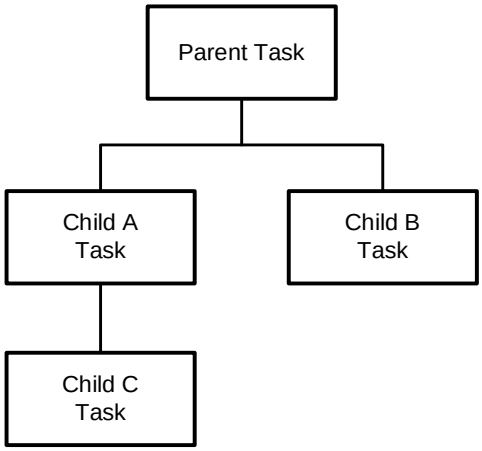

Figure 4.3 Parent and Child Tasks

A parent utask can create, start, stop, delete and perform other functions on its child tasks. It cannot perform these functions on its own parent nor its siblings and their children. In addition to this, a child inherits its parent's mode, MPA template, and limitations. In addition, its priority cannot be higher than its parent's priority. Basically, a child can do only what its parent can do. As will be discussed later, the ability to create child tasks is helpful to overcome MPU slot limitations. Parent, child, and sibling tasks are discussed further in section 4.3 MPA Templates. Also see Appendix D: SMX API Limitations.

The rules governing parent and child tasks are as follows:

1. A ptask creates a utask by specifying the SMX_FL_UMODE flag when it creates the utask. However, it cannot create a child utask, because the child utask could then access whatever the parent ptask could access.
2. A ptask creates a child ptask by specifying the SMX_FL_CHILD flag when it creates the ptask. Child ptasks are useful when developing a pmode partition that will become a umode partition. They also are useful for reducing MPU regions required per ptask.
3. A ptask creates a normal ptask if neither flag is specified when it creates the ptask – i.e. it is neither a parent nor a child.

4. Specifying both SMX_FL_UMODE and SMX_FL_CHILD causes an SMXE_INV_PAR error.
5. A child ptask can only create another child ptask, and it becomes the child's parent. In this case, the SMX_FL_CHILD flag need not be specified.
6. A task created by a utask is always a child utask, and the creating task becomes its parent task. The SMX_FL_UMODE and SMX_FL_CHILD flags are ignored, in this case.
7. In the case where a parent task creates a child task, which in turn, creates another child task, the first parent task is referred to as the *top parent task*, and subsequent parent tasks are referred to as *parent tasks*. Top parent tasks are important for runtime limiting and tokens.

4.2 MPU Control

4.2.1 Memory Protection Arrays and Tasks

As shown in Figure 4.4, smx has a Task Table consisting of a task control block (TCB) for each task that has been created. This table is not in a fixed order, but rather in the order in which tasks were created.

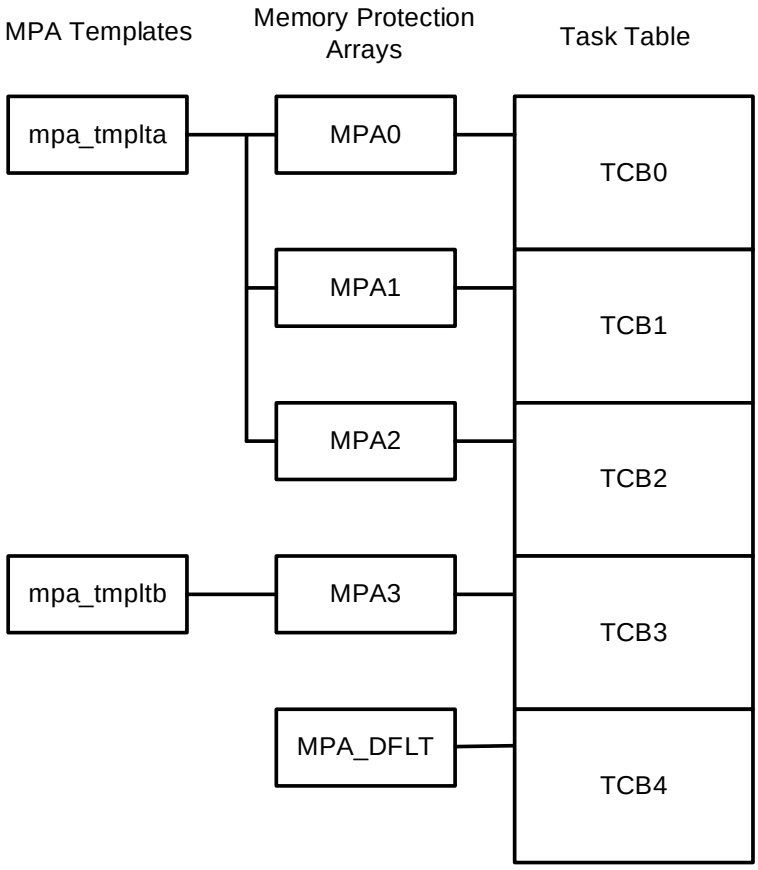

Figure 4.4 MPA Templates, Memory Protection Arrays, and Tasks

Each task also has its own *Memory Protection Array (MPA)*, which is loaded into the MPU when the task starts to run. Hence, each task has its own regions in the MPU when it is running. This applies to both ptasks and utasks.

Each MPA is an array of structures. For ARMM7, each element of this array is a structure consisting of two 32-bit fields named *rbar* and *rasr* that are exact copies of the MPU RBAR and RASR registers in each MPU slot, except that the valid bit is set in rbar, but not in RBAR. For ARMM8, the fields are named *rbar* and *rlar* and are exact copies of the MPU RBAR and RLAR registers in each MPU slot.

If MP_MPA_DEV is set, an additional *name* field is present in the MPA structure. This name field allows assigning a unique name to every region (e.g. “taskA_code”), which is helpful during debugging. The name appears in the smxAware MPU and MPA displays and in the debugger watch window when looking at MPA slots.

As shown in Figure 4.4, *MPA Templates* determine the contents of the MPAs. A template may be shared between MPAs, as shown for MPA0, 1, and 2. This would be the case for tasks within the same partition. In this case, the tasks are likely to have some unique regions and to share other common regions. Alternatively, a task may have its own template, as shown for TCB3. In this case, the task has its own code and data regions.

When a task is first created, it is assigned a *default MPA*, as shown for TCB4. For the debug version, the default MPA usually consists of *super regions* that permit access to all memory in use. This way a task can be debugged without being concerned about the exact regions it needs. For the release version of an application, the default MPA normally contains all NULL regions. Thus if a task has not been assigned an MPA, it cannot run.

Task creation usually is implemented as in the following example:

```
ut2a = smx_TaskCreate(SR04_ut2a, TP2, TS_SSZ, SMX_FL_UMODE, "ut2a");
mp_MPACreate (ut2a, &mpa_tmplt_ut2a, tmsk, msz);
```

This creates task ut2a with main code SR04_ut2a(), priority TP2, permanent stack of TS_TSSZ bytes, named “ut2a”. The SMX_FL_UMODE flag causes the umode flag to be set, which causes the task scheduler to dispatch it as a utask. If the umode task flag is not set, the task is dispatched as a ptask.

mp_MPACreate() allocates an MPA of msz regions from the main heap and loads regions from the mpa_tmplt_ut2a into it. tmsk is a bit mask which selects the regions from mpa_tmplt_ut2a. This allows a single partition template to be used for tasks within the partition that require different regions. Construction of templates and template masks is discussed in the next section.

4.2.2 MPU / MPA Relationship

The Cortex-M architecture allows for MPUs with 8 or 16 slots. 8 slots is too few in many cases and 16 slots is more than adequate. 10 to 12 would be optimal. The vast majority of Cortex-M processors with MPUs have 8-slot MPUs. Hence, there often is a problem with too few slots for a particular partition. The relationship between the MPU and an MPA, as shown in Figure 4.5, helps to deal with this problem.

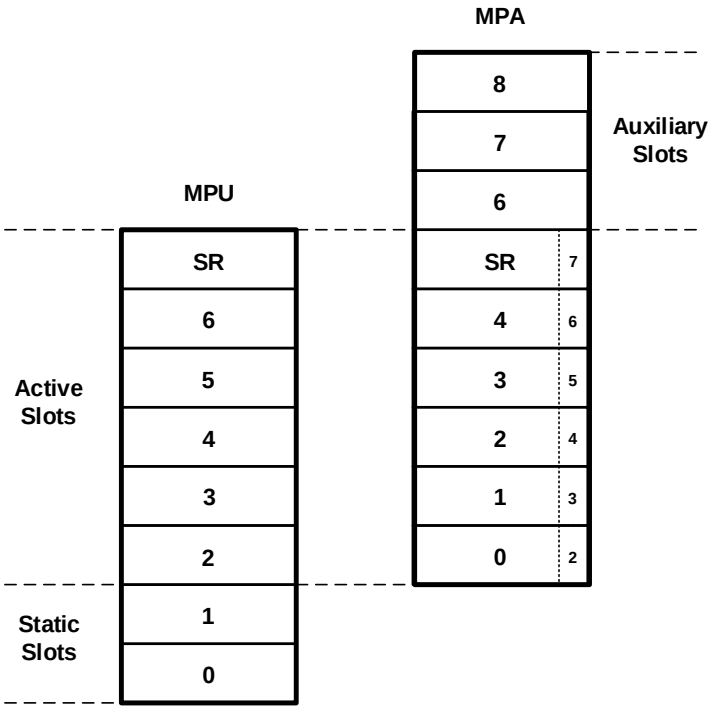

Figure 4.5 MPU / MPA Alignment

Figure 4.5 shows an 8-slot MPU and a 9-slot MPA. The active slots of the MPU are loaded from the active slots of a task's MPA when the task starts running. Task MPAs can vary in size, but each one must be large enough to hold the regions for the active MPU slots. In the above example, less than all MPU slots are used for active slots. This would be unusual for an 8-slot MPU, but not for a 16-slot MPU. The MPU static slots are loaded one time during initialization and are not changed due to task switches. The MPA auxiliary slots serve to increase the effective number of MPU slots, but are not loaded directly into the MPU. Slot types are discussed in the sections that follow.

4.2.3 Active Slots

For most systems, all 8 MPU slots are needed for active slots, and there are no static slots. Whenever an active MPU slot is changed, the corresponding MPA slot is also changed. Thus, when a task switch occurs, it is not necessary to save the MPU slots. It is only necessary to load the MPU active area from the new task's MPA active area. This is done automatically by the smx scheduler. Note that reducing the number of active slots will reduce task switching time, but this cannot be done on a task basis because all tasks in a system must have the same number of active slots.

The top active slot is designated "SR" for Stack Region. For ARMM7, this slot always holds the *task stack region*; for ARMM8 it may not. When a task with a *permanent stack* of ssz bytes is created by:

```
smx_TaskCreate(fun, pri, ssz, flags + heapn, name);
```

a block of sufficient size to hold the stack and to meet MPU region size and region alignment requirements is allocated from heapn. Task region information is temporarily stored in the task's TCB. This information is transferred to the task's SR slot when its MPA is created.

For utasks, heapn is usually mheap, the main heap. mheap is normally in the sys_data region, which is not accessible by utasks. For ptasks, a different heap that is not in the sys_data region should be used. Otherwise, the SR slot will not be effective to catch accesses outside of the stack block since it is included in the sys_data region. Worse, for ARMM8 an MMF would occur due to overlapping regions. The latter is guarded against by SecureSMX as follows: if the stack is in the sys_data region, its region is not loaded into the SR slot. Instead, this slot can be used for another region. PSPLIM is supported for ARMM8 to catch stack overflows – see section 4.11.7 PSPLIM and MSPLIM for more information. However, this is not as good as a stack region, which also detects underflows and may have different attributes than the sys_data region.

If a task without a permanent stack is created by:

```
smx_TaskCreate(fun, pri, 0, flags, name);
```

when the task is started, a stack block is allocated from the *stack pool* and the region information is generated and loaded into the SR slot of the task's MPA. Blocks in the stack pool must meet MPU region requirements. Whenever the task is stopped, its stack block is returned to the stack pool and the SP slot in its MPA is cleared. See the smx User's Guide for more information on permanent and temporary task stacks.

Since the stack pool is normally in the sys_data region, there is no problem using it for utasks. However, for ptasks, loading the stack region into the SR slot of the MPA is inhibited and this slot can be used for another region. PSPLIM is supported for ARMM8 to detect stack overflows.

For ARMM8, the foregoing is not a perfect solution for either type of task, but it is workable if you are careful to not allocate a task's stack from another region accessible by the task. For example a ptask stack could be allocated from heap1, if heap1 is not in any other region of the ptask.

4.2.4 Static Slots

MPU static slots, if any, are normally loaded one time, during system initialization using:

```
mp_MPUSlotLoad(u8 sn, u32* rp);
```

where sn is the slot number and rp points to the region to load. The following is an example of usage:

```
mp_MPUInit();                                          /* initialize the MPU */
mp_MPUSlotLoad(0, (u32*)&mpa_tmplt_sys[0]);  /* MPU[0] = sys_data */
mp_MPUSlotLoad(1, (u32*)&mpa_tmplt_sys[1]);  /* MPU[1] = sys_code */
```

mp_MPUInit() is called first. It initializes the MPU and loads all slots with NULL regions. Then slots 0 and 1 are loaded with sys_data and sys_code. These are privileged regions necessary for interrupts and exceptions to run and cannot be accessed by utasks.

In the case of a 16-slot MPU there are likely to be more than two static slots. These could be used for common regions such as C-library functions, SVC shell functions, tables, etc. Using

static slots in this manner reduces the number of active slots which must be switched on task switches. However, common regions reduce partition isolation. If they are pure code or fixed tables this is probably not a problem.

4.2.5 Auxiliary Slots

As mentioned, previously, 8-slot MPUs are most common, by far, but unfortunately, 8 slots are often not enough. The auxiliary slots shown in Figure 4.5 are intended to alleviate this problem. A given task's MPA may have one or more auxiliary slots, as shown, or none. Hence, MPA sizes vary from task to task.

Auxiliary slots can be used for two purposes:

1. Auxiliary slots for protected blocks and messages.
2. Expansion slots.

Auxiliary slots are explained in section 4.11.11 Protected Messages.

Expansion slots are primarily of value for I/O regions. Two or more I/O regions may be stored in expansion slots and one active slot shared between them. When necessary to access an I/O device,

```
mp_MPASlotMove(asn, esn)
```

moves its region from expansion slot esn to active slot asn in both the MPU and the task MPA, as illustrated in Figure 4.6. Also shown is a pmsg auxiliary slot.

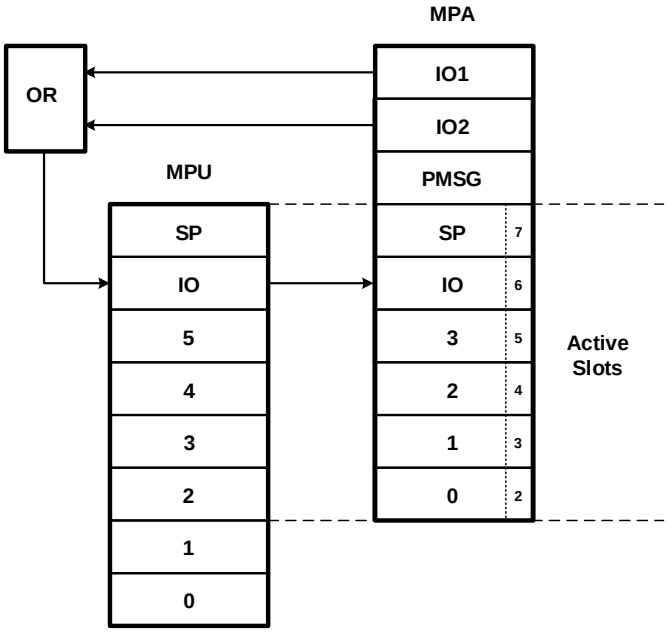

Figure 4.6 Using Expansion Slots

If expansion slots were not used, the single IO slot would need to span from IO1 to IO2. This could include many in-between memory-mapped peripherals that should not be accessible by the current task.

4.2.6 MPU Slot Numbers & Region Overlaps

For ARMM7 MPUs the slot number of each region in the MPA must be the MPU slot number, not the MPA slot number. These are the small numbers in the MPA slots in Figure 4.5. This unfortunate complexity is due to the fact that if two regions overlap, the access permission and attributes of the higher numbered region prevail in the overlap area. Static slots are often privileged, so if they are higher than a task's active slots, the task would not be able to access the overlapped portions of its own regions. This can result in puzzling MMFs — the task regions look good, so what's wrong? To prevent this, active slots are put above static slots. In the above diagram, slots 0 and 1 are static slots.

ARMM8 MPUs do not have slot numbers, nor do they allow region overlaps. If two regions overlap, access to a location in the overlap area causes an MMF. This, too, can be difficult to diagnose. A system could be running fine for hours, days, or weeks, when a rare access is made to a small sliver of overlap and BANG — an MMF brings the system down. This was an unfortunate design decision made by Arm Ltd. smxAware warns about region overlaps in the MPU and MPA displays to help avoid this problem.

4.2.7 Task Stack Slot

For ARMM7, MPU[N] is reserved for task stacks, where N is the highest active MPU slot. This is so that the stack region's attributes cannot be overridden by another region that overlaps it. For example, overriding the XN stack restriction would enable executing code from the stack, which is a common malware tactic. Also, anything that alters the task stack is likely to cause a difficult problem to find.

See section 4.11.6 Task Stacks for more information on task stacks.

4.3 MPA Templates

4.3.1 Creating and Loading MPAs

As shown above in Figure 4.4, the MPAs for tasks are loaded from templates. This usually occurs after task creation by calling:

```
mp_MPACreate(task, tmp, u32 tmsk, mpasz);
```

where tmp is a pointer to the template, tmsk is a bit mask which determines which regions in the template are loaded into the MPA for the task, and mpasz is the size of the MPA, in slots.

Normally templates are associated with partitions, as shown in Figure 4.7. Such templates consist of all regions needed by all tasks in the partition. Regions are selected for a given MPA by tmsk.

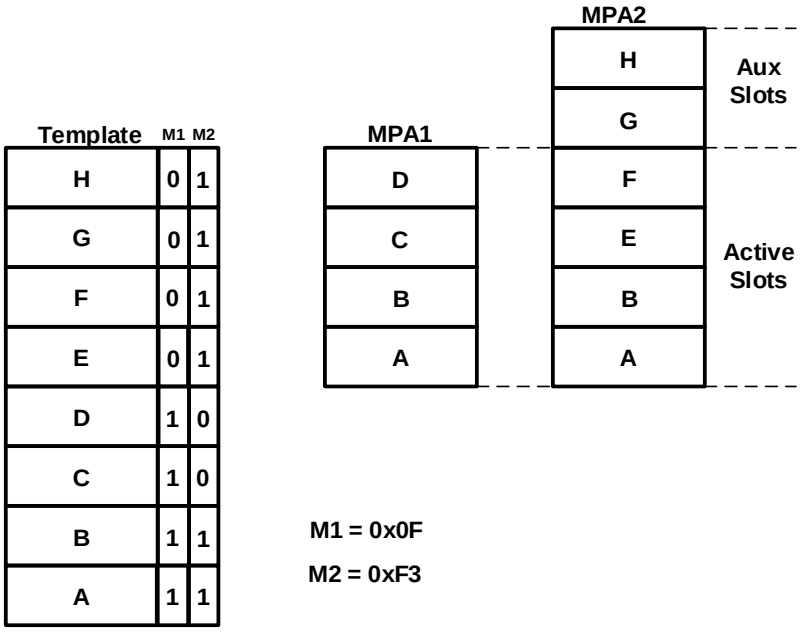

Figure 4.7 Template Loaded into MPAs

In the above figure, the template consists of 8 regions, A through H. M1 represents tmsk for MPA1. Note that it selects regions A through D, which are loaded into MPA1, in order. M2 represents tmsk for MPA2. Note that it selects regions A, B, and E, F, G, H, which are loaded into MPA2, in order. Regions A, B, C, D and A, B, E, F are loaded into MPA *active slots*. Regions G and H are loaded into MPA2 *auxiliary slots*. For ARMM7, each region in the template must have the MPU slot number where it goes. Hence, A, B, C, D would have 2, 3, 4, 5 and E, F would have 4, 5, assuming the active area starts at MPU[2], as shown in Figure 4.6. Regions G and H do not require slot numbers. For ARMM8, regions do not have slot numbers.

If there are NULL regions in an MPA between active regions they must have NULL place holders in the template. For ARMM7, NULL place holders look like:

```
RGN(4 | V, 0, "spare"),  /* reserved for dynamic region */
RGN(5 | V, 0, "stack"),   /* reserved for task stack */
```

For ARMM8 they look like:

```
RGN(4, 0, 0, "spare"),  /* reserved for dynamic region */
RGN(5, 0, 0, "stack"),   /* reserved for task stack */
```

If the defined regions end before the end of the MPA, as determined by mpasz, mp_MPACreate() will automatically place NULL place holders in the remaining slots. Thus they need not appear in the template. Hence, a ARMM8 template could look like:

```
MPA mpa_tmplt_t2a =
{
  RGN(0, RA("sys_data") | DATARW, RLA("sys_data") | AI(0) | EN, "sys_data"),
  RGN(1, RA("sys_code") | CODE,   RLA("sys_code") | AI(0) | EN, "sys_code"),
  RGN(2, RA("t2a_data") | DATARW, RLA("t2a_data") | AI(0) | EN, "t2a_data"),
  RGN(3, RA("t2a_code") | CODE,   RLA("t2a_code") | AI(0) | EN, "t2a_code"),
//  RGN(4, 0, 0, "dynamic"),
//  RGN(5, 0, 0, "spare"),
//  RGN(6, 0, 0, "spare"),
//  RGN(7, 0, 0, "aux"),
};
```

The commented-out lines are present only to document what the slots do – they can be omitted. Commenting them out saves a little bit of space for the template (36 bytes). Note that region numbers are ignored and present only for readability. The code for creating an MPA for the above template is:

```
mp_MPACreate(t2a, &mpa_tmplt_t2a, 0xF, 8);
```

It is very important that tmsk = 0xF have exactly the same number of 1's as the template has regions. If it is too small, regions will be omitted; if it is too large, regions will be included from the next template. For ARMM8 this is likely to cause MMFs due to region overlaps.

4.3.2 Using Parent and Child Tasks

As previously discussed, in section 4.1.8 Parent and Child Tasks, a partition may contain a parent task and its child tasks. The parent task is created in pmode, and its MPA is loaded in pmode from the partition template. As previously noted, it is convenient to create a single template for each partition containing all regions that the partition needs. This provides an overview of the partition. Then the regions are allocated to the parent and its child tasks, with each getting an appropriate subset of regions to do its job.

The child tasks can be viewed as specialists that help the parent task to do the work of the partition. For example, one child task might handle input for the parent and another child task might handle output. This helps to overcome the MPU slot limitation, since the parent task need not access the input and output regions and the child tasks need not access all of the parent's regions.

Child tasks can be created in pmode. But it is more convenient, for the parent task to create the child tasks and load their MPAs in umode, after it has initialized the partition and switched to umode, itself. Switching the parent to umode is done as follows:

```
TCB_PTR smx_ct;

smx_TaskSet(smx_ct, SMX_ST_UMODE, 1);
smx_TaskStart(smx_ct);
```

This sets the parent umode flag, then restarts the parent task so the umode flag takes effect. When the parent MPA is loaded, a pointer to its template is saved in the parent TCB. A child utask must inherit the parent's template. Thus when the parent calls:

```
TCB_PTR child;

mp_MPACreate(child, tmp, tmsk, mpasz);
```

tmp is ignored, and the parent's template is used instead. (For good form, tmp should be NULL.) tmsk is used, as usual, and mpasz determines the size of the child MPA. Note that the child cannot access anything outside of the partition's regions. This is a convenient way to avoid errors that create security problems.

As development of a partition proceeds, it often happens that new regions are needed that exceed available MPU slots. SecureSMX has been designed to facilitate dividing an initial single parent task into parent plus child tasks in order to deal with this, as noted above.

A parent task can create a (non-child) utask in pmode and create its MPA before setting its umode flag, for example:

```
TCB_PTR task;

mp_MPACreate(task, tmp, tmsk, mpasz);
smx_TaskSet(task, SMX_ST_UMODE, 1);
smx_TaskStart(task);
```

In this case, the tmp parameter is required. This is the method normally used to create a umode partition's top task. It can be also used to create independent tasks for the partition instead of creating child tasks. Independent tasks might share a common partition template, or not. Hence, a partition may have a mixture of independent tasks, parent tasks, and child tasks in order to do its job efficiently.

4.3.3 Using ARMM7 MPU Subregions

The ARMM7 MPU region size and alignment requirements can result in large wasted areas of memory between regions, as shown below for A.

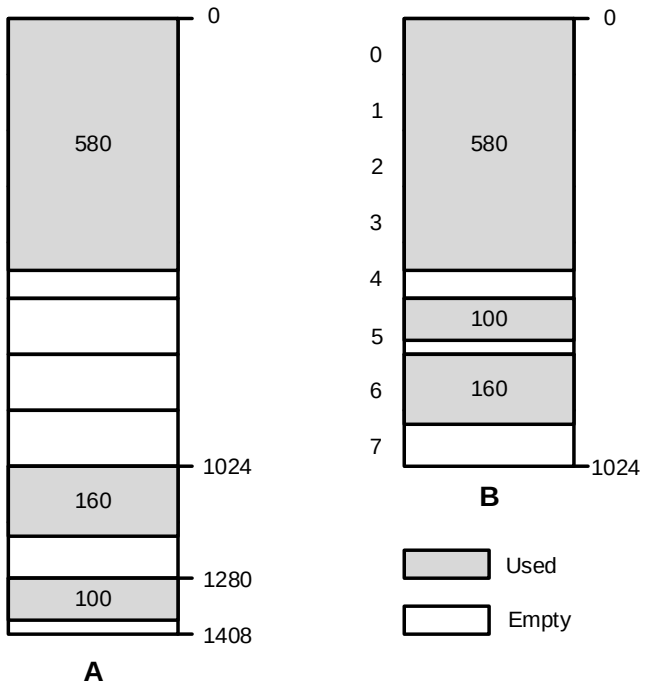

Figure 4.8 Using ARMM7 Subregion Overlays

Figure 4.8A shows allocating memory for three regions requiring 580, 160, and 100 bytes. Each allocated region size has been increased to the next power of two: 1024, 256, and 128, respectively. The result is that 1408 bytes of memory are required whereas only 840 bytes are actually used. Hence, 568 bytes = 39%, is wasted. By comparison, ARMM8 would allocate 608, 160, and 128 bytes = 896 total, resulting in 56 wasted = 6%.

Figure 4.8B shows the use of ARMM7 subregions within the 1024-byte region. There are 8 subregions, each 128-bytes in size. These subregions are numbered on the left side. Subregions 0 to 4 = 5 * 128 = 640 bytes, which is large enough for 580 bytes. So, this region can be defined as starting on a 1024-byte boundary, with a size of 1024 bytes, but with subregions 5, 6, and 7 disabled. The region definition is as follows:

```
RGN(0 | RA("r1_data") | V, DATARW | RSI("r1_data") | N57 | EN, "r1_data"),
```

where N57 = N5 | N6 | N7, which disables subregions 5, 6, and 7.

Now the 100 byte data will fit into subregion 5, which is on a 128-byte boundary, and the 160 byte data will fit into subregions 6 & 7, which is on a 256-byte boundary. Entries in the linker command file would be:

```
define block r1_data  with size = 640,  alignment = 1024  {rw section .r1.data};
define block r2_data  with size = 128,  alignment = 128   {rw section .r2.data};
define block r3_data  with size = 256,  alignment = 256   {rw section .r3.data};
define block sram_block with fixed order {block r1_data, block r2_data, block r3_data,…}
```

The first 3 lines define the blocks, their sizes, their alignments, and the sections within them. The fourth line forces the linker to put the blocks in the order shown in Figure 4.8B. So now, total memory required is 1024 bytes and wasted memory has been reduced to 184 bytes = 18% — not as good as ARMM8, but a big improvement.

Using the linker command file is discussed in detail in section 4.4 Linker Command File.

4.3.4 Creating ARMM7 MPA Templates

Templates are composed of regions. A typical ARMM7 template looks like this:

```
MPA mpa_tmplt_ut2b =
        {
          RGN(0 | RA("ucom_data")  | V, DATARW | SRD("ucom_data") | RSI("ucom_data")  | EN, "ucom_data"),
          RGN(1 | RA("ucom_code") | V, CODE      | SRD("ucom_code") | RSI("ucom_code") | EN, "ucom_code"),
          RGN(2 | RA("ut2b_data")   | V, DATARW | SRD("ut2b_data")    | RSI("ut2b_data")    | EN, "ut2b_data"),
          RGN(3 | RA("ut2b_code")  | V, CODE       | SRD("ut2b_code")   | RSI("ut2b_code")   | EN, "ut2b_code"),
          RGN(4 | V, 0, “spare”),  /* reserved for dynamic region */),
          RGN(5 | V, 0, "spare"),  /* reserved for dynamic region */
          RGN(6 | V, 0, "stack"),  /* reserved for task stack */
        };
```

This is a template for a single utask, named ut2b. ucom_data and ucom_code are regions that are shared with other utasks within the same partition or other partitions. The ut2b_data and ut2b_code regions are unique to ut2b. Regions 4 and 5 are NULL regions reserved for dynamic regions. Note that this template matches Figure 4.5, except that it has no auxiliary regions. The mask for this template is 0xF. The template regions 0 to 3 are loaded one to one into the MPA of ut2b then NULL regions are automatically loaded into MPA[4] and [5].

The above template definition must be preceded with:

```
#pragma section = "ucom_data"
#pragma section = "ucom_code"
#pragma section = "t2b_data"
#pragma section = "t2b_code"
```

These pragmas link the region names used in the template to blocks defined in the linker command file, discussed below in section 4.4 Linker Command File.

Each line in the above template is a structure: {rbar, rasr, name}. The macros RGN(), RA(), SRD(), and RSI() automatically create the necessary MPA fields. The macros and constants are defined in mpatmplt.h. It is recommended that all templates be put into mpa.c and to avoid including mpatmplt.h in other files, due to its ultra-short macro names.

In the above array of struct’s, the first field is *rbar* (*region base address register*). Looking at rbar for region 0, we see that it starts at ucom_data, the V flag is set, and MPA[0] is selected to be loaded by the RGN() macro. The slot numbers in the template are MPA slot numbers. When the MPA slots are loaded, the constant MP_MPU_FAS (First Active Slot) is added to each MPA slot number to convert it to its corresponding MPU slot number. (For example, in Figure 4.5, MP_MPU_FAS = 2.) When the task scheduler later loads a task’s MPA into the MPU for a task switch, the V flag and MPU slot number are used for fast MPU loading – see below.

MP_MPU_FAS is defined in mpu.h. As previously noted, for 8-slot MPUs, it is normally 0 because at least 8 active slots are needed for most systems. Hence MP_MPU_FAS is primarily of use for 16-slot MPUs, where it might be 6, leaving 10 active slots.

The second structure field is *rasr* (*region attribute and size register*). Looking at rasr for region 0, we see that it is a DATARW region. The SRD() macro automatically sets subregion disable

flags, the RSI() macro sets the size index, and EN means that the region is enabled. Automatic generation of subregion disable flags greatly reduces errors and is discussed more fully in section 4.4 Linker Command File.

The third structure field is *name.* It is loaded into the task's MPA only during debug and it is never loaded into the MPU. The name field helps one to remember what the region is for. It primarily useful in the smxAware MPA and MPU displays.

The region attribute macros are defined as follows:

```
#define DATARW (XN | RW | C)
#define CODE   (RO | C)
```

where XN, RW, and RO are MPU attributes that mean *execute never*, *read/write*, and *read-only*. C from TEX C B = 0 1 0 and defines *normal memory* — see Yiu. If normal memory is not defined, alignment errors occur. These and other attributes are defined in mpatmplt.h. For example, the DATARW attribute macro means that a region using it cannot be loaded with executable code then executed – a favorite hacking technique. This greatly impedes a hacker, especially if CODE regions are in ROM and therefore inalterable. A DATARW region can be used only to store and retrieve data.

4.3.5 Creating ARMM8 MPA Templates

The corresponding ARMM8 template to the above ARMM7 template is as follows:

```
MPA mpa_tmplt_ut2b =
{
  RGN(0, RA("ucom_data") | DATARW, RLA("ucom_data") | AI(0) | EN, "ucom_data"),
  RGN(1, RA("ucom_code") | CODE,   RLA("ucom_code")   | AI(0) | EN, "ucom_code"),
  RGN(2, RA("ut2b_data")   | DATARW, RLA("ut2b_data")   | AI(0) | EN, "ut2b_data"),
  RGN(3, RA("ut2b_code")  | CODE,   RLA("ut2b_code")     | AI(0) | EN, "ut2b_code"),
  RGN(4, 0, 0, "spare"),  /* reserved for dynamic region */
  RGN(5, 0, 0, "spare"),  /* reserved for dynamic region */
  RGN(6, 0, 0, "stack"),  /* reserved for task stack */
};
```

This is much simpler than the ARMM7 template. Slot numbers are ignored and only present for readability. There is no V flag, and DATARW and CODE appear in rbar. There is no SRD() since the ARMM8 MPU has no subregions. RLA() replaces RSI(), and specifying an end address rather than an encoding for the size is much easier for I/O and other fixed regions. EN is still present.

The attribute index (AI) is set to 0, which selects MAIR0, *Memory Attribute Indirection Register 0*. This is one of 8 registers that can be selected. It is defined in mpatmplt.h as 0x44, which means "normal memory, outer non-cacheable, normal memory inner non-cacheable". MAIR1, which is used for I/O regions, is defined as 0x4, which means device attributes nGnRE = non-gathering, non-reordering, early write acknowledgement. See the ARMv8-M Architecture Reference Manual for more information on MAIR attributes.

DATARW and CODE are defined slightly differently than for ARMM7, but do basically the same things. Other than these differences, the operation of templates, MPAs, and tasks is the same for ARMM7 and ARMM8.

4.3.6 Fast MPU Load

Since the active region of a task's MPA is loaded into the MPU every time a task is started or resumed, fast MPU loading is important. Both ARMM7 and ARMM8 permit loading up to 4 slots at a time using registers R2 thru R9 and not requiring setting of the MPU_RNR register for each region load. (ARMM8 fast loading is a little more complicated due to the lack of the region number in RBAR and because loading cannot span any 4 consecutive regions, e.g. regions 2-5 cannot be loaded in one operation. MPU_RNR must be set to 0, 4, 8, etc. ahead of each bulk load.)

Fast load is enabled if MP_MPA_DEV == 0. This allows mp_MPULoad() to use the MPU region override feature. During debug, MP_MPA_DEV == 1 and a slower load method uses the MPU_RNR register to load one region at a time. In this mode, some checking is done to help find ARMM7 template errors. No checking is done for ARMM8.

For ARMM7, MPA entries must be in increasing order by MPU slot number and the V bit must be set. Unused slots and those reserved for dynamic regions or the task stack must be defined with the slot number, V bit, and 0 for RASR, such as:

```
RGN(n | V, 0, “name”),
```

where n is the slot number.

For ARMM8, MPA entries, slot numbers are unused and only present for readability, and there are no V flags. Unused slots and those reserved for dynamic regions or the task stack must be defined as follows:

```
RGN(n, 0, 0, “name”),
```

4.3.7 Template Errors

Template errors are likely to be difficult to track down and may cause unexpected system vulnerabilities. The MPU and MPA displays in smxAware are helpful to find these errors. They show the contents of each region and its name, as well as its starting address, ending address, size, and attributes. smxAware also provides *overlapping region* and *adjacent region alerts*.

Overlapping regions in ARMM7 cause difficulty because the attributes of the higher-numbered region take precedence over those of the lower-numbered region. For example, if the lower region is DATARW and the upper region is PDATARW, when a utask attempts to access an address in the overlap area, an MMF will occur. This can be puzzling because there is nothing wrong with the lower region and the overlap may not be noticed. (The reason why static slots are below active slots is to avoid this problem, because static slots are usually privileged slots.)

The situation is even worse in ARMM8. In this case, the first access to the overlap area causes an MMF. Once again, the cause of the MMF is difficult to determine since nothing seems to be wrong. Worse than lost debug time, this could be an Achilles heel that brings a shipped system down when a once-in-a-blue moon event occurs that causes access to overlapped areas.

Adjacent regions pose a more subtle problem. In their case, overflows may not be detected because one adjacent region picks up where the other region leaves off. Systems could easily ship with this exploitable flaw undetected by the manufacturer, but not by a hacker. An example of this vulnerability in a ARMM7 system is a ptask stack taken from mheap. Since mheap is in the sys_data region and this region is in every ptask's MPA, stack overflow will not be detected.

In the case of a ARMM8 system, dispatching this task will cause an MMF before the task can even begin to run. Yet the task main function is in a code region of the MPU, the stack is in the stack region of the MPU, and everything else looks ok. So what's wrong?

Whenever a puzzling MMF occurs, check the smxAware MPU window for a region overlap alert. Prior to final release, smxAware should be checked for all static and dynamic regions used by every task for both overlap and adjacent alerts. See Chapter 8 Implementation for more information on this.

4.3.8 Standard Regions

The following standard regions are used in SecureSMX and in this manual:

sys_code	system services code and constant data, exception handlers, and ISRs.
sys_data	system global variables, and MPA templates.
ucom_code	Common umode functions, including SVC shell functions for system services from umode, C library functions, some portal functions, and portal shell functions.
ucom_data	Data for ucom functions. Note that this region represents an inter-partition vulnerability and should be minimized, if not eliminated.

sys_code and sys_data allow ptasks to access system services. They are present in all ptask MPAs. As a consequence, Background Region (BR) is not needed by ptasks and is off when they run.

ucom_code is an execute-only region that should be placed in ROM so that it cannot be altered. It contains code shared by utasks and ptasks. The existence of this shared region would seem to violate partition isolation. However, we feel what is in it should be safe for most systems. If very high security is needed for your system, its contents should be carefully considered. The same is true for any new code added here. There are ways to avoid common code such as duplicating functions and making separate SVC shells and jump tables for each partition.

In addition to these standard regions, each task may have it own code, data, and IO regions. These are defined by the application. See mpa.c for examples of templates using standard and application regions.

4.4 Linker Command File

The linker command file may seem intimidating, at first, but it is not that bad. Most programmers have little experience with it, and it has unique commands that may seem strange. However, IAR ILINK is a powerful linker and it is well-suited to the task as hand. You will need

to get as familiar with modifying your linker command file as you are with writing C code. And, with the introduction of the MPU, expect the linker command file to become pretty large.

As noted in the previous section, the regions used in templates correspond to blocks created in the linker command file (.icf file for EWARM). Appendix B contains two complete linker command files, for reference. The first is for ARMM7 and the second is for ARMM8. Both are for the same code – the smx/SecureSMX regression test. The only difference is the memory structures of the two processors. All blocks are the same. The description that follows is primarily for the ARMM7 linker command file, since it is the more complex of the two. Since the linker *blocks* become MPU *regions* the terms are used interchangeably in the discussion that follows. For maximum clarity, the term *region block* is used.

4.4.1 First Sections

At the top of the ARMM7 linker command file, memory regions are specified, for example:

```
define region SRAM      = mem:[from 0x20000000 to 0x2004FFFF];
```

Linker regions are not the same as MPU regions — don't let the terminology confuse you.

After the memory regions come the size definitions for MPU regions, such as:

```
define exported symbol scsz          = 0x80000;   /* sys_code size */
define exported symbol sdsz          = 0x40000;   /* sys_data size */
define exported symbol ucomcsz       = 0x8000;    /* ucom_code size */
```

We recommend choosing abbreviated symbol names and also using hex sizes. Sizes must be powers of two for ARMM7, so hex sizes can have only one non-zero digit and it must be 1, 2, 4, or 8. This helps to avoid size errors, which can be hard to find. For ARMM8, we also recommend using hex sizes. Sizes must be multiples of 32. Hence, each size should end in 0x00, 20, 40, 60, 80, A0, C0, or E0. Any other size ending is prohibited and will cause an MMF. It is relatively easy to scan even a long list, as shown, and see wrong endings.

By "exported" it is meant that the symbol can be used in code, merely by adding externs ahead of it:

```
extern u32 scsz;
extern u32 sdsz;
```

Next come empty block definitions, such as:

```
define block CSTACK with size = 0x200,       alignment = 8     { }; /* Main Stack */
define block EVT        with size = EVT_size, alignment = 512 { }; /* Exception Vector Table <1> */
define block mheap      with size = 0x4000,     alignment = 16   { }; /* mheap if SMX_CFG_SSMX */
```

The first allocates space for the *main stack*, the second allocates space for the exception vector table, EVT, and the third allocates space for the *main heap*. Space for other system buffers and tables are also allocated in this section. Empty blocks that are not referenced in the code require the keep {} command.

4.4.2 Region Block Definitions

Next come the *region block* definitions. These are linker blocks that become MPU regions. For ARMM7 for example:

```
define block ucom_code with size = ucomcsz*5/8, alignment = ucomcsz
        {ro section .svc.text, ro section .svc.rodata, ro object strncpy.o, ro object strlen.o};
```

where ucomcsz is the smallest power of two big enough to contain the region. Note that the block size is ucomcsz*5/8. This means that SubRegion Disables, SRDs 5, 6, and 7 are 1 – i.e. these subregions are disabled. This is shown in Figure 4.9

Figure 4.9 Region with Subregion 5-7 Disables

In this figure, the shaded area shows the portion of the region to which the MPU will allow access. Attempted accesses in the white area will result in MMFs. Hence, the white area is available for use by another region or regions. As design progresses, regions will grow and the linker may complain that a block is too small. This is easily handled by increasing 5/8 to 6/8, then to 6/8, then to 7/8, and finally to no fraction (=8/8). After this, double the size constant, e.g. ucomcsz and use the 5/8 fraction. This process becomes as natural as fixing syntax errors when the compiler complains.

As described in section 4.3.3 Using ARMM7 MPU Subregions, the SRD macro automatically calculates the SRD field in the RASR from the region block size. Hence, this process is error-free.

In the above block definition, notice that alignment is ucomcsz, as required by ARMM7. Within the {} there is: ro section .svc.text. Sections are defined in the source code and will be discussed next. Note also:

```
ro object strncpy.o, ro object strlen.o
```

This is how standard C library functions are included in ucom_code for use by all tasks with ucom_code in their MPAs. ro means read only, and .o is means an object file. It is not desirable to include the entire C library because this would increase memory requirements and there are some C library functions that are undesirable. Embedded systems normally require very few

standard C library functions, so this is a practical approach. You might want to put the C library functions into their own region. However, the limited number of MPU slots works against this.

Continuing to the next sections of the linker command file, notice that blocks are defined for sys_code and sys_data, and then region blocks are defined for the tasks. t2a is a ptask; ut2a is a utask. These cryptic names are used in the smx regression test. You would, of course, pick more meaningful names for your tasks. Note that there are both code sections and data sections.

Most of the above is not applicable to ARMM8. Looking a the ARMM8 linker command file in Appendix B note that all region block sizes are the sizes defined in the region sizes section and all alignments are 32 bytes.

4.4.3 Block Ordering

Unfortunately ILINK does not order blocks for best memory efficiency. For example, if the region size in Figure 4.9 is 256K, then the subregions sizes are 256K/8 = 32K. If the next block size is 128K, it cannot start after subregion 4, 5, or 6 because these are not on 128K boundaries. Therefore, the linker will put this block after subregion 7 resulting in an unused gap of 96K.

To fix this problem, use *block-ordering blocks* with fixed order keywords, such as:

```
define block rom_block with fixed order, size = romsz*5/8, alignment = romsz
                    {block sys_code, block ut1a_code, block ut2b_code,
                     block ut2a_code, block t2a_code, block t2b_code,
                     block ut2c_code, block ut2d_code, block ut2s_code,
                     block t2c_code, block t2s_code, ro};
```

This puts the blocks in the order specified, so that smaller blocks are put into gaps between larger blocks. Typically, every time a region block is increased in size, gaps will get larger and the ordering will need to be changed. This is can be time-consuming, if there are many blocks.

SecureSMX includes a tool called *MpuPacker* which determines the block order to achieve the most efficient memory usage. The blocks can then be reordered, in the above, and the linker run again. See section 8.11.1 Using MpuPacker for more information.

However, there will still be gaps between region blocks for ARMM7. These can be partially filled with *plug blocks* consisting of code and data not in region blocks (see 8.11.4 Using Plug Blocks). Another solution is to define smaller regions by using more tasks (see 4.1.8 Parent and Child Tasks). This will result in smaller gaps. There are several other methods to reduce memory waste, which are discussed in appropriate sections of this manual.

The foregoing is not necessary for ARMM8 since all region blocks are multiples of 32 bytes and are 32-byte aligned, so wasted memory is minimal and there is no advantage to moving region blocks around. Looking at ARMM8 linker command file in Appendix B, rom_block and ram_block are defined for use in templates such as mpa_tmplt_tinit. This is true also for ARMM7.

4.5 Defining Sections

As can be seen in the linker command files in Appendix B, *sections* define the contents of the linker region blocks, which, in turn, become MPU regions. Sections are defined in the source code. There are three ways to do this:

4.5.1 Section Prefixes

The easiest way to define sections is to use section prefixes. For example, at the top of xsmx.h:

```
#pragma section_prefix = ".sys"[3]
```

This prepends .sys to the default section names like this:

```
.sys.bss, .sys.data, .sys.noinit, .sys.rodata, and .sys.text
```

Since the section prefix was put into a header file, it applies to all smx C files that include xsmx.h. If this is not desired in a particular C file, use:

```
#pragma diag_suppress=Ta168  /* ignore warning that next line overrides existing prefix */
#pragma section_prefix = ""         /* cancel .sys prefix in code that follows */
```

to revert to the standard names:

```
.bss, .data, .noint, .rodata, and .text
```

Note that this pragma applies to the whole file, and the last one encountered is the one that is used. Also, the --section_prefix command line switch does the same, but it is better to use the pragma so it is visible in the source code, rather than hidden in the project. It also avoids overriding project nodes, which saves having to make project option changes to every overridden node and file below each.

4.5.2 Command Line Switches

The --section command line switch can be used to rename individual sections. For example, to rename .bss to .sys.bss:

```
--section .bss=.sys.bss
```

Any or all of the sections can be renamed. Also, this can be done in a file that is passed as input. For example, right clicking a node in the project window, such as XSMX, then selecting: Options, C/C++ Compiler, Extra Options with Override inherited settings checked and enter this:

```
-f $PROJ_DIR$\..\..\..\CFG\mpi_sys.xcc
```

[3] The dot at the start is not required, but is used for consistency with standard section names, such as .text. This is a useful convention to identify names as being section names.

Then in CFG\mpi_sys.xcc:

```
--section .bss=.sys.bss
--section .data=.sys.data
--section .rodata=.sys.rodata
--section .noinit=.sys.noinit
--section .text=.sys.text
```

The downside of command line switches is that they are hidden in the project file rather than in the source code, and it is necessary to make project options changes to every overridden node and file below each.

4.5.3 Section Pragmas

Sections can also be defined by putting pragmas into the code. For example:

```
#pragma default_variable_attributes = @ ".ut2s.bss"
TPSS  pssa;      /* portal server structure A */
u32  sz;         /* data size */
u8*   sbp;       /* server buffer pointer */

#pragma default_variable_attributes = @ ".ut2s.rodata"
TPCS* tp_pcl[] = {&pcsA, &pcsB};
#pragma default_variable_attributes =

#pragma default_function_attributes = @ ".ut2s.text"
void tpA_ut2s(void)
{
  mp_TPortalServer(&pssa, STMO);
}
#pragma default_function_attributes =
```

In the above examples, uninitialized data is put into .ut2s.bss, and initialized data is put into .ut2s.rodata. The variable pragma with no section name ends the variable pragma area and reverts to the default for the file. Then the tpA_ut2s() function is put into the .ut2s.text section. Additional functions can follow this function. The function pragma with no section name ends the function pragma area. The default variable sections (e.g. .bss) apply outside of the variable pragma areas and the default code sections (e.g. .text) apply outside of the function pragma areas.

The pragma approach can be used to override either of the above two methods. This allows different sections to be grouped logically together in the same file. For example, t2a_init can put be put into .text, whereas the t2a_main can be put into .t2a.text. Keeping the two functions together in a source file helps to reduce errors.

Pragma section assignments can also be spread around. For example,

```
#pragma default_variable_attributes = @ ".ut2s.bss"
```

can appear multiple times in the same module and it can appear in other modules, as well. The linker will combine all of them into a single .ut2s.bss section.

Note: The pragma approach requires special handling for string literals, so the command line technique may be preferable. See the end of section 4.5.4 String Literals for more information.

4.5.4 Template Macros

The template macros used in section 4.3.4 Creating ARMM7 MPA Templates use the __section_begin() and __section_size() pragmas (see mpatmplt.h). These actually work with section block names as well as on section names. In templates, they are used only for section block names. Using a section name in __section_begin() would be wrong unless it is the first section in the block. Using it in _section_size() would be wrong unless the section is the only thing in the block.

4.5.5 String Literals

The following:

```
#pragma default_variable_attributes = @ ".fs.rodata"
"This is STM32 working with FatFs"
#pragma default_variable_attributes =
```

does not put the string literal into .fs.rodata, as expected. Instead, the compiler puts it into the standard .rodata section. Thus when this string literal is accessed in the fs partition an MMF will occur. In order to fix this it is necessary to use the section command line switch:

```
--section .rodata=.fs.rodata
```

Then the string literals in the fs partition can be put into the fs_code region block, as follows:

```
define block fs_code   with size = fscsz*7/8, alignment = fscsz {ro section .fs.text,
                                                                 ro section .fs.rodata}
```

Following this with

```
initialize by copy with packing = none {rw};
```

in the linker command file may be necessary to avoid an error message, such as the following:

```
Error[Lp017]: the address of ".fs.rodata41 (tportal.o)" was needed when
      computing compressed initializers for section .data (ffdemo.o #8),
      but that address hasn't been set yet, since the size of the
      compressed initializers are needed in order to set it.
```

In some cases, the compiler puts string literals into the .text segment, such as when they are passed as parameters:

```
mp_FPortalCreate(fpsh, cp_pcl, cp_pclsz, ssn, “cp”, "cp_sxchg");
```

In order to get “cp” into .cp.rodata, define a variable to point to it, as follows:

```
const char* const cpname = "cp";

mp_FPortalCreate(fpsh, cp_pcl, cp_pclsz, ssn, cpname, "cp_sxchg");
```

Now the:

```
–section .rodata=.cp.rodata
```

command line switch in Extra Options works thus allowing utasks to access “cp”.

Another problem arises if a module has string literals that must go into different sections, since there can only be one section command line switch for section type (e.g. .rodata) in a module. This might occur, for example, if both client and server code are in the same module. The simplest solution, in this case, is to split the module into a client module and a server module. But if this is not practical, use a section command line switch for the most common string literals, and for each of the others, assign a string literal to an array variable (notice the brackets) and use section pragmas, as follows:

```
#pragma default_variable_attributes = @ ".cp.rodata"
const char* const cpname[] = "cp";
#pragma default_variable_attributes =
```

Multiple strings, such as an error message table, can be handled as follows:

```
#pragma default_variable_attributes = @ ".cp.rodata"
static const char err0[]  = "ERROR 0";
static const char err1[]  = "ERROR 1";
static const char err2[]  = "ERROR 2";

const char* const errmsg[] = {err0, err1, err2};
#pragma default_variable_attributes =
```

4.6 Map Files

4.6.1 ARMM7

If you are creating a system of significant complexity, it is easy to get confused with all the section definitions in the code and region block definitions in the linker command file. The smxAware MPA and MPU displays are a big help with this. The map file is also helpful to see exactly what is happening, for example:

```
rom_block                          0x20'0000   0xa'0000  <Block>
  sys_code                         0x20'0000   0x1’4000  <Block>
    .intvec             const      0x20'0000      0x1c8  vectors.o [1]
    .sys.rodata         const      0x20'01c8        0x4  bspm.o [1]
    .sys.rodata         const      0x20'01cc        0x4  bspm.o [1]
    .sys.rodata         const      0x20'01d0       0x28  isrshells.o [1]
    .sys.rodata         const      0x20'01f8        0x8  mpu.o [1]
    .sys.rodata         const      0x20'0200        0x8  mpu.o [1]
     ...
    .sys.text           ro code    0x20'06b0      0x368  bspm.o [1]
    .sys.text           ro code    0x20'0a18      0x190  clock.o [1]
    .sys.text           ro code    0x20'0ba8       0xd0  bbsp.o [5]
    .sys.text           ro code    0x20'0c78      0xafc  isrshells.o [1]
    .sys.text           ro code    0x20'1774      0x6d8  xprof.o [5]
    .sys.text           ro code    0x20'1e4c      0x304  xsys.o [5]
     ...
    sys_code            uninit     0x21'3c6c      0x394  <Block tail
```

The above corresponds to stm32746g_tsmx.icf in Appendix B. Note that the rom_block starts at 0x200000 with size 0xA0000, and the sys_code block is inside of it and also starts at 0x200000 with size 0x14000. sys_code contains the .intvec and .sys.rodata and .sys.text sections, as well as other sections that are not shown. sys_code ends with a *Block tail* of 0x394 unused bytes. This 916 bytes of wasted memory is a consequence of the ARMM7 MPU region power-of-two size requirement.

The *region size* for sys_code is 0x20000 bytes; the *subregion size* is therefore 0x4000 bytes (i.e. 0x20000/8), and the first 5 subregions = 0x14000 bytes are used for sys_code. Clearly, one less subregion would be too small (0x394 compared to 0x4000), so this is the best we can do.

Continuing in the map file:

```
ut1a_code                        0x21'4000       0x20  <Block>
  .ut1a.text          ro code    0x21'4000       0x10  tmpu.o [1]
  ut1a_code           const      0x21'4010       0x10  <Block tail>

   ...

ut2b_code                        0x21'4200      0x200  <Block>
  .ut2b.text          ro code    0x21'4200       0x60  tmpu.o [1]
  .ut2b.text          ro code    0x21'4260      0x168  tpmsg.o [1]
  .ut2b.text          ro code    0x21'43c8       0x22  theap.o [1]
  ut2b_code           const      0x21'43ea       0x16  <Block tail>
ut2a_code                        0x21'5000      0xa00  <Block>

   ...

t2b_code                         0x21'6c00      0x100  <Block>
  .t2b.text           ro code    0x21'6c00       0x24  tpmsg.o [1]
  t2b_code            const      0x21'6c24       0xdc  <Block tail>
ut2c_code                        0x21'8000     0x1400  <Block>

   ...
```

the next region, ut1a_code starts at 0x214000, and its size is only 0x20. Going further down in the table we find t2b_code followed by ut2a_code. ut2b_code ends at 0x2143ea + 0x16 = 0x214400. But ut2a_code starts at 0x215000 – 0xC00 = 3072 bytes above. Why is this? It is because ut2a_code has a size of 0xa00. Looking at the linker command file: ut2acsz = 0x1000 and:

```
define block ut2a_code  with size = ut2acsz*5/8, alignment = ut2acsz
```

so ut2a_code must be aligned on 0x1000 and 0x215000 is the next higher 0x1000 boundary.

Unlike block tails, block gaps can be reclaimed. This done by filling them with other blocks. From tsmx.icf we see that t2bcsz = 0x100, so t2b_code could be relocated to 0x214400, between ut2b_code and ut2a_code. Other small regions could also be moved into the gap to reduce wasted memory. It is unfortunate that ILINK does not do this, and it is obviously tedious to do it manually – especially if it must be done every time a block size changed. Our *MpuPacker* tool takes care of this problem – see section 8.11.1 Using MpuPacker.

Looking further down in the map file, we find:

```
ucom_code                    0x21'4000    0x3800  <Block>
  svc_code                   0x21'4000    0x2000  <Block>
    .svc.text       ro code  0x21'4000     0x1b8  svc.o [1]
    .svc.text       ro code  0x21'41b8      0x10  tmain.o [1]
    .svc.text       ro code  0x21'41c8     0x2bc  fportlc.o [5]
    .text           ro code  0x21'4484      0x66  ABImemset.o [4]
    .svc.text       ro code  0x21'44ec     0x242  tportls.o [5]
    .svc.text       ro code  0x21'4730     0x5c0  tportlc.o [5]
    .text           ro code  0x21'4cf0      0xa6  ABImemcpy.o [4]
    .text           ro code  0x21'4d98      0x18  strcpy.o [4]
    .text           ro code  0x21'4db0      0x36  strlen.o [4]
    .text           ro code  0x21'4de8      0x70  strncpy.o [4]
    .svc.text       ro code  0x21'4e58     0x170  cpcli.o [1]
    .text           ro code  0x21'4fc8      0x1e  strcat.o [2]
    svc_code        const    0x21'4fe6    0x101a  <Block tail>
```

Here we see that C library functions such as ABImemset.o, the smxu shell functions in svc.o, and portal functions have been put into svc_code which is in ucom_code.

Further down in the map file, we find:

```
sram_block                   0x2000'0000   0x2'8740  <Block>
  sys_data                   0x2000'0000   0x2'8000  <Block>
    CSTACK                   0x2000'0000      0x200  <Block>
      CSTACK          uninit 0x2000'0000      0x200  <Block tail>
    EVT                      0x2000'0200      0x1c8  <Block>
      EVT             uninit 0x2000'0200      0x1c8  <Block tail>
    mheap                    0x2000'03d0     0x4000  <Block>
      mheap           uninit 0x2000'03d0     0x4000  <Block tail>
    EVB                      0x2000'43d0     0x8000  <Block>
      EVB             uninit 0x2000'43d0     0x8000  <Block tail>
    heap1                    0x2000'c3d0     0x6144  <Block>
      heap1           uninit 0x2000'c3d0     0x6144  <Block tail>
    heap2                    0x2001'2520     0x2000  <Block>
      heap2           uninit 0x2001'2520     0x2000  <Block tail>
    heap3                    0x2001'4520     0x2f60  <Block>
      heap3           uninit 0x2001'4520     0x2f60  <Block tail>
    ucom_data                0x2001'7600      0x200  <Block>
      ucom_data-1            0x2001'7600       0x30  <Init block>
        .ucom.data    inited 0x2001'7600       0x2c  mfxstm32l152.o [1]
        .ucom.data    inited 0x2001'762c        0x4  mfxstm32l152.o [1]
      .ucom.bss       zero   0x2001'7630       0x4c  stm32756g_eval.o [1]
```

We see that sys_data is first in sram_block, and the main stack, CSTACK, is first in it. Next is the Exception Vector Table, EVT. This is where the sb_irq_table (see irqtable.c) is copied from ROM by __low_level_init() in startup.c, then ARMM_NVIC_VTOR is set to point to it. Next is mheap, the main heap, followed by EVB and other heaps. Because none of these are regions they need only be aligned as specified in tsmx.icf. Then comes ucom_data, which is a region block and thus aligned on its size, udsz = 0x100.

None of the blocks above ucom_data are initialized (see tsmx.icf). ucom_data consists of .ucom.data-1, which is initialized, and .ucom.bss, which is not. The ucom_data-1 Initializer_bytes are at:

```
    Initializer bytes     const      0x24'6780       0x30  <for ucom_data-1>
```

They are in ROM since they are constants and they are copied into the .ucom.data variable during system startup. The other variables are in .ucom.bss section, which means they are initialized to 0.

In some cases regions are nested within other regions, such as ucom_data within sys_data. This is done since some tasks may only need access to the inner region(s), but other tasks need access to the inner and outer region(s), but don't have enough MPU slots for all regions. In the above case, ucom_data might be shared between utasks, which do not access sys_data. But ptasks may need access to ucom_data and sys_data. Look at task templates to see what each task can access. For information on nesting regions, see section 4.7.2 Combined Regions.

4.6.2 ARMM8

The following map file corresponds to lpc55s69evk_tsmx.icf in Appendix B:

```
  sys_code                              0x0  0x1'1f00  <Block>
    .intvec               const           0x0     0x130  vectors.o [1]
    .sys.rodata           const         0x130       0x4  bspm.o [1]
    .sys.rodata           const         0x134       0x4  bspm.o [1]
    .sys.rodata           const         0x138      0x20  clock.o [1]
       ...
    .sys.text             ro code       0x734     0x358  bspm.o [1]
    .sys.text             ro code       0xa8c      0x66  term.o [1]
    .sys.text             ro code       0xaf4      0xa0  uart.o [1]
    .sys.text             ro code       0xb94      0xac  clock.o [1]
    .sys.text             ro code       0xc40      0xd0  bbsp.o [5]]
       ...
    sys_code              uninit     0x1'1eec      0x14  <Block tail>
```

sys_code starts with intvec. It also contains cp_code and ucom_code, like ARMM7. Note that the sys_code tail is a tiny 0x14 = 20 bytes!

Further down the map file, we find rom_block with ut1a_code:

```
  rom_block                           0x1'1f00  0x2'75dc  <Block>
    ut1a_code                         0x1'1f00      0x20  <Block>
      .ut1a.text          ro code     0x1'1f00      0x10  tmpu.o [1]
      ut1a_code           const       0x1'1f10      0x10  <Block tail>
    ut2b_code                         0x1'1f20     0x200  <Block>
      .ut2b.text          ro code     0x1'1f20      0x60  tmpu.o [1]
      .ut2b.text          ro code     0x1'1f80     0x168  tpmsg.o [1]
      .ut2b.text          ro code     0x1'20e8      0x22  theap.o [1]
      ut2b_code           const       0x1'210a      0x16  <Block tail>
    ut2a_code                         0x1'2120     0x640  <Block>
```

These are similar to ARMM7, except note that the ut2b_code to ut2a_code gap is only 0x12120 – (0x1210a + 0x16) = 0 – i.e. no gap!

Further down, we find:

```
sys_data                          0x2000'0000  0x1'daa0  <Block>
  CSTACK                          0x2000'0000     0x200  <Block>
    CSTACK              uninit    0x2000'0000     0x200  <Block tail>
  EVT                             0x2000'0200     0x1c8  <Block>
    EVT                 uninit    0x2000'0200     0x1c8  <Block tail>
  mheap                           0x2000'03d0    0x4000  <Block>
    mheap               uninit    0x2000'03d0    0x4000  <Block tail>
  EVB                             0x2000'43d0    0x8000  <Block>
    EVB                 uninit    0x2000'43d0    0x8000  <Block tail>
  ucom_data                       0x2000'c3e0     0x280  <Block>
    .ucom.bss           zero      0x2000'c3e0       0x4  bbase.o [5]
```

These are similar to ARMM7, except the sys_data and ucom_data blocks are smaller. Note that sys_data size is not a power of two, but rather a multiple of 32.

4.6.3 MpuMapper

MpuMapper is a utility we developed to modify the IAR map file in order to see what region each variable and function is in. This greatly aids tracking down and fixing MMFs. The Placement Summary changes from this:

```
    .sys.text           ro code   0x800'171c      0x5be  mpu.o [1]
    .sys.text           ro code   0x800'1cdc      0x3a4  bspm.o [1]
          ...
```

to this:

```
    .sys.text           ro code   0x800'171c      0x5be  mpu.o [1]
          mp_MPACreate              0x800'171d
          mp_MPASlotMove            0x800'1919
          mp_MPUInit                0x800'19bd
          mp_MPULoad                0x800'1a05
          mp_RegionGetHeapT         0x800'1a75
          mp_RegionGetPoolT         0x800'1b0b
          mp_RegionMakeT            0x800'1b7f
          mp_RegionGetHeap()        0x800'1c29
    .sys.text           ro code   0x800'1cdc      0x3a4  bspm.o [1]
          sb_ClocksInit             0x800'1cdd
          ...
```

The Placement Summary can then be scanned from top to bottom to look for things out of place, which is not practical with the original map. The modified map file replaces the original map file, which is renamed as xxxx_sav.map.

If MpuMapper is run with no arguments, it opens a dialog to permit browsing to the input .map file. Otherwise, the input file can be specified as an argument. This is useful to run it automatically from the IDE after every build. To do this, click on Project, Options, Build Actions, and on the Post-build command line put:

```
$PROJ_DIR$\..\..\..\BIN\MpuMapper.exe "$TARGET_DIR$\$TARGET_BNAME$.map"
```

4.7 Regions

4.7.1 Insufficient MPU Slots

When working with an 8-slot MPU, insufficient slots can be a problem for some partitions. Some solutions to this are:

1. Auxiliary slots – See the discussion in section 4.2.5 Auxiliary Slots and see Figure 4.6.
2. Child tasks – See the discussion in section 4.1.8 Parent and Child Tasks.
3. Define server partitions and access them through portals – See Chapter 5 Partition Portals.

However, these solutions may not be available for a particular partition for various reasons, such as requiring too much code change. In that case it is necessary to fall back to the solutions that follow in this section.

4.7.2 Combined Regions

If there are not enough MPU slots, it may be necessary to combine regions that you had hoped to keep separate. Fortunately this is easy to do, and it would be easy to reverse if something changes that makes an additional slot available. This is done in the linker command file, as follows:

Consider the initial case of having separate FS and USBD code and data regions:

```
define block fs_code    with size = 0x7000, alignment = fscsz
                        {ro section .fs.text, ro section .fs.rodata};
define block fs_data    with size = 0x400,  alignment = fsdsz
                        {rw section .fs.bss, rw section .fs.data, rw section .fs.noinit};
define block usbd_code  with size = 0x10000, alignment = usbdcsz
                        {ro section .usbd.text, ro section .usbd.rodata};
define block usbd_data  with size = 0x1000,  alignment = usbddsz
                        {rw section .usbd.bss, rw section .usbd.data, rw section .usbd.noinit};
  ...
place in ROM  {..., block fs_code, block usbd_code, ...};
place in SRAM {..., block fs_data, block usbd_data, ...};
```

To reduce the number of regions, fs_code and fs_data blocks are added into the usbd_code and usbd_data blocks, respectively. Size and alignment are deleted from the fs blocks, and then they are simply added to the lists in usbd blocks and deleted from the place directives:

```
define block fs_code {ro section .fs.text, ro section .fs.rodata};
```

```
define block fs_data  {rw section .fs.bss, rw section .fs.data, rw section .fs.noinit};
define block usbd_code  with size = 0x10000, alignment = usbdcsz
                {ro section .usbd.text, ro section .usbd.rodata, block fs_code};
define block usbd_data  with size = 0x1000,  alignment = usbddsz
                {rw section .usbd.bss, rw section .usbd.data, rw section .usbd.noinit,
                 block fs_data};
  ...
place in ROM   {..., block usbd_code, ...};
place in SRAM {..., block usbd_data, ...};
```

Two problems with this solution are that a task requiring access to the USBD stack also gains access to the file system, and the file system regions are now buried in the USBD regions and thus cannot be used separately. With regard to the first problem, a reduction of partition isolation and thus security, has occurred. Risk analysis may determine that this is ok for a particular system. If not, another approach is to make fs and/or usbd separate partitions and access them via portals. See Chapter 5, Partition Portals

With regard to the second problem, it is possible to retain the size and alignment of the fs blocks and thus allow them to be used independently. In this case, the original blocks would be included in the usbd blocks, as follows:

```
define block usbd_code  with size = 0x10000, alignment = usbdcsz
                { block fs_code, ro section .usbd.text, ro section .usbd.rodata};
define block usbd_data  with size = 0x1000,  alignment = usbddsz
                { block fs_data, rw section .usbd.bss, rw section .usbd.data,
                  rw section .usbd.noinit};
```

Notice that they now appear first. This is necessary to avoid large gaps inside of the usbd blocks. For example, fs_data is aligned on 0x400, and since usbd_data is aligned on 0x1000, if fs_data is first in usbd_data, it will be automatically aligned and no gap will occur ahead of it. If more than one aligned block is to be included in a larger aligned block, the largest included block should be put first, then the next largest, etc.

4.7.3 Common Regions

Rather than putting whole regions into other regions to cope with the limited number of MPU slots, as discussed in the previous section, it may work better to put common code and data into *common regions*, such as ucom_code and ucom_data.

A example of this is a task that uses smxFS to write to a USB disk. This task needs to access to smxfs_code, smxfs_data, smxusbh_code, smxusbh_data, and the USBH I/O region = 5 regions. The task needs access to its own code region, data region, stack region, and ucom_code region = 4 regions. (The ucom_code region is necessary for system services and standard C library functions.) Thus, 9 regions are needed, which exceeds MPU slots available. A solution to this problem is to put the subset of smxUSBH code needed for USB disk access into ucom_code, creating a new region, ucomx_code. Now the task needs access to just 8 regions.

Those tasks needing access only to system services would be given the ucom_code region, not the ucomx_code region. Hence they could not access the common smxUSBH code. However

any task that does require ucomx_code would also have access to the common smxUSBH code, so there is a reduction in partition isolation with this solution.

The difference between this solution and that of the previous section is that only some of the USBH code is exposed to other tasks – namely that portion needed to access a USB disk. Other USBH class drivers, device drivers, and portions of the USBH stack are not exposed. Thus it would not be possible for a hacker to gain access to some other USB device such as a WiFi stack through ucom_code. On the other hand since basic USBH code is exposed, the hacker could disrupt USBH services.

As in the previous section, if these weaknesses are not acceptable, putting the file system and USBH into separate partitions and accessing them via portals is a possible solution.

4.7.4 I/O Regions

Unlike memory regions, I/O regions have fixed addresses. Thus it is easier to specify them with direct memory addresses and not involve the linker. For example:

```
RGN(6 | 0x40011000 | V, IO | ( 9 << 1) | EN, "USART1"),
```

is for MPU region 6 of the STM32F746 ARMM7 processor. Looking at the memory map in its Reference Manual, the I/O block for USART1 starts on 0x40011000 and its size is 1 KB. (Referring to Yiu, Table 11.7 b1001 -> 1 KB, b1001 = 9.) Note that subregion disables are not used in the above.

Looking at USART1, in more detail, we find that only the first 32 locations are used. Hence

```
RGN(6 | 0x40011000 | V, IO | ( 4 << 1) | EN, "USART1"),
```

is a better choice. Then a wrong USART1 register address would cause an MMF.

The foregoing is for ARMM7. For ARMM8, the end address is specified rather than an encoding of the limit, which is much easier:

```
RGN(4, 0x40086000      | IOR,   0x40086FE0       | AI(1) | EN, "USART0"),
```

Remember the low byte is 0xE0 because the low 5 bits are used for other purposes. 0xE0 means the limit is 0xFF. Note: the above is for a LPC55Sxx MCU.

Sometimes, partitions require more I/O regions than there are available MPU slots. Possible solutions are swapping regions dynamically and spanning multiple regions. See section 8.5.4 Too Many I/O Regions, for discussion of these approaches.

4.7.5 I/O Regions Using Subregions

In some cases the size or alignment of an I/O register block does not match ARMM7 MPU requirements. For example, for the above MCU, the Ethernet MAC register block is from 0x4002 8000 to 0x4002 93FF, so its size is 0x1400, which is not a power of 2. The next power of 2 is 0x2000, so this is the region size. The corresponding subregion size is 0x2000/8 = 0x400. 0x2000 – 0x800 = 0x1600, which is big enough. So set SRD = b1100 0000 = 0xC0 allows access to the full Ethernet MAC register block. Unfortunately, it also allows access 0x200 bytes above Ethernet MAC register block to address 0x4002 95FF. Fortunately, this area is unused.

The case where a register block starts on an alignment less than its size does not seem to occur for this processor. However, this could happen, perhaps with a custom I/O device. For example, if an I/O register block size is 256 bytes, but it is located on a 128-byte boundary, then it is necessary to look below for a suitable boundary. The next 128-byte boundary below is at least 256-byte aligned, but a 256 byte region starting there would not big enough to contain the full register block. If, however, the boundary is also 512-byte aligned, then a 512 region would be big enough. The subregion size would be 64 bytes. Hence, the subregion disables would be SRD = b11000011 = 0xC3. This prevents access outside of the I/O register block. If say only the first 140 bytes of registers were used, SRD = 0xC7 would be even better.

4.8 Interrupts and Exceptions

4.8.1 Priorities

PendSV has lowest priority, SVC has next highest, and IRQs have higher priority. SVC must be lower than IRQ priorities since otherwise, a system call via SVC would block peripheral interrupts the whole time the system call ran.

4.8.2 Enabling ISRs and Exception Handlers to Run

If there are no static MPU slots available, the Background Region (BR) must be on in umode in order to permit ISRs and exception handlers to execute when an interrupt or exception occurs. This is controlled by setting SMX_CFG_MPU_BR_EN to 1 in xarmm_iar.inc. BR has no effect in umode. It takes effect only when an interrupt or exception causes the processor to switch to hmode.

BR enables access to all of implemented memory with default attributes, except for regions in the MPU, which override the default attributes. Since any task might be interrupted, MPU attribute overrides have no value, yet might cause trouble. However, it appears that compiler and linker checks prevent the worst possible problems such as executing data or changing RO data.

Alternatively, if there are two MPU slots available to be static slots (in all MPAs), it may be preferrable to set SMX_CFG_MPU_BR_EN to 0 and load the sys_code and sys_data regions into them. These are privileged regions and thus cannot be accessed in umode. They should be chosen to be as small as possible, while allowing all ISRs and exception handlers to execute. This means that they should not include initialization nor ptask code and data regions. However, sys_code must include all ISRs, exception handlers, smx, and other system services, and sys_data must include all control blocks, buffers, etc. to support sys_code functions.

This makes utasks more similar to ptasks: ptasks have sys_code and sys_data in the first two MPU slots in order to access system services directly and to support exceptions and ISRs. Also, BR is off for ptasks.

Unfortunately BR off is not much protection against hackers. Rather than turn BR on, the hacker could simply turn the MPU off – only one instruction is required for either. However, keeping BR off does improve reliability (bugs are not as smart as hackers) and is worth doing for that reason, alone.

4.8.3 Interrupts

Interrupts cause an immediate switch to hmode and thus risk allowing hackers to penetrate it. Recalling that any pmode code is but one step away from opening the Vault by turning off the MPU, this is a major security concern. In many cases, only a few lines of carefully written code in an ISR or in an ISR + LSR[4] are needed to do the job. Figure 4.10 illustrates the two ways of handling interrupts.

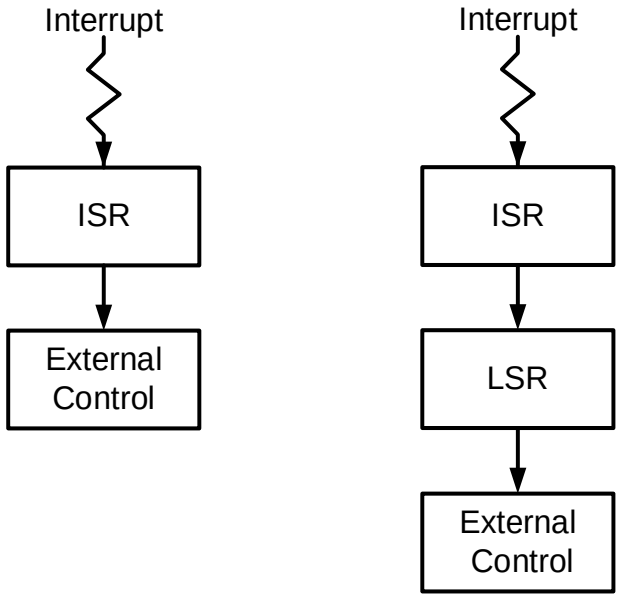

Figure 4.10 Minimal Interrupt Processing

In this case, carefully writing the ISR code and LSR (if needed) code may be adequate protection See section 8.10.6 ISRs and LSRs for suggestions for writing *fortified code* for ISRs and LSRs.

For complex ISRs, it may be a good practice to immediately swap the sys_code and sys_data regions into the MPU, if not already there, and to switch BR off, if it is on. Although this is not a large gain for security, it will improve safety and reliability by helping to catch programming mistakes. When exiting, the ISRs must, of course, restore the replaced regions and switch BR on, if it was on in the interrupted utask.

[4] smx Link Service Routines, LSRs, are used for deferred interrupt processing and to make system calls.

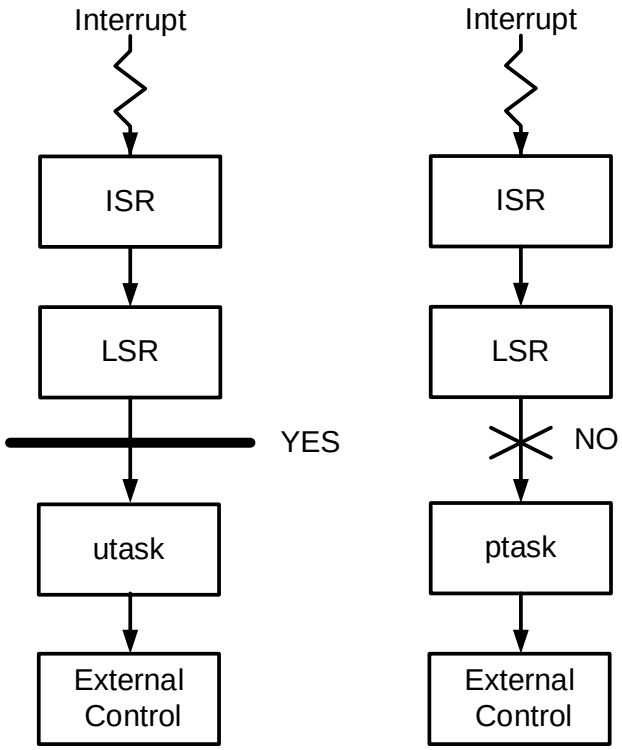

Figure 4.11 Task Interrupt Processing

When more than minimal interrupt processing is required, Figure 4.11 illustrates what to do on the left and what not to do on the right. The objective is to move as much processing as possible into a utask where hacking can be better constrained. As in Figure 4.10, the goal is minimal code in ISRs and LSRs.

Despite the foregoing caution, it may be necessary to do full interrupt processing in pmode (i.e. the above right-hand diagram). This is definitely faster and simpler, especially if there are critical sections of code and if system services are being called. In this case, it is preferable to do the processing in the ptask rather than in the ISR or LSR since it offers a bit more protection.

Another approach is discussed in section 6.7 Safe LSRs, which is using safe umode LSRs, uLSRs. This is an advanced technique and so is put off until the advanced chapter.

4.8.4 Writing ISRs

As a general rule, as little as possible should be done in ISRs. They should do the minimum necessary to reenable the IRQ and then invoke an LSR. The LSR should do the minimum necessary to start a task, preferably a utask. All ISR code should be in .sys.text, which is in sys_code.

Example of assembly ISR:

```
SECTION `.sys.text`:CODE:NOROOT
    THUMB
MyISR:
    ; ISR body or call C ISR here
    cpsid   f
    sb_INT_ENABLE
    pop     {pc}
```

Example of C ISR:

```
#pragma default_function_attributes = @ ".sys.text"
void MyISR(void)
{
   smx_ISR_ENTER();
   // ISR body here
   smx_ISR_EXIT();
}
#pragma default_function_attributes =
```

ISRs cannot make smx SSR calls, however they can make system service function calls, including smx_LSR_INVOKE(lsr, par). But, they must make direct calls not SVC calls. This is because IRQs have higher priority than the SVC exception. Attempting an SVC call from a higher priority IRQ will cause a Hard Fault. To avoid this, make sure that ISRs are preceded by xapi.h or xapip.h, not xapiu.h.

4.8.5 Exceptions

Because exceptions are internally generated, they are not as much of a concern for hacking as are interrupts. Handlers are implemented for the SVC, PendSV, MMF, and UF exceptions. The SVC exception occurs when an SVC N instruction is executed, as it is done in the system service shell functions in svc.c. SVC services are discussed in the next section. The PendSV exception is used for task switching by smx. The UsageFault exception occurs in ARMM8 due to a stack overflow.

The Memory Manage Fault exception occurs when the MPU detects a violation. The MMF handler halts operation if sb_handler_en is false. This normally is false for debugging (and set true in smx_Go() if SMX_DEBUG = 0). This allows seeing the call stack leading up to the MMF. Clicking on a level takes you to the instruction that called the level above. So clicking on the top level in the call stack window shows the line of code that caused the MMF. See Chapter 9 Debugging. Also there is information on MMF debugging in the pd2 discussion in Chapter 7 Partition Demos.

If SMX_DEBUG = 0, the smx_EM() error manager is called with an SMXE_MMF_VIOL error and severity = 1. Since this is considered to be an *irrecoverable error*, when smx_EMHook() is called at the end of smx_EM(), it stops the task causing the MMF and outputs “TASK STOPPED”. Stopping the task is necessary because attempting to return to the point of an MMF results in an infinite loop. Recovery code should be added to smx_EMHook() to delete, recreate, and restart the task or to reboot the system.

MMFs are not likely to occur in ISRs or trusted LSRs because they run in hmode, normally with Background Region ON. However safe LSRs can cause MMFs. In this case, the LSR host task should be stopped as well as any other tasks in the partition and the partition should be rebooted.

4.9 SVC API

4.9.1 SVC Calls

When a ptask is converted to a utask, it can no longer make direct system service calls. Instead, it must make indirect system service calls through the SVC Handler. This is accomplished via shell functions such as the following:

```
NI bool smxu_SemSignal(SCB_PTR sem)
{
  sb_SVC(SS)
}
```

where

```
#define sb_SVC(id) \
  { \
    __asm("mov r12, #0"); \
    __asm("svc %0" : : "i" (id)); \
  }
```

This shell function simply invokes the *SVC N* function with N = SS. r12 == 0 means that the function has 4 or less parameters; r12 == 1 means that it has more than 4 parameters. Abbreviations for system services are defined in the ssndx enum in svc.c:

```
enum ssndx {LIM, AS, ASL, HF, HM, LIF, MUCR, MUG, MUR, PICR, PIGW, PIPW, SS, ST,
            SEG, SSG, TCR, TSU, IRQM, IRQU, PK, PTMG, EM, END};
```

The abbreviations define indices into the smx_sst[] jump table, which is below ssndx in svc.c. The abbreviations are local to svc.c, hence they can be very short. The only requirements are that they be distinct and in the same order as the jump table. This makes it easy to add or to remove shell functions. After the jump table are the uSSR shell functions.

It is very important that only system services that are used in the application be included. Otherwise, unused services will be linked in. Another module, svctmplt.c, has the enum, jump table, and shell functions for every system service that can be permitted in umode. It is intended that these be copied from it to svc.c, as needed.

As discussed in a later section, it is possible to define partition-specific enums, system service table (SST), and shell functions that are included in the partition. In this case the partition does not require access to svc.c for system calls.

The svc.c code must be included in the MPA of every utask that makes system calls, unless the task is using a custom SST. Typically svc.c code is put into svc_code. C library code may also be included here. svc_data may be defined, but is likely to be empty. For partitions that need more than svc_code it can be included in ucom_code, which may include C library functions, partition

functions, and other shared code. Normally, ucom_code replaces the sys_code region in an active slot when a task is changed from a ptask to a utask. ucom_data contains common data and it replaces sys_data when a task is changed from a ptask to a utask.

To switch utask code to use indirect system service calls, it is necessary only to use #include “xapiu.h” instead of “include xapi.h” in its C modules. xapiu.h contains shell function prototypes and mapping macros, such as:

```
bool    smxu_SemSignal(SCB_PTR sem);

#define smx_SemSignal(sem)      smxu_SemSignal(sem)
```

Thus it is not necessary to change system call names in utask code, unless default parameters are being used in the calls. Macros cannot deal with default parameters – all parameters must be specified in a macro. If default parameters are being used, just replace smx_ with smxu_ in the utask code. Since smxu_xxxx() is a function, it can deal with default parameters, for example:

```
bool    smxu_SemTest(SCB_PTR sem, u32 timeout=0);
```

Other system calls besides smx calls are also implemented. These have prefixes such as sb_, mp_, etc., which become sbu_, mpu_, etc. Abbreviations for these calls are also included in the ssndx enum.

If you find the need to define system services of your own, define a unique abbreviation, such as MSC (my service call) to put in ssndx, and put the system call in the same position in smx_sst[]. Then create a shell function using the same abbreviation. See svc.h to pick the correct sb_SVC() macro, depending upon the number of parameters in your new service. Up to 7 parameters are currently supported. If your system service includes a heap call, it is necessary to use one of the sb_SVCH() macros. This is because a heap call may need to wait on the heap mutex. A double SVC call is necessary if the mutex wait succeeded.

4.9.2 SVC Call Mechanism

Figure 4.12 illustrates the system call mechanisms for both utasks and ptasks.

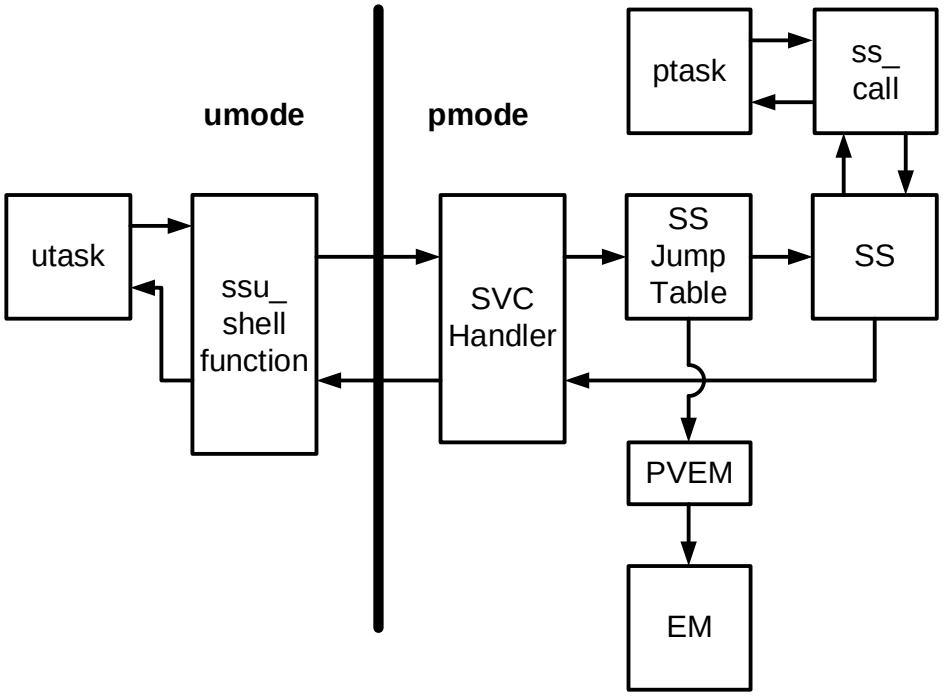

Figure 4.12 System Calls

As shown, utasks make indirect system calls via ssu_ shell functions[5]. Each shell function calls the SVC n instruction, where n identifies the system call. The SVC instruction causes an SVC exception, which causes the SVC Handler (SVCH) to run. This handler is located in xarmm_iar.s and is written in assembly language for best performance. Due to SVC exception handling by the processor, SVCH() runs in hmode with the main stack. As shown, it calls the system service (SS) via the smx_sst[] jump table located in svc.c. The return value from the system service is returned back through SVCH() and the shell function to the utask. Everything to the right of the heavy line runs in pmode.

Figure 4.12 also shows direct ss_ calls from ptasks. As can be seen, these are much simpler and consequently faster. If a ptask makes many direct system calls, it is possible that, when converted to a utask making indirect calls, it will run too slowly. In that case, it must remain a ptask or possibly be rewritten to make less system calls. Not shown in the figure is that ptasks can also make indirect calls via the ssu_ shell functions. This is done as for ucode, simply by adding #include “xapiu.h” ahead of the ptask code. Doing so before converting from a ptask to a utask is a good way to verify that the indirect call overhead will be acceptable.

For a simple system service, such as SemSignal(), 780 clocks are required for the smx_ version and 978 clocks are required for the smxu_ version – 198 more clocks or 25% overhead. For a

[5] ssu_ represents any system service, such as smxu_, sbu_ etc.

complex system service such as TaskCreate() the times are 3,876 and 4,777, respectively, leading to increases of 901 clocks or 23% overhead. The former is typical of services with 4 or less parameters and the latter is typical of services with more than 4 parameters. The latter requires copying parameters 5, 6, and 7 from the task stack to the main stack. Very few smx services require more than 4 parameters. In general, the performance hit for SVC calls is about 25%.

4.9.3 Restricted Services

Many system services are not desirable in umode due to potential hacker attacks. Allowing calls such as smx_SysPowerDown() or smx_LSRsOff() would obviously be a mistake. Such calls are omitted from svc.c. Note that ssndx[] in svc.c starts with LIM and ends with END. LIM is not a service. Instead, END is put into smx_sst[LIM]. SVCH() compares n to END and if not less, invokes smx_EM() with an SMXE_PRIV_VIOL error. This is treated as a recoverable error (sev = 0). After being logged and reported, control comes back to the point of call, but no service has been performed. Hence, a hacker cannot make up values of n in order to try to invoke restricted services. (Nor can he directly call such services without triggering an MMF.)

Since the privilege violation has been blocked from occurring, it is not considered to be a fatal error, and the task continues. This is appropriate during debug, but during operation it is probably a sign of hacking, and smx_EMHook() might take stronger action. However, the error is logged into the error and event buffers, so monitoring these may be sufficient to catch the hacker and to take appropriate action.

Restricted functions such as smx_SysPowerDown() generate an error message during compilation, if xapiu.h has been included:

```
"smx_SysPowerDown() not available in umode"
```

This helps to weed out restricted service calls when converting a ptask to a utask. As shown in Figure 4.12 a system call from a ptask goes directly to an SSR, and there are no disallowed service calls. As noted in the previous section, ptasks can also make indirect calls via the uSSR shell functions. Doing so, before converting to utasks, is a good way to weed out restricted services being made by the ptask. To do so, it might be necessary to split the task into a ptask that makes the restricted calls and then restarts itself as a utask. This is commonly done for initialization vs. normal operation.

4.9.4 Custom SSTs

svc.c contains smx_sst[] and shell functions for all services being used by utasks. As such, it needs to be fairly general, but normally quite a bit less than svctmplt.c, which contains all system services safe for umode.

As an even greater protection, custom ssndx enums, SST[]s, and shell functions can be defined for partitions when desirable. The following example shows how this can be done for partition pa:

```
/* pa system service table indices */
enum pa_ssndx {LIM, AS, MUF, MUG, SS, END};

/* pa system service table */
#pragma default_variable_attributes = @ ".sys.rodata"
```

```
const u32 pa_sst[] = {
  (u32)END,
  (u32)smx_SchedAutoStop,
  (u32)smx_MutexFree,
  (u32)smx_MutexGet,
  (u32)smx_SemSignal,
};

/* Initialization */
#pragma default_variable_attributes = @ ".sys.text"
void utpa_init(void)
{
  utpa = smx_TaskCreate(tmx1_utpa, TP2, 0, SMX_FL_UMODE, "utpa");
  mp_MPACreate(utpa, (MPA*)&mpa_tmplt_utpa);
  smx_TaskSet(utpa, SMX_ST_CBFUN, (u32)tmx1_utpa_cbf, 1);
  semx1 = smx_SemCreate(SMX_SEM_RSRC, 1, "semx1");
  mux1 = smx_MutexCreate(0, 2, "mux1");
  smx_TaskStart(utpa);
    …
}

/* utpa callback function */
void tmx1_utpa_cbf(u32 m)
{
  switch (m)
  {
    case SMX_CBF_START:
      smx_autostop = &pa_SchedAutoStop;
    case SMX_CBF_ENTER:
      smx_sstp = (u32*)&pa_sst;   /* change to pa_sst on entry to utpa*/
      break;
    case SMX_CBF_STOP:
    case SMX_CBF_EXIT:
      smx_sstp = (u32*)&smx_sst; /* change to smx_sst on exit from utpa */
  }
}

/************************************  UMODE PARTTITION A ************************************/

#include "svc.h"
#pragma default_function_attributes = @ ".utpa.text"

/* simulated xapipa.h */
Void    pa_SchedAutoStop(void);
bool    pa_MutexFree(MUCB_PTR mtx);
bool    pa_MutexGet(MUCB_PTR mtx, u32 timeout=0);
bool    pa_SemSignal(SCB_PTR sem);

#define smx_SchedAutoStop()        pa_SchedAutoStop()
```

```
#define smx_MutexFree(mtx)          pa_MutexFree(mtx)
#define smx_MutexGet(mtx, tmo)      pa_MutexGet(mtx, tmo)
#define smx_SemSignal(sem)          pa_SemSignal(sem)
/* end of simulated xapipa.h */

/* pa shell functions */
NI void pa_SchedAutoStop(void)
{
  sb_SVC(AS)
}

NI bool pa_MutexFree(MUCB_PTR mtx)
{
  sb_SVC(MUF)
}

NI bool pa_MutexGet(MUCB_PTR mtx, u32 timeout)
{
  sb_SVC(MUG)
}

NI bool pa_SemSignal(SCB_PTR sem)
{
  sb_SVC(SS)
}

void utpa_main(void)
{
  if (!smx_MutexGet(mux1, 0))
    tfailu();
  if (!smx_SemSignal(semx1))
    tfailu();
  if (!smx_MutexFree(mux1))
    tfailu();
}
#pragma default_function_attributes =
```

The first step is to define the xapipa.h, as shown above, which will convert smx calls to pa_ calls. Then define pa_ssndx and pa_sst[] as shown above. Note that there are only 3 smx services plus autostop. Many partitions may need very few services, like this, so the less a hacker has to work with, the better. (If a partition has a custom SST, it is not given access to svc.c.) Next the shell functions are defined. These all have less than 4 parameters, so only the sb_SVC() macro is used. The tmx1 function, which runs in pmode, shows how the utpa task is created, given an MPA, and given the tmx1_utpa_cbf callback function with utpa->flags.hookd set. It also creates the mux1 mutex and the sem1 semaphore, then starts the utpa task.

Ignoring autostop, for the moment, note that when utpa starts and each time it is resumed, smx_sstp is set to &pa_sst. Hence the pa_sst[] jump table rather than smx_sst[] jump table is used whenever utpa is running. Note that when utpa stops running smx_sstp is set to &smx_sst.

This restores use of smx_sst[] for utasks that do not have custom SSTs. utpa_main() shows how the pa services are called.

It is important to note that data and code above the UMODE PARTITION A are in .sys regions and code below is in the utpa_text region. The pa_sst[] table is used by SVCH() and is inaccessible by utpa. The pa shell functions are in utpa_text. Since there are very few, they do not add significant memory overhead to utpa_text. Task utpa does not need, nor should it have access to svc.c.

The above example is somewhat complicated by autostop, shown here just to illustrate how it can be done. Autostop is necessary only if a task is allowed to run off its last }, as utpa does. Such a task is called a *one-shot task*. Many partitions may not use one-shot tasks and thus not require autostop. However, if one does, note that tmx1_utpa_cbf calls

```
smx_ChangeAutoStop((u32)pa_SchedAutoStop);
```

on START. A complexity of the ARMM architecture is that a task is started by creating an exception frame, then doing an exception return. The LR slot in the exception frame contains the smxu_SchedAutoStop() address for a utask. So on START this is changed to pa_SchedAutoStop(). It is not necessary to put the LR slot back because the exception frame is in the task stack, and it is lost on autostop since the task stack is released to the stack pool.

4.9.5 Partially Restricted Services

Some system services are needed in umode, but they are limited in what they can do. For example, a parent task is allowed to start or stop one of its child tasks, but a child task is not allowed to start or stop its parent task. Limitations also apply to what interrupts a task can mask or unmask. These limitations have been added to smx. See Appendix D: SMX API Limitations and the smx Reference Manual for specifics.

4.9.6 Mixed Code Modules

It often is not desirable to segregate pcode into p modules and ucode into u modules. There may be a connection between a p function and a u function and thus it is desirable to keep them together. For example, a task may start in pmode in order to initialize its partition, then restart itself in umode to run. Thus there is need for both pmain() and umain() code and it is desirable to have one adjacent to the other.

This can be accomplished as follows:

```
#include "smx.h "

// pcode

/*+++++++++++++++++++++++++ LIMITED SVC API (UMODE) ++++++++++++++++++++++++*/
#include "xapiu.h"

// ucode

/*+++++++++++++++++++++++++ FULL DIRECT API (PMODE) ++++++++++++++++++++++++*/
#include "xapip.h"

// back to pcode
```

In the above, smx.h includes xapi.h, so pcode is first. Then xapiu.h switches to ucode. Then xapip.h switches back to pcode. It does this by reversing the mappings of xapiu.h. xapiu.h can be included next, then xapip.h after it, alternating as often as necessary. Headings can be included, as shown in order to make header file changes more prominent and thus avoid not seeing them.

If there are many alterations of pcode and ucode in a module, alternating #include statements may become messy. This can be avoided by using the *smx_* prefix for direct smx calls and the *smxu_* prefix for indirect smxu calls via the SVC Handler, as follows:

```
#include "xapiu.h"
#include "xapip.h"
…
void tm03(void)
{
  ut2a = smx_TaskCreate((FUN_PTR)tm03_ut2a, TP2, 0, SMX_FL_UMODE, "ut2a");
  smx_TaskStart(ut2a);
 …
#pragma default_function_attributes = @ ".ut2a.text"
void tm03_ut2a(void)
{
  bool rv;
  rv = smxu_SemTest(sbr2, INF);
```

xapiu.h defines the smxu_ function prototypes and xapip.h undefs the mapping macros in xapiu.h. Thus, they are not being used.

In the above example, it is helpful during debug to see the utask code immediately after the pmode code that created and started the utask. This makes it easier to follow interactions between the two, without having to scroll up and down in the module or switch from module to module. The pragma puts the utask code into its code segment.

Middleware modules run in umode, so generally a main header file in them includes xapiu.h so that all its C files start with ucode. Then xapip.h is included ahead of ISRs, LSRs, and initialization code, and xapiu.h is included again after them for the remaining code in the file. This technique works well for a subsystem or library that is mostly ucode.

4.10 Processor Control

4.10.1 smx Task Switching[6]

For the Cortex-M architecture, following execution of an smx System Service Routine, SSR, if the smx_sched flag is set, or if smx_lqctr is not 0, the PendSV Handler is triggered, causing a PendSV exception[7]. The smx_sched flag indicates that the current task should be stopped, suspended, or its priority tested vs. the top task in the ready queue. A non-zero smx_lqctr indicates that an LSR is ready to run. LSRs are dispatched ahead of all tasks.

---

[6] This is a simplified description – several complexities are omitted.

[7] When an SSR is called from SVCH(), operation is a bit different. See Section 4.9.2 SVC Call Mechanism.

When a PendSV exception occurs, the processor automatically switches to hmode and to the main stack. It first *stacks* the R0-R3, R12, LR, PC, and XPSR registers on the task stack, TS. The processor is now in handler mode and it is running the smx_PendSV_Handler() (PSVH()). If smx_lqctr > 0, PSVH() first calls the LSR scheduler to run LSRs in the LSR queue, smx_lq. When all LSRs have run, PSVH() tests the smx_sched flag: if 0 or if the current task is still the top priority ready task it returns to the current task, else it calls the task scheduler. The task scheduler selects the *top task* (longest waiting at highest priority) to run.

After the task scheduler runs, it returns to the *tail* of PSVH(). It is important to note that it will be doing so with a different *current task* since there has been a task switch. PSVH() does some processing, then returns to *thread mode*. If SMX_CFG_SSMX is set, the nPRIV bit in the CONTROL register is set equal to smx_ct->flags.umode. This determines whether the task runs as a utask or as a ptask. In either case, the task's stack is now used and if the task had been suspended, R0-R3, R12, LR, PC, and XPSR are *unstacked* from it before returning to the point of suspension. Otherwise, if the task is being started, unstacking is not required and control goes to the beginning of its main function.

4.10.2 From pmode to umode

If the ct->flags.umode flag is set in a task's TCB, the task is a utask; if the flag is 0 the task is a ptask. Normally the SMX_FL_UMODE flag is used to set its umode flag when a task is created:

```
ut2a = smx_TaskCreate(tm03_ut2a, TP2, 0, SMX_FL_UMODE, "ut2a");
```

Otherwise, the task created is a pmode task:

```
t2a = smx_TaskCreate(tm02_t2a, TP2, 0, 0, "t2a");
```

Alternatively, following task creation, while in pmode, the flag can be changed with:

```
smx_TaskSet(task, SMX_ST_UMODE, 1);
```

A ptask can set its own umode flag. Hence, a task can start running in pmode and then switch itself to umode, as follows:

```
#include “xapi.h”

#pragma default_function_attributes = @ ".t2a.text"
 void tm10_t2a(void)
 {
      /* perform pmode initialization */
      …
      /* create alias task */
      ut2a = t2a;

      /* Change template, set umode, and restart with code tm10_ut2a. */
      smx_TaskLock();
      smx_TaskSet(ut2a, SMX_ST_UMODE, 1);
      mp_MPACreate(ut2a, (MPA*)&mpa_tmplt_ut2a);
      ut2a->name = "ut2a";
      smx_TaskStartNew(ut2a, 0, TP2, tm10_ut2a);
}
```

```
#include “xapiu.h”

#pragma default_function_attributes = @ ".ut2a.text"
void tm10_ut2a(u32 par)
{
    /* Run in umode, after restarting. */
}
```

In this example, tm10_t2a() is the pmode main function for t2a, and tm10_ut2a() is the umode main function for ut2a. TaskSet() allows changing task t2a to task ut2a. (t2a and ut2a are the same task with different names – i.e. the TCB does not change.) Then MPACreate() allows changing to mpa_tmplt_ut2a. smx_TaskStartNew() allows changing a task’s main function. It also allows passing a parameter to the new main function and changing the task’s priority. Note that the task must be locked so it cannot be preempted while changes are being made. It is unnecessary to unlock the task because the scheduler will do that following smx_TaskStartNew().

Typically the top task of a partition may do quite a bit of initialization that can be done only in pmode. When this is complete, it switches itself to umode where it completes the initialization, such as creating and starting child tasks and then starts its normal operation.

4.10.3 Memory Protection Arrays, MPAs

Figure 4.13 shows an MPA for a ptask.

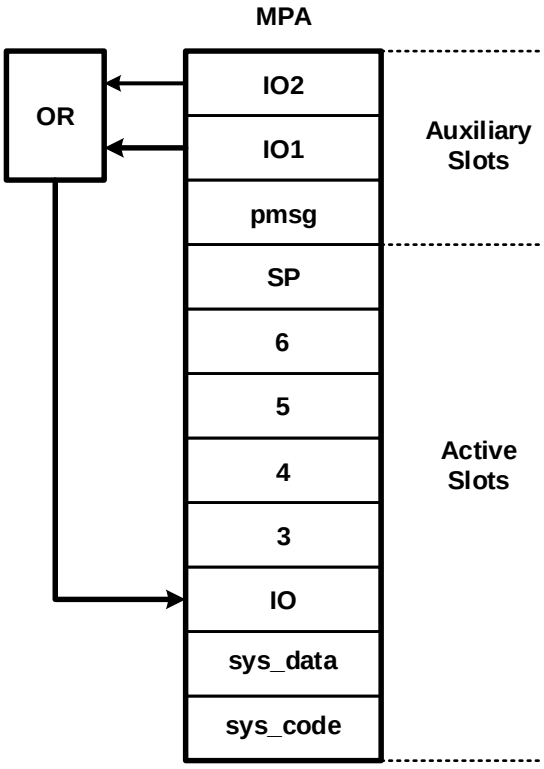

Figure 4.13 MPA for ptask

Notice that slots 1 and 0 hold sys_data and sys_code. This is necessary because Background Region, BR, is always off when a ptask runs. Slot 2 is an IO slot shared between IO1 and IO2. Slots 3 thru 6 hold ptask regions. Slot 7 holds the stack pointer region. Slots 8 thru 10 are *auxiliary slots* – i.e. outside of the MPU. Slot 8 is reserved for protected messages, pmsgs, and slots 9 and 10 hold IO regions that can be swapped to the active IO slot 2, as needed.

Figure 4.14 shows the MPA for the same task after it has been converted to a utask:

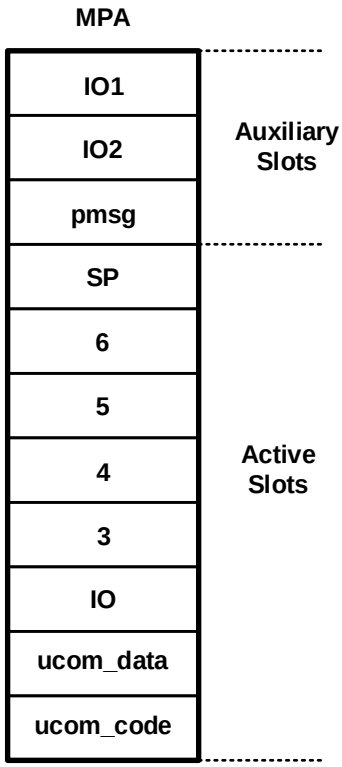

Figure 4.14 MPA for utask

Notice that ucom_code has replaced sys_code and ucom_data has replaced sys_data. In this case, BR is on in umode. BR is not effective in umode but becomes effective when an interrupt or an exception causes a switch to hmode. If ucom_data is not necessary, IO2 and IO1 can have their own active slots and switching between them is no longer necessary. Also the MPA would be smaller since two of the auxiliary IO regions are no longer needed. In both cases, slots 3-6 are available for task regions. How these are used will vary from task to task. Most likely, two slots will be used for task_code and task_data and the other two slots are available for other regions, such as dynamic regions or other common regions.

Note: The above represents conversion of a ptask into a utask, during development, and changing its template, accordingly. This is not the same thing as discussed in section 4.10.2 From pmode to umode. However, these figures also apply to that discussion since a switched task can change from a pmode template to a umode template.

4.10.4 What Good are ptasks?

Unfortunately, the MPU is totally unprotected in pmode. It is within the System Control Space (SCS), which is in the Private Peripheral Bus (PPB) memory area. Accesses to the PPB bypass the MPU. Hence the PPB, and all within it, have no protection from malignant pcode. If malware gains control of a ptask, it need only turn off the MPU or turn on BR in order to access whatever the hacker pleases. Thus, ptasks are not as secure as utasks[8]. What then are ptasks good for?

They are good for:

- Speed – direct access to all system and BSP services.
- Fine-tuned, mission-critical code that no one wants to change.
- Better reliability.
- Stepping stones in the conversion of ptasks to utasks.
- Cache and memory attribute control.

The fact that SecureSMX provides support for both ptasks and utasks and that they are largely treated equivalently means that it is easier to achieve a proper balance between security, performance, reliability, and other factors for an embedded system. As part of this equivalence, ptasks can make indirect SVC system calls and can also use portals, if desired.

ptasks are very important for improving the reliability and safety of legacy code, and they serve as stepping stones to moving to umode. Often, there is one area that presents a security or safety problem. It might be adding networking or it might be code that has proven vulnerable to hacking. The partition demos, pd0 to pd4, presented in Chapter 7 show a step-by-step process to define a partition in pmode and move it into umode. As time goes on, other partitions can be defined and moved into umode, thus steadily improving system security and safety. SecureSMX is designed to support incremental improvement.

4.10.5 Hacking a ptask

Although ptasks are not as secure as utasks, if a hacker gains access to a ptask, he is still not home free. Suppose, for example, that a ptask monitors a sensor. The sensor is connected to an A/D converter, which the ptask periodically samples. Assume that an 8-bit A/D converter is being used and that the samples are stored in a large buffer for later processing. In a typical attack, the hacker replaces the sensor with a D/A converter connected to his computer. Then, by synchronizing his D/A converter with the A/D converter, he can load a small program into the buffer, conversion by conversion.

Then he needs to cause the system code to branch to his program, and he is in! Normally this is done by replacing a return address in the task stack with the address of the just-loaded malware. However it is done, let's assume the hacker does it and the next subroutine return branches to his code in the buffer. But wait! The buffer is in an execute never, XN, region enforced by the MPU, so the branch will cause an MMF, and supervisory code will take over. Too bad for the hacker – foiled again!

[8] Because a utask cannot access the PPB, and BR has no effect in umode.

4.11 Dynamic Features

Previous sections of this manual have covered *static* features – i.e. those determined at compile and link time. This section presents dynamic features that are used while running.

4.11.1 eheap and smx_Heap

eheap is an RTOS-agnostic heap developed by Micro Digital specifically for embedded systems. It supports multiple heaps, in a simple manner, and can support simple, small heaps with a single bin up to large, complex heaps with up to 31 bins. Hence it is ideal for situations where multiple heaps of various sizes are required. It has other useful features for embedded systems such as:

- Configurable bin structures (number and sizes of bins).
- Aligned allocations on 2^n boundaries.
- ARMM7 region allocations with automatic subregion disables.
- Large bin sorting during idle periods.
- Integrated small-block, block pools for object-oriented programming objects.
- Automatic heap and bin scanning and fixing for broken links.
- Manual or automatic merge control.
- Fragmentation recovery.
- Debug chunks to assist in finding leaks and other problems.

smx provides smx_Heap shell functions which add task-safe support to eheap functions. Task-safe operation is achieved via a mutex per heap, rather than the SSR_ENTER() / SSR_EXIT() mechanism used by other smx services. This is necessary because heap calls can be slow. Using a mutex allows higher-priority tasks to run while a heap task waits for a heap operation to complete[9]. Using a heap mutex per heap is necessary to decouple partitions so that one partition cannot cause another partition to wait excessively for access to its own heap.

When a heap is initialized, if the mode parameter is NULL, the heap will not be mutex-protected. This reduces overhead for partition heaps in which operations cannot be preempted (e.g. only one task does heap operations). In this case, heap operations are still logged and smx heap error checking is still performed.

For C++ partitions where ultimate speed is necessary, integrated block pools can be added to eheap for small objects. In this case, there is no logging and no smx error checking. However, eheap still performs numerous error checks, and approximately 11 error types are reported in hvp[hn]->errno.

A single timeout, smx_htmo, applies to all heap accesses. It is intended only to prevent permanent task hang-ups and should be a long time. During debug it is best to set it to SMX_TMO_INF, in order to avoid heap timeouts, which may cause problems.

[9] Of course, there is only one processor, so the heap operation does not run while the higher-priority task runs. The important thing is that the higher-priority task is not forced to wait for a potentially long heap operation.

See the smx Reference Manual for more information on using smx heaps and creating and using multiple heaps. See the eheap User's Guide for more information on using eheap. This will help you understand the following sections.

4.11.2 The Need for Multiple Heaps

Using heaps in modern application code is popular and it is necessary to support object-oriented languages. This is a growing trend as embedded systems become more complex and are expected to do more functions – especially in IoT systems. Also, some middleware uses heaps.

It is unacceptable for utasks to have direct access to the main heap. A hacker could easily bring down the whole system simply by exhausting or corrupting the main heap. It generally is also not acceptable to have a heap that two or more partitions share, since malware in one partition could corrupt blocks owned by another partition, or it could access sensitive data of the other partition. Thus, each partition that needs a heap, must have its own heap. These heaps will generally be small, but not necessarily so.

An exception to the above is if one partition gets blocks from a heap, and another partition releases them back to the heap, then obviously the heap must be shared between the partitions. SecureSMX provides *protected messages*, *pmsgs*, for this purpose. Special services are provided to get, release, and handle pmsgs so that the task does not have direct access to the shared heap. pmsgs are used for portals, which are discussed in the next chapter.

4.11.3 Allocating Heap Space

The main heap is in sys_data and it is used by the smx kernel. Task stacks are allocated from it, as well as other objects that smx needs. Space for the main heap is allocated by the mheap block in the linker command file.

There are two ways to allocate heaps that are dedicated to partitions. One is to define the heap block in the linker command file, as follows:

```
define block heap2      with size = 0x1000,   alignment = 16  { };
```

This is quite a small heap (4096 bytes) but adequate for some partitions. Then in the initialization code for the heap:

```
hsa = (u8*)__section_begin("mheap");
hsz = (u32)__section_size("mheap");
```

The second way to allocate a heap is illustrated in Figure 4.15. In this case, it is allocated at run time from the main heap (or another heap), using:

```
hsz = 0x1000;
hsa = smx_PBlockGetHeap(hsz, sn, DATARW, “heap1”, mheap);
```

where sn is the slot number in the current task's MPA to place the new heap region. In the figure below, heap1 calls from TaskA operate only on heap1 and cannot go outside of it. TaskA can access the main heap only for a protected block or message, as shown by the dashed line, and cannot go outside of the protected block or message. (Protected blocks and messages are discussed in sections 4.11.10 Protected Data Blocks and 4.11.11 Protected Messages, respectively.) Hence the main heap is protected from TaskA.

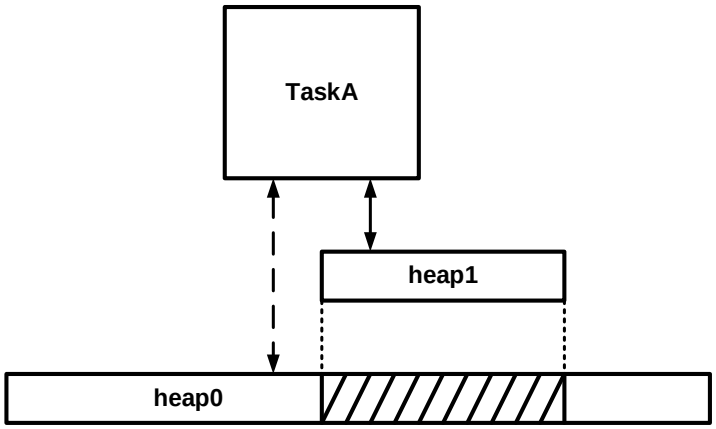

Figure 4.15 Dedicated Heap from Main Heap

Memory for a dedicated heap can also be a static block of memory, a block from a block pool, or Task Local Storage, TLS (see section 4.11.8 Task Local Storage).

4.11.4 Creating a Heap

Each partition that needs a heap creates its own heap. To do so, the partition must define a binsz[] array, allocate space for the bins, define an eheap variable structure, EHV, and get space for the heap, as follows:

```
/*==========================================================================
                              FAST HEAP (hn = 1) from SDRAM
==========================================================================*/
/* Medium bin size array consisting of five large bins with no SBA. The top
   bin contains chunks >= 2048 bytes. The array ends with 0xFFFFFFFF. */

u32 const binsz1[] =
/*bin  0      1       2       3       4     end */
     {24,  512, 1024, 1536, 2048,  -1};

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

HBCB  bin1[(sizeof(binsz1)/4)-1];   /* heap1 bins */

#if EH_STATS
u32   bnum1[(sizeof(binsz1)/4)-1]; /* number of chunks per bin */
u32   bsum1[(sizeof(binsz1)/4)-1]; /* sum of chunk sizes per bin */
#endif

EHV   hv1; /* heap1 variable array */
```

```
/* heap1 create and initialize control variables */
void h1_init(void)
{
  u8* hsa; /* heap starting address */
  u32 hsz; /* heap size */

  /* get heap space for h1 allocated in linker command file */
  hsa = (u8*)__section_begin("heap1");
  hsz = (u32)__section_size("heap1");

  /* initialize heap */
  smx_HeapInit(hsz, 0, hsa, &hv1, (u32*)binsz1, (HBCB*)bin1, EH_NORM, "heap1");
}
```

The EHV structure contains all of the variables that control the heap. The top task for a partition initializes certain variables, then calls smx_HeapInit(), as shown above. In this example, the second parameter is the donor chunk size, which is not used if there are no small bins. For simplicity, the above example does not show a Small Bin Array, SBA, but one can be added. An SBA consists of a single bin per chunk size, starting at 24 — e.g. 24, 32, 40, 48. An SBA provides faster allocations of small blocks since no bin searching is required.

This particular example is a simple heap, which has better performance than a one-bin heap (the smallest possible heap). In effect, each bin acts like a subheap and contains chunks starting at the bin size. So, for example bin1[0] contains chunks from 24 to 504 bytes in 8-byte increments = 61 different chunk sizes. The chunks in this bin are sorted, by increasing size, during idle periods, in order to speed up finding the first big-enough chunk for a desired block size.

If C++ is being used in a partition, then the partition's heap creation and initialization must be added to smx_HeapsInit(), which creates mheap. This function is called by C startup code prior to calling C++ initializers, which need the new heap. If C++ is used in the partition, then integrated block pools are normally added to provide fast allocation and deallocation for small objects.

4.11.5 Heap Manager

Each partition having a dedicated heap defines its own heap manager and assigns its address to the EHV mgr field. The heap manager does functions such as:

- Automatic chunk merge control to prevent excessive fragmentation.
- Automatic heap and bin scan and repair.
- Bin sorting to improve bin performance.

Simple partition heaps may not require heap managers. Large, heavily used heaps, such as mheap are likely to use managers for the above purposes to improve performance and reliability.

4.11.6 Task Stacks

smx provides three kinds of task stacks:

- Heap stacks (permanent)
- Preallocated stacks (permanent)
- Stack pool stacks (shared)

A heap stack block is allocated by smx_TaskCreate() from heapn, if the stack size, ssz, is non-zero and bp is zero. If bp != 0, bp is assumed to point to a preallocated stack block of size = ssz. If ssz == 0, the stack block is allocated from the stack pool when the task is dispatched and returned to the stack pool when the task is stopped.

Each stack block contains the following, in order of increasing addresses:

- Stack pad (optional)
- Stack
- Register Save Area (RSA)
- Task Local Storage (TLS) (optional)

A preallocated stack block must come from an existing task region, so no stack region is created for it. It also must conform to MPU size and alignment requirements. In this case, the top MPU slot is free for another region. For ARMM8, when a ptask with a heap stack is created from mheap, no stack region is created for it. This is because the stack region would overlap the sys_data region, which contains mheap, thus causing an MMF. If the stack block comes from another heap, a stack region can be created for it. If a utask is created, a stack region can be created for it because the sys_data region is either not present in the MPU for utasks or it allows privileged access, only. Thus region overlapping is not a problem.

If the stack is allocated from a heap, the stack block size is increased to meet the MPU size and alignment requirements, if does not already. The region for the heap stack is automatically created and stored in the task's TCB. When an MPA is created for the task, the stack region is loaded into the top active slot of the MPA and whenever the task is started or resumed, the stack region is loaded from the MPA into the top slot of the MPU.

For a heap stack, if the stack block size is increased, the additional space is put into the stack, itself. Hence task->ssz may be greater than the requested stack size in smx_TaskCreate(). As a consequence, in the event of a stack overflow, look at task->ssz, not at the requested size, to decide upon a new stack size. For ARMM8, increasing the requested stack size above task->ssz, may result in a slightly larger stack than expected, but for ARMM7 it could result in a much larger stack than expected. This is because the stack block size will increase by a subregion size. For example, if the current size is 1024*5/8 - 8-byte pad - RSA = 600 bytes and it is increased by 16 bytes (i.e. 616 requested in smx_TaskCreate()), another subregion must be added, so the new stack size becomes 1024*6/8 - 8 - 32 = 728 bytes – quite a bit more than expected! If fast RAM is scarce, it may be better to look for ways to reduce stack usage in the task, rather than increasing stack size.

If the stack size parameter is zero, a pool stack, or temporary stack, is assigned to a task by the task scheduler when the task first starts. The stack block size is determined by

STACK_BLK_SIZE in acfg.h, which must be a power of two for ARMM7 and a multiple of 32 for ARMM8. Then:

```
SMX_SIZE_STACK = (SMX_SIZE_STACK_BLK - SMX_SIZE_STACK_PAD
                                     - SMX_RSA_SIZE)
```

A temporary stack cannot have Task Local Storage, so TLS_SIZE is not included above. In addition, each stack in the stack pool must meet the MPU size and alignment requirements. For ARMM8, the stack pool comes from the sys_data region. Thus for a ptask, no stack region is created for it due to the no region overlap requirement. For ARMM8 utasks and all ARMM7 tasks, a stack region is automatically created and loaded into the top active slot of the task's MPA. Then, whenever the task is started or resumed, the stack region is loaded from the MPA into the top slot of the MPU.

For stack regions in the top MPU slot, overflows and underflows are immediately detected and reported as Memory Manage Faults (MMFs). If STACK_PAD_SIZE in acfg.h is not 0, stack overflow will not be caught until the stack pad is also overflowed (but it will show in smxAware). Stack pads are normally used during debugging so that the system will keep running despite stack overflows. Normal smx stack overflow detection by the scheduler is still enabled and reported, the first time, as SMXE_STK_OVFL by the smx error manager.

Due to the heap allocation method for a ARMM7 MPU, the following stack block sizes are available for 2048 bytes or less:

32			
40	48	56	64
80	96	112	128
160	192	224	256
640	768	896	1024
1280	1536	1792	2048

For example, if a stack block size of 232 bytes is needed, the closest larger size from the above table is 256, thus excess[10] space would be 24 bytes. If a stack block size of 1600 bytes were needed, the closest larger size from the above table is 1792, thus excess space would be 192 bytes. As apparent from the table, potential excess space gets larger, the larger the stack. For ARMM8, excess space is always less than 32 bytes.

[10] The term "excess" space is used instead of "wasted" space because the excess space is put into the stack where it might be used.

4.11.7 PSPLIM and MSPLIM

ARMM8 defines two new registers: PSPLIM and MSPLIM. The first is an overflow limit for the current process stack. The second is an overflow limit for the main stack.

PSPLIM is set equal to task->spp by smx_MPULoad() when a task is started or equal to clsr->stp by smx_MPULoad() when a safe LSR is started. During operation, if PSP >= PSPLIM, a Usage Fault occurs. MSPLIM is set = CSTACK + SB_SIZE_MS_PAD in the startup code (__low_level_init()). If MSP > MSPLIM a Usage Fault occurs. In both cases:

```
Usage Fault -> smx_UF_Handler() -> sb_UFM() -> smx_EM() -> smx_EMHook() -> halt
```

smx_UF_Handler(), in xarmm_iar.s, is written in assembly language in order to avoid an infinite stack-overflow/usage-fault loop caused by register pushes for C functions. If sb_handler_en is false, it halts. Otherwise, it first moves MSPLIM to the start of CSTACK in order to avoid further MSPLIM usage faults. Then it calls sb_UFM(). SB_SIZE_MS_PAD must be large enough to allow sb_UFM(), smx_EM(), and smx_EMHook() to run without MSPLIM usage faults. sb_UFM() tests for stack overflow. If PSP >= PSPLIM, it calls smx_EM(SMXE_STK_OVFL, 1); if MSP >= MSPLIM, it calls smx_EM(SMXE_MSTK_OVFL, 2); if neither, it calls smx_EM(SMXE_UF_VIOL, 1).

smx_EM() records the error, then calls smx_EMHook(), which stops the current task if sev == 1 or calls aexit() if sev == 2. Following this, recovery code should be added to reboot the task or aexit() should reboot the system. Unfortunately an exception stack is not created when a PSPLIM or MSPLIM violation occurs. Creating an exception stack and attempting an exception return does not work, so the processor is halted, instead, and a watchdog timeout is necessary to restart it and reboot the system.

Using PSPLIM can allow MPU[7] to be used for another purpose other than as a task stack region. In this case, the task stack would be put into the task_data region. However, PSPLIM does not protect against code execution from the task stack, which is a prevalent attack vector. Also it seems that simply rebooting the task or safe LSR experiencing the stack overflow is not possible and the entire system must be rebooted.

4.11.8 Task Local Storage

For a permanent stack, smx_TaskCreate() allows adding a Task Local Storage (TLS) area that follows the Register Save Area (RSA) and is part of the stack region, as follows:

```
smx_TaskCreate(fun, pri, tlssz_ssz, fl_hn, name);
```

tlssz_ssz is a double parameter: the upper 16 bits define the TLS size, tlssz, and the lower 16 bits define the stack size, ssz.[11] Both can sizes be up to 64 KB. TLS is available only if ssz > 0 – i.e. the task stack must be a permanent stack from heap, heapn (hn = fl_hn & 0xF) or a preallocated stack.

The TLS pointer is stored in the TCB of the task. It can be accessed as follows:

```
tlsp = (u8*)smx_TaskPeek(task, SMX_PK_TLSP);
```

[11] Some functions combine parameters, since the SVC Handler supports only up to 7 parameters.

This operation is permitted for utasks as well as ptasks.

TLS allows saving an MPU slot by combining space for static local variables with task stack space into a single region. In order to do so, local static variables must be organized into structures and arrays. If organized into a single structure, use:

```
tlsp->fieldn;
```

to access fieldn in the TLS. Using a TLS can save using an MPU slot for a task_data region. This is particularly recommended when a task has very few static variables. With the Cortex-M architecture, there is no time penalty to access a variable vs. a field in a structure – both require two LDR instructions. An index is added for a field, and no index is added for a variable. Depending upon the compiler and its settings, multiple field accesses may save LDR instructions vs. multiple variable accesses, and thus be faster.

Another possible use for a TLS is as a dedicated heap if a partition has only one task.

4.11.9 Dynamic Regions

There are times when it is beneficial if regions can be created during run time rather than defined statically at compile time. A good example is a case where a buffer size depends upon installation parameters and thus is not known at compile time. At initialization time, the code has already been compiled and linked, so it is not possible to change a static region. Hence a dynamic region is needed. Dynamic regions can only be created in pmode.

To create a dynamic region in the template for taskA:

```
mpa_tmplt_taskA[sn] = MP_DYN_RGN(dpr[n]);
```

where dpr[] is an array of n dynamic regions and MP_DYN_RGN() loads the address of dpr[n] and sets DRT flag in template slot sn. The DRT is bit 31 in the RASR, which is an unused bit, or = 0xFFFFFFFF in the RLAR, which is an illegal value. Thus a dynamic template slot does not look like a normal template slot.

SecureSMX provides three functions to dynamically create data regions, from pmode:

```
u8*   mp_RegionGetHeapR(rp, sz, sn, attr, name, hn);
u8*   mp_RegionGetPoolR(rp, pool, sn, attr, name);
bool  mp_RegionMakeR(rp, bp, sz, sn, attr, name);
```

where rp = &dpr[n], sz is the desired data block size, sn is the MPU slot number, attr is the attributes macro, name is an optional name for the region. For get from heap, hn is the heap number; for get from pool, pool is the pool handle; for make block, bp is the block pointer. A region can be made from any block pointed to by bp. This can be a dynamically allocated block or a static block such as sblk[128]. Both pool blocks and static blocks must meet region size and alignment requirements.

When a task's MPA is loaded, dynamic slots are loaded along with the static slots from the template. Hence, the above Region functions need to be called before the task's MPA is created. The above functions are **not smx SSRs** and thus they are not task-safe. They should be used only during initialization or while the calling task is *locked*.

The following illustrates creating a dynamic region in a template:

```
MPA mpa_tmplt_ut2a =
{
  RGN(0 | RA("ucom_data") | V, DATARW | RSI("ucom_data") | EN, "ucom_data"),
  RGN(1 | RA("ucom_code") | V, CODE   | RSI("ucom_code") | EN, "ucom_code"),
  RGN(2 | RA("ut2a_data") | V, DATARW | RSI("ut2a_data") | N7 | EN, “ut2a_data”),
  RGN(3 | RA("ut2a_code") | V, CODE   | RSI("ut2a_code") | EN, "ut2a_code"),
  MP_DYN_RGN(dpr[0]),   /* dynamic region */
  RGN(5 | V, 0, "stack"),     /* reserved for task stack */
};

MPR dpr[3];   /* dynamic protection regions */

bp = mp_RegionGetHeapR(&dpr[0], 200, 4, DATARW, "block1", mheap);
```

In the above, a dynamic region is obtained from mheap and loaded into dpr[0]. When mpa_tmplt_ut2a is loaded into task ut2a’s MPA, dpr[0] is loaded into MPA[4].

As with TLS, described in section 4.11.8 Task Local Storage, data must be organized into arrays and structs, or the entire region might be used as a working buffer. The major advantage of a dynamic region is that it does not require defining a section in the code and a block in the linker command file. It is thus easier to use and it is more flexible with regard to changing requirements.

4.11.10 Protected Data Blocks

Protected blocks are referred to as *pblocks*. They differ from the dynamic regions defined in the previous section in that they can be created and released at any time by utasks or ptasks. Protected data blocks are good for heaps, temporary buffers, work areas, structures, etc. that may vary in size.

The following smx SSRs allow creating and releasing protected data blocks:

```
u8*     smx_PBlockGetHeap(sz, sn, attr, name, hn);
u8*     smx_PBlockGetPool(pool, sn, attr, name);
bool    smx_PBlockMake(bp, sz, sn, attr, name);
bool    smx_PBlockRelHeap(bp, sn, hn);
bool    smx_PBlockRelPool(bp, sn, pool, clrsz);
```

Where sz is the desired data block size, sn is the MPU slot number for its region, attr is the attributes macro (e.g. DATARW), and name is an optional name for the block. For get from heap, hn is the heap number; for get from pool, pool is the pool handle; for make block, bp is the block pointer. For the release functions, bp is the block pointer returned by one of the get functions, and clrsz specifies how many bytes to clear after the free block link in the first word of the block. PBlock functions are similar to the region functions of the previous section. They are smx SSRs, so they can be safely called from either utasks or ptasks, at any time.

Basically, a pblock is obtained from a heap or a pool or made from a static block. A region is created for it and loaded into MPU[sn+fas][12] and into MPA[sn] of the current task. If MP_MPA_DEV, *name* is also loaded into MPA[sn]. The heap can be any heap, including the main heap, mheap. The block pool can be any block pool; the block can be any block, both as long as blocks meet size and alignment requirements. These are safe because if a hacker penetrates the task, the MPU prevents him from accessing memory outside of the protected block.

For more information on PBlock functions, see Appendix A.2.

4.11.11 Protected Messages

Protected messages are referred to as *pmsgs*. They are pblocks that can be sent from partition to partition via smx *message exchanges*. They carry their own region information with them. Thus they are portable regions.

The following smx SSRs are provided for protected messages:

```
MCB_PTR   smx_PMsgGetHeap(sz, bpp, sn, attr, hn, mhp);
MCB_PTR   smx_PMsgGetPool(pool, bpp, sn, attr, mhp);
MCB_PTR   smx_PMsgMake(bp, sz, sn, attr, name, mhp);
bool      smx_PMsgRel(mhp, clrsz);
MCB_PTR   smx_PMsgReceive(xchg, bpp, sn, timeout, mhp);
void      smx_PMsgReceiveStop(xchg, bpp, sn, timeout, mhp);
bool      smx_PMsgReply(pmsg);
bool      smx_PMsgSend(pmsg, xchg, pri, rxchg);
bool      smx_PMsgSendB(pmsg, xchg, pri, rxchg);
```

Where sz is the desired message data block size, bpp is the location in which to put the message block pointer, sn is the MPU slot number for the message block region, and attr is the attribute macro (e.g. DATARW). For get from heap, hn is the heap number; for get from pool, pool is the pool handle; for make block, bp is the block pointer and name is an optional name for the pmsg. These functions return the pmsg handle, which is a pointer to its Message Control Block (MCB).

For the release function, pmsg is the msg handle returned by one of the get functions, and clrsz specifies how many bytes to clear after the free block link in the first word of the message block. For the send and receive functions xchg is the message exchange to use, timeout is the maximum time to wait at an exchange, pri is the pmsg priority, and rxchg is the reply or resource exchange to send the pmsg when done.

For ARMM7, smx_PMsgGetHeap() minimizes the actual memory allocation by sizing to the nearest multiple of subregion size, and by setting the proper subregion disable bits in the region created. It can handle sizes that are not a power of 2, whereas smx_PMsgGetPool() and smx_PMsgMake() require blocks that meet size and alignment requirements.

These PMsg functions are smx SSRs, so they can be safely called from either utasks or ptasks, at any time. They are similar to standard smx message SSRs but incorporate message protection

[12] fas is an abbreviation for MP_MPU_FAS defined in mpu.h and means *first active slot* of the MPU.

mechanisms and are used to securely transfer information between partitions. They are the basis for portals discussed in Chapter 5 Partition Portals and are described more fully there. For detailed information on PMsg functions, see Appendix A2.

If the message handle pointer parameter, mhp, in the above calls is set to the address of the message handle, then the message handle will be loaded by the Get, Make, and Receive services and need not be loaded directly from the return value. The Rel and Send services load NULL into the message handle so it can no longer be used. In addition, if the message handle is not initially NULL, a Get, Make, or Receive service will be aborted and SMXE_INV_OP reported. This forces releasing or sending a message before getting or receiving another message, thus avoiding message and MCB leaks.

4.12 Miscellaneous

4.12.1 Standard C Library Functions

The standard C library contains a mixture of functions including utility, init/exit, and system services. Some should be only accessible during startup and shutdown, and some should not be used in embedded systems.

The MODULE SUMMARY section of the map file lists the IAR C libraries and files in them that were linked. For example:

```
rt7M_tl.a: [4]
    ABImemcpy.o                166
    ABImemset.o                102
    XXexit.o                    12
    cexit.o                     10
    cmain.o                     30
    cmain_call_ctors.o          32
    copy_init3.o                46
    cstartup_M.o                12
    data_init.o                 40
    strcmp.o                    18
    strcpy.o                    24
    strlen.o                    54
    strncpy.o                  112
    zero_init3.o                58
    ---------------------------------------------------
    Total:                     716
```

Standard C library functions that should be used in tasks should be put into ucom_code. If there are a large number of C lib functions being used, it is helpful to define a clib_code block and include it in ucom_code:

```
define block clib_code with alignment = 4
                    {ro object dl7M_tln.a, ro object m7M_tls.a, ro object rt7M_tl.a}
              except {ro object cstartup_M.o, ro object cppinit.o, ro object data_init.o,
                    ro object fpinit_M.o, ro object rle_init_single.o, ro object zero_init3.o,
                    ro object cmain.o, ro object cmain_call_ctors.o,
                    ro object exit.o, ro object cexit.o, ro object XXexit.o};
```

```
define block ucom_code   with size = ucomcsz*5/8, alignment = ucomcsz
                         {ro section .ucom.text, ro section .ucom.rodata,
                          block clib_code};
```

ucom_code is included in sys_code so ptasks can also access these functions.

The except clause omits init and exit routines that are used only during startup and shutdown and should not be accessible by ptasks and utasks. They are automatically put into rom_block via "ro" at the end of the block definition. rom_block and ram_block regions are in the MPA templates of startup and shutdown tasks. Another benefit of doing this is it reduces the ucom_code and ucom_data sizes to minimize the required ARM7 alignment.

Alternatively clib_code can be defined to specify all of the C library files to locate:

```
define block clib_code with alignment = 4 {
                         ro object ABImemclr.o, ro object ABImemcpy.o, ro object ABImemmove.o,
                         ...
                         ro object memchr.o, ro object memcmp.o, ro object mktime.o,
                         ro object strcasecmp.o, ro object strcat.o, ro object strchr.o,
                         ...
                         ro object xTzoff_nop.o, ro object xisdst_nop.o, ro object xttotm.o};
```

but this method is inconvenient, because as additional C library functions are used, they cause an MMF until they are added to the list. Generally speaking, use of C library functions should be minimized in embedded systems. Some have unexpected behaviors – e.g. printf() uses very large amounts of stack. Other functions are dangerous – e.g. gets(), sprintf(), abort(), and exit(). Standard C library functions were not designed with security in mind. Many have well-known flaws that are exploited by hackers. Including these in your code is to invite an attack. In many cases, you are better off to create equivalent functions of your own that hackers do not know about. At the very least, verify that what you are using is safe. Also, if a C library function is used in only one partition, include it only in one of that partition's regions, rather than in ucom_code.

Components, such as those downloaded from GitHub, tend to use clib functions freely without regard for whether they are vulnerable, have undesirable side-effects, or use static data. Initially during development, these functions may be put into ucom_code and their static data put into ucom_data, but soon they should be moved out. At the end of the project, corrections can be made. clib functions that are used only during boot, initialization, or exit should be removed from ucom regions and linked in by the ro and rw linker symbols. Or, if they are being called from code running under an initialization task, they should be put into the sys regions.

If possible, partition code should be rewritten to not use dangerous clib functions nor to use clib functions that require static data. However, for large, complex components, doing this may not be practical. In that case, if the clib function is used in only one partition, put it into a region of that partition using the linker command file. This does not require having the source code. If the clib function is used in other partitions and its source code is available, duplicate it in the other partitions with slightly different names and use header files to map the standard names to the duplicate names. This is a good solution for clib functions that require static data. For vulnerable clib functions, it confines the risk to the partition it is in. Most likely, the component in that partition has many other serious vulnerabilities, so this may not result in reduced security.

If the above solutions do not work, then it may be necessary to rewrite the clib functions to be safe and to make them system services accessible via svc shell functions.

4.12.2 Partition Isolation vs. ucom Regions

The ucom regions, especially ucom_data, are in violation of partition isolation. As shipped, ucom.code contains only svc shell functions and C library (clib) functions; ucom.data contains only static data needed by some clib functions. During development, it may be expedient to add other common code and data. However, for best security, it is recommended to minimize ucom_code and to eliminate ucom_data in the final design.

The svc shell functions translate into RTOS and system services, most of which check input parameters thoroughly and are written to avoid misuse. As shipped, services that might cause system damage are excluded from the svc shell functions. For security, when a project is complete, it is recommended that all unused svc shell functions be removed from ucom_code or, better, that their jump table entries be changed to report a security violation. Going even further, since most partitions use limited RTOS and system services, it is possible to create custom shell functions and jump tables to avoid using ucom_code for svc shell function, altogether. See 4.9.4 Custom SSTs for more information.

clib functions are quite a different matter. Many have well-known flaws that are exploited by hackers. Initially during development they may be put into ucom, but soon they should be moved out. See 4.12.1 Standard C Library Functions for discussion.

4.12.3 HAL Code

Hardware Abstraction Layer (HAL) code, such as that provided by chip vendors, needs special handling. Some guidelines are:

- Boot, initialize, and exit HAL code that does not run in a task and its static data should go into the default sections (.text, .data, etc.). These are not in a partition.
- Initialize and exit HAL code that does run in a task and its static data should go into .sys.code and .sys.data.
- HAL code used by a middleware module and its driver(s) should go into the same partition as the middleware module.
- Common HAL code and data used by two or more partitions should go into .ucom.code and .ucom.data. However, like the clib functions discussed above, this violates partition isolation. Some of the same remedies might be applied to them, including making HAL functions into system services.

Chapter 5 Partition Portals

Partition portals enable isolating client partitions from server partitions. They utilize an alternate form of API between clients and servers that is built upon smx protected messages.

5.1 Introduction

5.1.1 Isolated Partitions

Figure 5.1 shows the desired isolation between a file system (FS) partition and two application partitions (App1 and App2) that use the file system.

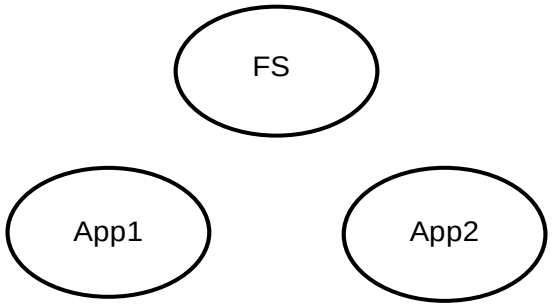

Figure 5.1 Desired Isolation Between Partitions

5.1.2 Function Call APIs

Figure 5.2 shows the result when the App1 and App2 partitions must access the file system via the file system's normal *function call API* – i.e. loss of partition isolation.

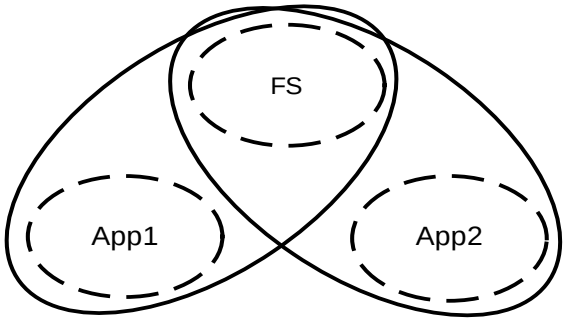

Figure 5.2 Loss of Partition Isolation

Function calls are the predominant API between clients and servers. This creates a problem for partitioning. In the above diagram, the file system's API functions must be accessible to both application partitions. And subroutines in the file system must be accessible to the API functions and driver functions must be accessible to the subroutines. Hence nearly the entire file system must be accessible to both App1 and App2. Also, file buffers and global variables must be accessible to the file system functions. So, the whole file system and driver(s) end up in a code region shared by the App partitions and file buffers and globals end up in a data region shared by the App partitions. Thus none of these partitions is isolated from the others.

If a hacker penetrates any one of the partitions, he has access to the other partitions via the common file system regions. Although he cannot necessarily control the other partitions, he can certainly bring them down and possibly disrupt the whole system. The solution to this problem is to use *partition portals*, which are discussed next.

5.1.3 Partition Portals

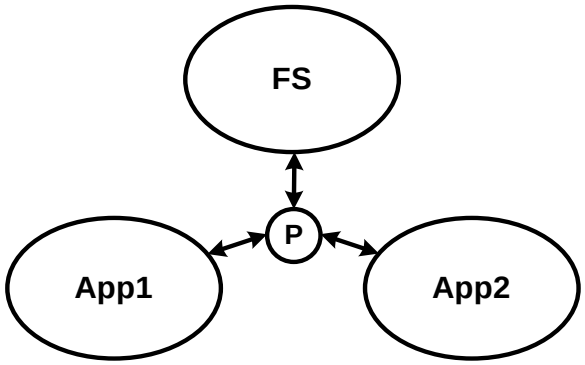

Figure 5.3 Partition Isolation Using FS Portal P

Figure 5.3 shows the isolation achieved with file system partition portal, P. The double arrows represent bidirectional data transfers. Note that there is no overlap between the partitions. Hence, the desired full level of isolation shown in Figure 5.1 has been achieved. Partition portals are indirect calling mechanisms which permit full isolation between partitions. They are based upon *smx protected messages, pmsgs,* working in combination with the Cortex-M MPU.

SecureSMX provides two types of partition portals:

- Free message portals, *fportals* – see section 5.3 Free Message Portal.
- Tunnel portals, *tportals* – see section 5.4 Tunnel Portal.

Protected messages are discussed first.

5.2 Protected Messages

5.2.1 pmsg Structure

SecureSMX portals are implemented using *smx protected messages* (*pmsgs*). Figure 5.4 illustrates the structure of a pmsg. A normal smx message consists of a Message Control Block (MCB) linked to a data block that contains the actual message. An smx pmsg is the same as a

normal smx message with the addition of special fields to the MCB, and the data block is an MPU block region. The MCB, itself, is in sys_data and therefore inaccessible to utasks.

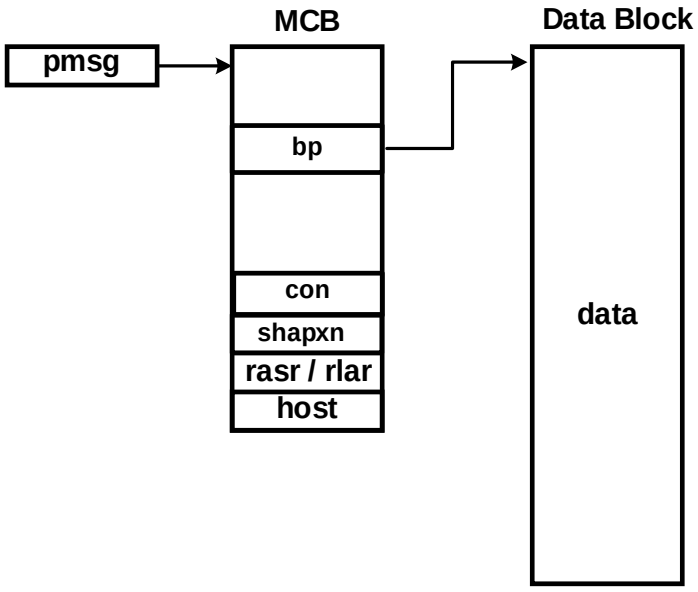

Figure 5.4 Protected Message Structure

The above figure shows the additional pmsg MCB fields along with bp, which are:

- **con** is the control structure which contains:
    - **hsn** host slot number.
    - **osn** owner slot number.
    - **bnd** bound message flag.
    - **sb** system block flag, which means the pmsg data block is in sys_data.
- **shapxn** ARMM8 shareability, access permissions, and execute never.
- **bp** points at the pmsg data block and is used to calculate rbar for ARMM7 and ARMM8.
- **rasr/rlar** ARMM7 *region attribute and size register* or ARMM8 *region limit address register*.
- **host** task handle – loaded when server receives pmsg.

5.2.2 Sending a pmsg

A pmsg can be sent from partition to partition, via a message exchange, and it carries with it its own MPU region information for its data block. Thus it is a self-contained data region, and hence the name *protected message*. For ARMM7 the data block consists of 5, 6, 7, or 8 contiguous subregions aligned on the subregion size, 2^n, all within a region of size 2^(n+3) aligned on its size. For ARMM8, the data block is a multiple of 32 bytes, aligned on 32 bytes. For more information on pmsgs, see Section 4.11.11, Protected Messages, and Appendix A.2 smx Protected Block & Message Services.

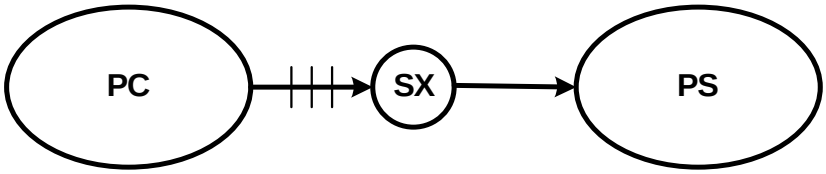

Figure 5.5 pmsg Transfer

Figure 5.5 shows how a pmsg is sent from a client partition, PC, to a server exchange SX. The hash marks on the arrow represent pmsgs waiting at SX for service from the server partition, PS. A pmsg is sent by a task in PC with a smx_PMsgSend() function. pmsgs can wait at an exchange either in FIFO order or in priority order. A pmsg is received by a PS task with a smx_PMsgReceive() function. These are discussed in detail below.

If SX is a pass exchange, PS will run at the priority set in the pmsg by PC. If this priority is higher than that of the PC task, the PS task will preempt the PC task and process the pmsg immediately. This is analogous to a direct function call, such as fwrite(). If the pmsg priority is equal to or lower than the PC task priority, the PS task will not run until the PC task suspends or stops and all other tasks of higher priority or precedence[13] have also suspended or stopped. This flexibility is not available with direct function calls. Delayed action can be useful if the pmsg is a non-urgent message, such as a status message that PS is just going to log or display. In this regard, it is important to note that pmsgs can accumulate at SX in priority and precedence order and will be processed in that order.

sn = pmsg->con.osn, where sn is the PC task's MPA slot number that contains the pmsg region. When a *free protected message* is sent, the current task's MPA[sn] is cleared. If sn is an active slot, MPU[sn+fas] is also cleared. In the case where sn is an active slot, even if the sending task retains a pointer to the message block, it can no longer access it. This prevents changing a message after it has been validated by a receiving task in another partition. It also prevents the sender from reading the message after it has been updated by a receiving task in another partition. This is the recommended approach for free messages – i.e. free pmsg blocks should not be taken from sender regions. It is best if they are taken from mheap or a block pool in sys_data.

For a *bound protected message* (pmsg->con.bnd == 1), the sender retains access to the pmsg, so the foregoing is not applicable. In this case, it is more efficient, slot-wise, for the pmsg block to come from a sender region. Bound protected messages are used for tunnel portals (see section 5.4, Tunnel Portal).

The slot numbers in the sending and receiving tasks need not be the same. The send slot can be an auxiliary slot (see Figure 4.13) if the data block comes from a client data region. Otherwise, it must be an active slot.

[13] a task has *precedence* if it has equal priority and has been waiting longer. It has *priority* if its priority is higher. In smx 0 is the lowest priority.

5.2.3 Receiving a pmsg

smx_PMsgReceive() is normally used by a server to receive a pmsg from an SX exchange. When a pmsg is received by a server task, its MPA[sn] is loaded with RBAR and RASR/RLAR from the MCB of the pmsg, where sn is the slot number parameter in the pmsg receive function. If sn is an active slot, MPU[sn+fas] is also loaded. If the pmsg data block is in a server region, the server slot can be an auxiliary slot for ARMM7. However, it must be an auxiliary slot for ARMM8 because overlapping regions cause MMFs for ARMM8. Otherwise the receive slot must be an active slot.

The receiving task can read and modify a free pmsg data block, then send it to another exchange. For example, a pmsg could be created and loaded with data by a data acquisition task, passed to another task to encrypt the data, then passed to yet another task to send the encrypted data out on a network. There is complete isolation between each sending and receiving task. This is possible because the pmsg carries its own region information as it moves from place to place. Thus a pmsg is protected at all times. Of course, an infected sender could send a disruptive message. Thus receivers should perform validation checks before accepting pmsgs.

smx_PMsgReceiveStop() allows a server task to release its stack prior to again waiting at SX. This kind of task is called a *one-shot task*. (See the smx User's Guide for a full explanation.) Since at any given time, there are likely to be many idle portals where servers are waiting, using one-shot tasks for portal servers can save significant RAM. This helps to compensate for the additional tasks introduced by portals.

Two portal protocols have been implemented using pmsgs:

- Free Message Protocol.
- Tunnel Protocol.

The free message protocol is the simpler of the two and provides the best security. It is discussed in section 5.3 Free Message Portal. The tunnel protocol is discussed in section 5.4 Tunnel Portal.

5.2.4 Message Priority Inheritance

Like mutexes, portals can cause unbounded priority inversions resulting in tasks missing their deadlines. Consequently, smx implements *message priority inheritance* for pass exchanges, which are used in portals. This means that if a pmsg is sent to a portal that has higher priority than the task serving the portal, the task priority is raised to that of the pmsg All portals have priority inheritance. For more information, see **message priority inheritance** in the smx User's Guide.

5.2.5 Dual MPA Slots for ARMM8

Due to the fact that overlapping slots generate MMFs for ARMM8, it is necessary to implement dual MPA slots for pmode servers. The following discussion applies only to ARMM8 processors.

If a pmsg[14] is allocated from mheap and sent to a ptask, region overlap is likely to occur resulting in an MMF. This is because mheap is in sys_data and sys_data is usually a ptask region. The simplest way to avoid this is to allocate ptasks from a heap or block pool that is not in sys_data. However, that may not always be convenient. For example, a pmsg that was allocated from mheap by a umode task might be sent to a pmode task only under exceptional conditions (e.g. a system error has occurred). Then, the resulting MMF might be puzzling to figure out. (Note that the MPU window in smxAware shows overlapping regions, so it is good to watch it.)

SecureSMX implements the following solution to this problem: when a pmsg is allocated from sys_data, the pmsg->con.sb (system block) flag is set. Then, if sb is set, a ptask receiving the pmsg must use an auxiliary slot for the pmsg region. (In this case, the pmsg region is not actually needed to access the pmsg.) If sb is not set, the ptask must use an active slot for the pmsg region.[15] (In this case the pmsg region is needed to access the pmsg). This is implemented as follows:

When a ptask does a receive, it specifies a dual slot number (dsn), as in the following example:

```
dsn = 0x64;
while (pmsg = smx_PMsgReceive(psh->sxchg, &bp, dsn, SMX_TMO_INF))
```

where:

```
dsn = xsn << 4 + asn;
```

and xsn = auxiliary slot number; asn = active slot number. When a pmsg is received, if pmsg->con.sb = 1, the pmsg region is put into MPA[xsn]. Otherwise it is put into MPA[asn] and into MPU[asn+fas]. When a utask does a receive, sb is ignored since utasks can not use sys_data as a region because it is privileged..

There are, of course, other cases where receiving a pmsg could cause a region overlap and thus an MMF. These are unlikely, but must be guarded against if an ARMM8 processor is being used. One way to proceed is to always allocate pmsgs from mheap.

5.3 Free Message Portal

A portal is limited to a single server, but it may be accessed by many clients.

Server API:

```
bool  mp_FPortalCreate(FPSS* psh, FPCS** pclp, u32 pclsz, u8 ssn,
                                const char* pname=NULL, const char* sxname=NULL);
bool  mp_FPortalDelete(FPSS* psh, FPCS** pclp, u32 pclsz, u8 xsn=0);
```

Client API:

[14] Here, and elsewhere, “pmsg” refers to the pmsg data block as well as to the whole pmsg.

[15] Of course, if the pmsg is allocated from another ptask region, then these rules do not apply. Therefore, to avoid unexpected MMFs when using an ARMM8, it is advisable to decide, in advance, upon a consistent pmsg allocation strategy and then to stick with it. This unfortunate complexity is due to an architectural flaw in ARMM8.

```
bool  mp_FPortalClose(FPCS* pch, u8 xsn=0);
bool  mp_FPortalOpen(FPCS* pch, u8 csn, u32 msz, u32 nmsg, u8 pri,
                                u32 tmo=SMX_TMO_INF, const char* rxname=NULL);
MCB_PTR  mp_FPortalReceive(FPCS* pch, u8** dpp=NULL);
bool  mp_FPortalSend(FPCS* pch, MCB* pmsg);
bool  mp_FTPortalSend(FPCS* pch, u8* bp, MCB_PTR pmsg);
```

See Appendix A.1 SecureSMX Services for descriptions of parameters and other information. Use of these functions is discussed in the following sections.

5.3.1 Configurations

The free *message portal* gets its name from the fact that the pmsgs are not bound to the client as they are for the *tunnel portal*. Hence they are *free*. Its protocol primarily makes use of the smx protected messaging capabilities discussed previously. Figure 5.6 illustrates three of many possible free message protocol configurations. Hash marks indicate waiting messages at the message exchanges shown.

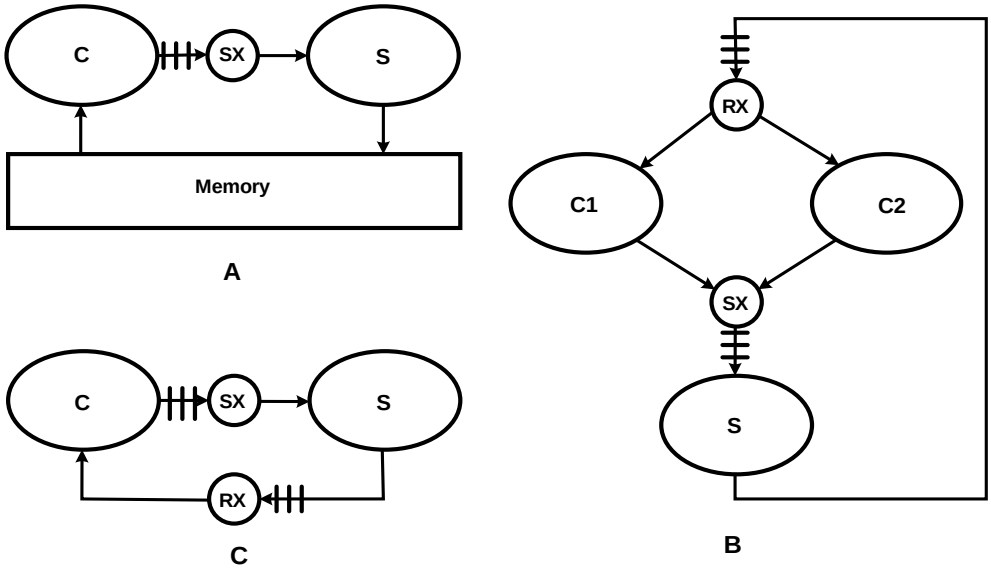

Figure 5.6 Free Message Protocol Configurations

Figure 5.6A is the most basic configuration in which client, C, obtains a pmsg from memory, loads it, then sends it to server exchange, SX. Server, S, receives the pmsg from SX, processes it, then returns it to memory. As previously explained, in section 4.11.11 Protected Messages, pmsgs can come from any heap or any block pool, or be made from any standalone block. Note that the latter two must meet MPU size and alignment requirements, whereas if calling smx_PMsgGetHeap(), smx_PBlockGetHeap(), or mp_RegionGetHeapT(), smx heap allocations automatically meet MPU region requirements.

The memory shown in Figure 5.6A can be global memory or local memory in the client partition. In the global case, an active slot in the client MPA is required; in the local case an auxiliary slot may be used (see section 4.2.5 Auxiliary Slots) for the client, to save a slot, but an active slot is

required for the server[16]. The 5.6A configuration is suitable for infrequent messages such as notifications of rare events, or it may be necessary if memory is limited.

Figure 5.6B shows two clients, C1 and C2, that are obtaining recycled pmsgs from a resource exchange, RX, loading them, then sending them to a server exchange, SX. The server processes the messages and sends them back to RX. In this case, performance is better than 5.6A at the expense of RAM permanently allocated to pmsgs. This configuration might be useful where data rates are moderate, such as input from multiple UARTs or multiple low-speed USB devices. Client could send multi-block messages, in which case, the server would need to separate intermingled blocks into their client streams.

Figure 5.6C shows a bidirectional data configuration. In this case, as many messages as needed are obtained from memory and sent to the reply exchange, RX. Client C receives messages from RX, processes and/or loads them, then sends them to the server exchange, SX. Server S receives messages from SX, processes and/or loads them, then sends them to RX. A return message may contain only status such as ACK or NAK or it may contain processed data. If the RX exchange is being used to return an ACK or NAK or a return value, there can be only one pmsg in circulation or the messages need to be numbered

In a case like 5.6C, the server might be a special processor for the client that could be replaced dynamically, when conditions change. Note that although data exchange is bidirectional, the client is the master and must initiate all transactions. Should the server need to initiate transactions, such as for a client callback, then a separate portal would be required.

5.3.2 Portal Creation

To create a free message portal, it is first necessary to define a Free message Portal Server Structure (FPSS) for the server, a Free message Portal Client Structure (FPCS) for each client permitted to access the portal, and a permitted client list, pcl[], containing pointers to the client FPCSs:

```
FPSS        pssa;                   /* portal server structure a */
FPSS*       pssah;                  /* portal server structure a handle */
extern FPCS pcs1, pcs2;             /* portal client structures 1 & 2 */
FPCS*       pcla[] = {&pcs1, &pcs2} /* permitted client list a */
u32         pclasz = sizeof(pcla)/4; /* permitted client list a size */
#define SSN 5                       /* server slot number for pmsg */
```

Then call the free message portal create function:

```
mp_FPortalCreate(&pssah, pcla, pclasz, SSN, “portla”, “sx_portla”);
```

The pssa, pssah, pcla[], and pclasz definitions are best grouped with the server code because they must be in a data region accessible to the server. Similarly, the pcs1 and pcs2 definitions are best grouped with their respective client code, because each must be accessible by its respective client. All portal structures are statically defined at compile/link time and are not accessible in

[16] In all portal discussions, clients and servers are assumed to be separate partitions. If not, there is no need for portals – the client can directly call server functions. This may be useful as an intermediate step during partitioning but not as the final implementation, in most cases.

umode. If another client is defined for this portal, its FPCS address would be added to pcla[] and pclasz would be increased. Hence, it is easy to see which clients are permitted to access a given portal. These are called *authorized clients*. Only they are allowed access to the server. The amount of overhead and complexity for a portal may seem excessive, but it is necessary for full isolation of client and server partitions in order to provide high security.

mp_FPortalCreate() creates the *server exchange, sxchg*, and loads its address into pssa. It then loads the sxchg address into pcs1 and pcs2 to enable each client to access the portal[17]. (For malware to access the portal, it would need to guess its sxchg address.) The portal name is also loaded into the server's FPSS and into each client's FPCS. Since a client may access multiple partitions via its portals and thus have multiple FPCSs, portal names are helpful to avoid confusion during checkout.

Once the server portal has been created, it is necessary to create and start the server task, as follows:

```
TCB* taskA;

if (taskA = smx_TaskCreate(mainA, PRI_MAX, stksz, SMX_FL_UMODE, "taskA"))
{
   mp_MPACreate(taskA, &mpa_tmplt_taskA);
   pssa.stask = taskA;
   smx_TaskStart(taskA);
}
```

The server portal and its task should normally be created in pmode during system initialization. taskA is given maximum priority so that it will run immediately after system initialization, then wait at sxchg for the first pmsg. After that, taskA assumes the priority of each pmsg that it receives since sxchg is a priority-pass message exchange. It is assumed above that taskA is a utask, but it could be a ptask, in which case replace SMX_FL_UMODE with 0.

5.3.3 Client Open

Once a portal and its task have been created, an authorized client can open the portal, as follows:

```
FPCS     pcs1;       /* portal client structure 1 */

mp_FPortalOpen(&pcs1, csn, msz, nmsg, pri, tmo, "pcs1_rxchg");
```

where pcs1 is the same client handle used in the permitted client list, pcla[], above. At this point, the client structure already contains the alias sxch handle and the portal name. csn is the client's MPA slot number for pmsg data block regions, msz is the pmsg block size, nmsg is the number of pmsgs to create for this portal, pri is the priority at which to send these pmsgs to sxchg, tmo is the rxchg wait timeout, and "pcs1_rxchg" is the rxchg name. rxchg is an RX resource/reply exchange as shown in Figure 5.6. At this point, csn, pri, and tmo are just stored in pcs1.

---

[17] A *handle*, such as *sxchg*, is a location in memory that stores the address of the sxchg control block. In this case, the sxchg handle is in a server data region. If a client calls mp_FPortalSend() using this handle, an MMF will occur because sxchg is not in a client region. To remedy this problem, each client must have its own *alias sxchg handle* that is accessible via one of its own data regions. An *alias handle* is another location in memory that stores the same control block address as the *real handle*.

If nmsg == 0, no rxchg is created and no pmsgs are created. This allows pmsgs to later be created separately, as shown in Figure 5.6A, or to allocate them from somewhere other than mheap such as the client's address space in order to use a auxiliary slot for the pmsg region. If nmsg > 0, rxchg is created and nmsg pmsgs are created and sent to rxchg. If nmsg == 1, rxchg is probably operating as a *reply exchange*, as shown in Figure 5.6C. If nmsg > 1, rxchg is probably operating as a *resource exchange* as shown in Figures 5.6B or C.

The following code supports the operation shown in Figure 5.6B:

```
FPCS      pcs1, pcs2;        /* portal client structures */

mp_FPortalOpen(&pcs1, csn1, pri1, tmo1, msz, nmsg, "RX");
mp_FPortalOpen(&pcs2, csn2, pri2, tmo2, msz, 0, NULL);
pcs2->rxchg = pcs1->rxchg;
```

In this case pcs1 is first opened with nmsg pmsgs – enough for both clients. Note that each client has its own MPA slot, priority, and timeout, but msz must be the same. Since nmsg == 0 for pcs2, no rxchg and no pmsgs are created for it, but the rxchg handle is loaded from pcs1 to pcs2. Of course, C1 and C2 can have separate rxchgs and different pmsgs, if desired. In that case, smx_PMsgReply(), used by server S to release a pmsg, will send the pmsg to the correct rxchg for each client.

5.3.4 Client Operation

A client receives a pmsg from an rxchg with:

```
FPCS*     pcs1h;  /*portal client structure 1 handle */
u8*       pbp;     /* pmsg block pointer */

pmsg = mp_FPortalReceive(pcs1h, &pbp);
```

where pcs1h is the portal client structure handle and &pbp is the address of pointer pbp, which is used to load the pmsg data block. If the portal were opened with no pmsgs created, then a pmsg would be obtained with one of the pmsg get functions, such as:

```
pmsg = smx_PMsgGetHeap(msz, &pbp, pcs1h->csn, MP_DATARW, mheap);
```

This approach is useful if the client seldom sends messages to this portal, if the messages vary in size, or if the client sends messages to many portals.

In either case, the pmsg is sent to the free portal message exchange, pcs1h->sxchg, with:

```
mp_FPortalSend(pcs1h, pmsg);
```

When processing of the pmsg is complete,

```
smx_PMsgReply(pmsg);
```

is called by server S to release the pmsg. In the first case the pmsg is returned to RX, as shown in Figures 5.6B & C. In the second case, it is deleted and its block is returned to mheap, as shown in Figure 5.6A. Which of these occurs is predetermined by the client and hidden from the server. This prevents malware in the server from sending pmsgs to inappropriate places.

5.3.5 Server Operation

As previously described in section 5.3.2 Portal Creation, the portal server task is separately created and started. The code for it should be similar to the following:

```
FPSS cpsvr; /* cp server structure */

void cp_main(void)
{
  cp_server(&cpsvr);
}

void cp_server(FPSS* psh)
{
  CPSH* chp;  /* console header pointer */
  MCB*  pmsg;
  u32   par1, par2, par3, par4, par5;
  bool ret;

  while (pmsg = smx_PMsgReceive(psh->sxchg, (u8**)&chp, psh->ssn, SMX_TMO_INF, &pmsg))
  {
    switch (chp->fid)
    {
      case CP_CLR_SCREEN:
        sb_ConClearScreen();
        break;
      case CP_DBG_MODE:
        ret = sb_ConDbgMsgMode();
        chp->ret = ret;
        break;
      case CP_WRITE_STRING:
        par1 = (chp->p1>>24)&0xEF;
        par2 = (chp->p1>>16)&0xFF;
        par3 = (chp->p1>>8)&0xFF;
        par4 = (chp->p1)&0xFF;
        par5 = (chp->p1)&0x80000000 ? 1 : 0;
        sb_ConWriteString(par1, par2, par3, par4, par5,(ccp)chp->dp);
        break;
    }
    smx_PMsgReply(pmsg);
  }
}
```

where the pmsg service header is defined as follows:

```
typedef struct CPSH {      /* CONSOLE PARTITION SERVICE HEADER */
  u32       fid;       /* function ID */
  u32       p1;        /* parameter 1 */
  u32       p2;        /* parameter 2 */
  u32       dp;        /* data pointer */
  u32       ret;       /* return value */
```

```
    void*       caller;     /* caller addr (debug) */
} CPSH;
```

The pmsg data block consists of the above header followed by the actual message. The header is loaded by the client sending the pmsg. This is normally done with *shell functions* that convert the function call to a pmsg — see section 5.5 Shell Functions.

The above code has been borrowed from the console partition portal. Note that once started, the portal server task, cp_task, waits for the next pmsg at its sxchg, with infinite timeout. When a pmsg is received, cp_server() switches on the function id, chp->fid, where chp is the console header pointer. The first case shown is a simple case of performing a command to clear the screen. The second case reads the console mode and returns it. This shows how to handle a return value. The third case shows a function with many small parameters packed into a single header parameter. This not only saves space, but also improves performance. cp_task then returns the pmsg to its rxchg using smx_PMsgReply(pmsg). It is necessary to use this function because the rxchg handle is hidden in the pmsg and the server cannot access it.

The foregoing is an example of Figure 5.6C, except that for a console portal there are likely to be many clients, rather than just one, accessing the console portal. In this case, because some functions have return values, the rxchg is serving as a reply exchange as well as a resource exchange. In order for this to work properly, there can be only one pmsg per client. This does not result in reduced performance because, in this case, output to the terminal is via a UART and thus very slow.

In other cases where communication is open-ended and no returns are expected, multiple pmsgs may be used as shown in Figures 5.6A and 5.6C. The advantage of that is to handle cases where client activity gets ahead of the server, possibly because the server has to wait for access.

The psh->ssn parameter in smx_PMsgReceive() is the slot in the MPU and in cp_task's MPA to put the data region for the pmsg. As noted previously, the pmsg data region is carried in its message control block, MCB. This allow cp_task to access the pmsg data.

Also important, but not shown, is the pmsg priority carried in the pmsg MCB. Because sxchg is a pass exchange, cp_task will run at this priority after receiving the pmsg. If it is higher than the client task, the server task will preempt immediately and run. If it is the same, the server task will not run until the client is suspended. (This could happen if the client waits on the rxchg to get the pmsg back.) If pmsg priority is lower, then the server will run sometime in the future. This would be appropriate for a non-critical activity, such as activity logging. In this case it would be likely that many pmsgs would build up from many different sources until more important processing is completed.

5.3.6 Client Close

To close a client portal, call:

```
mp_FPortalClose(&pcsA);
```

Doing this would be beneficial if pcsA is seldom used by the client, since all pmsgs created for it would be released, and its rxchg would be deleted. These and the associated memory could then be used for other purposes, such as another seldom-used portal. pcsA is not deleted, but it is cleared except for sxchg and the portal name. pcsA is permanently dedicated to the server portal

for which it is in the portal client list, unless the server portal is deleted. It is worth noting that all pcs's, pss's, and pcl's are static – i.e. they cannot be deleted – but they can be repurposed.

mp_FPortalClose() waits at rxchg for up to pcsA.tmo ticks for each pmsg. This is necessary because client pmsgs may be waiting at the portal exchange to be processed or one might be being processed. If a timeout occurs, it returns false, so other action can be taken to prevent a pmsg leak. A possible action would be to delay then call this function again. This works because pcsA.num is decremented each time a pmsg is released, thus it equals the number of pmsgs not yet released. Note that smx_PmsgRel() will not work because the client does not own these pmsgs.

For the example shown in 5.3.3 Client Open of opening two client portals that share an rxchg, closing the first client portal will release the pmsgs and delete rxchg. When it is done, the second client portal can be closed, but its FPCS rxchg field must be set to NULL as follows:

```
pcs2->rxchg = NULL;
```

mp_FPortalClose() has another parameter, xsn, which defaults to 0. This parameter is used only by the mp_FPortalDelete() function and is not intended to be used in normal operation.

5.3.7 Portal Deletion

Portal deletion should be a rare event and probably used only by recovery operations. It is intended to be used by system code running in pmode. To delete portal pssa, call:

```
mp_FPortalDelete(&pssa, pclp, pclsz, xsn);
```

This stops the portal task, for safety. Then it deletes the portal's sxchg, which releases all pmsgs waiting at it[18], and it clears the alias sxchg handles and portal names in every FPCS in portal's pclp[]. In addition, the portal is closed at each client. xsn is used for this. It is an available slot in the MPA of the ptask doing the deleting – preferably an auxiliary slot. xsn is necessary because mp_FPortalClose() must receive and release every pmsg at the client's rxchg. xsn holds the pmsg data region during this process. Now the FPCSs are available for use by other portals.

Normally, the portal task is deleted after its portal is deleted, thus freeing a TCB and the task's stack, if it had a permanent stack.

A portal and its task should not be deleted if the portal task is performing an operation. This can be determined by waiting for a reply pmsg from the portal at its rxchg. It can also be determined by testing if the task is in the *wait state*. However, if a partition is misbehaving, mp_FPortalDelete() can be used to forcibly shut down its portal and stop the portal task. This would probably be followed up by shutting down and restarting the whole partition.

It is not recommended that portals routinely be deleted, as this is likely to lead to programming errors. Portal delete is best reserved for recovery operations.

[18] This uses smx_MsgRel_F(), which releases the msg data block to its origin: heap, block pool, or nothing if it is standalone block, then releases its MCB to the MCB pool.

5.3.8 More Flexible Operation

The free message API presented above is intended to make creating and using free message portals easier. However, in some cases, these functions may prove to be too limiting. If so, smx_PMsg functions can be used directly to create a custom protocol.

Receive can use either of the receive functions: smx_PMsgReceive() or smx_PMsgReceiveStop(). The latter is good for seldom-used servers, since the server task releases its stack while waiting for the next pmsg. Considering that there may be many seldom-used servers in a system, sharing stacks can save significant memory.

Send can use either:

```
smx_PMsgSend(pmsg, rxchg, pri, reply); or

smx_PMsgReply(pmsg);
```

It is recommended that most servers use the latter, for better security and to minimize programming errors. For it, rxchg is derived from pmsg->rpx, ssn = pmsg->osn, pri = pmsg->pri, and there is no reply. Hence, the client controls the choice of rxchg and priority. osn is loaded into the pmsg MCB when the pmsg is received by the owner. (For an unbound pmsg, this is the last pmsg receiver; for a bound pmsg, this is the last pmsg sender.)

5.4 Tunnel Portal

Server API:

```
bool    mp_TPortalCreate(TPSS** pshp, TPCS** pclp, u32 pclsz, u8 dsn,
                                    const char* pname=NULL, const char* sxname=NULL);
bool    mp_TPortalDelete(TPSS* psh, TPCS** pclp, u32 pclsz);
void    mp_TPortalServer(TPSS* psh, u32 stmo);
```

Client API:

```
bool    mp_TPortalCall(TPCS* pch, u32 tmo=0);
bool    mp_TPortalClose(TPCS* pch, u32 tmo=0);
bool    mp_TPortalOpen(TPCS* pch, u32 msz, u32 thsz, u8 pri, u32 tmo=SMX_TMO_INF,
                                    const char* ssname=NULL, const char* csname=NULL);
bool    mp_TPortalReceive(TPCS* pch, u8* dp, u32 rqsz, u32 tmo=0);
bool    mp_TPortalSend(TPCS* pch, u8* dp=NULL, u32 rqsz=0, u32 tmo=0);
```

See Appendix A.1, SecureSMX Services for full call descriptions. Use of the above functions is discussed in the sections that follow.

A tunnel portal installs a single pmsg (the “tunnel”) between the server and the client, which alternate writing and reading its data block, as controlled by semaphores.

Tunnel portals offer the following features:

1. Full isolation between clients and servers.
2. Minimal change to client code.
3. Good performance.
4. No copy transfers available with change to client code.
5. Full protection of data.

The free message protocol is recommended for infrequent messages and low-to-moderate data rates, but it is too slow for large data transfers such as file and network data transfers. For these, the tunnel portal is better. Figure 5.7 illustrates the basic concept of the tunnel portal. It is called that because a pmsg data block, represented by T, in the figure, serves as a tunnel between the Client and the Server. This is possible because both have MPA regions containing the pmsg data block and thus both can read or write the data block. The pmsg data block is also referred to as the *portal buffer*, *pbuf*. The pmsg is sent by the client to open the tunnel. After that, it remains in place until the portal is closed.

Note: A tunnel portal can also accept free pmsgs – see section 5.6 Sending Free Messages to Tunnel Portals.

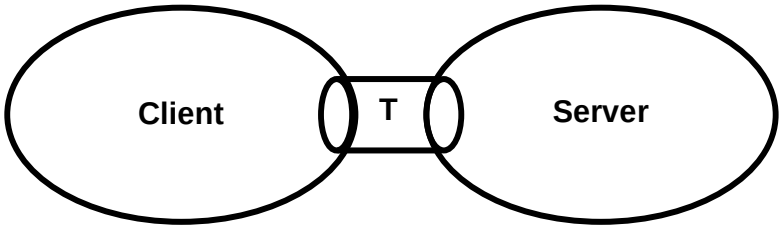

Figure 5.7 Tunnel Portal

pbuf is the only common region between the two partitions, and the MCB of the pmsg is not accessible by either. If a hacker gains access to one partition, pbuf cannot be used to access the other. pbuf is defined as XN (execute never) so the hacker cannot execute code from it. Since pbuf is an MPU region, he also cannot overflow it. Either case would generate an MMF. Access to pbuf alternates between client and server under control by two semaphores. Only one partition, at a time, is permitted to access pbuf.

When malware has access to pbuf, it can put anything into it. Also, the malware could ignore its semaphore and write into pbuf while the other end is reading it. (However, this would require preempting the task at the other end.) Therefore, the receiving partition should have data checks that prevent accepting harmful data. One possibility is to encrypt the data. This is not unlike sending data over the Internet.

In the following discussion, *client* means the task in the client partition that controls the client side of the portal, or it means the client partition, itself, and *server* means the task in the server partition that controls the server side of the portal, or it means the server partition, itself. Usage

should be clear from the context. Each partition has one tunnel portal task; each may also have other tasks.

Figure 5.8 illustrates the tunnel portal protocol. It is similar to other protocols with open, close, and data transfer operations. The operations are explained in the subsections that follow.

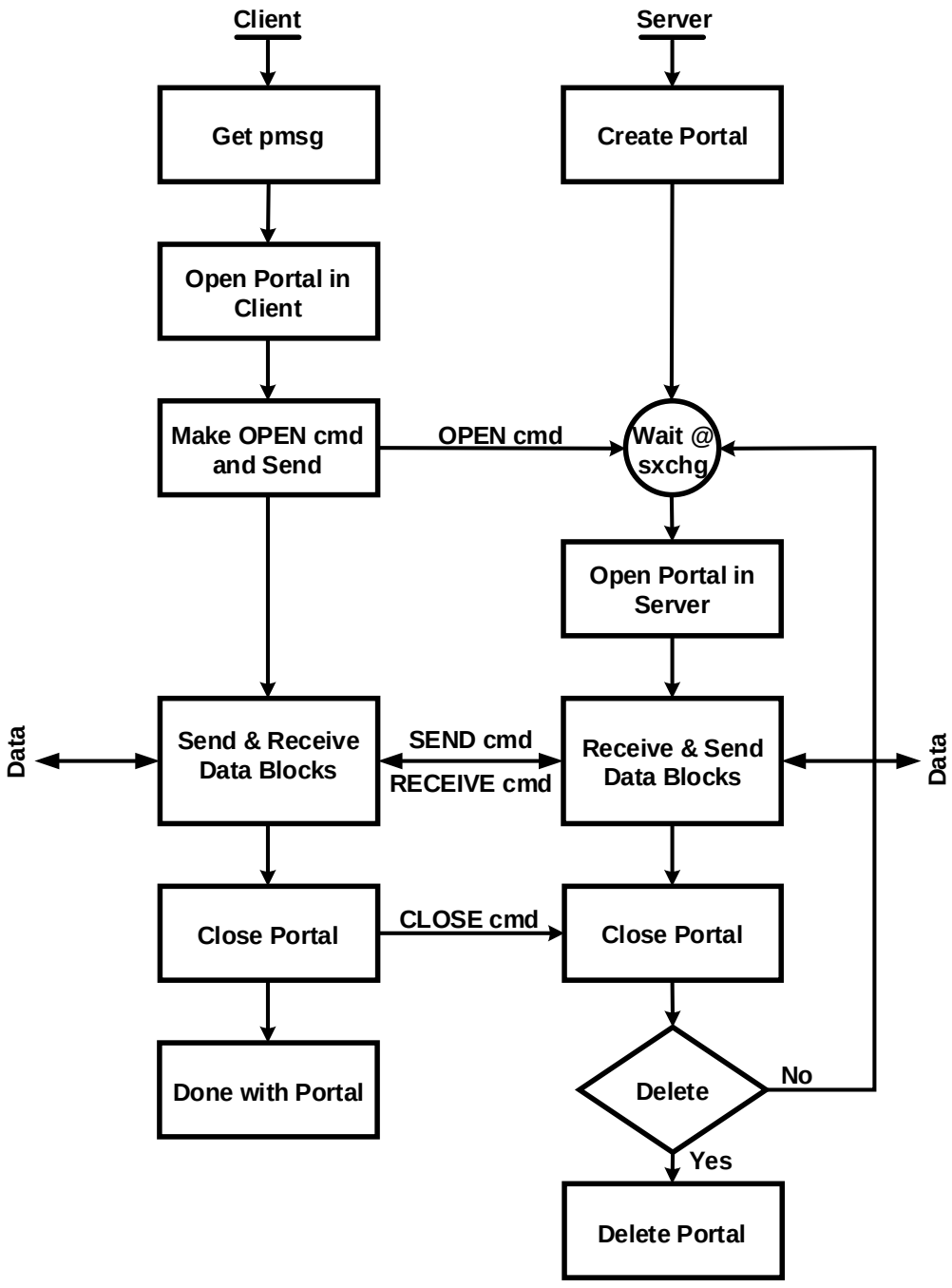

Figure 5.8 Tunnel Portal Operation

5.4.1 Get pmsg (by client)

As with the free message protocol, the pmsg data block can be obtained from any heap, block pool, or standalone block, as long as the block meets the MPU size and alignment requirements. The pmsg is created independently of the portal so that the client can use it for other purposes, such as first the work buffer, then pbuf. Doing this enables no-copy operation.

If the pmsg data block is already in an existing client region, an auxiliary MPA slot can be used to hold it, thus saving an active MPA slot (see Figure 4.6). For ARMM8, an auxiliary slot must be used if the pmsg data block is already in an existing client region. Otherwise, an active MPA slot must be used. This pmsg is *bound* (pmsg->con.bnd == 1) to the client so that only the client can send or release it.

5.4.2 Create Portal (by server)

A portal is normally created by the server's pmode initialization code and remains in existence as long as the server exists. Each tunnel portal has a Tunnel Portal Server Structure (TPSS) and a Portal Client List (PCL). In addition, every client authorized to access the tunnel portal has a Tunnel Portal Client Structure (TPCS) for the portal. The TPSS and TPCS structures contain the information necessary to control their respective ends of the portal.

A portal's Portal Client List is used, when the portal is created, to load the portal name and sxchg address into the TPCS of every client that is authorized to access the portal. It is also used, if the portal is deleted, to load "no portal" into each TPCS name field and NULL into each TPCS sxchg field. A portal cannot be opened at the client end if its TPCS sxchg is NULL.

The entire system of portals is predetermined at compile time and cannot be tampered with during run time. Thus it is virtually impossible for a hacker to access a portal not authorized for the client partition which he has penetrated. The network of connections to portals is the roadmap for a secure embedded system. In fact, when starting development of a system, a good plan is to draw a data flow diagram showing partitions linked by lines that represent portals.

The first step to create a tunnel portal is to create the tunnel portal server task:

```
TCB*  ptaskFS;  /* file server task */

ptaskFS = smx_TaskCreate(ptaskFS_main, TP2, SMX_FL_UMODE, "ptaskFS");
mp_MPACreate(ptaskFS, &mpa_tmplt_ptaskFS, tmsk);
```

Then the portal is created as follows:

```
TPSS   fs_pss;
TPCS*  fs_pcl[] = {&fs_pcsA, &fs_pcsB };

 fs_pss.stask = ptaskFS;
 fs_pss.sid     = FP;
 mp_TPortalCreate(&fs_pss, fs_pcl, sizeof(fs_pcl)/4, dsn, "fsportal", "fsportal sxchg");
```

where fs_pss is the TPSS for the file server portal, fs_pcl is the PCL array, which in this case contains two client structure addresses, dsn is a dual MPA slot number[19], "fsportal" is the portal name, and "fsportal sxchg" is the portal server exchange name. mp_TPortalCreate() creates a server exchange, sxchg, then loads the sxchg handle and the portal name into each client TPCS in the tunnel portal's PCL. (Each client needs an alias sxchg handle because it cannot access the real sxchg handle in the TPSS of the server.) Loading the portal name into each portal client TPCS helps to avoid confusion during debugging — a given client may be able to connect to more than one portal and thus would have more than one TPCS.

Note that the portal task handle and server ID are loaded into the portal TPSS. FP means File Portal. This is where the portal is assigned to a specific server partition.

In some cases, a partition may have more than one portal. This could occur if the partition allows simultaneous operations to take place. For example, a file system partition could have more than one portal in order to permit simultaneous accesses to different disk drives, such as an SD card and a thumb drive.

5.4.3 Open Portal (by client)

A client opens a portal when it wants to use it, by calling:

```
mp_TPortalOpen(tpch, msz, thsz, pri, tmo, ssname, csname);
```

where tpch is the handle of the TPCS for the portal, msz is the pmsg data block size, thsz is the total header size (msg header + service header), pri is the priority of pmsgs to be sent, tmo is the csem timeout, sname is the server semaphore, ssem, name, and csname is the client semaphore, csem, name. This function creates ssem and csem semaphores, loads information into the TPCS, sets the TPCS open flag, and creates the portal OPEN pmsg.

The OPEN pmsg includes portal information that the server needs, such as the ssem and csem addresses (the server cannot access these in the client's TPCS), portal type, message data size, mdsz, (msz – thsz), thsz, and control flags. It then sends the OPEN message to the sxchg of the portal and waits at csem up to tmo ticks for a signal from the server.

Note: tmo may need to be very large if other clients are using the portal or if the portal server has low priority. On the other hand, tmo should not be infinite because the client task would stop running indefinitely if there were a problem with the portal or the server, which could happen if the server were hacked or had a bug that stopped it.

5.4.4 Open Portal (by server)

When a pmsg is received at sxchg, a pointer to its data block is stored in the local variable, mhp (message header pointer), and the pmsg data block region is loaded into MPA[hsn] of the server and into MPU[hsn+fas] if hsn is an active slot. *hsn* means *host slot number*. The host, in this case, is the tunnel portal server. hsn is derived from tpch->dsn, which was loaded by mp_TPortalCreate(), as shown in section 5.4.2 Create Portal (by server).

---

[19] See section 5.2.4 Dual Slots for ARMM8 for a description of dual slot numbers and why they are necessary for ARMM8.

The first pmsg received from a tunnel portal client must be an OPEN pmsg. Otherwise it is discarded[20], the INV_CMD portal error is reported, and the server goes back to sxchg to get or to wait for the next pmsg. If an OPEN pmsg has been received, information is loaded from it into the portal TPSS. This information includes the message header pointer, mhp, service header pointer, shp, the message data pointer, mdp, message data size, mdsz, the csem and ssem addresses, and the TPSS open flag is set. The server then signals csem and waits at ssem for the next operation with the timeout specified by mp_TPortalServer(). Unlike a free message server, it does not go back to sxchg for another pmsg. The message data section of the pmsg data block becomes the *portal buffer, pbuf.*

It is important to note that there is an open flag at each end of a tunnel portal – in the client's TPCS and in the server's TPSS. This is necessary because the client and server partitions are isolated from each other – neither can see the other's data. Only pbuf is common between them. An important flag, such as the open flag, cannot be put in pbuf because it sometimes contains garbage. Of course, having two flags creates a flag synchronization problem – one end of the portal may think it is open and the other end may think it is closed. This is resolved via semaphore timeouts – the open end times out and closes its end. Then the two ends agree.

Note: Protocol timeouts are typically set to infinite during debug, else they may cause confusion in the debug process. However, they must be set to finite values in shipped systems, else a vulnerability will exist. Timeouts must be carefully chosen – too short and they can trigger shut-downs during normal operation; too long and they can cause excessive system stalls.

5.4.5 Send and Receive Data (by client and server)

A client sends and receives commands and data with:

```
mp_TPortalSend(tpch, dp, rqsz, tmo);

mp_TPortalReceive(tpch, dp, rqsz, tmo);
```

The portal server task is normally very simple, as follows:

```
TPSS   pssa;

void ptaskA(void)
{
  mp_TPortalServer(&pssa, STMO);
}
```

where pssa is the TPSS for the portal, and STMO is the timeout for the *ssem* semaphore of the portal. mp_TPortalServer() is part of the tunnel protocol and is discussed in section 5.5.3.

Send and receive are performed by the SEND and RECEIVE commands, respectively, as shown above in Figure 5.8. Figure 5.9 shows multiblock send, in more detail. (Note that operation in this figure begins with the Start ellipse in the center of the figure and that flow is upward on the left half of the figure. It was done this way in order to minimize line crossings.)

[20] Unless it is a free message – see section 5.6 Sending Free Messages to Tunnel Portals.

The tunnel protocol is capable of sending data blocks larger than pbuf size, as illustrated by the client work buffer in this figure. In this example, the work buffer is a little more than three pbuf sizes. Therefore, four sends are necessary. Of course, best performance is achieved if pbuf is large enough to send the whole work buffer at once. However, in tight memory systems that may not be feasible, or the amount of data may be far too large for a buffer.

As shown in Figure 5.9, the client sends a block of pbuf-size to the server. This involves copying it from the client's work buffer to pbuf, setting the end of data, eod, flag, if it is the last block, signaling ssem, and then waiting at csem.

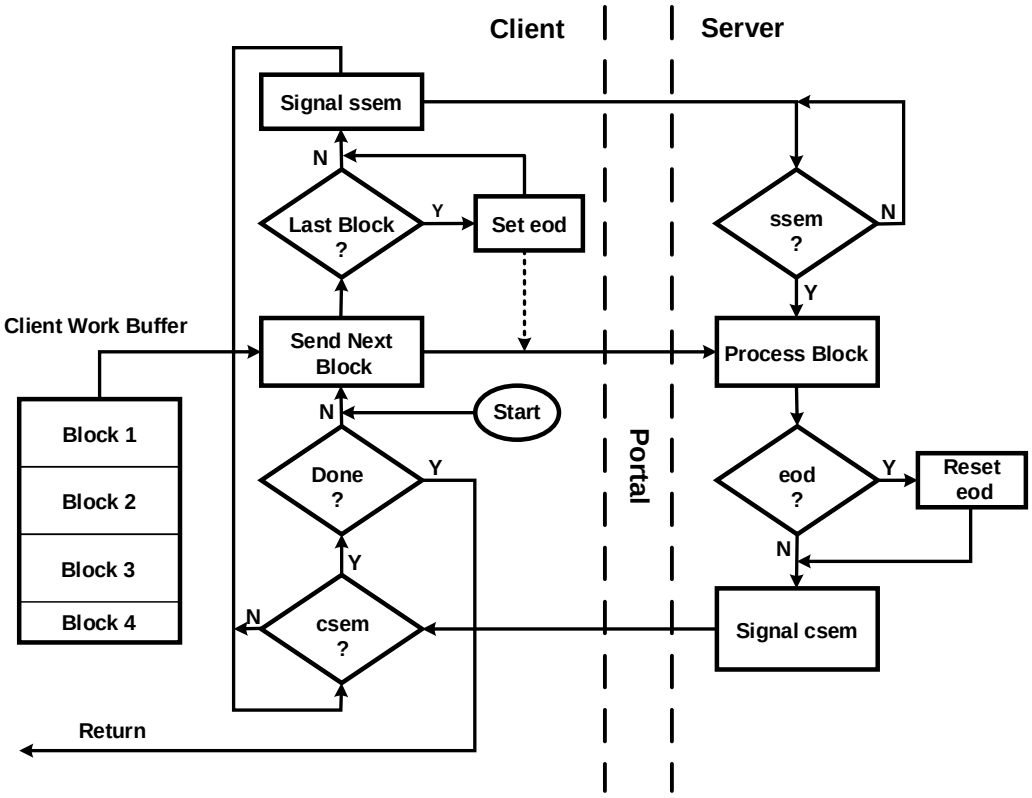

Figure 5.9 Multiblock Send

The ssem signal causes the server to start running. It accepts the data block in pbuf and processes it. In the case of a file write, for example, this means calling the file write function to write what is in pbuf to the disk. For example, if the client were writing a 32 KB block and pbuf were only 512 bytes, then 64 file writes would be necessary. That would clearly impact performance vs. a direct write with no portal. If the server had a larger working buffer, it would be more efficient to copy pbuf-size blocks to it until it was full, and then call the server's disk write operation. This could be handled in the server's switch statement (see discussion in the next section).

After it has handled the block in pbuf, the server resets eod, if it is set, then signals csem. This causes the client to resume running. If the send operation is done, control returns to the point of call (i.e. the shell function – see section 5.5 Shell Functions). Otherwise, the next pbuf-size block, or the smaller ending block, is loaded into pbuf, and the process repeats.

Figure 5.9 might seem to contradict client/server isolation. The line from “Send Next Block” to “Process Block” is implemented by pbuf. How are the “Signal ssem” and “Signal csem” lines implemented? They are implemented via the smx_SemSignal(sem) and smx_SemTest(sem) functions, where sem is the real csem or ssem handle in the Client and it is the alias csem or ssem handle in the Server. Furthermore, if the Client or Server are utasks, then smx_SemSignal() and smx_SemTest() are called via the SVC Handler; however for ptasks, they are called directly. As a result of the foregoing the Client and Server are fully isolated from each other – there are no sneak paths between them.

Receive is the reverse process and its flow chart is nearly the same as for send and thus is not repeated here.

There are two ways that send and receive can be made faster:

- Increase pbuf size.
- No-copy operation. In this case pbuf is used as the work buffer at either or both ends. This can result in performance nearly as good as direct calls.

For calls that don’t send or receive data blocks but only make API calls with a small amount of parameter data, the following call is used:

```
mp_TPortalCall(tpch, tmo);
```

This is actually a macro that calls mp_TPortalSend(), but it omits unneeded parameters, and its name makes it function clear.

5.4.6 Close Portal (by client)

A client must close a portal when a transaction is done so another client can use the portal. Even if a portal is used by only one client, it may make sense to close it when not in use, in order to save resources (semaphores and memory if the pmsg is released). This would be particularly true if the portal is seldom used by this client. Closing a portal is shown in Figure 5.8 when the data transaction is finished. The client sends a CLOSE command to the server and then waits on csem. When the server signals csem, the client clears the message header in pmsg and unneeded fields in its TPCS and its open flag, and then deletes csem and ssem. At this point, the csem and ssem fields in the TPCS should be NULL, and the pname, sxchg, pmsg, and mhp fields should still be valid. (These fields are cleared by other operations.)

Note: When objects are deleted, their handles are replaced with NULL. Unfortunately, NULL shows false data in object fields in the watch and locals windows, which can be confusing during debugging. However, defining a nullcb with 0’s in every field to use instead, has proven to be too error prone.

5.4.7 Close Portal (by server)

When the server receives a CLOSE command, it clears all unneeded fields in its TPSS and its open flag. It then signals csem, loads NULL into TPSS.csem and waits at sxchg for the next OPEN or CONTROL pmsg from a client.

5.4.8 Delete Portal (by server)

If the portal is no longer needed, it can be deleted, as shown in Figure 5.8. This is usually done by a system function that runs in pmode rather than by the server, itself, by calling:

```
mp_TPortalDelete(tpsh, pcpl, pclsz);
```

tpsh is the TPSS handle, pclp is a pointer to the permitted client list for the portal, pclsz is the size of the pcl list. mp_TPortalDelete() first stops the server task with an INF timeout, so it cannot run. Then sxchg is deleted. For each TPCS in pcpl, the pname field is replaced with “no portal” and NULL is loaded into the sxchg field. The latter prevents the portal from being reopened by a client. Then, in this portal’s TPSS, “no portal” is loaded into its pname field, and NULL is loaded into its sxchg, csem, and ssem fields.

Deleting the task is left up to application pcode. This is because the task was created outside of the portal and may need a clean shut down before deleting it, or it may be desired to restart it or to leave it stopped until it is needed again. Deleting a portal has the advantage of freeing an exchange for other use, and if the task is a one-shot task[21], its stack is also freed for use by another one-shot task. When the task is deleted, NULL is loaded into tpsh->stask.

5.5 Shell Functions

When using a portal, an API call made from a client is not a call to the service but instead a call to a shell function that loads the function parameters into a message and sends it to the server to perform the actual call. This is true of both free message and tunnel portals, except that for a tunnel portal the message is not actually sent because it remains in place as the portal buffer.

5.5.1 Mapping Functions to Shell Functions

Ideally, the shell function called on the client side would have the same name as the actual function, so the application code is the same whether calling through a portal or calling directly. However, for embedded software running in Micro-Controller Units (MCUs), all code is linked into a single executable and thus has a single name space. To solve this problem, a mapping file is used that defines macros to translate API calls to similarly named shell functions. Figure 5.10 shows how this works. Although shell functions may be used for either type of portal, they are more likely to be used for tunnel portals, so tunnel portals are assumed in the discussion that follows.

[21] One-shot tasks are a unique feature of smx. Rather than having an internal infinite loop, a one-shot task has a straight-thru main function. Hence, when done it is able to release its stack back to the stack pool since it is no longer needed. One-shot tasks are recommended for servers that seldom run.

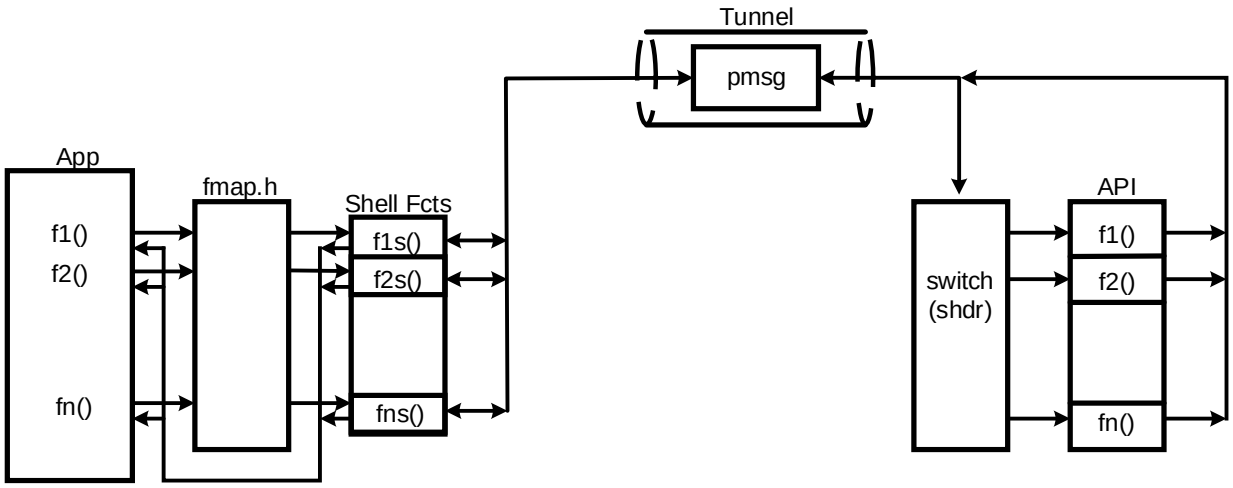

Figure 5.10 Shell Functions

The application makes API calls, f1(), f2(), etc. These are converted to shell function calls f1s(), f2s() etc. when fmap.h is included in the app modules.[22] fmap.h consists of statements for each API function such as the following:

```
#define f1(a, b)  f1s(a, b)
```

To go back to direct calls, it necessary only to comment out the header file:

```
//#include “fmap.h”
```

(Of course, direct calls work only if the client and server partitions are still joined into a single partition, as shown in Figure 5.2, or if they have common regions for the API functions and data in their templates.)

5.5.2 Creating a pmsg

When fmap.h is included in the application modules that make API calls, the calls are directed to the shell functions, as shown in Figure 5.10. Each shell function creates a portal message, with the format shown in Figure 5.11 and sends the message to the tunnel portal, as shown in Figure 5.10[23].

[22] In SecureSMX portals, the prefix ends with p (e.g. sfsp_fread() vs. sfs_fread()) to indicate it is the portal version of the function, and there is generally a separate fmap file for each server (e.g. fpmap.h for smxFS portal).

[23] For simplicity, a pmsg is said to be *sent* and *received*, but for a tunnel portal, the message is actually written into pbuf and read from pbuf, as shown in Figure 5.10.

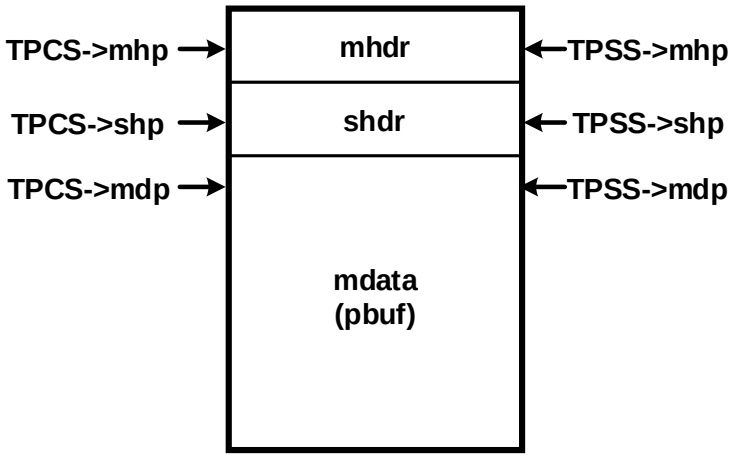

Figure 5.11 pmsg Format

The *message header, mhdr*, is used by the tunnel portal protocol and is not used by the application. The *service header, shdr*, is unique to the server and is used by the application. The rest of the message, mdata (message data) or pbuf, contains the data (if any) needed for the service (e.g. write) or produced by the service (e.g. read).

A typical service header is as follows:

```
typedef struct SFSP_SH {      /* FILE SYSTEM SERVICE HEADER */
    u32    fid;      /* function ID */
    u32    p1;       /* parameter 1 */
    u32    p2;       /* parameter 2 */
    u32    p3;       /* parameter 3 */
    u32    p4;       /* parameter 4 */
    u32    ret;      /* return value */
    void*  caller;
} SFSP_SH;
```

A similar header must be defined for each server or subserver in an embedded system. A shell function fills in this header for the server function that it is translating into a pmsg. The shell functions for a given server are put into a common file, such as fcli.c, and shared between all clients using that server. The header file, fmap.h can also be shared between the clients. Shell functions are typically small, so memory overhead for shell functions is small. For example, a shell function for smxFS is:

```
FILEHANDLE sfsp_fopen(const char* filename, const char* mode, TPCS* tpch)
{
  mp_PTL_CALLER_SAV(tpch);
  SFSP_SH* shp = (SFSP_SH*)tpch->shp;
  char* mp1p = (char*)tpch->mdp;
  char* mp2p = mp1p + strlen(filename)+1;
  strcpy(mp1p, filename);
  strcpy(mp2p, mode);
  mp_SHL2(SFS_ID_FOPEN, (u32)mp1p, (u32)mp2p, NULL);
  mp_TPortalCall(tpch, 0, 0, SFSP_CTMO);
  return (FILEHANDLE)(u32)shp->ret;
}
```

where tpch is the TPCS handle for the file system client. mp_PTL_CALLER_SAV() causes the caller return address to be saved in the SFSP_SH caller field. This field is useful during debugging to see where the call was made in the client. (The debugger call stack window does not show this due to switching from the client stack to the server stack. See 9.2.10 Portal Debugging.) sfsp_fopen() has two literal string parameters. These are pointers to the strings stored in a client code region and thus are not accessible to the server. So, two pointers into pbuf are created and these are used by two strcpy operations to load the actual strings, filename and mode, into pbuf. (How to handle different types of parameters is discussed in section 8.9.3 Tunnel Portal Client Shells and Server Cases for Most Calls.)

Now the mp_SHL2() (Service Header Load 2 parameters) macro is called to load the portal service header: SFSP_SH. SFS_ID_FOPEN is loaded into *fid*, pbuf string pointers are loaded into *p1* and *p2*, and NULL is loaded into *ret.* (NULL is the default error return value for sfs_fopen(), in case the portal operation fails.) Other mp_SHLn() macros are available for n parameters. Note that SFSP_SH allows for up to 4 parameters – the most needed for any file system function.

Then mp_TPortalCall() “sends” pbuf to the tunnel portal and waits to “receive” the return value and data, if any, from the sfs_fopen() function called in the server. This is accomplished by signaling ssem, where the server waits, then waiting on csem for a signal from the server. If, for some reason, the server is not waiting on ssem, csem will timeout and a "CLIENT TIMEOUT" error will be reported. In this case, the client would normally reopen the portal and try the file operation again. If the server is still unresponsive, an appeal needs to be made to higher-level recovery software.

5.5.3 Portal Server Operation

Assuming all is going well, the tunnel portal server processes the contents of pbuf, as shown on the right side of Figure 5.10. Its function is structured as follows:

```
void mp_TPortalServer(TPSS* psh, u32 stmo)
{
  while (psh->pmsg = smx_PMsgReceive(psh->sxchg, (u8**)&mhp, psh->dsn, SMX_TMO_INF))
  {
    if (mhp->type == TUNNEL)
    {
```

```
if (mhp->cmd == OPEN) /* first command must be OPEN */
{
  do /* continuously loop while open */
  {
    switch (mhp->cmd)
    {
      case OPEN:
        …
      case SEND: /* from client */
        mp_TPortalCallServerFunc(psh);
        …
      case RECEIVE: /* to client */
        mp_TPortalCallServerFunc(psh);
        …
      case CLOSE:
        …
      default:
        mp_PortalEM((PS*)psh, INV_CMD, &mhp->errnum);
    }
    smx_SemSignal(psh->csem);

    /* if portal is open, wait for client request */
  } while (psh->open && smx_SemTest(psh->ssem, stmo));
    …
```

The foregoing implements the tunnel protocol at the server end. smx_PMsgReceive() can be replaced with smx_PMsgReceiveStop() so that ptaskA releases its stack while it is waiting for the next pmsg (i.e. ptaskA is a one-shot task). This can significantly reduce memory required if there are many tunnel portals in a system.

mp_TPortalCallServerFunc() is a *call-back function* to which server call-back functions can be added as follows:

```
void mp_TPortalCallServerFunc(TPSS* psh)
{
  switch (psh->sid)
  {
    case SFSP:
      sfsp_server(psh);
      break;

    case …
```

This function interprets the *server id, sid*, field in the server's TPSS and branches to the sfsp_server() or other server function. (The sid field is loaded when the portal is created – see section 5.4.2 Create Portal (by server).) The server function consists primarily of a switch statement that interprets the function id, fid, in the file system service header (see SFSP_SH above) and calls the appropriate function in the server, such as:

```
shp->ret = (u32)sfs_fopen((const char *)shp->p1, (const char *)shp->p2);
```

with the parameters from the service header. Note that, as described in section 5.5.2 Creating a pmsg, the literal strings are actually in pbuf and the p1 and p2 point at them. The return value of the server function is put into the service header ret field, as shown. Data or status information to be returned to the client, if any, is loaded into pbuf. The return value is routed back to the application point of call by the shell function, as shown in Figure 5.10. Information in pbuf is passed back to the caller, as expected. Thus the portal is transparent to the application, except for some added delay. (How to handle different types of parameters is discussed in section 8.9.3 Tunnel Portal Client Shells and Server Cases for Most Calls.)

Addition of a server portal requires creating the following:

- A mapping header file.
- Shell functions.
- Server task with a switch statement to call actual functions in the server.

As noted in the beginning of this section, shell functions are primarily used with tportals, but can also be used with fportals. The console portal is an example of the latter — see cpcli.c and cpsvr.c modules for sample code.

5.6 Sending Free Messages to Tunnel Portals

There are times when it is advantageous to send a high-priority message to a partition to perform an urgent action, such as shutting the partition down or reconfiguring it. There are other times when it is desirable to send frequent low-priority status inquiry messages. In cases like these it is desirable to bypass the normal tunnel protocol. For these reasons, tunnel portals can accept free pmsgs.

A pmsg received from a free message client portal must have a CONTROL command. Any other free pmsg is discarded, the INV_CMD portal error is reported, and the server goes back to sxchg to wait for the next pmsg.

Free pmsgs can be sent to tunnel portals using mp_FTPortalSend(). This automatically creates a message header for the pmsg. A service header can be created by application code, if necessary. Free pmsgs are used for sending commands or single data blocks to a tunnel portal or receiving single data blocks from a tunnel portal.

When a free CONTROL message is received, the tunnel protocol code is bypassed, so operation is simpler and faster than normal tunnel portal operation. pmsgs are sent to the sxchg for the portal and can be intermingled with other tunnel portal OPEN and free CONTROL messages. If a tunnel portal operation is in progress a free CONTROL message will not be received until the tunnel portal is closed. After the server processes control or data in the pmsg it can load the pmsg with status or data and send it to an rxchg using smx_PMsgReply(). This works as shown in Figure 5.6C. If no rxchg is specified, the pmsg is released back to memory as shown in Figure 5.6A.

The following illustrates how to get and send a free pmsg via a tunnel portal to a file system server:

```
FPCS*    fpch;
u8*      dp;
SFSP_SH* shp;

fpch->pmsg = smx_PMsgGetHeap(SPBUF_SIZE, &dp, CL_SLOT, DATARW);
shp = (SFSP_SH*)((u32)dp + sizeof(TPMH));
shp->fid = SFS_ID_SUPCLOSE;
mp_FTPortalSend(fpch, dp, fpch->pmsg);
```

In the server, the free pmsg bypasses the tunnel protocol code:

```
void mp_TPortalServer(TPSS* psh, u32 stmo)
{
  while (psh->pmsg = smx_PMsgReceive(psh->sxchg, (u8**)&mhp, psh->dsn, SMX_TMO_INF))
  {
    if (mhp->type == TUNNEL)
    {
       // tunnel protocol code
    }
    else if (mhp->type == FREEMSG)
    {
      switch (mhp->cmd)
      {
        case CONTROL:
          mp_TPortalCallServerFunc(psh);
```

and goes directly to mp_TPortalCallServerFunc();

```
void mp_TPortalCallServerFunc(TPSS* psh)
{
    …
  switch (psh->sid)
  {
    case SFSP:
      sfsp_server(psh);
      break;
```

which interprets the sid field in the server's TPSS and branches to:

```
void sfsp_server(TPSS* psh)
{
  switch (shp->fid)
    …
    case SFS_ID_SUPCLOSE:
       // close USBH portal
       smxu_PMsgReply(ph->pmsg);
```

This closes a portal between a file system partition and a USBH partition. This command would normally be issued by system software, not by a file system client. After closing the portal, smx_PMsgReply() sends the free pmsg to the client's rxchg.

The following shows how the client fetches the pmsg, obtains the result of the USB portal close operation, then releases the pmsg to memory:

```
pmsg = mp_FPortalReceive(fpch, &dp);
ret = shp->ret;
smx_PMsgRel(&pmsg);
return(ret);
```

Hidden in the above code is the reply exchange, rxchg It is created by mp_FPortalOpen() and stored in the client's FPCS. When a pmsg is sent, the index of rxchg is calculated and loaded into pmsg->rpx. This is accessible to smx_PMsgReply(), but not to the server. mp_FPortalReceive() fetches the pmsg from the rxchg, the return value is extracted from it, then it is released..

As noted above, free pmsgs can be intermingled with tunnel pmsgs. A free pmsg must wait for the current tportal operation to finish. Then the server will come back to the sxchg for the next pmsg. If the free pmsg has higher priority than other waiting pmsgs or is ahead of waiting pmsgs at the same priority level, it will be accepted next. This is useful for system control functions such as shutting down a server quickly or obtaining statistics about it.

A free pmsg cannot interrupt a current server operation, regardless of its priority. Some other mechanism must be used for that, such as stopping the server portal task.

5.7 Other Portal Topics

5.7.1 Portal Access Delays

For shared portals, transaction times must be controlled so that every client gets fair access to the server. An important client can receive preferential treatment by assigning it a higher priority when it opens a portal. This priority is passed in pmsgs sent to the portal by the client. An sxchg will deliver to its portal the longest waiting pmsg at the highest-priority, and that priority will be adopted by the server while it is processing the pmsg.

However, high-priority clients cannot interrupt low-priority clients. Hence, priority inversions are possible. To minimize this, clients should not wait at mutexes, semaphores, etc. when they have a portal open. However, a client can be preempted by higher-priority task. As with mutexes, priority inheritance is implemented for pass exchanges in order to minimize priority inversions. (See smx Reference Manual for smx_MsgXchgCreate()). The server exchange that is created by mp_TPortalCreate() is a pass exchange with priority inheritance.

It may seem that direct service calls are better because a higher priority task can preempt a task in the middle of a service call. However, service calls are generally protected from reentry by smx_srnest or by mutexes, so the preempting task will end up waiting, anyway. Portals have an advantage over smx_srnest and mutexes, in that it is the client's pmsg, not the client itself that waits for the service. Furthermore, the pmsg priority can be less than that of the client. Since the server inherits the pmsg priority, it will wait, in this case, for the client to complete its work. This may be advantageous if there is work that the client can do independently of the service.

5.7.2 Portal Errors

The following portal errors are detected and reported:

```
CLIENT TIMEOUT
INVALID COMMAND
INVALID FUNCTION
INVALID SID
INVALID SSID
INVALID SIZE
INVALID TYPE
NO PMSG
PORTAL CLOSED
PORTAL NOT EXIST
PORTAL NOT OPEN
SERVER TIMEOUT
TRANSMISSION ERROR
TRANSFER INCOMPLETE
```

When an error is detected,

```
mp_PortalEM(ph, errnum, ep)
```

is called, where ph is the portal structure handle. Only the name in these structures is used and it is the first field in all portal structures. errnum is the error number and ep points at the location to store errnum in either the pmsg header or in the portal structure. This function runs in pmode. Currently mp_PortalEM() saves errnum in *ep, displays the portal name, if any, followed by the error name, and makes an entry in the smx Error Buffer consisting of: etime, errnum, and ph.

5.7.3 Chained Portals

Chained portals occur when a partition is accessed via a portal and it accesses another partition via another portal. An example is when the file system accesses a mass storage USB device (e.g. a thumb drive) via the USB host server portal. Chaining portals increases overhead. It can also result in poor performance because of incompatibilities between portals. For example, the first portal may be using a pbuf size which is smaller than the optimum sector or block size used for pbuf in the second portal. Causing a device to accept blocks that are smaller than its optimum block size can seriously impact its performance.

Problems like this may require more complex shell functions and switch functions. For example, the write shell function for the second portal could buffer up small blocks received from the first portal until either its pbuf is full or the last block has been received from the client. Then the second portal write shell function would write its pbuf and the device would operate more efficiently. Probably the first portal switch function would need to alert the second portal shell function when the last block was being sent. So a little complexity needs to be added to the portals. However basic partition code (e.g. file system code) may not be changed.

5.7.4 Server Callbacks

Synchronous server callbacks can be implemented using free message portals by sending callback messages to reply exchanges and callback response messages to server exchanges. If done this way, the portal belongs to the server, and all callback clients are in its PCL list. So each

client expecting a callback would send a pmsg to the server callback portal and the server would send the callback pmsg to each client rxchg, in turn.

If this is not the desired mode of operation, then each callback client could have a callback portal and the server would send its callback pmsg to each client's callback sxchg. Then each client would send its response pmsg to the server's rxchg.

The above operations can be made more efficient by using smx message *multicasting* or *broadcasting* (see the smx User's Guide).

Support for synchronous server callbacks will be added to the tunnel protocol, if a need develops. This would be more efficient since the callback would be through the tunnel portal and no additional portals would be needed. Asynchronous callbacks require a separate portal.

5.7.5 Who's The Boss?

Despite the fact that the server creates the portal, the client is always the boss (master). All transfers are initiated by clients. Even though the client might prime a portal for a callback to itself, as discussed above, the client is still the boss. Keeping this in mind helps to see when a second portal is needed in the opposite direction.

5.7.6 Client Data

Client shell functions usually do not need any global data, but if they do, it is best to avoid using a precious MPU slot, such by as adding a ucom_data region. The client structures have a 32-bit data/pointer field that can hold a word or a data pointer. In many cases, a single word is sufficient for a shell function. In cases where it is not, the data/pointer field can be defined as a pointer to a structure in the TLS of the current task (see section 4.11.8 Task Local Storage). Then, the client structure data/pointer field is typecast to a pointer to the structure stored in the TLS. The address of the TLS can be obtained by calling:

```
dp = smxu_TaskPeek(smx_ct, SMX_PK_TLSP);
```

An example of using a client structure data field for one word, occurs in the smxUSBH mass storage portal, where it is used to store the disk sector size used by client functions. The file sector size varies from device to device; thus it is necessary to call a driver function to get it. Rather than calling the driver function every time the sector size is needed, the driver function is called once and the sector size is stored in pch->data.

5.7.7 Window Portal

A window portal is like a pane of glass: it maintains isolation between partitions, yet what is written on one side of the pane can be read (and overwritten) on the other side. A window portal is simply a dynamic region (see 4.11.9 Dynamic Regions) that is shared between two or more tasks. Each task has the region in one of its active MPA slots and thus can access the data block of the window region. If the tasks have the same priority, then flags in the window can be used to coordinate writing and reading. Otherwise, a mutex or semaphores can be used.

Unlike the free message and tunnel portals, a window portal tends to be a fixed region for each task using it. The window region could be kept in an auxiliary MPA slot and switched into an active slot, when needed.

5.8 Console Portal

The console functions are implemented in bcon.c, and their use is described in the smxBase User's Guide Section 2.8. If the console partition is implemented, bcon.c, a UART driver, and cp_main() and cp_server() (both in cpsvr.c) are located in cp_code and related data in cp_data.

The following console functions are available via the console portal:

```
void    sb_ConClearScreen(FPCS* pch);
void    sb_ConClearScreenUnp(FPCS* pch);
bool    sb_ConDbgMsgMode(FPCS* pch);
void    sb_ConDbgMsgModeSet(bool enable, FPCS* pch);
int     sb_ConPutChar(char ch, FPCS* pch);
void    sb_ConWriteChar(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink,
                                                                    char ch, FPCS* pch);
void    sb_ConWriteString(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink,
                                                        const char *in_string, FPCS* pch);
void    sb_ConWriteStringNum(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink,
                                                const char *in_string, u32 num, FPCS* pch);
```

cpmap.h must be included in any module calling these function to convert the function names to shell function names in cpcli.h, which is contained in ucom_code. Each shell function converts one of the above function calls to a pmsg and sends it to the console portal exchange. There it is converted back by cp_server() and executed in the console partition. Console portal calls can be made by both ptasks and utasks.

The console portal is enabled by setting CP_PORTAL in xpcfg.h. This results in cp_init() being called to create a free message portal for the console partition, including its portal exchange, cp_sxchg and portal task, cp_task. It also starts cp_task waiting at cp_sxchg for a pmsg and initializes the cp portal server structure, cpsvr.

The cp following system-level variables are defined in smxmain.c, if CP_PORTAL:

```
#pragma default_variable_attributes = @ ".cp.bss"
FPSS    cpsvr;      /* cp server struct */

#pragma default_variable_attributes = @ ".cp.data"
FPCS* cpcli_lst[] = {&cpcli_t1c, &cpcli_t2c};     /* permitted client list for cpsvr */
u32     cpcli_lstsz = sizeof(cpcli_lst)/4;            /* size of cpcli_lst */
```

The above assumes two client tasks, t1c and t2c. Any number of client tasks is permitted. cpcli_lst[] is used to initialize the client FPCS structures:

```
#pragma default_variable_attributes = @ ".t1c.bss"
FPCS    cpcli_t1c;      /* cp client structure for t1c*/

#pragma default_variable_attributes = @ ".t2c.bss"
FPCS    cpcli_t2c;      /* cp client structure for t2c*/
```

When the system is running, only these clients can access the console portal. Other tasks lack the address of the portal exchange and therefore cannot send pmsgs to it.

Before making console calls, a client must call:

```
bool  mp_FPortalOpen(pch, csn, msz, 1, pri,  tmo, rxname);
```

to open its side of the console portal. In the above, pch is the handle of the client portal structure, csn is the MPU slot to use for the pmsg data region, msz is the size of pmsg data, only one pmsg is allowed, tmo is the timeout to wait on the rxchg for the pmsg, and rxname is the name of the rxchg.

The console portal can be used with minimal code changes in the client. This is accomplished by preceding client console function calls with:

```
#include "cprtl.h"
#include "cpmap.h"
#define CP_PCH &cpcli_t1c
```

The first provides defines and prototype functions, the second maps functions to shells, and the last provides the cp client structure handle for shell functions to access the t1c's rxchg.

5.9 Middleware Portals

SMX middleware has already been partitioned using portals. The following sections summarize what was done for each. For all but smxNS, the portal is at the API layer, meaning API calls are done via protected messages which are handled by a server task and the results are returned. Also, the client and server code run in umode. For smxNS, the portal was done at the TCP/UDP layer, and the smxNS high level partition runs in pmode, and low-level runs in umode.

5.9.1 smxFS

A tunnel portal is created for each disk. This is done by calling sfsp_init() from sfs_devreg(), which is the function to register each disk with the file system. Similarly sfs_devunreg() deletes the portal via sfsp_exit().

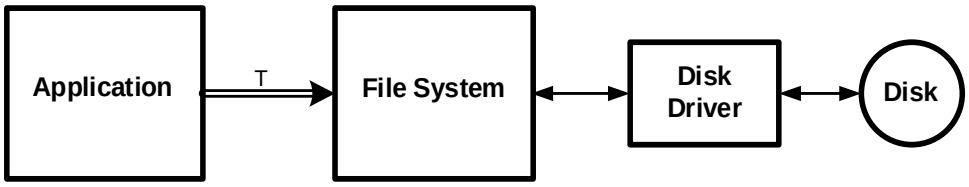

Figure 5.12 smxFS Portal

Demo Configuration: SMXFS and SMXFS_DEMO enabled (iararm.h). SFS_DRV_MMCSD=1 (fcfg.h). SFS_PORTAL=1 (xpcfg.h). Link fsdemo.c and XFS files.

Arrays of portal server and client structures and permitted client lists are indexed by the disk ID (0, 1, etc.). Server structures are defined in fpsvr.c, and client in the demo and user application files.

fpsvr.c has the init/exit routines that create and delete the portals, and it has the server task main function, which is a large switch statement that calls the actual services requested by ID with the parameters passed via the portal message.

fpcli.c has all of the shell functions that convert API calls into protected messages and passes them through the portal.

fpmap.h has the mapping macros that convert API calls into portal calls by adding a p to the prefix and passing the PCH (portal client handle) as a last parameter.

fprtl.h has definitions including configuration values for timeouts and MPA slots, the service header structure (SH), and portal shell function prototypes.

Application code calls mp_TPortalOpen() to open the portal from a task that wants to access a disk. If another task needs to access the same disk, the first task needs to close the portal with mp_TPortalClose(), and the other task must open it. See fsdemo.c.

USB disks add complexity because they require smxUSBH which must be accessed via its portal. So file operations go through the smxFS portal, and smxFS then does operations through the smxUSBH mass storage portal. We call these chained portals.

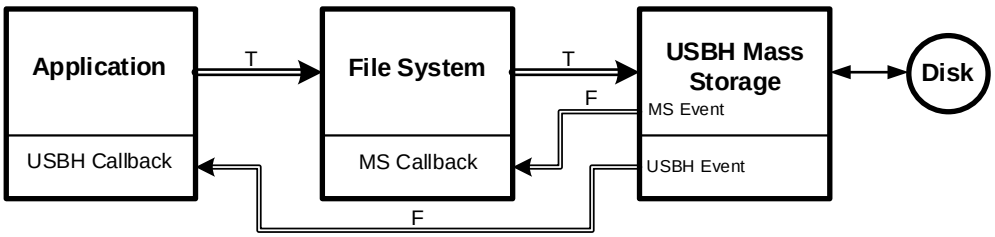

Figure 5.13 smxFS and smxUSBH Mass Storage Chained Portals

Demo Configuration: SMXFS, SMXFS_DEMO, SMXUSBH, and SMXUSBH_DEMO enabled (iararm.h). SFS_DRV_USB=1 (fcfg.h). SFS_PORTAL=1 and SU_PORTAL_MS=1(xpcfg.h). Link fsdemo.c and XFS files and usbhdemo.c and XUSBH files. usbhdemo.c is needed for an event callback portal.

There is some special handling at shutdown for this case (which is not necessary for your system if it never shuts down and deletes the portals). When the application task (e.g. in fsdemo.c) exits, it closes the FS portal. Later smxfs_exit() is called by smx_modules_exit(), which calls sfs_devunreg() to unregister the device, and it calls sfsp_exit() to delete the FS portal. Before doing this, though, it needs to send one last message to the USBH mass storage portal to shut it down, but the portal is closed. Rather than reopen a tunnel, it sends a free message request with command SFS_ID_SUPCLOSE. The smxFS portal server switch statement has this as its last case, and it closes the USBH mass storage portal. It's not immediately clear why this free message PCS/PCH has access to the FS tunnel. The key is that it (sfsp_fmpcs) is listed in the permitted client list for the portal. When the portal was created during startup, the exchange handle and name were loaded into all of the PCSs in the list by the loop in mp_TPortalCreate(). At the top of fpsvr.c:

```
FPCS  sfsp_fmpcs0;
…
u32* sfsp_pcl0[] = {(u32*)&sfsp_demo_lt_pcs0, (u32*)&sfsp_demo_rwt_pcs0,
                    (u32*)&sfsp_fmpcs0};
```

The first two in the list are the fsdemo tasks that normally use the portal.

Another issue related to USB disks regards event callbacks. A disk can be inserted or removed and the write protect status can change at any time, so a file system has to check these each time before doing an operation. This results in a lot of calls through the USB mass storage portal. An alternative to polling status is to use a callback to notify smxFS when these events occur. In the USB disk interface file (fdusb.c) we implemented a free message portal used for the USBH mass storage driver to send status updates. The disk driver (fdusb.c) maintains Inserted, Changed, and WriteProtect flags which the driver routines consult. See USBMSCallbackServerMain().

USB device disks: The foregoing covers the case of smxFS implementing a portal server for its API functions (sfs_fopen(), sfs_fread(), etc.). It also must implement a disk-level portal server to handle accesses from smxUSBD, as we have done for the SD/MMC driver. In fpsvr.c, below the API portal functions, a second portal is defined for the SD disk driver. This handles calls from the smxUSBD mass storage driver.

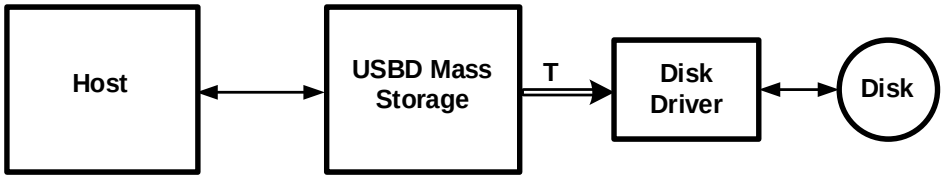

Figure 5.14 smxUSBD Mass Storage Portal

Demo Configuration: SMXUSBD and SMXUSBD_DEMO enabled (iararm.h). SMXFS and SMXFS_DEMO disabled, but link XFS files in project. MSTOR_RAMDISK=0 (usbddemo.c), and SFS_DRV_MMCSD=1 and SFS_DRV_USB=0 (fcfg.h). SUD_MSTOR=1 (udcfg.h), and to also use Serial (section 5.9.3 smxUSBD) set SUD_SERIAL=1 and SUD_COMPOSITE=1. SUD_PORTAL_MS=1 (xpcfg.h). Link usbddemo.c and XUSBD files and XFS files.

5.9.2 smxNS

Unlike the other middleware, this portal is within the TCP/IP stack, at the transport layer (TCP/UDP). smxNS API code, server applications, and user application code that uses networking run in pmode on the “hi” side of the portal, while the transport layer and drivers run on the “lo” side of the portal. The idea is to protect the system from network intrusions, which are the main risk for connected systems.

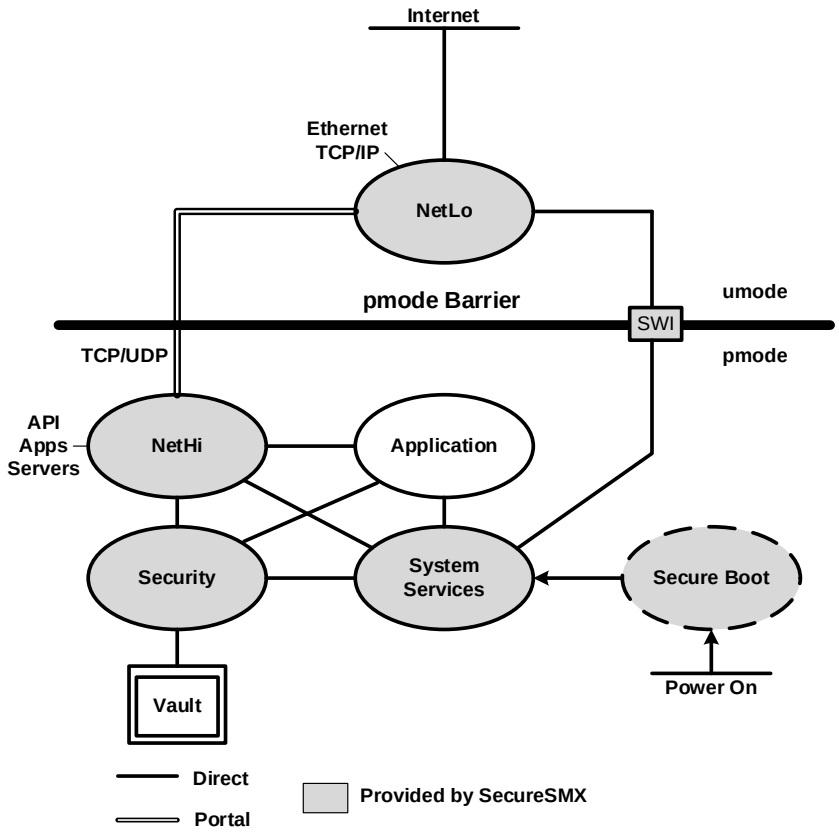

Figure 5.15 smxNS Transport Layer (Hi/Lo) Portal

Demo Configuration: SMXNS and SMXNS_DEMO enabled (iararm.h). SNS_PORTAL=1 and SNS_PORTAL_TCB=1 (xpcfg.h). Link nsdemo.c and XNS files. To use the web server, also link APP\DEMO\WEBPAGE files.

npsvr.c has the init/exit routines that create and delete the portals, and it has the server task main function, which is a large switch statement that calls the actual services requested by ID with the parameters passed via the portal message.

npcli.c has all of the shell functions that convert API calls into protected messages and passes them through the portal.

npmap.h has the mapping macros that convert API calls into portal calls by adding a p to the prefix and passing the PCH (portal client handle) as a last parameter.

nprtl.h has definitions including configuration values for timeouts and MPA slots, the service header structure (SH), and portal shell function prototypes.

smxNS was done differently than our other middleware. Instead of an API portal, it implements a portal at the transport (TCP/UDP) layer to isolate the low-level part of the stack from the high-level part to keep intruders from the Internet out. The high-level part runs in pmode and is directly called by the networking application code, which is also in pmode. For our other

middleware, the whole package runs in umode and is isolated from the application. Data encryption/decryption runs on the pmode side, so it is not possible to access it from the network.

A separate tunnel portal is needed for each application and network server task that uses the TCP/IP stack. During initialization, Ninit() calls NMTinit() which creates all of the portals, and then each task that uses them (application, telnet server, web server, etc.) opens one of the portals.

5.9.3 smxUSBD

smxUSBD function drivers are the APIs to the device stack, so portals are needed for them. Three have been implemented initially, mass storage, mouse, and serial. Mass storage functions are called internally, not by the application, and they use the SD/MMC portal discussed at the end of section 5.9.1 smxFS.

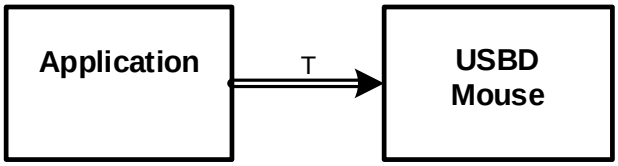

Figure 5.16 smxUSBD Mouse Portal

Demo Configuration: SMXUSBD and SMXUSBD_DEMO enabled (iararm.h). SUD_MOUSE=1 (udcfg.h). SUD_PORTAL_MOUSE=1 (xpcfg.h). Link usbddemo.c and XUSBD files.

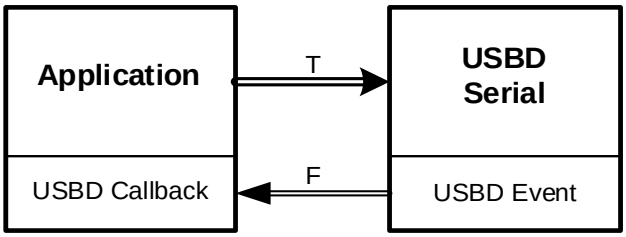

Figure 5.17 smxUSBD Serial Portal

Mass Storage: See end of section 5.9.1 smxFS

Demo Configuration: SMXUSBD and SMXUSBD_DEMO enabled (iararm.h). SUD_SERIAL=1 (udcfg.h). To run with Mass Storage as a composite device, see end of section 5.9.1 smxFS. SUD_PORTAL_SERIAL=1 (xpcfg.h). Link usbddemo.c and XUSBD files.

udpsvr.c has the init/exit routines that create and delete the portals, and it has the server task main function, which is a large switch statement that calls the actual services requested by ID with the parameters passed via the portal message.

udpcli.c has all of the shell functions that convert API calls into protected messages and passes them through the portal.

udpmap.h has the mapping macros that convert API calls into portal calls by adding a p to the prefix and passing the PCH (portal client handle) as a last parameter.

udprtl.h has definitions including configuration values for timeouts and MPA slots, the service header structure (SH), and portal shell function prototypes.

The application task (e.g. in usbddemo.c) opens the tunnel portal and then makes mouse or serial calls.

The serial driver implements a callback for data and line state/coding changes using a free message portal. See USBDSerCallbackServerMain() task function which wraps USBDSerialNotification(). The smxUSBD code that calls the callback calls sud_SerialRegisterPortNotify() to get the address of sud_SerialCallStackCallback() to call it.

5.9.4 smxUSBH

smxUSBH class drivers are the APIs to the host stack, so portals are needed for them. Two have been implemented initially. The mass storage portal is described in section 5.9.1 smxFS. The FTDI232 (serial) portal is the other. It supports connecting an FTDI USB to RS232 adapter.

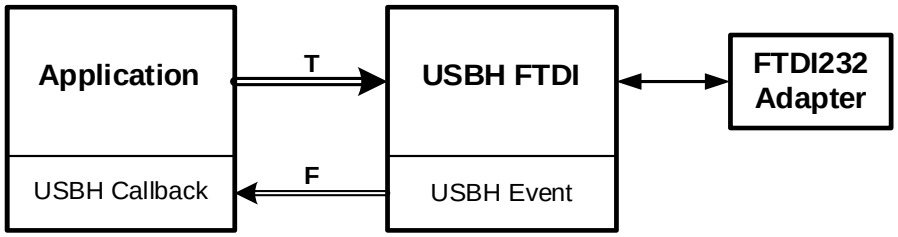

Figure 5.18 smxUSBH FTDI232 Serial Portal

Demo Configuration: SMXUSBH and SMXUSBH_DEMO enabled (iararm.h). SU_FTDI232=1 (udcfg.h). SU_PORTAL_FTDI232=1 (xpcfg.h). Link usbhdemo.c and XUSBH files.

upsvr.c has the init/exit routines that create and delete the portals, and it has the server task main function, which is a large switch statement that calls the actual services requested by ID with the parameters passed via the portal message.

upcli.c has all of the shell functions that convert API calls into protected messages and passes them through the portal.

upmap.h has the mapping macros that convert API calls into portal calls by adding a p to the prefix and passing the PCH (portal client handle) as a last parameter.

uprtl.h has definitions including configuration values for timeouts and MPA slots, the service header structure (SH), and portal shell function prototypes.

For mass storage, smxFS opens the portal and makes USB mass storage calls, as discussed in section 5.9.1 smxFS.

For FTDI232 serial, the application task (e.g. in usbhdemo.c) opens the tunnel portal and then makes FTDI API calls. The demo has a send task and a receive task, and they alternate use of the portal. Each opens the portal, uses it, and then closes it to allow the other to use it. Each task has its own PCH to access the portal. Notice above each of the task main functions in usbhdemo.c (usb_ftdi232_receive_main() and usb_ftdi232_send_main()), SUP_FTDI_PCH is defined to that task's pch variable. This is what is passed as the last parameter to the macros in upmap.h.

smxUSBH has a global event callback to notify the application of several different events. When using portals, this uses a free message portal. See su_CallStackCallback() in udriver.c. usbhdemo.c implements the actual callback function to handle the events. See USBHEventServerMain() task function which wraps USBHEventCallback(), and USBHEventInit(), which creates the free message portal. The smxUSBH code that calls the callback calls su_GetStackCallback() to get the address of su_CallStackCallback() to call it.

5.10 Portal Tips

Portals are arguably the most complex part of SecureSMX, yet they are essential for partition isolation because of the single address space limitation of MCUs. Every partition is likely to require a portal, some may require more than one portal. For example, if a file system can support simultaneous accesses to every device, then a portal per device is needed for best performance. In some cases, portals may be necessary to report asynchronous events or for feedback. The result of all of these requirements could be a very large number of portals in a complex system.

A possible alternative to portals is to implement partition APIs as service functions. This might be applicable to a file system, for example. The downside of doing this is that hmode code must be able to access file system code. Hence the latter and its static data must be included in sys_code and sys_data. This brings potentially vulnerable code below the pmode barrier, thus reducing security. Accessing the file system via a portal is more secure. Portals are a necessary evil for secure systems.

Portals are confusing primarily because we are accustomed to using APIs, not portals, to access services. The following tips may help you to adapt to using portals.

1. It is best to start with a diagram showing all servers and all clients, and then connect the clients to the servers with arrows. We recommend that the arrowheads point at the servers. This is the best time to work out the partitioning and the interconnections in your system. If the diagram becomes very complicated, that may be a hint that you need to rework your system architecture. Once you have a workable diagram, you can begin programming portals and partitions, but not before.
2. Some servers may have multiple portals. For example, the file system may have multiple portals in order to simultaneously access multiple disks. For these servers, you need to decide if all clients access all portals or if some clients access one portal and other clients access a different portal. The latter structure may be simpler to implement.
3. Each client has a portal client structure (PCS) for each server that it can access. The PCS must be in a client MPU region. If many clients are accessing the same two servers, it

may be simpler to merge them into a single server, even though they provide different services.
Alternatively, if a server has a large number of clients waiting for different services, it may be a choke point and the server should be divided into more servers. Then it may be found that only a few clients connect to each of the new servers and the system diagram is simpler.

4. Each server has a portal server structure (PSS) for each of its portals. Each PSS must be in a server MPU region. If simultaneous operation is not necessary, it may be simpler to have a single portal per server.
   On the other hand, if the server is providing different groups of services it may be simpler, within the server, to have a separate portal and portal task for each service group. An example of this is USB host class drivers.
5. For each portal, there must be a permitted client list (PCL) containing the addresses of all the PCSs of the clients that the portal serves. During initialization, which must be done in pmode, the PCL is used to load the handle of the server's exchange and its name into each PCS in the PCL. This is necessary because clients and servers are isolated from each other when running. (If this is confusing see 8.2.6 The Handle Problem.)
6. Remember that it is a client task that opens the portal. Unlike the server side, which creates the portal during system initialization, the client side opens a portal from the task that will use the portal. If you get confused when trying to figure out how to use portals in your existing design, think about the tasks involved and consult your diagram.
7. Attempting to share PCSs between client tasks in the same partition and to share PSSs between server tasks in the same partition, though tempting, is likely to cause confusion, and thus is not recommended. It is better to dedicate each PCS or PSS to a single task.
8. Arrays of PCSs: On the client side, PCSs are typically defined as individual structures. However, PCSs in the same client, can grouped into an array so they can be selected via an index. This might be simpler when a client has access to multiple portals.
9. One-shot tasks: Due to the need for portal tasks, a partitioned system is likely to have many more tasks than a non-partitioned system. Using one-shot portal tasks is recommended because they do not consume stack memory while waiting and portal tasks are likely to do a great deal of waiting.
10. Timeouts: Set csem and ssem timeouts to INF during debug, else debugging is likely to be difficult. However, after debugging is complete, all csem and ssem timeouts must be set to reasonable finite values. Otherwise, a hacker could freeze a client or server and this could propagate throughout the system, bringing it to a halt. Also, if unexpected events or latent bugs occur, timeouts provide a means of recovery.
11. Callbacks: Asynchronous callbacks (e.g. plugging in a USB device) normally require free message portals from the server to the client. If more than one client needs to be notified, smx broadcast or multicast capabilities might be useful. Synchronous callbacks, which occur while a tunnel portal is open probably can be handled via the tunnel portal.
12. Debugging: See section 9.6 Debug Tips.

Chapter 6 Advanced Theory

The topics in this chapter are not necessary to start using SecureSMX but may become useful later in a project. They are intended to provide a little more background information.

6.1 System Services

A *system service* is either an smx System Service Routine (SSR), such as smx_SemSignal(), or a function that performs a system service, such as sb_IRQMask(). SSRs can cause task switches. System service functions cannot.

System services operate the same regardless of whether they are invoked from ptasks or utasks. For example, a utask may test a semaphore and become suspended upon it. A ptask may later signal that semaphore, and the utask will be resumed. Or vice-versa. A ptask may have higher or lower priority than a utask. The smx task scheduler dispatches the tasks according to their priorities, regardless of privilege. What is different is that the ptask executes pcode, whereas the utask executes ucode. Both are limited to access only the memory that the MPU permits.

Although the results are the same, the mechanisms by which utasks and ptasks execute system services are different. These are explained in the next two subsections.

6.1.1 System Calls from pmode

System calls from pmode are the simpler case, and it is the same as when running smx without an MPU. Figure 6.1 shows the operation when a system service is called directly from a ptask:

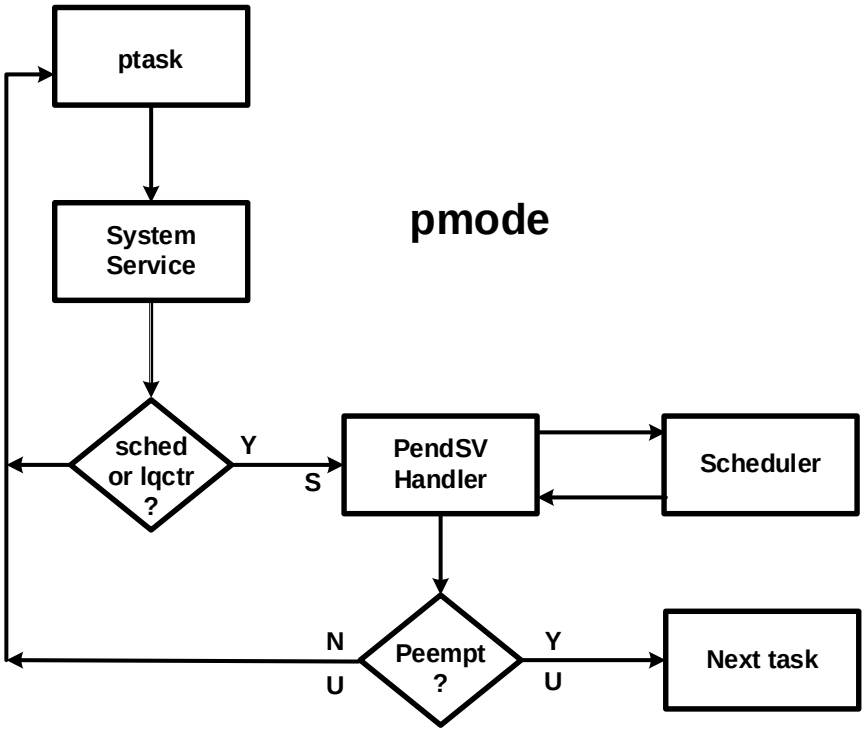

Figure 6.1 System Call from ptask

The system service is executed in pmode using the task stack. If *smx_sched* or *smx_lqctr* is non-zero, indicating that the ptask may need to be preempted, the smx_PendSV_Handler() is invoked. It runs immediately since it is an exception. The S beside the line means *stacking* occurs. Stacking is the process whereby the processer creates a stack frame in the task stack, TS, and then switches to the main stack[24], MS. The PendSV Handler runs in the main stack.

The PendSV Handler then calls the scheduler, which determines if the current ptask should be preempted. If so, the ptask is suspended or stopped, the next ptask or utask is resumed or started, and the scheduler returns to the PendSV Handler. This second part of the PendSV Handler is called its *tail*.

The PendSV Handler performs an exception return to the current ptask if it has not been preempted. Note: only smx SSRs can cause task preemption. The U next to the line means *unstacking* occurs. Unstacking is the process whereby the processer loads saved processor registers from the stack frame in TS, then switches to TS.

[24] The term *main stack* is used by Arm Ltd. smx uses the term *main stack*. These are the same stack.

If the current ptask has been preempted, the PendSV Handler performs an exception return to the next task, with unstacking. However, this is not the same task stack. It is the stack of the task that was suspended and that now is being resumed. The result of unstacking for this task is as though it were never suspended. If the task is being started there are no registers to restore. However the scheduler does a head fake and creates a bogus stack frame, thus fooling the PendSV Handler.

6.1.2 System Calls from umode

Figure 6.2 illustrates how a system call is made from umode. This is clearly more complicated than the system call from pmode. The utask calls a shell function, which in turn invokes the smx_SVC_Handler() with a 1-byte parameter, which is the call ID. The S beside the line means *stacking* occurs.

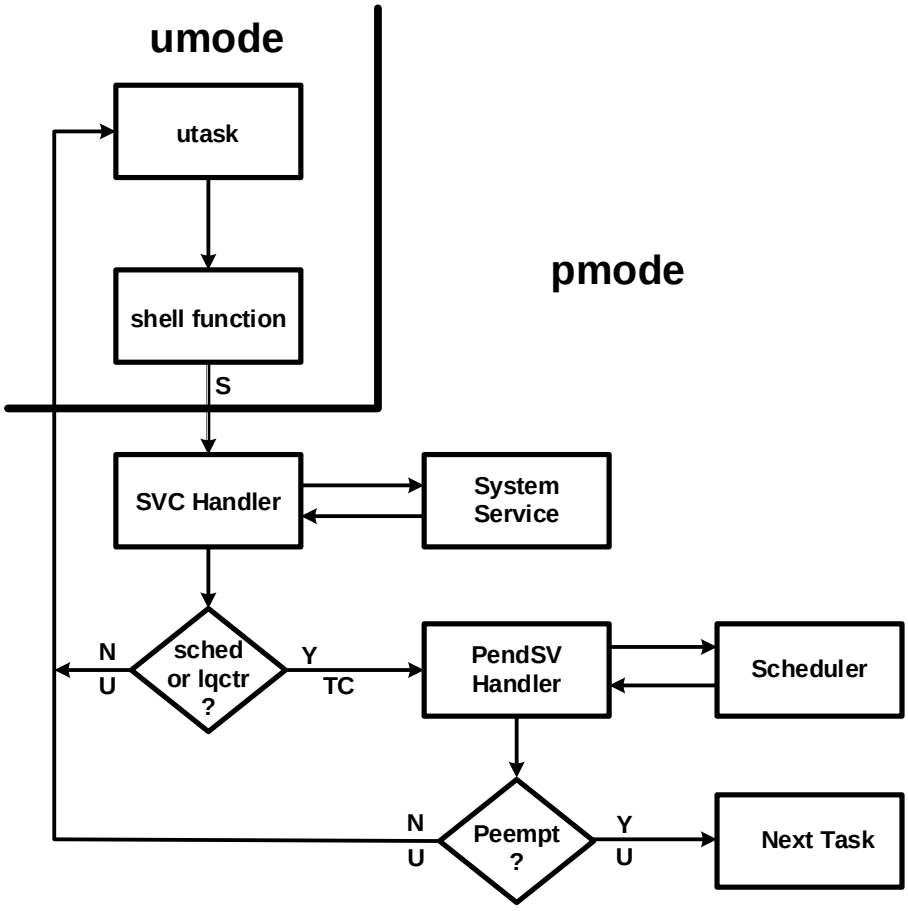

Figure 6.2 System Call from umode

The SVC Handler runs in the main stack. It loads parameters 0 thru 3 from TS into r0 thru r3 and it moves parameters 5 to 7, if any, from below the exception frame in the task stack to the top of the main stack. (This is where the system service expects to find these parameters, by the

ATPCS[25].) The SVC Handler then calls the system service via the SST jump table that is in use. (See section 4.9.2 SVC Call Mechanism.)

If the system call is not *restricted* (see sections 4.9.3 Restricted Services and 4.9.4 Custom SSTs) the system service executes normally and returns to the SVC Handler, which moves the system service return value from r0 to its correct position in the exception frame. This second portion of the SVC Handler is called its *tail*. If smx_sched has not been set by the system service and if no interrupt invoked an LSR (smx_lqctr == 0) during the system service, the SVC_Handler performs an *exception return* back to the utask. The U beside the line means *unstacking*. The net result is that register r0 contains the return value, r1 thru r3 and r12 are restored, and the utask continues running from after the point of call.

If smx_sched or smx_lqctr is non-zero, indicating that the utask may need to be preempted, the PendSV Handler is invoked. This is a little tricky. The SVC Handler first finishes running, then it *tail chains (TC)* to the PendSV Handler. As a result, no unstacking/stacking occurs. The PendSV Handler inherits the stacking done for the SVC Handler. The PendSV Handler then calls the scheduler, which determines if the utask should be preempted. If so, the utask is suspended or stopped, the *top task*[26] is resumed or started, and the scheduler returns to the PendSV Handler tail.

If the utask has not been preempted, the PendSV Handler performs an exception return to the utask. The U under the line means *unstacking* occurs. If the utask has been preempted, the PendSV Handler performs an exception return to the new task with unstacking. This operation is the same as described in the previous subsection with regard to resuming or starting the next task, which can be a utask or a ptask.

Eventually the suspended utask will be resumed, unstacked, and continue to run (assuming it was not stopped nor deleted by a preempting task).

A subtle, but important, difference in the above two diagrams is that for a call from a ptask, the branch to the PendSV Handler is made immediately after the system service finishes. Whereas for a call from a utask, this is not possible – the PendSV Handler cannot run until SVC Handler completes, because the SVC Handler has higher priority. Then the SVC Handler tail chains to the PendSV Handler.

This causes a problem for certain RTOS services. The task scheduler cannot run if the PendSV Handler does not run, and a task cannot be suspended or stopped if the scheduler does not run. Hence, smxu services cannot wait upon an RTOS object, such as a mutex. For example, the smxu_Heap shell functions cannot wait on the heap mutex. As a consequence, smxu_Heap shell functions make the heap call, assuming the heap mutex is free. If it is, the heap service proceeds normally and returns an appropriate value. However, if the mutex is not free, the utask is suspended on it for the timeout period. If a timeout occurs, 0 is returned and the heap service fails. If the mutex becomes free before the timeout, a special value, SMX_HEAP_RETRY, is returned, and the heap shell function calls the heap service again.

---

[25] Arm Thumb Procedure Call Standard

[26] Top task is defined as the longest-waiting, highest priority task.

Consequently the SVC Handler may need to run twice to perform one heap operation, which obviously hurts performance. This is a consequence of the Cortex-M architecture and there does not seem to be a solution to this problem. However for the vast majority of heap calls, the heap mutex will be free and the SVC Handler will be called only once.

6.2 Critical Sections

This section provides theoretical background concerning critical sections. For detailed guidance about what to do to protect critical sections in certain situations, see section 8.10.7 Critical Sections.

6.2.1 SecureSMX Object Priorities

Figure 6.3 illustrates the relative priorities of objects in a SecureSMX system:

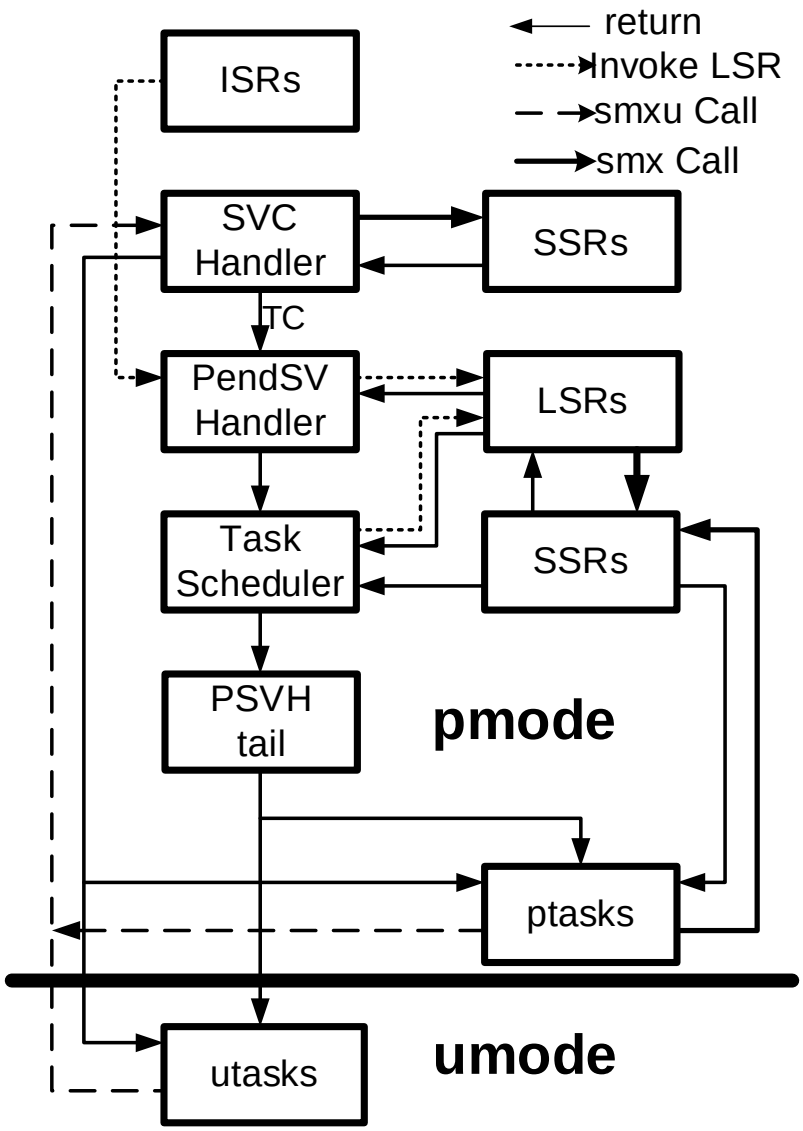

Figure 6.3 System Hierarchy

Figure 6.3 illustrates the operation of SecureSMX. This is complicated, but it is helpful to understand the basic concepts:

- Priorities decline from top down.
- An ISR can preempt lower priority ISRs and all objects below it unless interrupts are disabled or masked. SSRs never disable interrupts. Handlers and the Task Scheduler disable interrupts only in small critical sections. LSRs and tasks should also disable interrupts only in small critical sections. Minimizing interrupt disable or masking times is important because they directly affect *interrupt latency.*
- Except for tasks, an ISR returns to the point of interrupt even though it may have invoked an LSR. Invoking an LSR consists of putting its handle and a parameter into the LSR queue, *smx_lq*, and incrementing the LSR counter, *smx_lqctr*. LSRs are intended to do *deferred interrupt processing* in order to permit short ISRs and to reenable interrupts as quickly as possible (since ISRs normally disable interrupts).
- If an ISR has interrupted a task and smx_lqctr > 0, the ISR branches to the PendSV Handler (PSVH()), which runs all LSRs in smx_lq in the order they were enqueued (i.e. LSRs have no priorities, except that they are higher priority than any task. Due to having no priorities, LSRs preserve *temporal integrity*.)
- If an ISR has interrupted a task and smx_lqctr == 0, the ISR returns to the point of interrupt in the task – no more work to do.
- LSRs can call System Service Routines (SSRs) but cannot wait on them. For example, an LSR could call smx_SemSignal() or it could call smx_SemTest() with a 0 wait time. Thus, an SSR returns to the LSR, as shown in Figure 6.3.
- If an SSR has made a task switch likely by suspending or stopping the current task (ct) or by enqueueing a new task in the *ready queue, smx_rq,* that might have higher priority than ct and ct is not *locked*, the SSR sets *smx_sched*.
- When all LSRs have run and smx_sched > 0, PSVH() branches to the Task Scheduler (TS). If smx_sched == 0, TS is bypassed (not shown). TS determines if a higher priority task is ready to run, and if so the higher priority task becomes the current task, ct.
- While TS is running, it tests smx_lqctr prior to starting ct, and if greater than 0, executes a *flyback* to run all LSRs in smx_lq. Control then returns to TS and interrupts are disabled until ct resumes or starts. This is essential in order to insure that deferred interrupt processing is not overly delayed.
- When done, TS branches to the PSVH() tail, which dispatches ct with interrupts disabled until ct begins running. As soon as ct begins running, it can be interrupted to permit ISRs and LSRs to run.
- As shown in Figure 6.3, ptasks can make direct SSR calls. If smx_sched is not set by the SSR, the SSR returns directly to the ptask. Otherwise, it branches to TS, as shown.
- Both utasks and ptasks can make indirect SSR calls via the SVC Handler (SVCH()). If smx_sched is not set by the SSR and smx_lqctr is 0, SVCH() returns directly to the utask or ptask. Otherwise, SVCH() *tail chains (TC)* to PSVH().

6.2.2 Interrupt Disabling and Masking in Tasks

Tasks can be preempted by ISRs and LSRs, so it is necessary to protect critical sections shared between tasks and ISRs or LSRs. Interrupts can be disabled in ptasks with sb_INT_DISABLE() and reenabled with sb_INT_ENABLE(). This is the customary way to protect critical sections. Note that this also disables LSRs from preempting tasks because they must be invoked by ISRs[27].

In umode, the processor enable and disable instructions become NOPs. So, if interrupts are being disabled to protect a critical section in a ptask and that task is changed to a utask, the critical section is no longer protected.[28] This problem can be difficult to find. It is likely to occur when converting legacy pcode to ucode. To avoid the problem, use the alternate versions of sb_INT_DISABLE() and sb_INT_ENABLE() which stop the debugger if called in umode. These are enabled by SB_ARMM_DISABLE_TRAP in barmm.h.

The solution to the absence of interrupt disabling and enabling in umode is interrupt masking and unmasking, using the smxBase functions sb_IRQMask(irq_num) and sb_IRQUnmask(irq_num). Each task is limited to masking an unmasking only specific interrupts. Otherwise, a hacker could cause trouble by masking any interrupts he wished. This limitation is accomplished with *interrupt permission tables*.

Each task's TCB has a tcb.irq pointer, which points to an IRQ permission table consisting of an array of IRQ_PERM structures, defined as:

```
typedef struct {
  u8 irqmin;
  u8 irqmax;
} IRQ_PERM;
```

where irqmin and irqmax define a range of IRQ numbers. For example, the IRQ permission table for smxNS is:

```
const IRQ_PERM sb_irq_perm_ns[] = {
  {61, 62},         /* Ethernet */
  {37, 37},         /* USART1 for terminal output */
  {0xFF, 0xFF},  /* terminator */
};
```

Each row specifies a range of permitted IRQ numbers; the table is terminated with two 0xFF's. Then, set the task's irq pointer to point to the table, for example:

```
smx_TaskSet(task, SMX_ST_IRQ, &sb_irq_perm_ns);
```

Now sb_IRQMask(irq_num) and sb_IRQUnmask(irq_num) can be used by this task to mask and unmask its IRQs, in order to protect its own critical sections. Note that only ptasks and initialization code that runs in pmode can call smx_TaskSet() to do this. Child tasks inherit the parent's IRQ permissions.

---

[27] LSRs can also be invoked by tasks, but such LSRs will not preempt other tasks.

[28] It is good that interrupts cannot be disabled in umode, else hackers would have a field day!

A good place to put IRQ permission tables is at the end of irqtable.c in the BSP directory. To find the IRQ numbers of IRQs, for your processor, see the sb_irq_table[] in irqtable.c.

Interrupt masking is useful for regulating ISRs, LSRs, and tasks which are performing related functions, such as within the same partition. For example, it is necessary to prevent ISRs and LSRs from loading information into a buffer while a task is reading the buffer. If a ptask is expected to become a utask, it is preferrable to use interrupt masking rather than disabling.

6.2.3 Other Methods to Protect Critical Sections

Mutexes are a good way to protect critical sections, except they cannot protect against LSRs accessing the critical section. See section 8.10.7 Critical Sections for a detailed discussion of alternatives.

6.3 Cache Control

Cortex-M MPUs provide cache control. However, most Cortex-M MCUs do not have caches. Hence SecureSMX does not currently implement MPU cache control. But, if needed, it can easily be added to mpatmplt.h by defining a new attribute such as DATARW_NC (not cached) and used in, for example:

```
mp_RegionGetHeapT(taskA, sz, sn, DATARW_NC, name, 0);
```

This would get a non-cached block of sz bytes from the main heap and load its region into MPA[sn] of taskA. This block could be used for a DMA controller buffer. Since MCUs do not usually maintain cache coherency, DMA controller buffers cannot be cached.

6.4 Porting SecureSMX

At the current time, SecureSMX runs only on smx and supports only Cortex-M ARMM7 and ARMM8 processors using the IAR EWARM toolchain. If this does not match your requirements, you may be interested in the following.

6.4.1 To Another Toolchain

SecureSMX utilizes IAR Embedded Workbench for ARM v9.40, or later, from IAR Systems. Heavy use is made of the ILINK linker, and smxAware is integrated with the C-SPY debugger to provide a powerful debug environment.

smx has been ported to GNU C, and SecureSMX could be ported to GNU C provided that its linker has equivalent capabilities to the IAR linker. However, smxAware is not available for debuggers used with GNU C, so you would lose the ability to display MPAs and the MPU in clearly readable forms, as well as all of the other textual and graphical displays. Hopefully the GNU debuggers have similar features to IAR C-SPY for debugging MMFs and other exceptions. Supporting GNU C is probably feasible, although a lot would be given up.

We have no plans to support other toolchains, at this time.

6.4.2 To Another RTOS

The smx kernel is a feature-rich RTOS kernel with many unique features that have been utilized in SecureSMX. In addition, many modifications have been made to smx to support SecureSMX. As a consequence, porting SecureSMX to another RTOS is likely to be much more difficult than porting an application to SMX.

To support the latter we have created FRPort and TXPort porting layers for FreeRTOS and ThreadX. Each converts at least 90% of service calls likely to be used in applications to smx service calls. In most cases, the smx services are likely to provide better operation and better performance. Thus porting an application to smx is likely to improve its operation as well as opening the security features of SecureSMX to it.

6.4.3 To Another Processor

SecureSMX requires that a processor have the following features:

1. A Memory Protection Unit comparable to the ARM v7-M or ARM v8-M MPU.
2. Privileged and unprivileged processor operation modes.
3. An SVC-equivalent instruction to support a SWI API for system services.

We have not supported other processors due only to lack of manpower. If you are interested in using SecureSMX on another processor, please contact us.

6.5 Runtime Limiting

Unfortunately isolation, alone, is not sufficient to thwart hacker attacks. For example, a hacker could merely put a task into an infinite loop thereby denying processor access to all but higher-priority tasks. This alone would be sufficient to bring down most systems. Hence the RTOS must support runtime limits as well as isolated partitions. Runtime limits act like *time partitions*.

6.5.1 Guidelines

In operating systems like Windows or Linux, runtime limiting is implemented by giving each process a slice of processor time. This is simple to do and generally works well to enforce *fairness* between processes. Realtime systems are different. Rather than fairness, the main concern is for all tasks to meet their deadlines. Assigning runtime limits to tasks can be at odds with this. Further complicating the matter, is that task workloads may vary significantly over time. To deal with these problems, the following guidelines have been used for developing the runtime limit algorithm used by SecureSMX:

1. Mission-critical tasks should not have runtime limits because such limits could cause more harm than good. If a hacker reaches mission-critical code, it is game over, anyway. Therefore, all tasks are not required to have runtime limits. It is up to the developer to decide which tasks should have them.
2. To simplify development, child tasks share their parent’s runtime limits. This way, as child tasks are spawned it is not necessary to readjust the limits. Assuming that the typical partition has only one parent task, this amounts to assigning runtime limits to partitions, which makes sense, since they are usually well-defined

subsystems for which it is possible to determine reasonable runtime limits. Child tasks may vary considerably in their use of processor cycles from time to time and thus it is hard to assign individual runtime limits to them.

3. If child tasks spawn their own child tasks, the runtime limit of the *top parent* task is used by all child tasks below it. This preserves the idea of runtime limits being associated with partitions rather than with individual tasks within the partitions.
4. The runtime frame cannot be fixed. Introducing fixed limits into real-time systems, invariably causes complexities that must be programmed around. It generally works best to design in flexibility, and then let the system run freely. Such systems adapt better to changing conditions than do rigidly designed systems.
5. The idle task must be allowed to run some minimum number of times per runtime frame in order to perform its background functions. Hence the idle task's limit is actually the number of times it is allowed to run per frame. When the idle count reaches 0, the runtime frame ends and all runtime suspended tasks are resumed. The idle limit depends upon what the idle task is doing. For example, it might be desirable for it to do some minimum number of heap scan increments per runtime frame. Generally, when the idle task catches up on its work, it will decrement the rest of its count rapidly, so a relatively high count should not waste time.
6. A task is only suspended if it reaches its or its top parent's limit. If it is not running nor attempting to run, it is not suspended. This is simpler to implement and results in fewer tasks to resume at the end of the runtime frame.
7. When the runtime frame ends, all top-task and normal task runtime counters are restored to their limits. Child task counters are actually pointers to top task counters and thus are not changed.

Some goals of the above guidelines are to make assigning runtime limits easier for the developer and to provide the best possible visibility into the consequences of specific assignments. Even so, it may necessary to allow automatically changing runtime limits, if some partitions are not getting their work done on time.

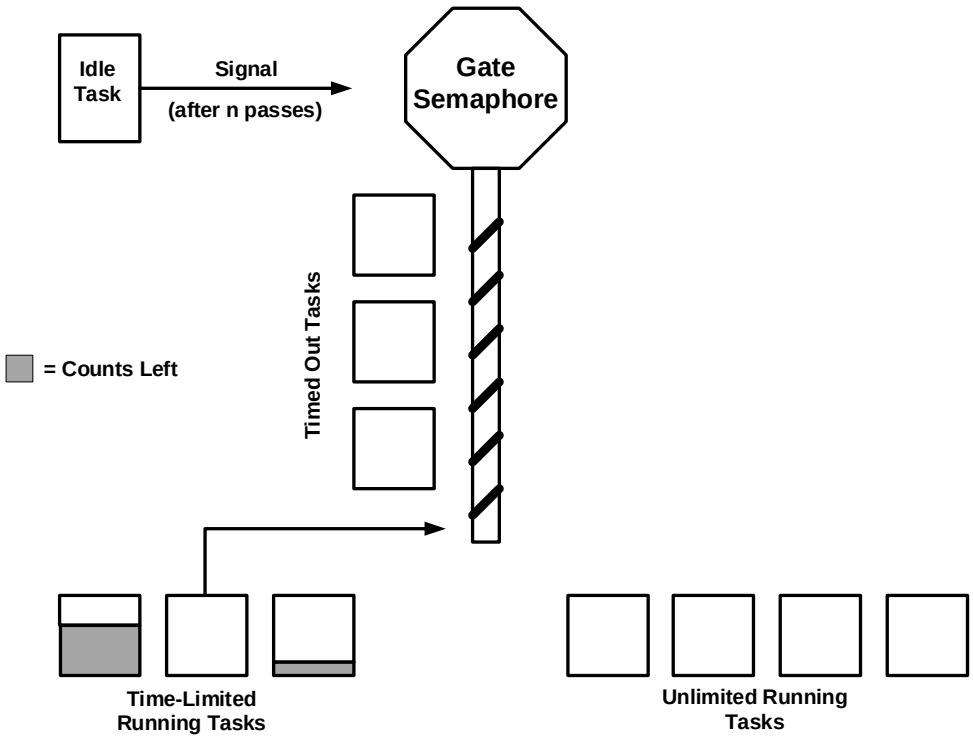

Figure 6.4 Runtime Limiting

6.5.2 Approach

Figure 6.4 illustrates the basic approach. It turns out that specifying runtime limits in ticks is not fine enough, so tick timer clocks are used, instead. This results in large numbers. For example, for an STM32F7 MCU with a 200 MHz clock rate 1 millisecond is 200,000 clocks. To make this more tractable, it is suggested to define a unit such as:

```
#define MSEC 200000
```

and to use this to specify limits, such as:

```
smx_TaskSet(t2a, SMX_ST_RTLIM, 10*MSEC);
```

This has the added advantage of making runtime limits independent of the processor being used. u32 fields *rtlim* and *rtlimctr* have been added to the task TCB. As indicated above, rtlim is set by means of smx_TaskSet(), except for child tasks. When a child task is created its rtlim and rtlimctr become pointers to the top parent's[29] rtlim and rtlimctr. smx_TaskSet(task, SMX_ST_RTLIM, lim) returns SMXE_OP_NOT_ALLOWED if task is a child task.

When initialization is complete, all non-child, runtime-limited (RTL) tasks have their rtlims set and their rtlimctrs cleared. Non RTL tasks have 0 rtlims and are ignored by the runtime limit code. During operation, each time an RTL task stops running, the number of clocks it has used is measured and added to its rtlimctr. The current task's rtctr is checked on each tick. If its rtlimctr >= its rtlim, the task is suspended on the smx_rtlsem *gate semaphore*, or, if it stops running

[29] See Section 4.1.8 Parent and Child Tasks for the definition of *top parent task*.

before the next tick, it will be suspended on smx_rtlimsem, if it attempts to run. Hence only RTL tasks which exceed their rtlim's or attempt to exceed them are suspended on smx_rtlimsem.

For a child task, the foregoing description applies to its top parent's rtlim and rtlimctr. Hence, any of a top parent's children can run out the parent's rtlimctr. If the child continues running it will not be suspended on smx_rtlimsem until the next tick. But if the child stops running and it, the parent, or one of the parent's children attempts to run, that task will be suspended on smx_rtlimsem. It may seem non-intuitive that a child can run out its parent's rtlimctr and yet the parent is not suspended on smx_rtlimsem. This is done to save time, since it is not necessary to suspend the parent, unless it attempts to run.

Note that an RTL task may exceed its rtlim by (MSEC - 1), or 199,999 in the above case. If objectionable, the overrun could be reduced by increasing the tick rate, but that might be undesirable. Another alternative would be to use another timer that generates faster interrupts to catch errant RTL tasks sooner.

6.5.3 Enabling Runtime Limiting

In xcfg.h:

```
#define SMX_CFG_SSMX     1
#define SMX_CFG_RTLIM    1
#define SMX_IDLE_RTLIM   2  /* number of idle passes per runtime limit frame */
```

Examples:

1. Typical code to create and start a ptask with a runtime limit of 10 milliseconds:

   ```
   t2a = smx_TaskCreate(ttM01_t2a, TP2, TS_SSZ, 0, "t2a");
   mp_MPACreate(t2a, &mpa_tmplt_t2a);
   smx_TaskSet(t2a, SMX_ST_RTLIM, 10*MSEC);
   smx_TaskStart(t2a);
   ```

2. Typical code to create and start a child ptask *t2ac1* of parent ptask *t2a*:

   ```
   t2ac1 = smx_TaskCreate(ttM01_t2ac1, TP2, TS_SSZ, SMX_FL_CHILD);
   mp_MPACreate(t2ac1);
   smx_TaskStart(t2ac1);
   ```

   Note that no template is specified in mp_MPACreate() because a child task uses its parent's template. Also no rtlim is set for the child task because it uses it top parent's rtlim and rtlimctr.

3. Typical code to create and start a utask with a runtime limit of 20 milliseconds:

   ```
   ut2a = smx_TaskCreate(ttM01_ut2a, TP2, TS_SSZ, SMX_FL_UMODE, "ut2a");
   mp_MPACreate(ut2a, &mpa_tmplt_ut2ax);
   smx_TaskSet(ut2a, SMX_ST_RTLIM, 20*MSEC);
   smx_TaskStart(ut2a);
   ```

   This code must run in pmode.

4. Typical code to create and start a child utask ut2ac1 of parent utask ut2a:

```
ut2ac1 = smxu_TaskCreate(ttM01_ut2ac1, TP2, TS_SSZ);
mpu_MPACreate(ut2ac1, NULL, 0xc3, 6);     /* 0xc3 = b1100 0011 */
smx_TaskStart(ut2ac1);
```

   This code must run in umode, and it can create only a child task, which shares its parent's rtlim and rtlimctr. In this case slots 0, 1, 6, and 7 of mpa_tmplt_ut2ax are loaded into MPA[2-5], 0 is loaded into MPA[6], and the stack region is loaded into MPA[7]. There are only 6 active slots in the MPU. MPU[0,1] are static slots.

6.5.4 Adaptive Time slicing

Most RTOSs support time slicing among tasks at a single priority level. smx previously supported time slicing only at priority level 0. It turns out that the above implementation of runtime limiting permits a more general form of time slicing, which we call *adaptive time slicing*. Specifying runtime limits for tasks, in effect, time slices them if they do not voluntarily give up the processor. As previously noted, the accuracy of the time slice is +/- 1 tick. Note that the time sliced tasks need not all be at the same priority level and that when they are done, lower priority level tasks are automatically allowed to run. Also, time sliced tasks may voluntarily give up the processor and other tasks will run, instead. In addition, other tasks can still be runtime limited, if desired.

6.6 Tokens

6.6.1 General

During World War II, families were given red tokens for meat, blue tokens for fish, and silver tokens for the trolley. *Tokens* governed how much of each resource they could use. Similarly, we must govern how much of each resource a partition can use, lest a hacker gain control and use up resources needed by other partitions. As with runtime limiting, it is necessary to find a simple approach that is practical for users to implement.

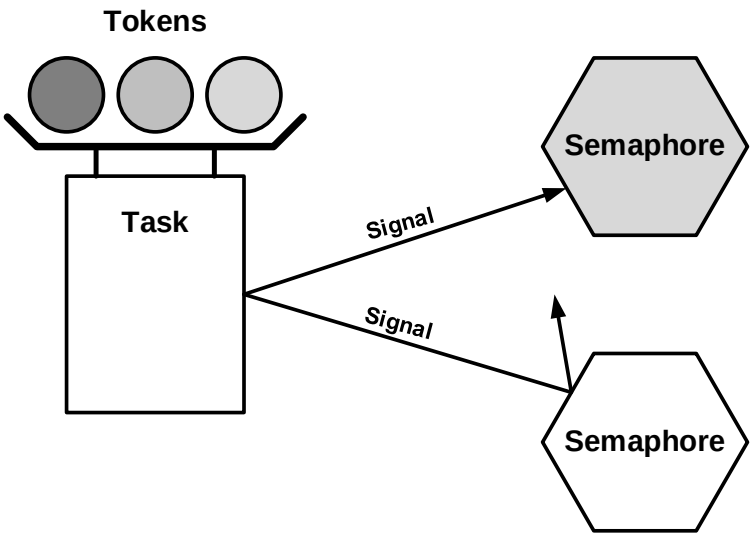

Figure 6.5 Tokens

Figure 6.5 illustrates the basic token approach taken by SecureSMX. This approach does not require that all tasks use tokens. Tokens are enabled by setting SMX_CFG_TOKENS in xcfg.h The TCB of a task has a field that contains a token array pointer (tap), which points to the array of tokens that the task is allowed to use. A child task shares the token array of its top parent. As with runtime limits this is done to make token assignments unaffected by child task spawning. It also means that token assignments tend to be for a partition, not for an individual task. When a task is first created, its tap is NULL. This means no tokens are required for it to access objects. This is appropriate for mission-critical tasks and system tasks such as those for initialization and recovery. Similarly to runtime limits, we do not want a disaster caused due to lack of a token! For token-limited tasks, the user must create a token array and load its address into the TCB tap field.

So, how are tokens defined and what do they control? In many RTOSs, the control blocks for tasks, semaphores, etc. are statically defined. In smx, only object *handles*, which are pointers to control blocks, are statically defined. Control blocks are taken from pools and initialized as needed. A token array consists of the addresses of the handles of the objects that a task is allowed to access, as follows:

```
u32 ut2a_ta[] = {(u32)&sema+1, (u32)&semc, 0};
u32 ut2b_ta[] = {(u32)&sema, 0};
```

These are token arrays for two tasks, ut2a and ut2b. The +1 denotes a high-privilege token, which permits task ut2a to create, delete, and perform other privileged operations on sema. ut2b has only a low-privilege token for sema and thus it can perform only operations such as signaling or testing sema. A 0 marks the end of each array. In practice, token arrays will usually be larger than these examples. However, they are easy to create, since it is well-known what kernel objects a task must access and control at the time the task code is written. Since token arrays must be searched from beginning to end, those tokens needed most often should be placed first. However, since mission-critical and highly-trusted tasks are not required to have tokens, system performance impact from token array searching should be minimal. Token arrays are stored with control blocks in pmode and thus are not accessible from umode.

Should a task attempt access to an object for which it has no token or an insufficient privilege token, the operation is aborted and a SMXE_TOKEN_VIOL error with sev = 1 is reported, causing smx_EMHook() to stop the task.

6.6.2 Blocking Excessive Creates

Tokens, alone, cannot prevent a hacker from repetitively creating or getting an object until he uses up all control blocks for that object, thus denying them for other partitions. To prevent this, an object cannot be created or gotten again until it has been deleted or released. Thus a hacker cannot use up control blocks for an object type — he can create or get only one object, for which he has a high token. In addition, he can access other objects only if he has tokens for them.

Unfortunately, messages are an exception to this. It is normal practice to get or receive a message, fill it, then send it to an exchange. When the message has been sent, its handle is loaded with NULL. This allows another message to be obtained. Normally this is desirable since the server task to which messages are being sent may be busy, so extra messages are needed to avoid loss of data. However, by getting messages repetitively, a hacker could use up all MCBs in the system.

The best way to block this hack is by pre-allocating a set number of messages to a resource exchange, and then limit the task to obtaining messages only from the resource exchange. Thus, the partition that the task is in would not be permitted to get or make messages. It could only receive them from the resource exchange. This limitation can be done for utasks by omitting the message get calls from the smx_sst[] jump table. It also can be done for ptasks, if they are required to use the SVC exception for kernel calls.

6.6.3 Handle Verification

As a standard feature, smx verifies that all handle parameters are valid handles. Each handle is range-checked vs. its object control block pool and the cbtype field of the object is checked to make sure the handle is aligned on an object boundary. If handle verification fails, the service is aborted and an SMXE_INV_xCB error is supported, where x is the object type (e.g. “S” for a semaphore). For messages, the owner field is also checked. If the current task is not the message owner, the service is aborted and the SMXE_NOT_MSG_ONR error is reported. These are not considered to be irrecoverable errors, so severity = 0, and smx_EMHook() allows the task to continue. Handle verification is performed ahead of token verification in order to avoid MMFs due to out-of-range handles.

6.7 Safe LSRs

6.7.1 The ISR Problem

A fundamental security flaw in the Cortex-M architecture and most other processor architectures is that ISRs must run in *handler mode*. This pretty much opens up everything to a hacker coming in via an interrupt. TrustZone might be employed to avoid this problem, at great complexity, but it seems that Arm Ltd. could have come up with a simpler solution.

The flaw is further compounded by most RTOSs, which allow either kernel services or reduced kernel services to be called from ISRs, thus greatly increasing the attack surface for the ISRs. This also violates the generally accepted good programming practice of keeping ISRs short. Doing so minimizes their attack surfaces and permits faster interrupt response times. By contrast, smx permits only one small service call from an ISR and that is to invoke a *link service routine* (*LSR*). This permits performing only the absolute minimum work in an ISR and deferring other processing to an LSR. LSRs run like mini tasks that have priority over all tasks, are immune to priority inversion, and can call any smx service that does not wait.

So far so good, except that LSRs, themselves, can be hacked and they also run in hmode. To deal with this SecureSMX introduces LSRs that run in umode, called *uLSRs*.

6.7.2 LSR Types and Operation

Summary of the 3 types of LSRs:

- Trusted LSR: **tLSR** runs in handler mode and works like previous LSRs
- Safe LSR: **uLSR** runs in umode and **pLSR** runs in pmode, and both operate like mini tasks.

The previous smx LSR consisted only of a function pointer and a parameter. Both were placed in the LSR queue, smx_lq. smx_lqin pointed to where to put the next LSR, and smx_lqout pointed to where to get the next LSR. smx_lq was a cyclic queue. When all ISRs finished running, all LSRs would be run in the order invoked, then tasks would run.

In order to run an LSR in umode, it must have its own stack, its own MPA, and several other things. Hence, it requires an *LSR control block* (*LCB*), similarly to other smx objects. The LSR queue operation is the same as before, except that LSR handles, instead of function pointers are loaded into smx_lq. A parameter is still loaded after each handle. The function pointer is the first field in the LCB. Hence tLSRs that run in hmode, such as smx_KeepTimeLSR, get the function pointer from the LCB and run with it. These LSRs are low-overhead *trusted LSRs* (*tLSRs*) that are used for smx operations and can also be used for applications, when low overhead is necessary.

When an LSR is to be run, if flags.sys in its LCB is 1, then it is run as a tLSR and most of the LCB fields are not used. If flags.sys is 0, then flags.umode is tested. If flags.umode is 1, the LSR is run as a umode LSR, uLSR. If it is 0, the LSR is run as a pmode LSR, pLSR. This is similar to how tasks are run. When either of these types of LSR is started, its MPA is loaded into the MPU, its stack pointer is loaded into the PSP register, an exception frame is built in the LSR stack, and an exception return is made to the LSR function, with the parameter passed in r0.

Normally, an LSR will run in the partition that it serves. As such, it can use regions from the partition's MPA template and thus is able to directly access variables and functions in the partition. In fact, the LSR is actually running in the partition. At the same time, the LSR will probably have IO regions that tasks in the partition do not have. The LSR can be directly invoked by any task in the partition in order to start an IO operation, after which it may be repeatedly invoked by an ISR until the operation is complete (e.g. sending a multi-block message).

A uLSR or pLSR is very much like a mini task. Its control block is only 9 words, compared to about 32 words for a task. LSRs typically require only very small stacks of 25 words, or less. The smallest possible stack is the size of an exception frame, 8 words. One of the fields in the LCB is a *host task handle, htask*, which normally points to the top parent task in the partition. The LSR shares the token array, runtime limit, and runtime counter of htask. The LSR can only access smx objects permitted by htask tokens, and if it runs down the htask runtime counter, it will be stopped until the runtime frame ends. Hence, it is protected the same as a task.

Like a one-shot task, when an LSR runs through its final brace, it triggers auto stop. This, in turn, triggers a PendSV exception in order to return to hmode and to the PendSV Handler. The latter will run all LSRs in lq, then continue the current task or resume or start a preempting task. If the current task is continued, its MPA is reloaded into the MPU and it stack pointer is reloaded into the PSP register. If a preempting task is to be run, the scheduler loads its MPA into the MPU and its stack pointer into the PSP register.

See the smx Reference Manual for LSR create and delete function descriptions.

pmode LSRs serve primarily to allow partitions to be developed in pmode, then moved into umode. However, like ptasks, they do improve security slightly and reliability significantly.

6.7.3 Performance

The overhead for the LSR types, measured on a typical MCU, in clock cycles are:

- tLSR 338
- pLSR 800
- uLSR 1086

As would be expected, the tLSR has much less overhead. For a very simple LSR that just starts or resumes a task, it may be preferrable to use a tLSR. But, for more complex LSRs that do processing and make smx calls, the pLSR and uLSR overheads should not be a problem – they are much less than starting or resuming a task via a tLSR. Thus they present a good alternative for deferred interrupt processing. On the other hand, if performance is not an issue, it may be simpler to use a tLSR to restart or resume a task to do deferred interrupt processing.

6.7.4 Resulting Security

As noted above, the best that can be done is to minimize the amount of code in ISRs. Doing so permits carefully writing the code to make it very hard, hopefully impossible, to hack. Moving as much interrupt processing code as possible into uLSRs provides the best-possible security. uLSRs are subject to the same limitations as utasks and thus can be fully isolated. At the same time, invoking uLSRs is much faster than starting or resuming utasks. Hence they represent a good compromise between security and performance.

6.8 Task Privilege Levels

6.8.1 Description

Tasks must have privilege levels in order to enforce the principle of least privilege, for security. For example, a file server must be able to limit file access depending upon the privilege level of the client making a request. This is accomplished by transferring the client's privilege level to the pmsg being sent to the server's port, then transferring the pmsg's privilege level to the server when it accepts the pmsg. Server code can then test the privilege level of the current task with:

```
priv = smx_TaskPeek(smx_ct, SMX_PK_PRIV);
```

When a task is created, its privilege level is set to 0. This can be changed as follows:

```
smx_TaskSet(smx_ct, SMX_ST_PRIV, priv, fix);
```

where priv is the new privilege level, and if fix = 1, the privilege level is fixed. Fixed privilege levels are intended for non-server tasks. When a task is created, its privilege level is not fixed. Privilege levels can be set from 0 to 255. The meaning of a level is determined by the application. utasks cannot change their own or other tasks' privilege levels, except via pmsgs, as discussed above.

Chapter 7 Partition Demos

This chapter presents a series of demos that demonstrate how to create an isolated partition in pmode and then move it to umode. It is intended to provide a quick introduction to SecureSMX and how to use it.

Note: This chapter and demos were created before the decision was made in v5.40 to make SecureSMX open source (see 1.8 SecureSMX Licensing). We have retained this and the demo on our website (link below), as a useful tutorial despite being an older version.

7.1 Getting Started

The partition demos are in a single file at www.smxrtos.com/securesmx/demo. After you have downloaded and unzipped this file, you will observe five complete demos labeled pd0 thru pd4. Each of these can be made and run using the IAR EWARM tool suite. If you do not have EWARM, you can download a free evaluation copy from www.iar.com. The demos run on the STM32F746G-Discovery board, which is very low cost and widely available from online distributors.

The sections that follow contain instructions for stepping through each demo and observing how it works. This is the best way to learn. Each demo is derived from its preceding demo and then new code is added and changes are made. For example pd1 is derived from pd0. Hence, if you prefer, you can use a comparison tool such as Beyond Compare to see what is changed from one demo to the next.

smxAware provides insight into the demos. The smxAware files follow the demos in ssdemos. To install smxAware see the installation section of its user's guide.

Note: SecureSMX v5.2.0 and IAR EWARM v8.50.5 were used when writing the following sections, so if a newer version is used, addresses and sizes are likely to be a little different. We tested with v9.40.2 and verified it works (and addresses and sizes differ a little). Also, it is possible we may have made fixes or adjustments to the demos since this was written causing these to differ.

7.2 Creating an Isolated Umode Partition Demo

pd0 thru pd4 illustrate how to take a typical embedded system, identify a vulnerable partition, then move the partition from pmode to umode, where it is fully isolated.

pd0 is intended to represent a typical, unprotected, embedded system running in hmode and pmode. It contains three tasks: idle, mctask, and ffdemo. The mctask is intended to represent a mission-critical task, which must be changed very little. The ffdemo task uses FatFs, which is third party code and thus may be considered to be vulnerable. Our goal is to move ffdemo and

FatFs into an isolated umode partition from which mctask is protected. This is done in a sequence of steps represented by pd1, pd2, etc.

Running the Demos

The demos write a simple file to SD card repeatedly. Use a card that is blank or has nothing important on it, and ensure it is inserted before starting the demo. In the IAR debugger, add passcnt and failcnt to the Live Watch window, and you should see passcnt increasing as it runs. If a terminal emulator is connected to the eval board, the top line tells what demo is running and the bottom line shows % Idle, % Work, % Overhead, and Seconds running. Note that % Work is quite high and overhead is very low. This line is presented so that you can see that the demo is running.

7.2.0 pd0

We recommend that you trace through pd0 just to see what is there and how it works. Starting at main(), certain startup code has already run. (A breakpoint can be put at __low_level_init() in startup.c and restart, if you wish to trace it to main()). When tracing main() and following code, note that SMX_CFG_SSMX is off as are configuration constants dependent upon it (see xcfg.h). main() initializes a few things, then calls smx_Go(), which initializes smx. smx_Go() initializes the error manager, event buffer, LSR[30] queue, ready queue, timer queue, timeout array, several system LSRs, the idle task and a few other things. Then idle is started at PRI_MAX with ainit() as its main function. The system LSRs perform time functions, profiling, timeouts, and task self-delete.

ainit() does application initialization, including tick enable, (portals are not enabled because SMX_CFG_SSMX is 0), profiling is enabled as is event monitoring. At this point the mctask and ffdemo tasks have been created and started. However, their priorities are less than PRI_MAX, so they do not run. Put breakpoints at ffdemo_main(), mctask_main(), and smx_IdleMain(). Then ainit restarts idle at PRI_MIN with smx_IdleMain() as its code, and the other tasks run.

All the tasks run in while loops. Since mctask has PRI_HI, it runs first. It just loops for a msec, then suspends itself for 2 ticks. This allows ffdemo to run. It performs file operations, then suspends itself for 25 ticks so idle can run. Idle performs a stack scan, a profile display, a heap manager function, and bumps another task at PRI_MIN, if any is there. Since SMX_CFG_RTLIM is 0, runtime limiting is not performed. Power down is inhibited. Idle runs when the other tasks do not run.

The smxAware Event Timeline window provides a picture of the above tasks running. It can be very helpful during debugging to see exactly in what order tasks, LSRs, and ISRs are running. Let the system run for a few seconds, click the debugger pause button, then click on smxAware in the task bar. Click on Graph in the pulldown window to see the timeline window. Other smxAware windows present a great deal of useful information that may interest you. Clicking on Event shows the entire Event Buffer. Clicking on Memory Map shows how read/write memory is structured.

---

[30] An LSR is an smx object that is used to perform deferred interrupt processing. It runs after all ISRs complete and before any tasks resume. Hence it is immune to problems such as task priority inversion.

7.2.1 pd1

At this point we have three tasks: *idle*, *ffdemo*, and *mcon*. The first step is to turn on SMX_CFG_SSMX[31]. This enables the MPU. When it is on, smx_TaskCreate() assigns the default memory protection array (MPA), *mpa_dflt*, to the task it creates (i.e. task->mpap = mpa_dflt). This is used for ffdemo and mcon. (idle is discussed later). So the next step is to define mpa_dflt.

Defining Region Blocks

From the map file (see "Unused Ranges, From" in it), we see that the memory sizes are as follows when using the linker command file from pd0 (sizes may vary slightly):

```
ROM     0x136eb
SRAM    0x6cac
DRAM    0x8000
```

(Do not try to run it yet until we change to pd1.icf below. Using pd0.icf was just to see these sizes in the map.)

The first step is to define *region blocks* for these. For ARMM7, a region block must have a size that is a power of two and it must be aligned on its size. The first step is to determine the size. Then we use subregion disables (by n/8 multiples) to make the region blocks fit as closely as possible to the required sizes. The results are:

```
ROM     0x136eb  <= 0x20000*5/8 = 0x14000
SRAM    0x6cac   <= 0x8000*7/8 = 0x7000
DRAM    0x8000   <= 0x8000
```

These values were used to create region blocks in the linker command file at pd1\APPM\IAR.AM\STM32\stm3264g_pd1.icf. **Now change the linker to use pd1.icf**. Right click on pd1 top node, Options, Linker, and change:

```
$PROJ_DIR$\stm32746g_pd0.icf   to

$PROJ_DIR$\stm32746g_pd1.icf
```

Looking at pd1.icf, note the MPU region sizes. These must be powers of two, which is easy to do in hex. The rule for this is: There can be only one non-zero digit and it must be 1, 2, 4, or 8. Below this are the region block definitions. Note the size and alignment taken from the above calculations. Ignore (MPUPACKER) – it is a marker for our MpuPacker utility, discussed in section 8.11.1 Using MpuPacker. The rest of pd1.icf is the same as pd0.icf.

As development proceeds, region blocks will grow in size. The linker will inform you if the allocated size is exceeded. Then it is a simple matter to increase the allocated size by 1/8 or go to the next power of two and 5/8. This permits keeping region block sizes tight during development.

[31] Be sure the configuration constants dependent upon SMX_CFG_SSMX are off – see xcfg.h. We are not ready for them, yet.

Default MPA

mpa_dflt is defined in pd1\BSP\ARM\STM32\mpa7.c. At the top, the region block names and sizes have been brought over from pd1.icf. Below this, mpa_dflt is defined. The macros making this possible, such as RGN and RA are defined in pd1\SSMX\ARMM\mpatmplt.h. Each line in mpa_dflt defines RBAR, RASR, and a name for one region. The name is used only during debugging – it is very helpful, in smxAware, for example.

The first three regions of mpa_dflt are the memory regions defined in pd1.icf. The next three are IO regions. ffdemo requires all three of these. mcon requires just USART1. These IO memory mapped sections are 1K in size and 1K aligned, so there is no problem making them into ARMM7 regions. However, the memory-mapped registers in USART1 and SDMMC1 are contained within the first 64 bytes and the registers in DMA2 within the first 128 bytes. So from Liu Table 11.7 we see 64 bytes corresponds to 5 and 128 to 7, both as used in mpa_dftl. It is not essential to tighten down regions, like this, but it does improve reliability vs. bugs and soft errors.

Idle Task

For idle, an MPA template, mpa_tmplt_init, has been defined in mpa7.c. It is similar to mpa_dflt, except that it has a large IO region. This is because idle first runs with ainit() as its main function and ainit() does many different IO accesses. In smx_Go(), following creation of idle, mp_MPACreate() is called to create a custom MPA for idle, using mpa_tmplt_init. After this, smx_Idle->mpap -> MPA for idle and smx_Idle->tp = mpa_tmplt_init.

MMFs

When the MPU is enabled, a task's MPA is loaded into the MPU whenever the task is dispatched. Hence the task is limited to the memory regions and their attributes in its MPA. For pd1 the regions are much smaller than the implemented memory sizes:

ROM	0x14000	vs. 0x100000
SRAM	0x7000	vs. 0x50000
DRAM	0x8000	vs. 0x2000000

This is useful because an access outside of used memory into implemented memory will not trigger a Bus Fault, but it will trigger an MMF. To see this, start pd1. You will get an MMF, which causes an immediate halt. To find the cause of the MMF, open the Call Stack window and click on the top entry. You will see that there is some code attempting to access location 0x20008100. Selecting MPU in the smxAware Text window you can see that this address is not in any MPU region, hence the MMF. Comment out the two lines of assembly code, and pd1 will run ok.

An MMF causes a system halt only when debugging. For a released system, it causes a branch to the smx_EM() with an SMXE_MMF_VIOL error and severity = 1. It calls smx_EMHook(), which stops the task causing the error.

Task Stacks

Slot 7 is reserved for the task stack region. The main benefit of having a separate stack region is that stack overflow is caught immediately, causing an MMF. This protects whatever is "above" the stack such as a heap control block, another stack, or a global variable.

During debug it may be desirable for the system to continue running despite a stack overflow. This can be accomplished by adding a *pad* above the stack. smx will report a stack overflow when it scans the stack or when the current task stops running, but no MMF will occur unless the pad is exceeded. Below the stack is the *register save area*, and below that is optional *task local storage*. The latter may be helpful if you run out of MPU slots. See section 4.11.8 Task Local Storage.

smx supports two types of tasks: *normal* and *one-shot*. A normal task has a permanent stack, which may be pre-allocated or allocated from a heap by smx_TaskCreate(). In the first case, the stack block must be a region block. In the second case, eheap is able to find and allocate a region block from a heap. Either way smx_TaskCreate() creates the stack region and stores it in the task's TCB. Then mp_MPACreate(), which is called next (see smx_Idle in smx_Go), moves the region into MPA[7].

For a one-shot task the stack block is taken from the stack pool when the task is dispatched by the scheduler. The stack block must be a region block. The scheduler creates the stack region and loads it into MPA[7]. Then its MPA is loaded into the MPU, and the task is started. A one-shot task does not have an internal infinite loop like a normal task. When a one-shot task stops it releases its stack. Yet the one-shot task can be waiting at any smx object (e.g. semaphore, mutex, etc.) just like a normal task. As a consequence, many one shot tasks can share a single stack as long as they do not need to run concurrently. Since partitioning tends to increase the number of tasks in a system, one-shot tasks can help to limit memory growth.

Summary

At this point, all tasks are running under the MPU, and no application code changes have been made. Although not much has been done so far, there is already some benefit, namely: a latent bug or two might have been found and soft error protection has been improved.

7.2.2 pd2

In this step we put FatFs into a *pmode partition*. Since a partition must have at least one task, we will add ffdemo to the partition, for now.

Define Sections

The first step is to define regions for this new partition, which we will call *fs*. Regions are composed of *sections*. The C compiler puts everything into the well-known sections: *.text*, *.bss*, *.data*, *.rodata*, and *.noinit*. There are two ways to create our own sections (which have pros and cons):

1. Compiler section switches.
2. Section pragmas.

Compiler section switches can be put into .xcc files. \CFG shows three .xcc files. (In the Open dialog box, change the filter to All Files (*.*) to see them, since .xcc is not a standard extension type.) These simply rename the sections created by the C compiler, such as:

```
--section .text=.sys.text
```

To apply these files to a group (folder), such as RTOS: in the project window, right click RTOS, Options, C/C++ Compiler, check “Override inherited settings”, Extra Options, check “Use command line options”, and enter:

```
-f $PROJ_DIR$\..\..\..\CFG\mpi_sys.xcc
```

which applies mpi_sys.xcc to all files under RTOS. The simplicity of the project window belies the complexity of the underlying project file. Unfortunately, when “Override inherited settings” is checked in a subgroup, all of the settings of groups that include the subgroup are copied into it. Then any changes made to a group must also be made to every overridden file or group under it, which is easy to forget. For this reason, the .xcc option should be used sparingly. Here it is used for RTOS and System, where it saves adding pragmas to a large number of modules, and for some STMicro HAL files, where it helps reduce the number of pragmas added to third party code, which is inconvenient when the code is revised by the third party. Note that overridden files are indicated by a checkmark in the gear column of the Workspace (project) window.

For other modules, it is preferrable to use section pragmas in the module. This avoids the above problem and it is necessary when not all functions or variables belong in the same regions. For example, in ffdemo.c, it is preferrable for ffdemo_init() and ffdemo_exit() to go into sys_code, since they are called during initialization and exit, respectively. Hence they do not belong in the fs partition. Another example is in sd_diskio_dma_rtos_bspv1.c, where the transfer completed callbacks and the trusted LSRs belong in sys_code. To do this,

```
#pragma default_function_attributes =
#pragma default_variable_attributes =
```

are placed ahead of them to end .fs.text and .fs.data from the start of the module.

However, section pragmas do not work for string literals, which is discussed in Eliminating MMFs below and section 4.5.4 String Literals.

Upgrade pd2.icf

The next step is to add new region blocks to the linker command file. Four new MPU region sizes have been added: fscsz, fsdsz, scsz, and sdsz.

Below them is a new block, *clib_code*. This is an *ordinary block* with alignment of 4. It is necessary in order to bring clib functions into sys_code. (clib_code is included in sys_code below.) Adding unknown code, such as FatFs, is likely to bring clib functions with it. The Module Summary in the map file is helpful to find the new modules holding these functions. They are likely to appear in one of the lower groups. Some clib functions require variables, so *clib_data* is defined for these and it is included in sys_data.

Next come the new region blocks, *fs_code*, *fs_data*, *sys_code*, and *sys_data*. The first two are for the fs partition. Notice that each code block includes .xx.text and .xx.rodata sections, and each data block includes .xx.bss, .xx.data, and .xx.noinit sections. Although it’s not necessary to specify ones that are not used, we strongly recommend always listing all of them to avoid wasting time debugging MMFs when the code changes and they become necessary later.

Next are sys_code and sys_data. These are included in all ptask templates. Their purpose is to allow direct access to system services and other services. For sys_code, note that .intvec is included first. This is necessary to enable the CPU to access the first two vectors in the vector

table on startup – see \BSP\ARM\STM32\STM32F7xx\vectors.c. Next are two .sys code sections, then the clib_code block. For sys_data, CSTACK (the main stack) is included first so that overflowing it will trigger an MMF. Next are three .sys data sections and the clib_data block.

Now rom_block and ram_block are completely different than before: they include the region blocks defined above. ro adds all code not in the code region blocks, and rw adds all data not in the data region blocks. The sys blocks have been placed ahead of the fs blocks to minimize the gap between them. In the map, sys_code ends at about line 900. The *Block tail* is wasted space inside of the sys_code region block. It is 0x36ce (14,030) bytes. scsz = 0x20000 so a subregion (1/8) is 0x4000 bytes, which is larger than the tail. 0x8010392 + 0x36ce = 0x8014000, which is the starting address of fs_code, so it is not possible to use the space wasted between sys_code and fs_code region blocks, by disabling a subregion.

Memory Overflow

Memory sizes have grown as follows, due to organizing code and data into region blocks that have wastage at their ends and gaps between them due to the processor's alignment requirements:

ROM	0x136eb	to	0x18000	24%
SRAM	0x6cac	to	0xc000	143%
DRAM	0x8000	to	0x8000	0%

These are much larger increases than we will see in the end. There are many methods to improve memory efficiency, however it is too early to apply them now. If you are experiencing memory overflows at this stage, the best plan is to get a processor/board with more memory. If this is not feasible, the next best plan is leave out portions of code, as you work.

fs MPA template

mpa_tmplt_fs is shown in BSP\ARM\STM32\mpaf7.c. Note that it has been necessary to combine the USART1 and SDMMC1 IO regions in region 4, since there is no spare region. The first is located at 0x40011000 and the second is at 0x40012c000. The range to be covered is 0x2c00 - 0x1400 = 0x1800. The next larger power of 2 is 0x2000. The region must start at 0x40010000 and it will cover to 0x40012000, which is not enough, so region size 0x4000 must be chosen. It must start at 0x40010000 and will cover to 0x40014000, which is sufficient. This covers from TIM1 to EXTI, which is twelve IO regions! In region 4, there are 6 region disables (N0, N1, N2, N3, N4, and N67) leaving windows at 0x1000 to 0x1800 and at 0x2800 to 0x3000. The first admits USART1 & 6; the second admits SDMMC1. This is acceptable, if UART6 is not used.

Also note that the SDMMC1 and DMA2 regions were removed from the default template, mpa_dflt, since they are now in the fs template.

fs MPA

In ffdemo_init(), ffdemo create is followed by:

```
mp_MPACreate(ffdemo, &mpa_tmplt_fs, 0xFF, 8);
```

This gets a block for the MPA from the main heap that is large enough for 8 slots, transfers the first 7 slots from mpa_tmplt_fs, and loads the stack region from the ffdemo TCB into slot 7. In

AppDbg.map search for fs_code in the Placement Summary to see its size. Then scan down to the end of the section where you find <Block tail>. This shows how much spare space is left. Do the same for fs_data. The sizes for the fs regions are (decimal):

```
fs_code   0x3000 (12288)     spare  0x4d4 (1236)     10%
fs_data   0x2800 (10240)     spare  0x294 (660)       6%
```

sys_code and sys_data allow the fs partition to directly access system services and data. These are temporary and will be replaced when the fs partition is moved to umode. The IO regions and the stack region have been previously discussed. The Event Buffer, EVB, requires a separate region since it is in DRAM. If it were moved into SRAM, it could be combined with sys_data, freeing up an MPU slot so USART1 and SCMMC1 could be separated.

Eliminating MMFs

Despite having tightened down the ffdemo task regions, pd2 runs smoothly. This is because we have already fixed all of the MMFs that normally occur when regions are tightened. If you were working with your own project, you would need to do this yourself. So, here is how to do it:

When an MMF occurs, open the Call Stack window and click on the top function (ignore <Exception frame>). This shows where the MMF occurred. Put a breakpoint there and run to it from the start. It generally works better to trace for an MMF in the disassembly window – tracing in the C source code window can be misleading. If you have left a variable out of your regions, you will generally find code like this:

```
ldr     rx, =variable
ldr     ry, [rx]
```

The first instruction will execute, but the second will refuse to execute. This is the sign of an MMF. Compare the address in rx to the MPU regions in the smxAware Text window of smxAware. You should find that it is not in any of them. The disassembly or C window will give you the variable name. Find it in your code and move it into one of your data regions. This is generally done with a section pragma, such as:

```
#pragma default_variable_attributes = @ ".fs.bss"
```

This is for a non-initialized variable in the fs_data region. For an initialized variable use ".fs.data".

If you have left out a function you generally find code like this:

```
bl      function
```

To the right of this is the address of the function. Comparing to the MPU regions in smxAware, you should find that it is not in any of them. Find the function in your code and move it into one of your code regions with:

```
#pragma default_function_attributes = @ ".fs.text"
```

This is for the fs_code region.

Handles, as parameters in system service calls, tend to cause difficulty. For example, a semaphore is created in hmode, then a utask attempts to signal it and gets an MMF. The problem is that the compiler attempts to pass the handle, not its address, as the parameter. This results in

an attempt to access an address outside of the MPU. To avoid this problem, it is necessary to create an *alias handle* in a region of the utask and copy the actual handle into it after creating the object in hmode. Then specify the alias handle as the parameter in the system service call, instead of the actual handle.

A big problem is *string literals* (e.g. “abc”). The compiler puts all literals into section .rodata no matter where they occur in the code. This can be perplexing – everything else works, except the string literals. Often the string literal is staring you in the face, but you fail to recognize it. The best way to get them into one of your sections, for example .fs.rodata, is to use a .xcc file, as discussed previously. For example, this has to be done for “SDQueue” in smx_PipeCreate(), in SD_initialize(). You would think that literal would be put into .fs.rodata, but it’s not! In this case, -f $PROJ_DIR$\..\..\..\CFG\mpi_fsd.xcc has been put into Extra Options for FatFs.

An alternative is to define an array for a string, such as in ffdemo.c:

```
#pragma default_variable_attributes = @ ".fs.rodata"
const char hdr[] = "This is STM32 working with FatFs";
```

Then in ffdemo_main():

```
strcpy((char*)wtext, hdr);
```

puts the string literal into the beginning of wtext.

For ARMM8, a region overlap causes an MMF when the overlapping area is accessed. This can be particularly puzzling during debug because you see that the object causing the MMF is in a region, so what’s the problem? The MPU window in smxAware, flags overlapping regions, so watch for this. Otherwise, you need to carefully compare MPU regions. Stacks and pmsgs are the primary cause of overlapping regions. If a stack comes from the main heap, do not create a separate stack region. This leaves MPU[7] available for another region. In this case, PSPLIM is used to detect stack overflows. For pmsgs, it is best to use an auxiliary slot in the current task’s MPA.

Sometimes, to find the cause of an MMF, it is necessary to trace in assembly. Tracing in C often gives misleading results. For example, in C, it may look like a function is out of range, whereas actually a parameter is the cause of the problem.

Summary

We now have FatFs and ffdemo running in an isolated partition with fairly tight regions. It is left as an exercise to the reader to do the same for mcon. However, since mcon is highly-trusted code, there is no reason to do this other than to improve reliability or possibly catch latent bugs. Idle is left as is because it must perform functions such as heap management and profiling, which require wide memory access. Also, in case of a system shutdown, aexit() runs under idle.

7.2.3 pd3

Before moving the fs partition to umode, we must get it to make system calls via the SVC Exception, because it will not be able to access system calls directly.

SVC Functions

SSMX\ARMM\svctmplt.c contains shell functions for all services considered safe for use from umode. It does not contain shell functions for services that could disrupt system operation, such as smx_SysPowerDown(). In some cases a service may be allowed from umode, but is limited in what it can do. For example, smx_TaskCreate() can be used to create a umode child task, but not a pmode task. svctmpl.c cannot be used in the project file because the jump table in it brings in all smx and other services whether they are used or not. So svc.c is derived from svctmplt.c to include only services actually used, in this case, by fs partition. As can be seen, it is also considerably smaller. This is useful for IO services that may or may not be needed.

The ssndx enum in svc.c has 23 entries. The first, LIM, is the limit, the last, END, is the number of entries, excluding itself, and in between are symbols for the 21 services provided. Each symbol defines the n in the SVC N instruction. At the top of svc.c is the jump table, smx_sst[], used by the SVC Handler (SVCH) with n as its index. Here the service function names are listed. These must be in the same order as the ssndx enum. Note that the first entry is the limit = END = 22. This is used by SVCH() to determine if n is valid. If not SMXE_PRIV_VIOL is reported to the Error Manager, which takes control.

It is pretty easy to get ssndx and smx_sst[] out of step. When this happens the actual function activated will not be what you expected. It is fairly simple to find and fix this problem.

Following the jump table are the shell functions, which call SVC N via one of the sb_SVC macros, using the symbols defined in the ssndx enum. (The sb_SVC macros are defined in svc.h.) Each shell function has the same name as the service it represents, with a u added to the prefix, e.g. smxu_. The header file, xapiu.h, defines the shell functions, and then maps each service to a shell function using mapping macros.

All that is required to cause the fs partition to make service calls via SVCH() is to add

```
#include “xapiu.h”
```

after other includes in each module that calls a service. Since, xapiu.h uses mapping macros, all parameters in each service call must be specified. This means that default parameters must be added. For example:

```
smx_TaskStart(ffdemo);
```

must be changed to:

```
smx_TaskStart(ffdemo, 0);
```

Default parameter values are specified in xapi.h.

Never in hmode

SVC functions can be called from pmode or umode, but must not be called from hmode. Since ISRs and trusted LSRs run in hmode, this is an easy mistake to make. The problem is that the n parameter should be stored in the task stack. But in hmode, there is no task stack, only the main stack, so n is stored in the main stack. However, here it is not protected and the results can be pretty wild. Usually a SMXE_PRIV_VIOL will be reported because the n delivered to SVCH() is too large. At other times, the wrong service will be called, which might report some other error such as SMXE_INV_TCB. This can be very confusing until you realize what is wrong.

At the top of sd_diskio_dma_rtos_bspv1.c we have:

```
#if SMX_CFG_SSMX
#include "xapiu.h"
#endif
```

But starting at line 645 is ISR and LSR code. So, ahead of this put:

```
#if SMX_CFG_SSMX
#include "xapip.h"
#endif
```

This reverses the effect of xapiu.h.

Summary

We now have the fs partition making system calls via the SVC exception.

7.2.4 pd4

Finally we are ready to move the fs partition uptown to umode!

ucom Regions

The first thing is to define the *ucom_code* and *ucom_data* regions. These regions are common to utasks and replace the sys_code and sys_data regions used by ptasks. In RTOS\SSMX\svc.c, note that smx_sst[] is left in sys_code because it is used by SVCH(), which runs in hmode. Below this, the shell functions are put into .ucom.text, and xapiu.h is included for prototypes.

For the STM32F746 group containing the HAL files, we removed the project override on the C/C++ Extra Options tab that used mpi_sys.xcc to locate all of the files in sys_code and sys_data regions, and instead we added this override to some files and pragmas to others to locate them elsewhere. The SD driver files are put into the fs sections since they are used only by the file system, and other files and routines are put into ucom sections since they may be needed by any code. (Keep in mind that putting things in ucom goes counter to good security, because it is shared by multiple partitions. For higher security systems, an alternative is to duplicate the HAL routines and data needed by multiple partitions, giving them slightly different names, so each can be located in only one partition.) The remaining HAL files do not have project overrides nor pragmas, so their code and data fall into the default sections (.text, .data, etc.) which is fine because those routines are called only during startup, which runs in pmode.

Next, we have modified pd2.icf to produce pd4.icf (pd3 uses pd2.icf). In particular, uccsz and ucdsz have been defined, and below them ucom_code and ucom_data are defined. ucom_code includes ucom sections and clib_code. The *.ucom.reset* section is a special section defined in BSP\ARM\STM32\STM32F7xx\reset.c. It has the first two elements of the *intvec* (interrupt vector) table, which are needed for system startup. Note that ucom_code is included in sys_code, which is expected to be at the start of rom_block, and that is where the processor expects to find pointers to CSTACK and to __iar_program_start, when it first starts running. After initialization, the VTOR register points to the real intvec table (see vectors.c). Including the ucom sections in the sys sections allows ptasks to access the clib functions, SVC shells, and other common functions and data.

New MPA

Next, a new MPA is required for fs in umode. In \BSP\ARM\STM32\mpaf7.c under UMODE TEMPLATES is a new template *mpa_tmplt_ufs*. Note that ucom_code and ucom_data have replaced sys_code and sys_data, the EVB region has been removed, and nothing else has changed from mpa_tmplt_fs. EVB is accessed only by system services and thus its region is not needed here. Note: the smx_EVBLogUser() functions, used to log user functions, can be called in umode, but they are accessed via SVC shell functions.

In mp_MPACreate() in ffdemo_init(), mpa_tmplt_ufs has replaced mpa_tmplt_fs.

fs_heap

One more change is necessary because FatFs requires a heap. Since the main heap cannot be used from umode, we must create a new heap, *fs_heap*. At the top of smxmain.c, bins and variables are defined for the main heap, and space for the main heap, itself, is allocated. (It is necessary to allocate it here because its size is determined by SMX_HEAP_SPACE, defined in acfg.h.) Below this, fs_heap bins and variables are defined. Since this a small, low-activity heap, it is given only one bin. Its size is defined near the top of pd4.icf as 0x1000 bytes.

fs_heap is initialized after the main heap (mheap) in smx_HeapsInit(). This consists primarily of putting fs_heap in the fs_heap section, initializing three fields in the fs_hv structure, and then calling smx_HeapInit()[32] which is in xheap.c. This calls eh_Init() in \XBASE\eheap.c. eheap is an RTOS-agnostic heap, which is the basis for smx_Heap. eh_Init() creates the heap, loads the remaining fields in fs_hv. The fs_hv structure, *EHV*, is defined in eheap.h. Then eh_Init() assigns a heap number to fs_heap, which is fs_hn. Finally in XFMW\FatFs\option\syscall.c, ff_memalloc() calls:

```
smx_HeapMalloc(msize, 0, fs_hn);
```

and ff_memfree() calls:

```
smx_HeapFree(mblock, fs_hn);
```

umode

The final step is in smx_TaskCreate() in ffdemo_init() to replace 0 with SMX_FL_UMODE as the flags parameter. The fs partition is now running in umode. (You can verify this by checking *umode* in ffdemo_main vs. *umode* in mcon_main().) As a consequence, mission critical code and system code are protected from the fs partition by the *pmode barrier*. What that means is that any code, including malware, running in the fs partition can access hmode and pmode only via the SVC exception, and that access is limited by the SVC shell functions that have been provided in svc.c.

Background Region (BR)

When BR is on, except in umode, the processor can access all implemented memory. BR on in umode has no effect. Put a breakpoint in ffdemo_main() and bring up the MPU Register window (it is near the bottom of the Group menu). You will see that MPU_CTRL = 5. This means that

---

[32] smx_HeapInit() is called by $Sub$$__call_ctors() in smxmain.c, which is called by __cmain() in the EWARM startup code, prior to its calling C++ initializers, which require a heap.

both BR and the MPU are on. This is true for all utasks. If an interrupt occurs, while in umode, the processor switches to hmode, and BR allows the ISR to access all implemented memory. The same is true for exceptions.

ptasks are different. BR is off in ptasks. For example, in mcon_main(), MPU_CTRL = 1, meaning that BR is off and MPU is on. ptasks rely on sys_code and sys_data to directly access the services they need. If an interrupt occurs, sys_code allows the ISR shell in vectors.c to run. (If not, it must be moved into sys_code.) Then smx_ISR_ENTER() saves the state of BR and turns it on; smx_ISR_EXIT() restores BR to its previous state if control returns to the point of interrupt. Otherwise, smx_PendSV_Handler() runs next, and BR remains on for LSRs that might be dispatched by it.

Reversion to MPU Off

You may be having difficulty finding a problem and you feel that the MPU or partitioning is interfering with your effort, or may be the cause. Or you may be making a major change and do not want to be interrupted with MMFs. Whatever the reason, reversion to MPU off is easy. First, set SMX_CFG_SSMX to 0 in xcfg.h and xarmm_iar.inc. This automatically disables several other SecureSMX features. Next: Top node Options, Linker, and change pd4a.icf to pd0.icf. It is not necessary to change section pragmas in the code because the Linker will now ignore them, and all #include “xapiu.h” statements are disabled. Then, since SMX is provided in library form in these demos, exclude the library from the project and add the nompu version, which was built with SMX_CFG_SSMX 0. We recommend that you give this a try to verify that pd4 runs normally with the MPU off. Remember to reverse these changes before continuing.

Where inherited settings have been overridden there will be a check mark in the gear column of the project window. It is recommended that you enable inherited settings in case the problem is due to changes not being made to all duplicated project sections. Doing this will eliminate duplicated sections in the project file. First save a copy of the .ewp file and restore it when done, to avoid having to redo the overrides.

Sizes

ROM and SRAM usage have continued to increase, as shown in the map file:

ROM	0x136eb	to	0x28000	106%
SRAM	0x6cac	to	0xc000	77%
DRAM	0x8000	to	0x8000	0%

This is using pd4a.icf, and the percentages are vs. pd0 sizes. Now is a good time to run MpuPacker to see if any improvement is possible. It is in the BIN directory, and it is documented in section 8.11.1 Using MpuPacker, but some information is presented here. Make sure it set for pd4. It generates two files: MpuPacker.txt and MpuPackerDiag.txt in APPM\IAR.AM\STM32. In the EWARM Open dialog, click “All files (*.*)” in the lower right corner to see these. Comparing the first file to **pd4a.icf**, we see that no improvement in ordering can be made.

The second file provides diagnostic information. For rom_block, there are no gaps, but there are 0x873b bytes free at the end. rom_block = 0x28000, so actual size used is 0x28000 – 0x873b = 0x1f8c5. The next smaller region block size (from 0x40000) is 0x20000, so reducing to that (and using 8/8 subregion multiple) would save 0x8000 bytes.

Looking at “Block Tails” in the Diag file, we see that the ucom_code tail can be reduced by changing the region size (i.e. opt = “R”). ucom_code is in sys_code, which is in rom_code. Somehow, ucom_code was way too big. It can be reduced to 0x2000*5/8. sys_code is a little too big and can be reduced to 0x10000*7/8. These can be determined by looking at their sizes in the map file and calculating the correct region size and multiple, but it is easier to let MpuPacker guide you. The R means to divide the region size by 2 at least once, and the S means to reduce the subregion size by 1/8 or more. This can be done iteratively. For example, if it says R, divide the region size by 2 and restore the multiplier to 8/8. Relink and run MpuPacker again, and if there is still an R there, do it again until the R is gone. If an S is there now, reduce the multiple by 1/8 and try again. Repeat if S remains. When no letter is indicated in the opt column, you are done.

By reducing ucom_code and sys_code, there is now more space in rom_block, so with region size 0x20000 and 8/8 multiple the map file shows 0x873b byte block tail. Subregion size is 0x4000, so this is more than 2 subregions, and we change 8/8 to 6/8, leaving 0x73b bytes in the block tail. Now:

```
ROM      0x136eb  to  0x18000  24%
```

This is a dramatic reduction. Now change the linker to use **pd4b.icf**. Looking at Block Tails in MpuPackerDiag.txt there are no tails larger than subregions. However, looking above in this file, we see rom_block now has a gap of 0x2000 and there is 0x73b free. The latter is less than rom_block subregion size = 0x20000/8 = 0x4000.

To work on the gap, the map file shows that fs_code ends at 0x8015000 and the last code ends at 0x80178c5, so there are 0x28c5 bytes not in a region block. Up to 0x2000 of this space can be formed into a *plug block* with 4-byte alignment and put into the gap, thus reducing rom block size by up to 0x2000. This plug block is called *pb1_code*. Now change the linker to use **pd4c.icf**. Notice it selects individual object modules to put into the plug block. (These must be files that do not contain pragmas to control section location or you will get link errors (see 8.11.4 Using Plug Blocks).) Looking at MpuPacker.txt, we see that pb1_code is located between sys_code and fs_code, as expected. Looking at MpuPackerDiag, we see that the gap is gone. This is because, pb1sz = 0x2000 in pd4c.icf. Also, End Free space went from 0x73b to 0x2525, adding 0x1dea more bytes at the end of rom_block for it to grow. If End Free had been greater than the subregion size of 0x4000 for rom_block, it would have allowed reducing the multiplier by 1/8 reducing rom_block by 0x4000.

This significant reduction to 24% plus more space for code in rom_block illustrates what can be done. Also there are other techniques that can reduce memory waste even further, as discussed in Section 8.11 Reducing Memory Waste for ARMM7. However, it is clear that the above work is best left until the end of the project, unless packing is so poor that things won’t fit in memory.

Looking at MpuPackerDiag.txt, we see that ucom_code, sys_code, and fs_code have “tails”. Tails are unused memory at the ends of blocks. Having unused memory at the ends of regions is better than having all of it after all code and all data because it allows code and data to grow within regions, thus enabling partition-only updates.

Performance

The difference in average 4096-byte file write and read performances from pd0 to pd4 is less than the jitter from one measurement to the next. Therefore the performance of the SD card is the limiting factor. Measured performances are: 1.17 mbps write and 3.72 mbps read.

Summary

The fs partition is now running in umode. Hence, mission critical code and system code are safe from malware that may have infected the fs partition. The price for this enhanced security in memory is small and in performance is none.

7.2.5 pd5

ffdemo is effectively a client and FatFs is effectively a server. Clients and servers are normally in separate partitions. ffdemo and FatFs were put into the same partition in pd2 for the sake of simplicity. Now we will separate them. In order to do so, the FatFs partition requires a task and a portal. The portal enables clients to communicate with the server, without calling server functions directly, which would violate isolation between server and client.

Server Tasks

One-shot tasks are ideal for servers because, after a server processes a request, it may sit idle for a long time. In addition, the process of receiving a request, processing it, then sending back a result is a one-shot operation, wherein no information need be carried over from one to the next. A one-shot task gives up its stack while waiting for the next request so that another task can use the stack.

7.2.6 pd6

The ISR Problem

Back in the good ol'embedded-systems days we had a rule that ISRs were to be as short as possible. Obviously, this rule has been forgotten, with dire consequences to security. The two ISRs for FatFs have a combined size of 972 bytes, not counting the functions they call. This provides a large target for a hacker and it is not easy to write so much unhackable code. The even worse news is that ISRs run in hmode. Hence, if a hacker breaks in, it takes him only two instructions to turn off the MPU and then the entire system is his to exploit!

To see how the ISRs work in FatFs, go to pd5\BSP\ARM\STM32\STM32F7xx\vectors.c. There you will see IRQ49, 59, and 69 have vectors to shell functions at the bottom of vectors.c. These shell functions wrap the FatFs ISRs in smx_ISR_ENTER() and smx_ISR_EXIT(), which are necessary to integrate the ISRs with smx. Put breakpoints on the FatFs ISRs. Then you can trace into these ISRs to see how complex they are. Eventually these ISRs call completion callback functions. These are at the bottom of sd_diskio_dma_rtos.bspv1.c. These callbacks have been modified for SMX to invoke LSRs such as BSP_SD_WriteCpltCallbackLSR, which, in turn, call the functions that perform the smx services necessary for completion. The foregoing only accomplishes allowing FatFs to run with smx – the ISRs and LSRs run in hmode, so no security gain has been made. This type of LSR is called a *trusted LSR*, meaning that it runs in hmode.

Safe LSRs

In this step, we have converted trusted LSRs to *safe LSRs*, which run in umode. Also we have moved as much code, as possible from the FatFs ISRs to the LSRs. Safe LSRs are like mini-tasks – they have very low overhead, very small stacks, and can share MPU regions with tasks in the partitions to which they belong. All LSRs run between ISRs and tasks. Hence they are ideal for high-performance code that normally would go into ISRs -- they cannot be delayed by priority inversions and other problem that beset tasks, such as high switching overhead.

To see how this works, place the same breakpoints in pd6\BSP\ARM\STM32\STM32F7xx\vectors.c and trace from them. You will see quite a difference in the ISRs and that all the complicated code is now in the uLSRs, which run in umode. If this code is hacked, the hacker is locked into a umode partition – he thus cannot turn off the MPU.

The preceding demos present a simple method to add isolated partitions to an existing system. The remaining sections that follow add more general implementation information to that of the demos.

Chapter 8 Implementation

This chapter provides implementation information that expands upon theoretical discussions and demos of the previous chapters. The intent here is to bridge the gap between theory and actually writing code. Design techniques and tips are presented.

8.1 Planning

8.1.1 Security Plan

Improving the security of a system requires a security plan. This normally consists of identifying threats, then figuring out how to counter them. For example, to counter the man-in-the-middle attack, all data to and from the system should be encrypted. This requires a secret key stored in the system. The next threat is that a hacker will steal this key. To counter this threat, the key must be stored in a secure pmode partition and all code accessing that key must also be in that partition. In addition, a portal must be used to securely transfer data in and out of the partition in order to encrypt or decrypt it. This encryption/decryption process is thus hidden from all client partitions that use or generate the data. SecureSMX provides the necessary tools for securing keys and other secret information.

Any place where the system contacts the outside world is a possible *attack surface*. Hence, it is desirable to enclose each of these, especially low-level drivers, in umode partitions. The latter provide greater isolation and security than pmode partitions. The attack vectors used by hackers are so numerous and powerful that it is not practical to defend against every one. It is better to assume that any I/O partition can be breached and to focus on containing the incursion. Fully isolated partitions can ensure that although the infected partition has become malicious, the rest of the system is able to shut it down, restart it or the whole system, continue its mission-critical function, and to report the attack. This requires careful planning of partitions and recovery methods.

Of course, an infected partition can be spewing bad data that can cause other partitions to become damaged and to cause damage. This must be guarded against in the coding of these partitions. Task stacks in their own XN partitions take away common hacking attacks such as code execution from stacks and stack overflow. This protection is part of SecureSMX. An attempt should be made to use only buffers that are in their own regions, in order to immediately catch buffer overflows – another common hacking technique. Attempts to go outside of partitions will generate immediate MMFs. These can stop a hacker in his tracks, provided that the MMFs are acted upon. This requires a plan for how to handle breaches of partitions, and this may determine how partitions are defined. Also a plan is needed for code to log MMFs for support team study and to perform system recovery.

Probably the greatest vulnerability in a system using SecureSMX is its ISRs. As noted in section 4.8.2 Enabling ISRs and Exception Handlers to Run it may be necessary for BR to be on when

an interrupt occurs. Even if BR is not on, a hacker can turn it on or turn the MPU off with a single instruction. The best approach to guard against ISR infection is to first minimize ISR code. This can be done by invoking LSRs, which perform all non-essential functions, before re-enabling the interrupt, and for LSRs to start or resume utasks to do the bulk of the processing that is not time-critical. In umode we are better able to contain the attack. Clearly there are performance trade-offs in doing this, and these may require careful planning. In addition, ISRs must be fortified – see section 8.10.6 ISRs and LSRs.

Unfortunately, hackers have extensive arsenals of tools, and they spend full time figuring out how to attack systems like yours. System developers, on the other hand, must spend most of their time creating and debugging useful software. Thus, the good guys are seriously outgunned by the bad guys. The best we can do is do guard against common hacking techniques and apply common sense.

8.1.2 Reliability Plan

It makes little sense to have a secure system that is unreliable. Although security and reliability are two sides of the same coin there are differences in planning. Reliability failures result from accidents, not deliberate actions. Hence we need not be concerned about attack surfaces but we do need to be concerned about code quality. (Of course code quality is also a factor in security.)

There is usually some code in a system that is questionable because it is poorly written, inadequately tested, or third-party code that no one understands. Typically there is not enough manpower nor time to carefully review and fix such code. In this situation, putting the code into a fully isolated umode partition and adding a portal to access it may be the best solution. Then, when the code malfunctions, it should not bring down the whole system. Instead, the malfunction can be logged for future repair, and the partition can be shut down or reinitialized and restarted.

Often, questionable code is the result of adding a feature for a large customer or a new feature requested by Marketing. Such code may not receive the careful attention nor talent that the main code has had, or the code may have originated from a third party. It is a good practice to put such code into an isolated partition – at least until it has proven itself worthy of your trust.

8.1.3 When to Add MPU Support

For an existing product or a project near completion, the answer for when to add MPU support is now. For a new project the answer may depend upon factors such as schedule pressure, market window, and preferences or talents of team members.

There is no question that the best results will be achieved if security and reliability are “baked in” to the system from the start. Using SecureSMX does introduce a learning curve to master the new methodologies that it offers. However, once that learning is achieved it may actually help to reduce debugging time, and it will produce a better design structure that is easier to maintain and to extend in the future.

Another point of view is that implementation of MPU support is best postponed until the main design goals have been achieved and the design is stable. Otherwise, introducing it may be too much of a distraction and may result in missing the schedule. If the schedule is tight and manpower is limited, this may be correct. In this case, we suggest that team members read this manual in order to get an idea of what is needed and that a security plan is created that sets forth

design rules to minimize redesign. At a minimum, partitions and their APIs should be identified, and at least one task should be created for each partition. Planning like this requires little time now and will reduce redesign time, in the future.

Unfortunately, in the late stage of project development, delivery pressure and long hours may have set in, and “niceties” such as safety, security, and reliability may get little or no priority. In this case, security can be delayed a little longer. It may even be acceptable for a project to go into early production without the full security plan having been implemented. See section 8.2.3 Iterative Process.

8.2 Project Approach

8.2.1 Legacy Code

Even in a new project, there is likely to be some legacy code. An effective method is to gradually convert less-trusted code to ucode and less-trusted tasks to utasks, while frequently verifying that the system still runs correctly. If a problem occurs, the last step can be reversed and the problem tracked down and fixed.

Sometimes it happens that you have some poorly written code or third party code that no one understands and which has or is expected to come under attack. This is often called SOUP (Software of Unknown Pedigree). Rather than rewriting the code, it is easier to put it into an isolated umode partition. Then at least a hacker cannot bring down the rest of the system if he gets into the SOUP. This may be an adequate first step with other partitions ported later.

See Chapter 7 and the partition demos for a step-by-step process to do this. This process is repeated for each partition to move to umode. Note that certain tasks will be left in pmode. These include security tasks (e.g. crypto tasks), mission-critical tasks, and tasks that cannot run in umode due to performance or other issues. Figure 8.1 illustrates the basic process.

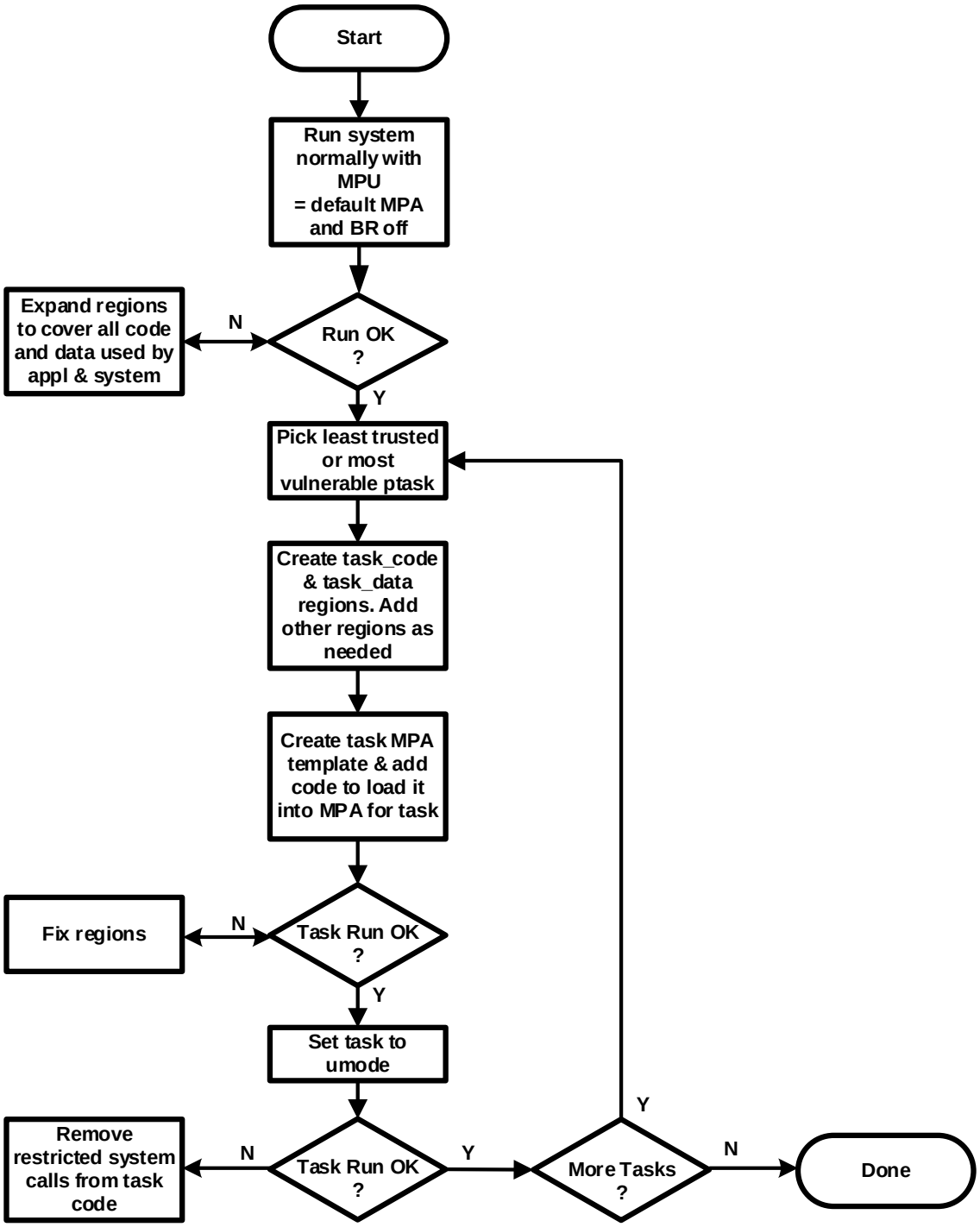

Figure 8.1 Converting ptasks to utasks

8.2.2 New Code

If you are starting a new project, the first step is to decide on partitions and whether the partitions should be in umode or pmode. For new code, the steps are:

1. Define all partitions for the project.
2. Create a working base.
3. Per partition:
   a. Define all tasks — utasks for umode and ptasks for pmode.
   b. Implement the code.
   c. Define regions for each task.
   d. Create a template for each partition.
   e. Debug tasks, and get partition working in its intended mode.
   f. Implement portals for the partition, if needed.
4. Repeat steps 3a-3f for each partition defined in step 1.

In this case, step 1 precedes writing code, and therefore each partition is defined in a written specification with data flow diagrams. The working base should be implemented and debugged next so there is an environment in which to debug the partitions as they are written.

If you prefer, you can write your new code in pmode, then use the approach in section 8.2.1 Legacy Code to convert it to umode. An advantage of this approach is that it is easier to write and debug code in pmode. In doing so, you avoid umode restrictions on system calls and interrupts. However, once you get your code running, you must port it to umode. We think it is more efficient to develop umode code in umode and pmode code in pmode. This saves reworking the code and results in a more sound design. However, the decision is up to you.[33]

Even in a new design, you may have some legacy code. It is quite possible that some of the code may be fine-tuned, mission-critical code. If moved to umode it will definitely run slower and thus may need to be rewritten. If this is not acceptable there is no problem leaving this code in pmode. Of course, if the code has been experiencing cyber attacks, then it needs to be moved to umode.

8.2.3 Iterative Process

As should be apparent from the foregoing, an actual project is likely to be a combination of converting legacy code and creating new code. Hence, adding security and improving reliability will most likely be an iterative process. It is not necessary to achieve perfectly isolated partitions on the first pass. Incorporating SecureSMX, even minimally, makes security and reliability better. Obviously shipment schedules must be met. Initially sales and manufacturing must ramp up, and then there will be a growing population of devices. When the population of units in the field is small, the payoff for hackers may be too little to motivate them. It is probably acceptable to gradually improve security and reliability over this time period. In fact, security improvement is typically an ongoing process for the life of the product, anyway.

[33] One difficulty with developing a partition first for pmode is the loss of two MPU regions for sys_code and sys_data that might require effort to work around, only to reverse that work when they are removed for umode.

8.2.4 Keeping a Log and Backups

It is recommended that you keep a log of changes made as you progress, test frequently, and save backups at each working point. Then when the system stops working, it is easier to figure out what change caused the problem. Keeping a Was/Is list by cutting and pasting old/new code is another helpful technique. Then when a wrong change has been made, it is easy to reverse it by doing a side-by-side diff to the last working backup and reviewing the change logs.

8.3 Working Base

The working base consists of boot, initialization, HAL, SecureSMX, other system services, idle task, and possibly legacy code. This code initially runs in pmode with the MPU off. See pd0 in Chapter 7 for an example of a working base.

8.3.1 Getting Started

SMX runs ainit() in smxmain.c under the idle task with maximum priority, PRI_SYS, in order to initialize smx and middleware. ainit() also calls appl_init(), which should be used to initialize application code. The idle task runs with the init MPA so it has access to all implemented memory and I/O.

Your tasks are assigned the default MPA in smx_TaskCreate(), which will be changed in a later step. They will run in pmode so there is no restriction on the services they can use. Turn on the MPU by calling

```
mp_MPUInit();
```

You can start your tasks in appl_init(), but they will not run until the idle task is restarted with smx_IdleMain() code and 0 priority at the end of ainit(). Your system should still run, and it is actually using the MPU. Simple as this change is, there is actually some gain: Any access outside of implemented memory or with the wrong attributes (e.g. execute from RAM) will result in an MMF.

Now you have a working base upon which to build and debug your new code.

8.4 Partitions

8.4.1 Creating Partitions

Partitioning has been introduced in section 4.1 Partitions and Tasks, and Chapter 6 Advanced Theory presents a complete set of demos that illustrate how to create a partition and move it to umode. A partition is usually formed from a body of code and data that performs a specific system-level function, such as a file system. The first step is to group all code and data that implement the system function into the partition. In the case of a file system, this would include all drivers that it uses, with the exception of ISRs and LSRs. Because the latter must execute in hmode, they are left out of a umode file system partition. It is important to get as much code out of ISRs and LSRs and into partition utasks, as possible. See section 4.8.3 Interrupts for more discussion of this.

When working with legacy code, the above grouping of code and data into a partition is normally performed in pmode. Then regions are defined for the partition and grouped into a partition template used to load the MPA for each task in the partition, as discussed in section 4.3.1 Creating and Loading MPAs. For simplicity, in the discussion below, we will assume that a partition has only one task. If the partition has no tasks, then a task must be created for it. For example, a file system normally does not have a task, but rather runs in the context of each client task that uses it. The file system task is a server task that provides a portal for its clients to get file system services. See Chapter 5 Partition Portals for discussion.

8.4.2 Partition Overlap

Existing application code normally already uses tasks, and new application code will normally be designed to use tasks. Hence, the tasks can serve as guide for defining partitions. Initially it may be a good idea to create a partition for each task. However, after doing this, it may become apparent that a great deal of code and data are common between some partitions. There are two approaches to dealing with this:

- Define common regions for code and data.
- Merge partitions.

The problem with the first approach is that the partitions are no longer fully isolated from each other. The problem with the second approach is that the partitions are larger and thus more of the system is vulnerable if a partition is breached. For more discussion see sections 4.7.2 Combined Regions and 4.7.3 Common Regions. In the case of legacy software, one may just have to live with these problems – at least for the first pass. They can be reduced through restructuring in future passes.

8.4.3 Using Region Tails

As noted in Section 1.3 Advantages of Isolated Partitions, there are more advantages to partitioning a system than increased security and reliability. Region updating is one of these. Referring to the map file in section 4.6 Map File, we see the following:

```
fs_code                          0x00260000    0x7000  <Block>
  .fs.rodata            const    0x00260000      0x24  fmount.o [6]
  .fs.text              ro code  0x00260024    0x100a  fapi.o [6]
    …
  fs_code               const    0x00264ee0    0x2120  <Block tail>
```

fs_code has a size of 0x7000 and an unused tail of 0x2120 due to ARMM7 MPU region requirements. This tail would normally be wasted space. However, it can be used as spare space to permit loading larger versions of the file system or to update the file system without changing other code. This is possible because the space has been permanently allocated to the file system by the linker.

To have interchangeable versions of a server, outside code must make its service calls via a jump table, so that entry points (i.e. the jump table) are the same from one server version to the next. With portals, this is achieved automatically because external calls go through a portal switch statement, which makes the actual file service calls. Addresses in the switch statement are automatically updated whenever the partition is recompiled and relinked.

This feature is not dependent upon ARMM7 block tails. ARMM8 regions can be assigned a larger space than needed in order to allow for different code versions and future updates. This is a better use for spare space than allowing it to accumulate at the end of memory regions.

8.4.4 Partition Updating

A partition that is fully isolated by using portals can be updated independently of the rest of the system provided that any external functions that it uses, such as ucom_code do not move. This is because other partitions that interact with the partition do so through its portals and do not access internal code nor data within the partition. Hence addresses of internal code and data within the partition are relative to each other and are free to change if the partition is recompiled since only the partition, itself, uses them. Tails in the regions used by the partition allow for expansion of code, data, or the task stack in those regions.

Partition updates have the following advantages over full system updates:

- Less code to deal with.
- Updating is faster and requires less bandwidth to download updates.
- Buffer sizes are significantly reduced.
- System code is untouched.
- Possible to run tests on the new partition before accepting it.
- May be possible to update non-critical partitions while the system continues running.
- If the entire code is being updated, the hacker can inject malware where it will do the most damage – most likely a critical partition. Since critical partitions have been carefully written and thoroughly tested they are not likely to be updated often. Non-critical partitions are more likely to be updated and partition updating allows them to be updated without exposing critical partitions to injection attacks.

Partition updating requires modifying and thoroughly testing the updated code with the full system at the home base, then extracting the partition object code from the full system object code, sending it to the devices in the field, which receive and patch it into their object.

8.5 Templates & Regions

See section 4.3 MPA Templates for discussion of how templates are defined and used. As shown there, the general case is to define a template per partition, which is shared between the tasks in the partition. This simplifies the process of deriving child tasks from parent tasks. However in the following, we will discuss task templates (i.e. a template per task) in order to simplify discussion.

8.5.1 Creating Templates

If you are going to start with a ptask, then the template must contain the sys_code and sys_data regions since BR is turned off when ptasks run. If these are already in static MPU slots, then they need not be added to the template. A good way to create a new template is to pick a template in mpa.c that is close to what you need and modify it.

When the template has been defined, load it into the task's MPA with:

```
mp_MPACreate(task, tmp, tmsk, mpasz);
```

Where task is the task, tmp is the template pointer, tmsk is the template mask, and mpasz is the MPA size. For 8 active slots with no auxiliary slots, tmsk = 0xFF, and mpasz = 8. Now that the task is no longer using the default MPA, it is likely to experience MMFs when it starts running. Solving the MMFs may consist, in many cases, of just moving taskA-specific code and data into. taskA_code and taskA_data regions, respectively. Assigning regions to tasks is task-specific. Some tasks may need fewer memory regions, but more I/O regions, etc.

8.5.2 Code and Data Regions

The first step is to group code and data into task-specific regions and to define blocks in the linker command file to hold these regions. It is convenient to name them after the task, e.g.: *taskA_code* and *taskA_data* or name them after the partition, *e.g. usbh_code* and *usbh_data*. If not already the case, it may be helpful to put all task-specific code into a single module, and it may also be helpful to put all task-specific static data, if any, at the start of that module. Or, you can leave functional partitioning of code and data into modules as is.

8.5.3 I/O Regions

Unlike memory regions, I/O regions have fixed addresses. It is thus easier to specify them with direct memory addresses and not involve the linker. For example:

```
RGN(6 | 0x40011000 | V, IO | ( 9 << 1) | EN, "USART1"),
```

Look at the memory map information in the MCU manual to find the memory range for a particular peripheral. For ARMM7 the region must be a power of 2, aligned on that size boundary, and for ARMM8 the region must be a multiple of 32 bytes on a 32 byte boundary. Put its starting address in the RGN() definition, e.g. 0x40011000 above. For ARMM7 fill in the size exponent, e.g. (9 << 1) above. The value of the size exponent is $\log_2$ block_size – 1. For example if the region is 0x400 = 1024 bytes, 1024 = 2^{10}, and 10 – 1 = 9. The << 1 shifts it to the correct position in RASR. For ARMM8, set RLAR to the address limit, rounded down to the previous 32-byte boundary. For example, if it ends at 0x…FF, set it to 0x…E0.

8.5.4 Too Many I/O Regions

Sometimes more I/O regions are needed than there are available slots in the MPU. For example, a particular task using smxNS needs the ETH, GPIO, RCC, and USART1 regions. The ETH region is the Ethernet controller, the GPIO and RCC regions are needed by the driver for pin and clock configuration, and the UART region is needed for status and error messages. In addition the task needs its code and data regions, common code and data regions, and an EMAC buffer region, totaling up to 9.

Clearly a creative solution is needed here. One possibility is to run the task in pmode for initialization, without the ETH, USART1, and EMAC regions, thus requiring 8 slots. Then run the task in umode, without the GPIO and RCC regions, thus requiring 7 slots. This might work, depending upon how the code is structured.

Another possibility is to create larger regions that span individual I/O regions, such as the following from mpa_tmplt_ns in mpaf7.c:

```
RGN(5 | 0x40020000 | V, IO | N3 | N57 | (15 << 1) | EN, "ETH, RCC, GPIO"),
RGN(6 | 0x40011000 | V, IO              | ( 9 << 1) | EN, "USART1"),
```

Note the use of subregion disables N3 and N57 to block intervening peripheral regions as much as possible. In this case, region 5 starts at 0x4002 0000 and goes to 0x4003 0000. This includes 16 other I/O regions, which this task should not access. The subregion disables block address ranges 0x4002 6000 to 0x4002 7FFF and 0x4002 A000 to 0x4002 FFFF, but 13 I/O regions are still exposed. Clearly this is not a good solution.

Using auxiliary regions may be a better solution. In this case active slot 5 would be shared by auxiliary regions 8, 9, and 10:

```
Region 8: 0x4002 8000 to 0x4002 93FF, size = 0x2000, SRD = 0xE0. ETH
Region 9: 0x4002 3800 to 0x4002 3BFF, size = 0x400, SRD = 0. RCC
Region 10: 0x4002 0000 to 0x4002 03FF, size = 0x400, SRD = 0. GPIOA
```

These are all exact fits – no outside I/O regions are exposed. Now the question is can the code be modified to swap in I/O regions as necessary? For example, using:

```
mp_MPASlotMove(5, 8);
```

when ETH access is needed. The places to put slot moves can be found by running the code and finding where MMFs occur. As long as putting slot moves in the code is viable, and performance is not seriously impacted, this is clearly a better solution than spanning I/O regions. However, a few words of caution are in order: (1) A slot move can only be put into code that is unique to a task. It cannot be put into a subroutine shared by two or more tasks unless the tasks have identical shared slots and corresponding auxiliary slots. (2) It cannot be put into a subroutine shared between a task and an LSR. (3) If a common subroutine requires a slot move, internally, one solution is to split the subroutine and to do the slot moves in task code between split routine calls. Implementing slot moves may require some creative programming.

If it is necessary to span I/O regions, here are steps to help determine the subregion disables:

1. List the peripherals needed by the task.
2. Write the addresses ranges next to each from the MCU manual.
3. Order them by address.
4. Consider the distances between them and decide how to group them based on the number of slots.
5. Subtract addresses from the beginning of the lowest to the end of the highest to see how big the range is. Round up to the next power of 2 (0x4000 for example).
6. Divide by 8 to get the subregion size (0x800 in this case).
7. Make a list of the subregion addresses, and mark the one(s) where each peripheral is. Depending on the size of the I/O space for the peripheral, it might span multiple subregions or it may be a small sliver of a subregion. In this case, each fit in a subregion but used only part of it (since each is only 0x400 bytes).

```
 0 0x40010000 - 0x400107FF
 1       800        1000
*2      1000        17FF  UART1
 3      1800        1FFF
 4      2000        27FF
*5      2800        2FFF  SDMMC1
 6      3000        37FF
 7      3800        3FFF
```

8. The unused subregions are excluded by N0|N1|N3|N4|N6|N7, in this case.
9. Check results in smxAware's Task display. Expand the task and scroll down to MPAs.

```
MPA[4]     rbar 40010014    rasr 1300db1b   "USART1, SDMMC1"
   Start           40010000
   End             40013fff
   Subreg Dis    0,1,3,4,6,7 (Size 0x800)
   Sub Start     40011000 (Size 0x800)
      End          400117ff
   Sub Start     40012800 (Size 0x800)
      End          40012fff
   Attributes    PIO
```

In this case, the peripherals were quite close, so subregion size was fairly tight, but it can be terrible if they are separated by much. You might get better results by changing the region grouping, so do a little experimentation.

8.5.5 MPU Region Details

MPU slots in which the EN bit is 0 are *inactive* and have no effect upon memory accesses. Hence a user is not forced to use all slots. Unused slots are usually filled with 0's to disable them. Regions should not overlap, unless the overlap is disabled with subregion disables in one of the regions. The problems with overlap are discussed in section 4.2.6 MPU Slot Numbers & Region Overlaps.

Background Region (BR) is enabled by setting MMU_CTRL.PRIVDEFENA. It flows around all other regions in the MPU so that all of memory is covered and accessible to pcode (but not to ucode). It is considered to have a slot number of -1. Hence the attributes of MPU regions take precedence over it. For example, if a region is RO, it cannot be written, even though BR is on. Outside of MPU regions, default memory attributes are in effect, as if the MPU were not present.

If regions that are simultaneously present in the MPU overlap, the attributes of the higher-numbered region prevail. This could cause an unexpected action, such as an MMF. Using the subregion disables of one region in the overlap area will prevent this problem. Another problem to be aware of is adjacent regions in the MPU. In this case, overflow or underflow from one region to the other will not be detected, unless there is a permission violation.

smxAware flags region overlap and adjacent region problems in the MPA and MPU displays.

8.5.6 ucom_code Region

The ucom_code region contains the shell functions necessary for SVC calls, tunnel portal functions, and C library functions that are needed by more than one utask. It should be the only region shared between umode partitions in the final system implementation. It is a read-only region that should be stored in ROM, if possible. Even if loaded into RAM, it is protected from change by the MPU RO attribute. Given that dangerous C library routines are not included, we think that ucom_code is sufficiently safe from hacking.

The SVC shell functions do little more than invoke the SVC N instruction with a value of n. The actual system functions are performed in pmode out of the reach of the hacker. smx SSRs have thorough parameter testing to avoid problems with bad parameters. Possibly it will be necessary to add this for other system functions that currently do not test all parameters. Tunnel portal functions must be run in umode due to technical difficulties. Free portal functions run in pmode.

It is possible that C library functions will need to be rewritten to be safe (e.g. checking all parameters) and also moved to pmode.

8.5.7 Using TLS to Reduce Regions

As described in section 4.11.8 Task Local Storage, TLS can be defined for a task when the task is first created. It is a section of memory in the task stack block which is below[34] the Register Save Area, RSA. Since it is in the stack block region, it is protected. TLS can replace a task data region, thus saving an MPU slot. This can be particularly useful for an I/O task, which may need more I/O regions than otherwise available slots.

As an example of using TLS, suppose that we have taskA that needs two static variables, var1 and var2 and an 80-byte static buffer separated by a pad to protect against buffer underflow. Then define:

```
typedef struct {
    u32    var1;
    u32    var2;
} taskA_var;

taskA_var tv = taskA->tlsp;                  /* variables */
u8* pp = taskA->tlsp + sizeof(taskA_var);    /* pad pointer */
u8* bp = pp + PAD_SIZE;                      /* buffer pointer */
u32 tlssz = sizeof(taskA_var) + PAD_SIZE + 80;
```

Then:

```
taskA = smx_TaskCreate(taskA_main, PRI3, (tlssz<<16 + 200), h0, “taskA”);
memset(pp, PAD_PATTERN, PAD_SIZE);
```

creates taskA with a 200-byte stack and a TLS with tlssz bytes, and puts a pointer to the TLS in taskA->tlsp. Then the pad is filled with PAD_PATTERN.

[34] As usual, “stack below” means higher memory and “task top” means lowest memory address.

Now, to use the TLS refer to the variables as:

```
tv->var1 = a;
tv->var2 = b;
```

and access the buffer using bp:

```
bp[i] = x;
```

Putting the buffer at the bottom ensures that a buffer overflow (see footnote 34) will generate an immediate MMF. A pad of a recognizable pattern guards the variables from buffer underflow. During debug, watching pad patterns can help to find bugs. In the final system, it is advisable to frequently check that pads have not been overwritten in order to detect hacking.

Using the above technique, multiple structures and buffers may be defined in a TLS. These are for static variables. Auto variables are located in the task stack, as usual.

8.6 Using the Linker

See section 4.4 Linker Command File for discussion of how create a linker command file. This section covers some implementation details.

8.6.1 Block in Block

Ideally, ucom_code should be the only common region between umode partitions. However, as explained in section 4.7.3 Common Regions, in some cases it is necessary to have some other common regions. This reduces the security of the regions, but it may be that too much recoding would otherwise be required. This is a tradeoff that may need to be made between security and feasibility, especially on the first pass.

For pmode, there may be a need to access functions in ucom_code (e.g. SVC shell functions), yet no slots are available. In this case, ucom_code could be included in sys_code, which is accessible by all ptasks, as follows:

```
define exported symbol scsz        = 0x20000;  /* system */

define exported symbol ucomcsz  = 0x8000;   /* ucom */

define block ucom_code with size = ucomcsz*5/8, alignment = ucomcsz
        {ro section .svc.text, ro section .svc.rodata, block cp_code,
         block clib_code};

define block sys_code with fixed order, size = scsz*5/8, alignment = scsz  /* <3> */
        {block ucom_code, ro section .intvec, ro section .sys.text, ro section .sys.rodata};
```

Since ucom_code is an aligned block it is important to put it first in sys_code and to add “with fixed order” to make sure that the linker puts it first. Otherwise a large gap may occur ahead of ucom_code in sys_code. Note that the scsz = 0x20000 and ucomsz = 0x8000, so ucom_code will be aligned on a ucom_code boundary and there will be no wasted space within sys_code due to a boundary gap. If a block being included is not aligned (e.g. clib_code), then it can be put anywhere in the larger block. In case sys_code is first in rom_block, it must have the initial stack pointer and start address in as the first two words, so a mini EVT is put at the start of ucom_code, as is done for other large blocks. See section 8.10.5 Reset Vector.

8.6.2 Initialized Variables

IAR EWARM compilers v7.50, and later, put explicitly zero-initialized data with data that is non-zero-initialized, instead of with bss data that is cleared in blocks. If you want to put such data with bss data, delete the explicit initializer from these variables (i.e. =0, =NULL, =false) or use the compiler's --section switch to set the section names for the whole file, instead of using the section pragma.

Statically initializing variables to 0 or NULL after a #pragma that specifies the section:

```
#pragma default_variable_attributes = @ ".ucom.bss"
u32 my_var = 0;
```

causes the following compiler error:

```
Warning[Be006]: possible conflict for segment/section "<section>"
```

It's unnecessary, so delete the initializer (=0). IAR confirmed the behavior is by design. The reasoning is that when the user places variables in a specified section, one of the important use cases is to handle initialization for those variables in some special way. If you want to handle initialization yourself, you typically need to put zero-inited variables in one section and initialized variables in another. That is the reason for the warning, and also the reason why an explicit zero is treated differently from an implicit zero. This gives the user control over zero-inited variables. For variables that are not placed in user-specified sections, the assumption is that they will be initialized by the normal mechanism, and for those variables, explicit zeros are treated the same as implicit zeros.

8.7 Tasks

8.7.1 Creating ptasks

Most systems start with all ptasks and no utasks. In this case, all tasks will be using the default MPA, dflt_mpa, and they should run with no problem. The following sections explain how to get a ptask running with its own template. At this point, if the task is to remain a ptask, then the work is done. Converting a ptask to a utask is explained in the next section.

8.7.2 Converting from ptask to utask

The first step is to change the template. See section 4.3 MPA Templates for how to do this. Next, add

```
#include "xapiu.h"
```

ahead of all utask code that is supposed to execute in umode. This forces the SVC API to be used by the utask for system service calls. The final step is to move taskA into umode. Creating a task, creating its MPA, and assigning it to umode are usually done during initialization in pmode as in the following example:

```
ut2a = smx_TaskCreate(tm06_ut2a, TP2, 300, SMX_FL_UMODE | h0, "ut2a");
smx_MPACreate(ut2a, mpa_tmplt_ut2a, tmsk);
smx_TaskStart(ut2a);
```

This creates a utask, ut2a, with a main function tm06_ut2a, priority TP2, stack size 300 bytes from the main heap, h0, for umode, and name ut2a. Next an MPA is allocated and loaded with template mpa_tmplt_ut2a, and ut2a is started.

Now, when ut2a is first dispatched, the sb_PendSV_Handler() tail will set the processor CONTROL register = 0x3 just ahead of starting the task. This causes the processor to run in unprivileged thread mode (umode) using the task's stack. Later, when ut2a has been suspended and is then resumed, the EXE_RETURN value (e.g. 0xFFFF FFFD will cause it to run in umode.

8.7.3 Dealing with Restricted and New Services

When a task first starts running as a utask, you are likely to see PRIVILEGE VIOLATION errors. The latter indicate that *restricted service calls* are being made by the utask. See section 4.9.3 Restricted Services for discussion of this. These must be removed from the utask code. Frequently this can be done by moving them to a ptask, such as the initial ptask version of the current utask. If this cannot be done then the utask may need to remain a ptask or it might be split into a utask and a ptask.

If the restricted service is actually a new service that is safe in umode, it needs to be added to the SVC services, as follows:

- Add an abbreviated service name to the ssndx enum in svc.c
- Add the service to the smx_sst[] jump table in svc.c in the same position as the abbreviated service name is in ssndx.
- Add a shell function to svc.c using the abbreviated service name.
- Add the shell prototype function in the upper part of xapiu.h.
- Add the service to shell function conversion in the lower part of xapiu.h.

8.7.4 Dealing with Shared Code and Data

When converting a ptask to a utask, it may occur that the ptask shares code or data with another ptask. Potential solutions to this problem are:

- Move the code or data into a common region accessible by both tasks. This may be acceptable for common code, but may not be acceptable for common data. Also, there may not be an MPU slot available for the new region.
- Put all tasks into one partition and convert the whole partition at once instead of task by task. This is good if all of the ptasks are destined to be utasks.
- Splitting a task into a ptask and a utask, where the ptask shares pcode and pdata with other ptasks and the utask does not.
- Starting a task as a ptask to perform functions required for task initialization, then switching it to a utask that performs only operational functions.
- Replicating common routines using different names.
- Passing global variables via messages or pipes.

8.7.5 Permanent ptasks

Once you have moved everything you can into umode, the ptasks left behind are the royalty, so to speak, that you are trying to protect. They perform security functions, mission-critical functions, and system functions. You have several options for dealing with them:

- Allow them to continue running in the default MPA.
- Put them into one partition and develop a single MPA template for it.
- Put them into individual partitions with individual MPA templates.

System reliability increases with each of these steps.

Implementing ptasks having limited regions does produce some security gain, because many hacker tricks, such as executing code from the task stack or a buffer will trigger an MMF before the hacker can gain access. The MMF handler can then stop the task and sound an alarm. However, best protection comes from moving as much code as possible into umode where a hacker definitely cannot get to the MPU nor to pmode code and data.

8.7.6 Using Child Tasks to Reduce Regions

See Section 4.1.8 Parent and Child Tasks for general discussion of parent and child tasks. Here we consider spawning a child task to deal with a partition that has too many regions. This is a particularly effective method in smx for two reasons: (1) The smx task control block, TCB, is very efficiently implemented and requires only about 25 words, plus another 6 for the MPA, and (2) smx one-shot tasks give up their stacks when waiting. It is likely that only one child task can run at a time in a partition, hence they can all share one stack.

Consider the following template:

```
MPA mpa_tmplt_usbd =
{
  RGN(0 | RA("ucom_code") | V, CODE   | SRD("ucom_code") | RSIC(ucsz)    | EN, "ucom_code"),
  RGN(1 | RA("ucom_data") | V, DATARW | SRD("ucom_data") | RSIC(udsz)    | EN, "ucom_data"),
  RGN(2 | RA("usbd_code") | V, CODE   | SRD("usbd_code") | RSIC(usbdcsz) | EN, "usbd_code"),
  RGN(3 | RA("usbd_data") | V, DATARW | SRD("usbd_data") | RSIC(usbddsz) | EN, "usbd_data"),
  RGN(4 | 0x40010000     | V, IO     | (9 << 1)      | EN, "USART1"),
  RGN(5 | 0x40012C00     | V, IO     | (9 << 1)      | EN, "SDMMC"),
  RGN(6 | 0x40026400     | V, IO     | (9 << 1)      | EN, "DMA2"),
  RGN(7 | V, 0, "stack"),
  RGN(8 | 0x40040000     | V, IO     | (17 << 1)     | EN, "USBHS"),
};
```

This template has 9 regions and will not fit into an 8 slot MPU. This problem can be solved by defining a child task to handle the serial output. The parent task, uptsk, is created in pmode, as follows:

```
/*============================== PMODE ================================*/
TCB_PTR uptsk;
u32 pmsk = 0x16F; /* = b1 0110 1111 parent mask */

uptsk = smx_TaskCreate(uptsk_main, TP2, 500, SMX_FL_UMODE, "uptsk");
smx_MPACreate(uptsk, mpa_tmplt_usbd, pmsk, 8);
smx_TaskStart(uptsk);

/*============================== UMODE ================================*/
#include "xapiu.h"
#pragma default_variable_attributes = @ ".usbd.bss"
TCB_PTR uctsk;
#pragma default_variable_attributes =

#pragma default_function_attributes = @ ".usbd.text"

/* parent task */
void uptsk_main(void)
{
    u32 cmsk = 0x1F; /* = b0001 1111  child mask */
    uctsk = smx_TaskCreate(uctsk_main, TP2, 0, 0, "uctsk");
    smx_MPACreate(uctsk, 0, cmsk, 8);
    smx_TaskStart(uctsk);
    …
}

/* child task */
void uctsk_main(void)
{
    …
}
#pragma default_function_attributes =
```

In this example, it is assumed that the uptsk parent task is created by pmode initialization code. Note that the MPA for uptsk task uses the above template with pmsk, which filters out the USART1 (4) and stack (7) regions. Thus region 4 = SDMMC, region 5 = DMA2, and region 6 = USBHS. Region 7 is automatically loaded with RGN(7| V, 0, 0).

uptsk_main() runs in umode and creates the child task, uctsk. This task inherits mpa_tmplt_usbd from its parent, uptsk, and loads it into its MPA with cmsk. Hence, region 4 = USART1 and slots 5, 6, and 7 are filled with RGN(5 | V, 0, 0), RGN(6 | V, 0, 0), and RGN(7| V, 0, 0), respectively. uctsk also inherits umode from uptsk as well as its IRQ permissions. If a priority above TP2 were specified, uctsk would be given TP2. A child task cannot be created in umode with higher priority than its parent.

Now when uptsk has a message to output via USART1, it signals the semaphore at which uctsk waits. Since uctsk has the same priority as uptsk, it sends the message at a later time after uptsk is done. This would be appropriate for unimportant messages. If uptsk needed to get an important message out quickly, it could send a higher-priority message to a pass exchange at which uctsk waited. This would result in a temporary priority boost for uctsk.

Since uctsk was created with 0 stack size, it is a *one-shot task*. As such, it releases its stack while waiting for the next message. This saves memory. However, uctsk does require a TCB, which takes about 100 bytes of memory.

In the case where an operation is unique to a partition, using a child task makes sense. However, if the operation is needed by more than one partition it may make sense to define a small, special-purpose partition for this operation.

8.8 Creating SVC Calls

You will probably need to create some of your own SVC calls, at some point. For example, your code may require changing I/O connections during operation. For example, if your processor has a USB On-The-Go controller, switching from the USB host stack to the USB device stack requires some GPIO calls. However, it is definitely undesirable to allow a utask access to the GPIO regions – a hacker could cause serious trouble with this, such as redirecting outputs to do the opposite of what they are supposed to. See sections 4.9.1 SVC Calls and 4.9.4 Custom SSTs for information on creating SVC calls.

8.9 Portals

8.9.1 Creating a Free Message Portal

1. Create prtl.h file to define the SLOT, service header struct (SH), and client shell prototypes.

```
#define DEMO_SSLOT   6

typedef struct DEMO_SH {   /* DEMO SERVICE HEADER */
  u32       fid;          /* function ID */
  u32       p1;           /* parameter 1 */
  u32       p2;           /* parameter 2 */
  u32       p3;           /* parameter 3 */
  u32       p4;           /* parameter 4 */
  u32       ret;          /* return value */
  void*     caller;       /* caller addr (debug) */
} DEMO_SH;

int demop_func1(u32 par, TPCS* tpch);
int demop_func2(u32 par1, u32 par2, u32 par3, TPCS* tpch);
…
```

2. Create svr.c file containing:

a. Portal server structure (pss) variable

```
FPSS  demo_pss;
```

b. Portal client list (pcl) with addresses of TPCS variables from the client app code (externed)

```
extern TPCS  demo_pcs;
extern TPCS  demo_pcs1;
TPCS* demo_pcl[] = {&demo_pcs, &demo_pcs1};
```

c. Portal init()

```
bool demo_init(u32 ssn)
{
  FPSS* psh = &demo_pss; /* demo_pss handle */
  TCB*  demo_task;       /* portal server task */

  if (mp_FPortalCreate(psh, demo_pcl, demo_pclsz, ssn, demoname, "demo_sxchg"))
  {
    /* create and start portal server task */
    demo_task = smx_TaskCreate((FUN_PTR)demo_main, DEMOS_PRI, 400,
                                                          SMX_FL_UMODE, "demo_task");
    if (demo_task)
    {
      mp_MPACreate(demo_task, &mpa_tmplt_demo);
      smx_TaskSet(demo_task, SMX_ST_IRQ, (u32)sb_irq_perm_uart, 0);
      demo_pss.stask = demo_task;
      smx_TaskStart(demo_task);
      return true;
    }
  }
  return false;
}
```

d. Portal exit()

```
bool demo_exit(void)
{
  /* shut down portal */
  ret &= smx_TaskDelete(&demo_pss.stask);
  ret &= mp_FPortalDelete(&demo_pss, &demo_pcl, MP_NPCL(demo_pcl));

  return true;
}
```

e. Portal server()

```
void demo_server(FPSS* psh)
{
  MCB*  pmsg;
  DEMO_SH* shp;  /* demo portal header pointer */

  while (pmsg = smx_PMsgReceive(psh->sxchg, (u8**)&shp, psh->ssn, SMX_TMO_INF))
  {
      switch (shp->fid)
      {
        case DEMO_ID_FUNC1:
          shp->ret = demo_func1(shp->p1);
          break;
```

```
            case DEMO_ID_FUNC2:
              shp->ret = demo_func2(shp->p1, shp->p2, shp->p3);
              break;
            ...
          }
          smx_PMsgReply(pmsg);
      }
    }
```

f. Portal server task main function

```
void demo_main(void)
{
  FPSS* psh = &demo_pss;
  demo_server(psh);
}
```

3. Create cli.c file for client shell functions that each send a command to the portal server to run the corresponding API function.

```
#include "dprtl.h"

#pragma default_function_attributes = @ ".svc.text"

int demop_func1(u32 par, TPCS* tpch)
{
  mp_PTL_CALLER_SAV(pch);
  char* dp;
  DEMO_SH* shp;
  MCB*  pmsg;

  if (pmsg = mp_FPortalReceive(pch, (u8**)&shp))
  {
    mp_SHL1(DEMO_ID_FUNC1, mtype, 0);
    dp = (char*)shp + sizeof(DEMO_SH);
    strcpy(dp, mp);
    shp->p2 = (u32)dp;
    mp_FPortalSend(pch, pmsg);
    return shp->ret;
  }
}

int demop_func2(u32 par1, u32 par2, u32 par3, TPCS* tpch)
{
...
```

4. Create pmap.h file to define mapping macros from API calls to client shells. For example:

```
#define demo_func1(par)  demop_func1(par, DEMO_PCH)
```

5. In your init code called by appl_init(), call the portal init function in svr.c, and in exit code called by appl_exit() call the exit function in svr.c.

```
demo_init(DEMO_SSLOT);
demo_exit();
```

6. Create MPA template in mpa.c (ARMM7 shown; ARMM8 similar)

```
#pragma section="demo_code"
#pragma section="demo_data"
...
extern u32 democsz;
extern u32 demodsz;
...
MPA mpa_tmplt_demo =
{
  RGN(0 | RA("demo_code")| V, CODE   | SRD("demo_code")| RSIC(democsz)| EN, "demo_code"),
  RGN(1 | RA("demo_data")| V, DATARW | SRD("demo_data")| RSIC(demodsz)| EN, "demo_data"),
  RGN(2 | RA("ucom_code") | V, CODE   | SRD("ucom_code") | RSIC(svccsz) | EN, "ucom_code"),
  RGN(3 | 0x40011000     | V, IO                    | ( 9 << 1)    | EN, "USART1"),
  RGN(4 | V, 0, "dynamic region"),
  RGN(5 | V, 0, "dynamic region"),
  RGN(6 | V, 0, "dynamic region"),
  RGN(7 | V, 0, "stack"),
};
```

7. Add sections to the .icf file

```
define exported symbol democsz  = 0x___;
define exported symbol demodsz  = 0x___;
...
define block demo_code  with size = democsz,  alignment = democsz  {ro section .demo.text,
                                                                    ro section .demo.rodata};
define block demo_data  with size = demodsz,  alignment = demodsz  {rw section .demo.bss,
                                                                    rw section .demo.data, rw section .demo.noinit};

define block rom_block with fixed order
                {..., block demo_code, ...}

define block sram_block with fixed order
                {..., block demo_data, ...}
```

8.9.2 Creating a Tunnel Portal

The following shows the steps to create a tunnel portal.

1. Create prtl.h file to define the SLOT and TMO settings, service header struct (SH), and the client shell prototypes.

```
#define DEMO_SSLOT   6                 /* slot for pmsg region */
#define DEMO_CTMO    SMX_TMO_INF       /* client timeout */
#define DEMO_STMO    500               /* server timeout */

typedef struct DEMO_SH {   /* DEMO SERVICE HEADER */
  u32       fid;          /* function ID */
  u32       p1;           /* parameter 1 */
  u32       p2;           /* parameter 2 */
  u32       p3;           /* parameter 3 */
  u32       p4;           /* parameter 4 */
  u32       ret;          /* return value */
  void*     caller;       /* caller addr (debug) */
} DEMO_SH;

/* TPMH is the tunnel portal message header */

#define DEMO_THDRSZ  (sizeof(TPMH) + sizeof(DEMO_SH))  /* total header size */

int demop_func1(uint nID, TPCS* tpch);
int demop_read(void* buf, int size, TPCS* tpch);
…
```

2. Create svr.c file containing:

a. Portal server structure (pss) variable.

```
TPSS  demo_pss;
```

b. Portal client list (demo_pcl) which contains the addresses of tunnel portal client structures (TPCSs) in the client code.

```
extern TPCS  demo_pcs;
extern TPCS  demo_pcs1;
TPCS* demo_pcl[] = {&demo_pcs, &demo_pcs1};
```

c. Portal init()

```
bool demo_init(u32 ssn)
{
  TCB_PTR demo_svr = &pss->stask;
  if (mp_TPortalCreate(&demo_pss, &demo_pcl, MP_NPCL(demo_pcl), 0, "demo_portal",
                                                          "demo_portal sxchg", 0))
  {
    /* create and start portal server task */
    demo_svr = smx_TaskCreate((FUN_PTR)demo_main, PRI_NORM, STACK_SIZE,
                                                  SMX_FL_UMODE, "demo_svr_task");
```

```
    if (demo_svr)
    {
      mp_MPACreate(demo_svr, &mpa_tmplt_demo);
      pss->ssn  = ssn;       /* server slot number */
      pss->sid  = DEMO;   /* server id */
      pss->ssid = 0;         /* no subserver */
      smx_TaskStart(demo_svr);
      return true;
    }
  }
  return false;
}
```

d. Portal exit()

```
bool demo_exit(void)
{
    u32 ret = true;

    /* shut down portal */
    ret &= smx_TaskDelete(&demo_pss.stask);
    ret &= mp_TPortalDelete(&demo_pss, &demo_pcl, MP_NPCL(demo_pcl), 0);
    return ret;
}
```

e. Portal server()

```
void demo_server(TPSS* ph)
{
  TPMH*        mhp = (TPMH*)ph->mhp;      /* msg header pointer */
  DEMO_SH* shp = (DEMO_SH*)ph->shp; /* file header pointer */

  switch (shp->fid)
  {
    case DEMO_ID_FUNC1:
      shp->ret = demo_func1(shp->p1);
      break;
    case DEMO_ID_FUNC2:
      shp->ret = demo_func2(shp->p1, shp->p2, shp->p3);
      break;
    case DEMO_ID_READ:
      shp->ret = demo_read((void*)ph->mdp, mhp->mdsz);
      break;
    case DEMO_ID_WRITE:
      shp->ret = demo_write((void*)ph->mdp, mhp->mdsz);
      break;
    ...
  }
}
```

f. Portal server task main function

```
void demo_main(void)
{
  mp_TPortalServer(&demo_pss, DEMO_STMO);
}
```

3. Create cli.c file for client shell functions that each send a command to the portal server to run the corresponding API function. See examples in mp_TPortalCall() and mp_TPortalSend() in Appendix A: API which show how to handle different scenarios for parameters.

```
#include "dprtl.h"
#pragma default_function_attributes = @ ".demo.text"

int demop_func1(uint nID, TPCS* tpch)
{
  mp_PTL_CALLER_SAV(tpch);                          /* save the portal caller address in TCB */
  DEMO_SH* shp = (DEMO_SH*)tpch->shp;               /* service header pointer */
  mp_SHL1(DEMO_ID_FUNC1, par1, ERROR_ID1); /* load service header with 1 parameter */
  mp_TPortalCall(tpch, 0, 0, DEMO_CTMO);            /* send pmsg */
  return shp->ret;                                  /* return received value */
}

int demop_read(void* buf, int size, TPCS* tpch)
{
  mp_PTL_CALLER_SAV(tpch);
  DEMO_SH* shp = (DEMO_SH*)tpch->shp;
  mp_SHL2(DEMO_ID_READ, 0, 0, 0);                   /* load service header with 2 parameters */
  #if NO_COPY
  mp_TPortalReceive(tpch, 0, size, DEMO_CTMO)           /* single block left in pbuf */
  #else
  mp_TPortalReceive(tpch, (u8*)buf, size, DEMO_CTMO)   /* copy multiple blocks to buf */
  #endif
  return shp->ret;
}
...
```

demo_write() would look the same as demo_read() except mp_TPortalSend() is called instead of mp_TPortalReceive(). Only data block read uses mp_TPortalReceive(). All others use mp_TPortalSend(). (mp_TPortalCall() uses mp_TPortalSend()).

Note that for data block read, the source/destination buffer address and the transfer size are passed in the Receive/Send call. In the copy mode, data is copied from/to the buffer passed to/from the portal buffer, and the server passes ph->mhp (field in TPSS) to the actual API call. Size is stored in the message header (mhp->mdsz) which is passed to the API call.

Integers and pointers are easily passed in the SH.par fields, but to pass a string or to get data returned in parameters requires special handling. See the examples in section 8.9.3 Tunnel Portal Client Shells and Server Cases for Most Calls.

4. Create pmap.h file to define mapping macros from API calls to client shells. For example:

```
#define demo_read(buf, size)  demop_read(buf, size, DEMO_PCH)
```

5. In your init code called by appl_init(), call the portal init function in svr.c, and in exit code called by appl_exit() call the exit function in svr.c.

6. In your app task that will use the portal, allocate a pmsg and call mp_TPortalOpen()

```
/* open demo portal with pmsg from heap */
demo_pcs->pmsg = smx_PMsgGetHeap(BLKSZ, (u8**)&demo_pcs->mhp, DEMO_CSLOT,
                                                                    MP_DATARW, 0);
u32 pri = smx_TaskPeek(demo_main, SMX_PK_PRI);
mp_TPortalOpen(demo_pcs, DEMO_BUFSZ, DEMO_THDRSZ, DEMO_CSLOT, pri,
                          DEMO_CTMO, "demo_portal", "demo_portal ssem", "demo_portal csem");
```

7. At the end of the task (if it exits) or at a point where it no longer needs the portal, close it and free the pmsg.

```
/* close demo portal and release pmsg */
mp_TPortalClose(demo_pcs, TMO);
smx_PMsgRel(&demo_pchs->pmsg, DEMO_CSLOT, 0);
demo_pcs->pmsg = NULL;
demo_pcs->mhp = NULL;
```

8. Add a new case to mp_TPortalCallServerFunc() in SSMX\tportls.c:

```
switch (fpsh->sid)
{
  case DEMO:
    demo_server(fpsh);
    break;
```

9. Create MPA template in mpa.c (ARMM7 shown; ARMM8 similar)

```
#pragma section="demo_code"
#pragma section="demo_data"
...
extern u32 democsz;
extern u32 demodsz;
...
MPA mpa_tmplt_demo =
{
  RGN(0 | RA("demo_code")| V, CODE   | SRD("demo_code")| RSIC(democsz)| EN, demo_code"),
  RGN(1 | RA("demo_data")| V, DATARW | SRD("demo_data")| RSIC(demodsz)| EN, demo_data"),
  RGN(2 | RA("ucom_code") | V, CODE   | SRD("ucom_code") | RSIC(svccsz) | EN, "ucom_code"),
  RGN(3 | 0x40011000     | V, IO  | ( 9 << 1)    | EN, "USART1"),
  RGN(4 | V, 0, "dynamic region"),
  RGN(5 | V, 0, "dynamic region"),
  RGN(6 | V, 0, "dynamic region"),
  RGN(7 | V, 0, "stack"),
};
```

10. Add sections to the .icf file

```
define exported symbol democsz  = 0x___;
define exported symbol demodsz  = 0x___;
...
define block demo_code  with size = democsz,  alignment = democsz  {ro section .demo.text,
                                        ro section .demo.rodata};
define block demo_data  with size = demodsz,  alignment = demodsz  {rw section .demo.bss,
                                        rw section .demo.data, rw section .demo.noinit};

define block rom_block with fixed order    {..., block demo_code, ...}
define block sram_block with fixed order  {..., block demo_data, ...}
```

8.9.3 Tunnel Portal Client Shells and Server Cases for Most Calls

Writing the client shells and server cases for each portal call gets easier with experience. Best is to find a similar case, then copy and edit it. This section shows some common examples.

General: In the mp_SHLn() macro, the n is the number of parameters for the function (not including the PCH par). The last value in the macro is the best error value to return if the portal operation itself (not the call) fails. A function that returns true/false would pass false; one that returns an error code should return the most appropriate error code.

These all use mp_TPortalCall() which is a macro that maps onto mp_TPortalSend(). This is used for most calls. The previous section shows data block transfers. The next section shows a more complex case of data block transfers for APIs that transfer a number of items not bytes.

1. Simple Values (input) (e.g. int, char, bool, handle, etc.): Pass the values in order and typecast to (u32) if necessary. These are passed in the service header (SH) par fields.

```
int sfsp_fseek(FILEHANDLE filehandle, long lOffset, int nMethod, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  mp_SHL3(SFS_ID_FSEEK, (u32)filehandle, lOffset, nMethod, 1);
  mp_TPortalCall(pch, SFSP_CTMO);
  return shp->ret;
}
```

   Note that a FILEHANDLE is a pointer, which is treated as a value like an smx handle and points to data on the server side not the client side. The corresponding server case is:

```
case SFS_ID_FSEEK:
  shp->ret = sfs_fseek((FILEHANDLE)shp->p1, shp->p2, shp->p3);
  break;
```

   p1 is typecast back to (FILEHANDLE), and others are simply passed on.

2. Simple Values (output): (Actually these are pointers to simple values, which is necessary for a function to return data via parameters.) These are returned via the portal buffer. Pass 0 for the parameters, and then the server case passes pointers into the portal buffer. The client shell then copies the values out of the portal buffer into the locations the parameters point to:

```
int sup_FTDIGetStatus(uint iID, u8 *pModemStatus, u8 *pLineStatus, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SUP_SH* shp = (SUP_SH*)pch->shp;
  mp_SHL3(SU_ID_FTDI_GET_STATUS, iID, 0, 0, -1);  /* portal returns pars 2, 3 in msg buffer */
  mp_TPortalCall(pch, SUP_FTDI_CTMO);
  if (shp->ret != -1)
  {
    *pModemStatus = *(u8*)(pch->mdp);
    *pLineStatus = *((u8*)(pch->mdp)+1);
  }
  return shp->ret;
}
```

The server case:

```
case SU_ID_FTDI_GET_STATUS:
  shp->ret = su_FTDIGetStatus(shp->p1, (u8*)ph->mdp, (u8*)ph->mdp+1);
  break;
```

Notice the values in this case are each a byte, so the second is one byte after (+1). For a word, it would be (+4). Also notice the client shell only copies the values if the function was successful.

3. Strings and Structures (input): These have to be copied into the portal buffer, and the server side passes the address of each within the portal buffer. Use strcpy() and memcpy() respectively.

```
FILEHANDLE sfsp_fopen(const char* filename, const char* mode, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  char* mdp1 = (char*)pch->mdp;
  char* mdp2 = mdp1 + strlen(filename)+1;
  strcpy(mdp1, filename);
  strcpy(mdp2, mode);
  mp_SHL2(SFS_ID_FOPEN, (u32)mdp1, (u32)mdp2, NULL);
  mp_TPortalCall(pch, SFSP_CTMO);
  return (FILEHANDLE)shp->ret;
}
```

Server case simply passes them on:

```
case SFS_ID_FOPEN:
  shp->ret = (u32)sfs_fopen((const char *)shp->p1, (const char *)shp->p2);
  break;
```

4. Strings and Structures (output (and input)): memcpy() is used before and after the call for input and output, respectively.

```
int sfsp_findnext(int id, FILEINFO* fileinfo, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  /* fileinfo is input and output par */
  void* mdp = (char*)pch->mdp;
  memcpy(mdp, fileinfo, sizeof(FILEINFO));
  mp_SHL2(SFS_ID_FINDNEXT, id, (u32)mdp, -1);
  mp_TPortalCall(pch, SFSP_CTMO);
  /* copy even if ret is -1 since pFindSpec and maybe other fields changed */
  memcpy(fileinfo, mdp, sizeof(FILEINFO));
  return shp->ret;
}
```

Server just passes the pointer:

```
case SFS_ID_FINDNEXT:
  shp->ret = sfs_findnext(shp->p1, (FILEINFO*)shp->p2);
  break;
```

5. Other (String In and Struct Out): Another detail is that since the struct has a fixed size, putting it first in the portal buffer lets us get the offset to the other simply by sizeof() rather than calling strlen(). Notice the pointers are passed in reverse order mdp2 then mdp1.

```
int sfsp_findfirst(const char* filespec, FILEINFO* fileinfo, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  /* put fileinfo first to avoid strlen(), and so same as other sfs_find calls */
  /* fileinfo is output par only so no need to copy here */
  u8* mdp1 = pch->mdp;
  char* mdp2 = (char*)mdp1 + sizeof(FILEINFO);
  strcpy(mdp2, filespec);
  mp_SHL2(SFS_ID_FINDFIRST, (u32)mdp2, (u32)mdp1, -1);
  mp_TPortalCall(pch, SFSP_CTMO);
  /* copy even if ret is -1 since pFindSpec and maybe other fields changed */
  memcpy(fileinfo, mdp1, sizeof(FILEINFO));
  return shp->ret;
}
```

8.9.4 Tunnel Portal Data Block Transfers in Item Units

Data block transfers use mp_TPortalReceive() and mp_TPortalSend() as shown in section 8.9.2 Creating a Tunnel Portal. Special handling is needed if the API transfers a number of items rather than a number of bytes. fread() and fwrite() are examples.

The client shells multiply size and items to pass to the portal, which operates in bytes, but then for the return value, they reverse this to return the number of items read/written by dividing the completed size by the item size. Notice they use the message header cmpsz field, which would be less than the requested size if there were some problem:

```
size_t sfsp_fread(void * buf, size_t size, size_t items, FILEHANDLE filehandle, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  mp_SHL4(SFS_ID_FREAD, 0, 0, 0, (u32)filehandle, 0);  /* portal server fills in pars 1, 2, 3 */
  mp_TPortalReceive(pch, (u8*)buf, size*items, SFSP_CTMO);
  return pch->mhp->cmpsz/size;  /* ret num items (not bytes) completed */
}

size_t sfsp_fwrite(void* buf, size_t size, size_t items, FILEHANDLE filehandle, TPCS* pch)
{
  mp_PTL_CALLER_SAV();
  SFSP_SH* shp = (SFSP_SH*)pch->shp;
  mp_SHL4(SFS_ID_FWRITE, 0, 0, 0, (u32)filehandle, 0);  /* portal server fills in pars 1, 2, 3 */
  mp_TPortalSend(pch, (u8*)buf, size*items, SFSP_CTMO);
  return pch->mhp->cmpsz/size;  /* ret num items (not bytes) completed */
}
```

Server passes to the actual calls as one item of total bytes:

```
case SFS_ID_FREAD:
  shp->ret = sfs_fread((void*)ph->mdp, mhp->mdsz, 1, (FILEHANDLE)shp->p4);
  if (shp->ret != 1)
     mhp->errnum = TRANS_INC;  /* alert mp_TPortalReceive() transfer incomplete */
  break;
case SFS_ID_FWRITE:
  shp->ret = sfs_fwrite((void*)ph->mdp, mhp->mdsz, 1, (FILEHANDLE)shp->p4);
  if (shp->ret != 1)
     mhp->errnum = TRANS_INC;  /* alert mp_TPortalSend() transfer incomplete */
  break;
```

8.9.5 Data Block Transfer Considerations

As mentioned in the note at the start of the Creating a Tunnel Portal, special care is needed for data block transfer API calls.

Notice in the example in the preceding section that special handling is needed to set the message header errnum field to TRANS_INC so that the portal can abort the operation in the case that this is part of a transfer that the portal automatically split because the requested size is larger than the portal buffer size. For example, if the original call to sfs_fread() in the application is 4KB but the portal buffer is only 1KB, the portal automatically splits it into 4 separate reads. If one of these reads fails, the whole operation is aborted by flagging TRANS_INC, as above.

Attention must be given to the return value. If it is a success/fail code, just return shp->ret in the client shell. If it is a completion size, return the amount completed to that point, such as a partial amount of a split transfer, using the cmpsz field of the message header. If the API call returns the number of units rather than bytes, it is necessary to divide by the unit size, as shown in the example in the previous section.

Test the case of failure of an early chunk of a split transfer by simulating failure in the actual API call (by patching the return value in the debugger just before the function returns), and step carefully through the code using breakpoints in mp_TPortalSend() and mp_TPortalReceive(), watching what is happening and ensuring the right value is returned.

Split transfer support is not appropriate for all portal types. It is suitable when success means the full requested amount is returned. For a serial driver that may return fewer, the technique described above doesn't work, and the portal buffer should be the largest request size, to avoid split transfers.

8.9.6 Portal Configuration Settings

Configuration settings are needed for client and server timeouts and slot numbers. Define CTMO, STMO, SSLOT, and CSLOT constants for each portal as shown in SMX code. The first three are characteristics of the portal and should be defined in the portal header file, e.g. fprtl.h for smxFS. CSLOTs are specific to each client task that uses a portal, so they should be defined in the client code.

Servers that wait on an event should have a maximum wait time for each such event. This is called a *timeout*. CTMO should be chosen to be much larger than the largest server timeout, since the purpose of CTMO is to detect that the server has failed or disconnected from the client. This allows the client to initiate a recovery operation. The purpose of STMO is the same in reverse – i.e. to allow the server to detect that the client has failed or abandoned the operation. Although the client is occupied with the server, it is possible that it might make a call, such as smx_MutexGet(), that could result in a wait. So STMO should be much larger than the largest client timeout that could occur during a server operation started by the client. This allows the server to abort the current operation and accept the next pmsg.

If CTMO or STMO are too small, they may interfere with normal operation, causing needless problems. Avoiding this may require careful study. For example, flash disks can have very long delays occasionally. On the other hand, it is not a good idea to set these timeouts to INF, for then no recovery is possible.

8.10 Miscellaneous

8.10.1 Heap Calls

Due to the greater power of smx_Heap calls, they have extra parameters vs. the Standard C Lib heap functions. For example,

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

where an specifies the power-of-two block alignment required and hn is the heap number.

Application code changes can be minimized by defining heap macros, such as:

```
#define malloc(sz)   smx_HeapMalloc(sz, 3, 2)
```

where 3 specifies the normal 8-byte alignment and 2 specifies heap 2, the heap being used in this partition.

8.10.2 Performance Measurements

The smxBase time measurement functions sb_TMInit(), sb_TMStart(), and sb_TMEnd() are designed to make accurate performance measurements in pmode. sb_TMInit() is called automatically during startup from idle/ainit. It determines the calibration constant sb_TMCal() to eliminate the overhead of the start and end functions.

Unprivileged time measurement functions sbu_TMInit(), sbu_TMStart(), and sbu_TMEnd() are provided to make accurate performance measurements in umode. The start and end functions use the sbu_PtimeGet() function in svc.c, which does an SVC call. Call sbu_TMInit() from a utask prior to using these functions in order to set sbu_TMCal to eliminate their overhead. Since there is quite a bit more overhead on sbu_TM calls than on sb_TM calls, accuracy may not be as good.

8.10.3 Where Am I?

sb_IN_UMODE() and sb_IN_SVC() are useful in special situations. For example, sb_IN_UMODE() == true means execution is in umode. It thus cannot be in an LSR or ISR. IN_SVC() means that the current system service was called from umode.

8.10.4 Event Buffer

The Event Buffer, EVB, captures events for smxAware to display. EVB is defined in the linker command file and put into sys_data. When debugging, it is desirable to make EVB as large as possible in order to maximize traces. This can be done by determining how much SRAM is free from the map file and increasing EVB by that much. If this is not enough, EVB can be moved into DRAM, if available. However, that may impact performance since EVBLog functions are called frequently.

8.10.5 Reset Vector

The reset vector (initial SP and IP) must be located at the start of internal flash. Since MpuPacker reorders blocks for efficiency, the block it puts first must have the reset vector. The reset vector is just the first 2 slots of the EVT, so in reset.c, we have created mini vector tables with just these two entries that are put at the start of their respective code sections, e.g. .usbh.reset for usbh_code. The placement and ordering are done in the .icf file like this:

```
define block usbh_code  with fixed order, size = usbhcsz,  alignment = usbhcsz
                        {ro section .usbh.reset, ro section .usbh.text, ro section .usbh.rodata};
```

Notice “with fixed order” and that .usbh.reset is first. This has been done for what is likely to be the biggest region block since MpuPacker locates the biggest region block first. If you create a larger region block, be sure to include ro section .usbh.reset at its start. If this region block includes another region block, include ro section .usbh.reset at the start of that block and put it first.

8.10.6 ISRs and LSRs

As noted in section 4.8.3 Interrupts, ISRs are probably the weakest link in a well-partitioned system. They can be manipulated by hackers from outside of the system. Therefore, it is best to make ISR code short and rugged. To fortify ISRs, LSRs, and other vulnerable code, the following ideas may be helpful:

1. Callees perform validity checks on all parameters.
2. Callers perform validity checks on returns.
3. Range check pointers before using.
4. Put pads (i.e. known patterns) before and after buffers.
5. Check pads frequently to catch overflows.
6. Report errors and abort.

If done well, measures like these will make a hacker's work more difficult and thus may thwart his attack. Clearly it is practical to implement measures like these only on small amounts of code. That is a good reason to keep ISRs and LSRs as short as possible. Any functions that can be deferred from ISRs to LSRs should be, and any functions that can be deferred from LSRs to utasks, should be.

8.10.7 Critical Sections

The introduction of umode results in changes to how methods that protect critical sections work. As a consequence it may require making changes to legacy code. This section gives you detailed guidance for each situation. For background information, please see section 6.2 Critical Sections.

Mutex: A mutex is the most common way to protect a critical section. It works to protect critical sections in tasks against reentrancy by other tasks. It can be used to protect tasks against LSRs. However, since LSRs cannot wait it does not work well for them.

smx_LSRsOff(): This is a more effective way to protect tasks against LSRs. It is better because it blocks LSRs from running until the task is out of the critical section. Then the LSRs can run. However, this service is not available for utasks; see SVC Call below for them.

smx_TaskLock(): Locking a task is an effective way to prevent reentrancy into its critical section by another task. It is much faster than a mutex. But it does not protect tasks against LSRs, and if the critical section calls a service that might suspend or stop the task, the lock is lost. Task lock is implemented with a counter and thus can be called multiple times without harm, provided that task unlock is called the same number of times to unlock the task.

Disabling Interrupts: This is the common approach to protect critical sections in tasks, LSRs, and ISRs from ISRs. However, it does not work in umode! In umode, the processor ignores the cpsie i and cpsid i. Hence, pmode code that relies on these for protection becomes unprotected when it is moved to umode. This is easily overlooked, since the sb_INT_ENABLE() and sb_INT_DISABLE() macros fail silently. To help deal with this problem, we have defined alternate versions of the ENABLE and DISABLE macros in XBASE\barmm.h that stop the debugger if called in umode. Set SB_ARMM_DISABLE_TRAP to 1 to use them.

Masking Interrupts: Typically, when and ISR and a task share a buffer, one is loading the buffer and the other is unloading it. Hence, there is only one ISR to protect against. This can be accomplished by masking the interrupt, which can be done from umode. To allow the utask to do this, it is necessary to create an IRQ permission table, as follows:

```
const IRQ_PERM sb_irq_perm_ns[] = {
  {61, 62},        /* Ethernet */
  {37, 37},        /* USART1 for terminal output */
  {0xFF, 0xFF}, /* terminator */
};
```

This enables IRQs 61 to 62 and 37 to be masked and unmasked. Each row specifies a range of IRQs to allow, and the table is terminated with an entry of 0xFF's. A good place to put IRQ permission tables is at the end of irqtable.c in the BSP. Next, permit the task to use it:

```
smx_TaskSet(task, SMX_ST_IRQ, &sb_irq_perm_ns);
```

Then sb_MaskIRQ() and sb_UnmaskIRQ() can be used by this task to mask and unmask these IRQs.

SVC Call: Study the code to determine what needs to be protected against. To protect umode code from LSRs, it can be made into a SVC call. To protect against one or a few interrupts, mask them as discussed in the previous note. If all interrupts must be disabled, one solution is make the critical section an SVC call, which DISABLEs interrupts ahead of the critical section and ENABLESs them after it. This works for utasks. ptasks can call the code directly.

8.11 Reducing Memory Waste for ARMM7

It is recommended that debugging be done with a test board having ample memory. This allows work to progress without concern for inefficient memory usage. Otherwise, it may be necessary to work with only portions of a system, at a time, or to frequently optimize memory usage.

8.11.1 Using MpuPacker

Due to ARMM7 region size and alignment requirements, region block gaps can be very large. Hence it is important to minimize them. MpuPacker aids in this process. It is in the \BIN subdirectory can be run from the command line.

The following are instructions for how to use it:

1. MpuPacker first opens a dialog box.
2. Enter the full path and name of the .icf and .map files or click the button at the end of the line to browse to them. After the first run, these need not be entered again.
3. Check the Create Diagnostic File box, and then click the Optimize button.
4. Both MpuPacker.txt and MpuPackerDiag.txt will be created in the same directory as the .icf file.
5. MpuPacker.txt shows the optimized block order for the collection blocks in the .icf file as marked by MPUPACKER:COLL. Copy and paste these into the collection blocks in the .icf to replace the original ones.
6. MpuPackerDiag.txt shows more information concerning block ordering, so you can see the efficiency of block placement. In addition, it shows the unused memory in each block, which is called a *block tail*. Reducing tails is discussed next.

8.11.2 Reducing Block Tails

Block tails are potentially wasted memory inside of region blocks. A tail can be as large as the subregion size of the block, minus one byte. In a particular test case, total ROM tails = 34,740 bytes, and total SRAM tails = 38,248 bytes. However, if available memory is not being exceeded, spare memory might as well be distributed among block tails. This is because they provide expansion memory for partition updates, thus allowing smaller and faster updates that may not disrupt normal operation, except for the partition being updated.

As noted above, running MpuPacker produces an MpuPackerDiag.txt file that has a Block Tails section. The section looks as follows:

```
rom_block
                                    addr          tail         subreg      opt
                                    ----          ----         ------      ---
     ucom_code          const       0x0800447a     0x1b86        0x1000       S
     sys_code           uninit      0x08016e60     0x91a0        0x4000       S
     fs_code            const       0x0802b994      0x66c        0x2000
     fsdp_code          const       0x0802c774      0xc8c         0x400       R
     fsdd_code          const       0x0802d400     0x6c00        0x4000       R
     usbhdp_code        const       0x08034cc8      0x138         0x200
     fpu_code           uninit      0x08035000      0x800         0x100       -
```

To reduce block tails, dDo the optimizations to the .icf file indicated in the opt column, and then run the linker and MpuMapper again. The meanings are:

- R Divide the region size by 2 because less than half of the region is being used. Also restore the multiplier to 8/8. If less than 1/4 of the region size were initially used, it would show as R after running again. Repeat the process until R no longer appears.
- S Reduce n in the n/8 multiplier to disable more subregions. This can be done iteratively, reducing by 1 each time until S no longer appears.
- \- Delete the region because it is empty.

The first time this is done it may take several iterations to optimize memory usage. However, after that, as regions grow due to adding code and variables, the linker will complain if a region is exceeded. It is usually easy to increase the region size and continue development.

8.11.3 Reducing Region Block Gaps

Gaps between region blocks can dramatically increase wasted memory, as shown by the MpuPackerDiag.txt file:

```
Details for Region  sram_block  ARMv7 MPU Alignment
       Block sys_data      Size   20000   6/8-size  18000   Start 20000000  End 20017fff  *
       Block lcd_data      Size     200   7/8-size    1c0   Start 20018000  End 200181bf  *
       (gap)               Size     240                     Start 200181c0  End 200183ff  *
       Block fpu_data      Size     400   7/8-size    380   Start 20018400  End 2001877f  *
       (gap)               Size    3880                     Start 20018780  End 2001bfff  *
       Block fpd_data      Size    4000   5/8-size   2800   Start 2001c000  End 2001e7ff  *
       (gap)               Size    1e40                     Start 2001e800  End 2002063f  *
       Block pb1_data      Size    20c0   8/8-size   20c0   Start 20020640  End 200226ff
```

```
       Total Gaps            Size    5900
```

In this case the gaps total to 0x5900 bytes = 22,784 bytes! The MpuPackerDiag.txt file shows the recommended optimum order for this collection block as:

```
Details for Region  sram_block  ARMv7 MPU Alignment
       Block sys_data      Size   20000   6/8-size  18000   Start 20000000  End 20017fff  *
       Block fpd_data      Size    4000   5/8-size   2800   Start 20018000  End 2001a7ff  *
       (gap)               Size    1cc0                      Start 2001a800  End 2001c4bf  *
       Block pb1_data      Size    20c0   8/8-size   20c0   Start 2001c4c0  End 2001e57f
       (gap)               Size     280                      Start 2001e580  End 2001e7ff  *
       Block fpu_data      Size     400   7/8-size    380   Start 2001e800  End 2001eb7f  *
       (gap)               Size      80                      Start 2001eb80  End 2001ebff  *
       Block lcd_data      Size     200   7/8-size    1c0   Start 2001ec00  End 2001edbf  *
       Total Gaps          Size    1fc0
```

Simply reordering the blocks results in a reduction of gaps to 0x1fc0 = 8128 bytes – a savings of 14,656 bytes!

The replacement block shown in MpuPacker.txt:

```
define block sram_block with fixed order, size = sramsz*4/8, alignment = sdsz
                {block sys_data, block fpd_data, block pb1_data, block fpu_data,
                 block lcd_data};
```

should be pasted into the .icf file in place of the initial block and the linker run again.

8.11.4 Using Plug Blocks

What if the above is not enough and memory is still being exceeded? After MpuPacker has done its job, there may still be large gaps between region blocks, as shown above. This can be remedied by defining *plug blocks*. These are blocks that are not used for regions, and thus can be aligned on word boundaries.

It is recommended to put all non-runtime code and data into two plug blocks, as follows:

```
define block pb1_code with size = pb1csz, alignment = 4 {ro section .pb1.text, ro};
define block pb1_data with size = pb1dsz, alignment = 4  {rw section .pb1.bss,
                 rw section .pb1.data, rw section .pb1.noinit, rw};
```

In this case, some non-runtime code has been put into .pb1.text, and ro brings in all remaining code that is not in region blocks. Similar for data. Initially, the plug blocks can be kept at the ends of collection blocks, such as:

```
define block sram_block with fixed order, size = sramsz*4/8, alignment = sdsz
                {block sys_data, block lcd_data, block fpu_data,
                 block fpd_data, block pb1_data};
```

As design progresses, if available memory is exceeded, even using MPUPacker recommendations, pb1_code and pb1_data can be split into smaller plug blocks that fit nicely into the gaps. In this example, pb1_data = 0x20c0 bytes, which is too large for any of the gaps in the optimized sram_block shown above. However, by redefining sections in the data, pb1_data can be split into pb1_data, pb2_data, etc. blocks that nicely fit into the gaps. Clearly, doing this takes time and is not recommended during debug, unless it is unavoidable.

8.11.5 Reducing Region Block Sizes

Larger regions tend to waste more memory. For example, if a region size is 0x20000 then its subregion size (1/8) is 0x4000 = 16KB. It may happen that the actual size is slightly over a subregion boundary with the result that nearly all of the rest of the 16KB subregion is wasted. There is nothing that the linker can do about this. The following techniques can be used:

1. The Tails section MpuPackerDiag shows all block tail sizes next to the subregion size. If the tail is very small compared to the subregion, then this region block is a good candidate for size reduction. In the case of a code partition, using more subroutines might achieve the needed size reduction. In the case of a data partition, reducing a buffer size or sharing a buffer might do the job.
2. If a spare slot is available in every partition template using the region, split the region block into two smaller region blocks, such that the sum of their tails is smaller than the original tail.
3. Split partitions into smaller partitions so that regions are smaller and the sum of the resulting tail sizes is smaller than the original sum of the tail sizes. This is most easily done by defining child tasks. (See parent/child task discussion in section 4.3.2 Using Parent and Child Tasks)

8.11.6 Restructuring Regions

The linker will omit unreferenced code and data, but it cannot distinguish what each task or partition needs. Hence, it may be necessary to do a careful analysis of each overly large region. For example, it might be possible to divide a large region shared by TaskA and TaskB into RegionA, used by TaskA, RegionB, used by TaskB, and RegionC, shared between them. TaskA would use A and C, and TaskB would use B and C.

But suppose TaskA has only one available slot. This can be accommodated by including region block C in region block A in the linker command file. Then TaskA would use region A (including C) in its MPA, whereas TaskB would use regions B and C in its MPA. Now both tasks can access region C, but neither can access the other’s private region, so task isolation has also been improved. At the same time, a large region has been divided into 3 smaller regions, which should reduce memory waste.

8.11.7 Handling Aligned Blocks within Aligned Blocks

A block may be defined to be a region for one task, but to be accessed by another task it must be put inside a larger block that is defined as a region for that task. This typically occurs due lack of an extra MPU slot for the second task. For example, ucom_code is used in utasks that have an available slot, and it is included in sys_code for ptasks that do not. In order to prevent an unnecessary gap within sys_code, put ucom_code first *with fixed order*:

```
define block ucom_code   with size = ucomcsz*5/8, alignment = ucomcsz
                         {ro section .ucom.text, ro section .ucom.rodata}

define block sys_code  with fixed order, size = scsz*7/8, alignment = scsz
               {block ucom_code, ro section .sys.text, ro section .sys.rodata};
```

This results in the following linker output:

```
sys_code                        0x21'2000     0x1c00  <Block>
  ucom_code                     0x21'2000     0x1400  <Block>
    .ucom.text      ro code     0x21'2000      0x340  ucom.o [1]
        ...
    .text           ro code     0x21'30f2       0x1e  strcat.o [2]
    ucom_code       const       0x21'3110       0x2f0  <Block tail>
  .sys.text        ro code      0x21'3400       0x84  startup.o [1]
```

This results in a 0x2f0 tail, but no gap after (0x2f0 + 0x213110 = 0x213400) ucom_code. Note also that ucom_code is aligned on sys_code. This is always the case for an inner region block since its region size cannot be larger than the outer region block.

If more than one region block is included in an outer region block, the internal region blocks should be in put order by decreasing size. However, gaps can still occur due to subregion disables. If so, manual ordering within the larger block may be necessary. For example, if decreasing block-size ordering is A, B, C, but C can fill the A to B gap, then A, C, B will give a better result. An additional technique is to fill gaps with sections. For this purpose, sections can be sized to better fill gaps. For example, if region ucom_code contains A, B, and .ucom.text, split .ucom.text into .ucom.textA and .ucom.textB using section pragmas in the code, so that .ucom.textA nicely fills the rest of the A to B gap after C, and .ucom.textB is put where it fits best in ucom_code. This is possible because section sizes do not have to meet region requirements, so they can be used as stuffing.

8.11.8 Reducing code and data sizes

If easier approaches do not achieve the size reduction goal, it may be necessary to make some code changes. This may be well worth the effort if a region only slightly exceeds a subregion boundary. For code regions, a few ideas to consider are:

- Refactor selected functions to reduce size and to improve performance.
- Reduce repeated code for less used functions by defining subroutines.
- Replace macros with subroutines.
- Be sure to use _pragma("inline=never") ahead of the above subroutines.
- Move out code not actually needed in a partition by using section pragmas. This could be the case if some code is in a region just because it is in the same module, but the code is actually used elsewhere, such as during initialization or exit.
- Move seldom needed functions to sys_code and access them via the SVC Handler from umode – i.e. add to svc.c. Since the number of SVC functions is limited, it is best to create a master function in which a parameter selects each function. This way many functions can be moved out of a partition region. Total parameters for the master function is limited to 7. Of course, this adds overhead to the functions.

For data regions, consider:

- Moving some data to a Task Local Storage, TLS – see Section 4.11.8 Task Local Storage.
- Reducing buffer sizes where reduced performance is acceptable.

- Sharing buffers rather than defining separate buffers. One way to do this is to create a local heap, allocate buffers as needed, and free them when not needed.
- If there is a local heap, use it instead of static variables. It is helpful, in this case, to define related variables as fields in structures and allocate blocks for the structures in order to maximize heap efficiency.
- Move out data not actually needed in a partition by using section pragmas.

8.11.9 Conclusion

Obviously many of the above cures are labor intensive and should be done only if code or data no longer fits within the installed memory in a system that otherwise is ready to be shipped. However, they offer hope that memory waste due to using a ARMM7 MPU can be held below a small percentage, thus allowing security to be improved even in tight-memory systems. In our experience, we find that memory waste can be reduced to about 20% without using extreme measures. If code rewriting and restructuring is permitted, it should be possible to go well below 20%.

However, if there is sufficient memory, it is better to have it distributed among partitions rather than all at the end of each memory since this enables partition-only updates.

8.12 Prerelease Checklist

It is recommended to make a checklist of settings to change prior to release. Here are a few for the list:

- Ensure SMX_CFG_SSMX_ENABLE and all portals (middleware and application) are enabled by XX_PORTAL settings (xcfg.h, xpcfg.h, etc.).
- SB_ARMM_DISABLE_TRAP = 0 in barmm.h to restore the normal DISABLE/ENABLE macros in barmm.h (versions with no DEBUGTRAP).
- MP_MPA_DEV = 0 in mpu.h for fast MPU loading and to eliminate the region name field used for debugging.
- Tune down region sizes in mpa.c by changing RASR SIZE field e.g. (nn << 1). Regions using RSIC() macro are tightly wrapping the code or data they contain. Super regions using (nn << 1) may be spanning whole memories, so tune them down to what is actually used. Check that I/O regions map the needed peripheral ranges as closely as possible. See sections 8.5.3 I/O Regions and 8.5.4 Too Many I/O Regions.
- Do subregion disables to optimize memory usage, according to section 4.3.3 Using ARMM7 MPU Subregions.
- Do code coverage testing to ensure there are no unexecuted sections of code which might generate an MMF due to referencing inaccessible symbols or doing non-permitted operations.

8.13 Design Tips

The following tips may help to reduce design errors:

- Prefix utask names and their main code with “u”, e.g. utaskA, utaskA_main(). This helps to keep the actors straight so you know who is privileged and who is not.
- Avoid static local variables. The linker may not put them where you expect.
- If a module has pcode and ucode, it is best to put the pcode first and follow it with #include “xapiu.h”. Then put the ucode. See section 4.9.6 Mixed Code Modules for more discussion and alternatives.
- If pcode follows ucode, use #include “xapip.h” ahead of it. It is ok to alternate xapiu.h and xapip.h as many times as necessary.
- Using hexadecimal numbers for sizes and alignments in the linker command file, helps to avoid errors — e.g. 0x800 instead of 2048. For a power of 2, only one digit is permitted in the number and the digit must be 1, 2, 4, or 8. So a wrong digit, like 5, stands out. Get a hex calculator if you don’t already have one.
- Make blocks just large enough to contain their section(s). As code or data grows, the linker will let you know if a block overflows and you can increase the size of the block.
- Defining a constant for a region size in the linker command file (e.g. fsdsz), and then using it for region block alignment and size helps to reduce errors.
- Through the use of subregion disables there are only 4 possible block sizes, for example: fsdz*5/8, fsdsz*6/8, fsdz*7/8, and fsdz*8/8. When a block overflows, go to the next size and the SRD macro will automatically generate the correct SRD pattern.
- If available memory is exceeded, use MpuPacker to suggest the most efficient block order to reduce gaps.
- For ARMM7, reducing region sizes reduces alignment requirements and thus results in more efficient memory usage. If an application no longer fits into a memory, it may be necessary to define smaller partitions or more child tasks.
- If converting legacy pcode to umode, it is best to resist unnecessary code changes until the code is running in umode. This way there are fewer changes that could be causing a problem.
- If creating new code, it is best to not optimize the code until it is running in the intended mode, for the same reason as above.
- If converting ptasks to utasks, after a ptask is running, include xapiu.h ahead of its code so that it makes SVC calls instead of direct system calls. This will filter out restricted system calls before switching to umode – one less problem.
- If a task needs to create other tasks and do other privileged operations, start it in pmode, and after it finishes initialization, restart it in umode, as shown in section 4.10.2 From pmode to umode.

- Special versions of sb_INT_DISABLE() and sb_INT_ENABLE() can be enabled to do DEBUGTRAP if used in umode. Set SB_ARMM_DISABLE_TRAP to 1 in XBASE\barmm.h. See section 6.2.2 Interrupt Disabling and Masking in Tasks.
- In a SVC function, for a system service that may suspend or stop on a heap mutex, use an sb_SVCH macro. If the heap is initially busy, but then becomes free before the timeout, these macros try again. See section 4.9.1 SVC Calls.
- IAR ILINK does not automatically order blocks to minimize gaps between them, so it may be necessary to do so manually with the help of MpuPacker and the fixed order keyword. See section 4.4.3 Block Ordering and 8.11 Reducing Memory Waste for ARMM7.
- IAR ILINK does not automatically fill gaps with code and data that is outside of regions, so it is necessary to create plug blocks and locate them in gaps.
- MPU_TYPE.DREGION indicates number of MPU slots. CM3/4 allow 8, CM7 allows 8 or 16.

8.14 Measurements

8.14.1 Size

The SecureSMX functions, if all are used, add about 8KB of code to SMX. The breakdown is: without portals 4KB, tunnel portals 3KB, and free pmsg portals 1KB. These numbers exclude portal shell functions and buffers. SecureSMX adds the sum of all MPA sizes to RAM and the sum of all MPA template sizes to ROM. In addition, it adds the sum of all portal server and client structures to RAM. It increases MCB sizes by 14 bytes and TCB sizes by 28 bytes. The introduction of partitioning and portals will usually increase the number of tasks in a system. For many of the new tasks, it is possible to use one-shot tasks, thus reducing additional memory needed for task stacks.

8.14.2 General Performance

Execution overhead via the SVC exception for typical system services, such as smx_SemTest() and smx_SemSignal() is about 100%. Percent overhead is less for longer system services, since the overhead is a fixed amount of time per system service. Overhead for starting a task is about 25%. This includes loading the task’s MPA into the MPU, and for other MPU conditional code. It is about 12% for task resumption and negligible for task suspension and stopping.

The main overhead for using the MPU is from calls to mp_MPULoad() at each task switch and for making SVC calls. A counter can be added to mp_MPULoad() to see how often it runs. If using smx_sst[] and SMX_CFG_DIAG is 1 in xarmm_iar.inc and xcfg.h, SVC calls are counted in smx_sst_ctr[]. The entries correspond to the enum values in ssndx in svc.c. The counts can be surprisingly high. For a 20MB file write to a thumb drive followed by a 20MB file read, we recorded:

[21]	10	smx_EQCount()
[33]	30926	smx_HeapFree()
[35]	30944	smx_HeapMalloc()
[63]	7	smx_MutexCreate()
[66]	284271	smx_MutexGet()
[68]	284271	smx_MutexRelease()
[87]	14	smx_SemCreate()
[88]	13	smx_SemDelete()
[89]	30895	smx_SemPeek()
[90]	30896	smx_SemSignal()
[91]	226406	smx_SemTest()
[93]	15512	smx_EtimeGet()
[96]	10245	smx_StimeGet()
[99]	3	smx_TaskCreate()
[130]	244	smx_SysPeek()
[141]	61873	smx_HeapAccessGet()
[149]	6	sb_DelayUsec()
[150]	195367	sb_IRQClear()
[151]	3611	sb_IRQMask
[152]	198978	sb_IRQUnmask
Total	1404492	

The mutex, semaphore, and IRQ operations are very high because this was measured on STM32, and the Synopsys USB host controller generates an extreme number of interrupts (many thousands per second) because of its design. The number would be much lower for EHCI and other controllers. The number of heap operations is also surprising and probably could be reduced with minor design changes.

8.14.3 Thumb Drive Performance

Despite the above, performance reduction is not bad:

	read/write		
no MPU	3274/2132 KB/s	100/100%	
no portal	2854/1846 KB/s	87/ 87%	100/100%
FS portal	2714/1790 KB/s	83/ 84%	95/ 97%
FS+USBH portals	2541/1709 KB/s	78/ 80%	89/ 93%

The first column shows what was measured; the second column shows transfer rates; the third column is percentages for portal + MPU; the fourth column shows percentages for MPU only. No MPU means that fsdemo, smxFS, and smxUSBH were running with the MPU turned off. MPU means fsdemo, smxFS, and smxUSBH were running in umode with the MPU turned on. No portal means direct file system and USBH calls due to the fsdemo, smxFS, and smxUSBH being in the same partition. Portal means file system calls via a portal with fsdemo and smxFS in mutually isolated partitions and USBH calls with smxFS and smxUSBH in mutually isolated partitions.

The first line shows a 13% reduction in performance due to turning on the MPU. This includes the SVC calls shown in the previous table. Of this reduction only 2% is due to MPU loading

during task switches. Using the smxFS portal results in another 3-4% performance reduction. In this case, an SD card is being accessed. FS+USB reflects adding a chained portal from smxFS to smxUSBH in order to access a thumb drive. This shows another 4-5% reduction in performance.

Overall, there is a 22% reduction in performance for file reads and a 20% reduction in performance for file writes. As mentioned in the previous section a lot of the overhead is due to the high number of SVC calls due to the particular USB host controller used on STM32. Even so, these are good results considering the large improvement in security due to putting fsdemo (a prototype application) into an isolated umode partition, putting smxFS and smxUSBD each into isolated umode partitions, and communicating between the three partitions via tunnel portals. The three partitions are fully isolated from each other and from all other partitions in the system.

8.14.4 SD Card Performance

Performance has been measured for the smxFS demo with an SD card. The following are KB/sec read and write times:

Direct:	R/W = 7305/3060
No-copy:	R/W = 6845/2935 = 93%/97%
Portal 4096:	R/W = 6780/2940 = 92%/97%
Portal 512:	R/W = 3780/2080 = 51%/69%

Direct is the direct call. The other three are via an smxFS portal. No-copy uses a 4096-byte pbuf as the working buffer in the client. Portal 4096 uses a 4096-byte pbuf, and Portal 512 uses a 512-byte pbuf. The demo continuously writes, reads, and compares a 20MB file. The above measurements are averages over 250 second periods. Read and write performance is very good in both 4096 byte pbuf cases, but much less for a small pbuf. However, in a tight memory system, this performance reduction might be acceptable.

If not, then the direct call API must be used with consequent loss of isolation, or the application (e.g. fsdemo) and the smxFS partitions must be merged. In this case some higher-level API would be presented to the rest of the system via a portal for the combined partition.

8.14.5 ARMM7 Memory Waste

Due to the ARMM7 MPU requirement that regions be power-of-two sizes and that they be aligned on their sizes, memory waste can be substantial. However, in an actual middleware port of about 20,000 lines of code, wasted code space was only 14.5% and wasted data space was an even smaller 2.5%. These numbers are not bad considering the importance of security and reliability as well as the other benefits of isolated partitions. However, other systems may experience much worse results and it may be necessary to work with larger memories during development or to put some code and data into external memory, which may result in performance degradation. See section 8.11.1 Using MpuPacker to minimize waste.

8.15 EWARM Tool Issues

EWARM does a very good job of supporting the features of SecureSMX. However, it was not designed for partitioning, hence there are some issues:

- Extra Options such as section name changes applied to a top node in the project must also be applied to sub nodes that have their Override inherited settings box checked. The “privileged” subfolders in the App project help to deal with this problem.
- *#pragma default_function_attributes* and *#pragma default_variable_attributes* have some limitations and may cause errors locating things. They cannot locate string literals, which are always put into .rodata by the compiler. See 4.5.4 String Literals for solutions to this.
- *const* in a variable definition might be ignored by the compiler, so locating it in an rodata section might be wrong.
- The linker might complain that a memory overflows even though you haven't moved or added any significant code or data. If you temporarily make the memory region bigger by increasing the end address in the mem statement, you can get it to link and then inspect the .map file to try to find out what is wrong. You may see it has reordered things, and alignment requirements may have caused big gaps, for example.
- The map file does not show symbols ordered by address and sections, which makes it difficult to check that things are located properly. MpuMapper fixes this. See 4.6.3 MpuMapper.

Chapter 9 Debugging

This chapter focuses on how to deal with debug problems unique to using partitioning, portals, and other SecureSMX features.

9.1 Using Configuration Constants

Debugging code is more challenging when security features are enabled. For this reason, configuration constants are provided to selectively disable some security features. Once basic security features are working well, the disabled security features can be individually enabled to test the functionality that they control. The following are the security configuration constants that are available:

9.1.1 SMX_CFG_SSMX

When 1, this enables all MPU and security features. When 0, all of these features are disabled, and the code for them is omitted. SMX_CFG_SSMX_ENABLE must also be 1 to turn on the MPU (see below). The following configuration constants are disabled if SMX_CFG_SSMX is off: SMX_CFG_SSMX_ENABLE, SMX_CFG_PORTAL, SMX_CFG_RTLIM, and SMX_CFG_TOKENS.

9.1.2 SMX_CFG_SSMX_ENABLE

When 0, the MPU is not turned on even though SMX_CFG_SSMX is 1. This is useful during debugging if MMFs are interfering with finding problems. It can also be useful to determine if a problem is caused by the MPU, such as interrupts not being disabled in umode. SVC calls are also disabled if this is 0, which makes debugging easier since system calls are direct, and the call stack isn't broken by the exception. However, if the problem goes away and use of SVC calls is needed to reproduce it, they can be easily enabled with the MPU off by a config setting at the USER tag in xapiu.h.

9.1.3 MP_MPA_DEV

Enables names for MPA regions and for static slots. Names are very helpful when looking at MPAs in global and local variable windows and also for use with smxAware. MP_MPA_DEV also selects a slower form of MPU loading that checks for MPA errors. MP_MPA_DEV should be 0 for final code in order to speed up MPU loading on task switches and to save memory by eliminating unnecessary names.

9.1.4 SMX_CFG_PORTAL

Enables use of portals. If 0, portals are disabled and portal code is omitted. When developing a portal, it is convenient for the client and server to initially be in the same partition or to have

common regions such that the client and server can be operated with or without the portal. SMX_CFG_PORTAL can be used to select between these modes. Once the portal code is running properly, the partitions can be isolated from each other.

9.1.5 SMX_CFG_RTLIM

Enables runtime limiting. Allows disabling runtime limiting during debugging to determine if a problem is being caused by it.

9.1.6 SMX_CFG_DIAG

Enables a system service counter to keep track of how many times the service has run. These are a good way to spot imbalances – if a system service is running far more often than other services, there is probably a problem. See section 8.14.2 General Performance.

9.1.7 SMX_CFG_TOKENS

Enables tokens to be used to control access to smx objects.

9.2 Debugging Techniques

9.2.1 Keep a Debug Log

It is helpful to keep a debug log as you go, such as:

```
->”->PMR->PSvr->ut2c: pmsg=5dd0, dp=5d8, pass ->tpA6: rxch->0, sxchg->ut2s, pmsg=5dd0,
fl=5df0,->tpAEnd: ut2s->ssn=0 wrong, Fixed, mpap=548. rok
```

(You are not expected to understand this.) The idea is to keep track of the path followed (e.g. ->”->PMR->PSvr->ut2c), what was observed (e.g. pmsg=5dd0), and what was “fixed” (ut2s->ssn=0 wrong). The nature of this work is that there are frequent task switches due to the introduction of tasks for partitions, portals, child tasks, etc. Each time a task switch occurs, the call stack is lost, so it is easy to get lost. A solution to this is to run to a starting breakpoint, then trace forward using breakpoints or run-to-cursor’s in the subsequent tasks and also to keep a debug log, so you ”remember” where you came from and where you are going.

9.2.2 Buy a Tracing Tool

Another solution is to invest in a tracing tool such as IAR I-jet Trace, which allows you to look at the history of code that executed up to the problem. I-jet Trace is used in place of an I-jet debug probe. It is a more expensive unit with a large memory to store execution history. It requires ETM support on the processor and board. It can be configured to start, end, or center the trace on the trigger condition, such as a breakpoint. It correlates the disassembly trace with the C code. You can navigate forward and backward through the trace and into or over functions. Unfortunately, SVC calls seem to confuse the navigation, so when going backward at the trace level, it will not step over calls that make an SVC call. As a consequence, it is necessary step backward through all the code before the SVC call point of return, even the SVC handler, itself.

9.2.3 Finding MMFs

Although MMFs are theoretically helpful for finding software bugs such as uninitialized pointers, etc., it can be difficult and frustrating to figure out what has caused the latest MMF.

The sections that follow are intended to help you with this process. See also Eliminating MMFs in 7.2.2 pd2.

9.2.4 MMF Storms

When isolating a task, you are likely to experience an *MMF storm*, if not a downright *blizzard*, due to references that are outside of the regions you have assigned to the task. This is normal and is worse for utasks than ptasks because ptasks have the sys_code and sys_data regions, which are large. It may seem that the MMFs will never end. All you can do is to keep plodding along, finding and fixing MMFs. There is an end to this tunnel, so be patient.

9.2.5 Using Debugger Windows

During normal operation, an MMF is directed to the smx_MMF_Handler in xarmm_iar.s. However during debug it is directed to MMF_Stop() in vectors.c. The latter simply stops and does not switch to the main stack, as does the smx_MMF_Handler(). As a consequence, the call stack leading up to the MMF is visible in the call stack window and you can see what function caused it. Double-clicking on that function takes you to the instruction causing the MMF.

In addition, the Fault Exception Viewer (new in C-SPY v8.20) shows what type of violation occurred (data or code) and provides the data address if it was a data violation. It also provides the instruction address causing the MMF. By entering this address into the “Go to” box at the top of the Disassembly window, you can see the actual assembly language instruction that caused the MMF. Generally, looking at the C statement is not sufficient.

If you are working with an older version of C-SPY, open System Control Block (SCB), in the Registers window. The CFSR register in it shows the cause of this fault. (See ARM Application Note 209.) The PC register points to the faulting instruction. The MMFAR register in SCB shows the address of a data violation, if any.

Study the faulting assembly instruction to see what memory it attempts to access and compare to the MPU regions for the current task. The smxAware MPU display in its Text window is very helpful with this because it shows the starting and ending addresses and the attributes for each region.

If you are not using smxAware, the Memory Protection Unit in the debugger’s Registers window allows looking at MPU registers. In order to see a specific slot, enter its number into MPU_RNR. Then RBAR and RASR or RLAR will display the desired region. Clicking on the +’s reveals the fields. Unfortunately, only one region can be seen at a time. With smxAware, the entire MPU or an entire MPA can be seen at once. This makes it much easier to verify that a particular address is not covered.

In some cases, you may find an attribute violation (e.g. trying to write to DATARO or to access PDATARW in umode). This requires a code fix. However, you may have discovered a latent bug, which needed to be fixed anyway.

For more discussion of how to debug MMFs, see Eliminating MMFs in section 7.2.2 pd2.

9.2.6 The Handle Problem

Why might:

```
smx_SemSignal(sema);
```

cause an MMF when

```
smx_SemSignal(semb);
```

does not? It may be that semb is defined in a current task region and sema is not. sema was probably defined in pmode or in another partition. Why does this matter? If you look at the disassembly for SemSignal(sema), you will see something like this:

```
LDR   R0, [PC, 0x830]      ; sema
LDR   R0, [R0]
BL    smx_SemSignal(struct SCB *)
```

Remember that sema is a *handle*. That means it is a memory location that stores the address of the sema control block – i.e. it is a special pointer. The first instruction above loads the address of sema into R0. This is ok, but the second LDR attempts to read this address, which is outside of the MPU regions. That is what causes the MMF.

In order to correct this problem, you must define an *alias handle*, semaa, for sema in a region that the current task can access:

```
SCB_PTR    semaa;
```

Then in code that has access to both handles (probably pcode), you need to load the alias handle after sema has been created:

```
semaa = sema;
```

This may seem so simple, why belabor it? Because this problem is certain to get you sooner or later.

In general, when an object is created in one partition and used in another, the second partition needs an *alias handle.* This is a location in the second partition that stores the pointer to the object's control block. The control block, itself, need not be in the second partition because it is accessed only by smx in pmode, and smx has access to all sys_data. For examples of alias handles see the csem, ssem, and sxchg fields in portal control structures.

9.2.7 Fixing an Easy MMF

Typically, you will find that a function or a variable that the current task is attempting to access is not in the task's MPU regions. This is fixed by adding the function or data to the appropriate region of the task's MPA template. For a pmode task, this often involves adding functions or variables that you did not realize the task needed (e.g. C library calls and BSP functions). For a umode task, it often involves moving functions from sys_code and variables from sys_data to regions accessible by the utask, such as ucom_code and ucom_data

It often occurs that some variables or functions are needed in two regions. But they cannot be in two places at once. The solution to this is to put them into a smaller region, then include the smaller region in the larger region, such as the following:

```
define block cp_data with size = cpdsz*5/8, alignment = cpdsz {block cp_heap,
                                 rw section .cp.data, rw section .cp.bss};
…
define block sys_data with size = sdsz*5/8, alignment = sdsz, fixed order
   {block EVT, block CSTACK, block mheap, block ucom_data, block cp_data,…
```

In this case, the cp_data region is needed by both the cp partition and by ptasks.

9.2.8 Region Overlaps

Region overlaps cause difficult problems to find, because the address of a variable or a function is in an MPU region for the current task. Hence, it seems that nothing is wrong.

For ARMM7, suppose that region 6 overlaps region 5 by one subregion. Then region 6's attributes take precedence over region 5's attributes in that subregion, because region 6 has a higher number than region 5. So if region 6 is read-only and region 5 is read/write, an attempt to write into the overlap area of region 5 will cause an MMF. This is can be a serious problem because it may seldom occur. Suppose, for example, that within the region overlap there are just a few variables in region 5 that are rarely accessed. Then MMFs will seldom occur.

For ARMM8, if two regions overlap, an MMF occurs only when an access is attempted to the overlap area. So the system can be running fine and suddenly have an MMF due to a rare access to a tiny overlap area. In this case attributes do not matter.

Both of the above are potential Achilles heels for systems being shipped, since either problem can easily escape even through testing.

smxAware is the best tool for finding this problem – it flags region overlaps in both its MPU and its MPA displays. You just have to remember to look.

9.2.9 Reversing Course

If umode restrictions are hindering debugging of a task or partition, you may wish to temporarily return the task or partition to pmode. Or maybe the design has changed, and you want to permanently return it to pmode.

To return a utask to operation as a ptask, remove the SMX_FL_UMODE flag in smx_TaskCreate() or comment out:

```
smx_TaskSet(utask, SMX_ST_UMODE, 1);
```

if it is being used and comment out #include "xapiu.h" in its module(s). Also, go back to the MPA template that was being used in pmode, which includes the sys_code and sys_data regions.

Alternatively, try setting SMX_CFG_SSMX_ENABLE to 0 temporarily so the MPU is not enabled and direct system calls are made instead of SVC, so the call stack isn't broken. See section 9.1.2 SMX_CFG_SSMX_ENABLE

9.2.10 Portal Debugging

Loss of the call stack due to isolating the client and server partitions causes a problem when debugging portals. The problem is that while tracing operation in the server it is not possible to see the origin of the operation in the client. To avoid this problem, we recommend that during the initial debugging of a new portal to combine the client and server partitions into a single partition or to use common regions between them. Then, by disabling or enabling the map header file, e.g. #include “pmap.h” and remaking the project, it is possible to quickly switch between direct calls or portal calls.

Once the portal is operating correctly, the client and server partitions can be separated. This will usually result in a host of MMFs to be tracked down and fixed. When that has been done, separation is complete.

After the client and server partitions have been separated, there may be times when it is necessary to be able to go back into the client and trace through to the server. To do this, run to the point of interest in the server, and then use the *caller field* in the service header of the pmsg. (To get this, if you are stopped in the function called via the portal, e.g. some_func(), then in the call stack window, double click on the server function (which should be one below some_func), and in it add shp to the watch window and expand it to see the caller. If it’s not named shp, it is whatever points to ret, p1, p2, etc in the call, e.g. xxx->p1.) Enter this address into the disassembly window. What you should see is an instruction like:

```
BL      some_func(int, …
```

at the cursor. Run to the line after it for the call stack window to show the calls leading up to it. Or put a breakpoint on this line and run from the beginning. When the debugger stops on this breakpoint, you can then step into the portal call to see what is loaded into the service header, then continue on into the server to find what is wrong. While in the client, enter smx_ct in the watch window to see what the current task is. Alternatively, the current task can be determined at the server side by opening the smxAware GAT Event Timelines, clicking the +>| (Zoom in Max) button, and finding the last task to run using the portal other than the server task.

When deploying a new driver for a middleware partition, it often happens that certain hardware “features” must be programmed around. If these features have time dependencies debugging through a portal is likely to be too clumsy. In such cases restoring the direct call API becomes necessary. SMX middleware portals can be disabled using the settings in XSMX\xpcfg.h, such as:

```
#define SFS_PORTAL      0  /* enable smxFS portal */
```

and then remaking the project. This excludes the mapping header file so that direct calls are made, and it removes code related to creating and using the portal. Note that it is also necessary to merge the client and server partitions or to use common regions between them.

9.3 Using smxAware Security Features

The following highlight smxAware features used to help debug SecureSMX-based software. For full information on smxAware, see the smxAware User's Guide.

The first three displays discussed below are in the smxAware Text window.

9.3.1 MPU Display

This displays the MPU for the current task. At the top, MPU ON/OFF, BR ON/OFF, privileged or unprivileged mode, and MSP (main stack) or PSP (task stack) are shown. Following this, active region overlaps and adjacent regions are shown. Both can cause problems. As a project progresses, the ordering of regions is likely to change and require careful attention. Next is display of the MPU regions. The following is a partial ARMM7 MPU display (ARMM8 is similar):

```
Current Task:   (fs_reader_writer0)    0x20013420
MPU  ON      BR ON      CPU:  UNPRIV MODE   PSP
MPU[0]  Enabled   rbar 08030000  rasr 0602e017  "fsdp_code"
    Start               08030000
    End                 08030fff
    Subreg Dis          5,6,7 (Size 0x200)
    Sub Start           08030000 (Size 0xa00)
        End             080309ff
    Attributes          CODE
MPU[1]  Enabled   rbar 20034001  rasr 1302e019  "fsdp_data"
    Start               20034000
    End                 20035fff
    Subreg Dis          5,6,7 (Size 0x400)
    Sub Start           20034000 (Size 0x1400)
        End             200353ff
    Attributes          DATARW
MPU[2]  Enabled   rbar 0804a002  rasr 0602c019  "ucom_code"
    Start               0804a000
    End                 0804bfff
    Subreg Dis          6,7 (Size 0x400)
    Sub Start           0804a000 (Size 0x1800)
        End             0804b7ff
    Attributes          CODE
...
MPU[7]  Enabled   rbar 2000a007  rasr 1302c115  "stack"
    Start               2000a000
    End                 2000a7ff
    Subreg Dis          0,6,7 (Size 0x100)
    Sub Start           2000a100 (Size 0x500)
        End             2000a5ff
    Attributes          DATARW
```

When MP_MPA_DEV is 1, region names are displayed, as shown above. These names are not actually in the MPU; they are in the current MPA.

9.3.2 MPA Displays

Each displays the selected task's MPA. The current task name is in parentheses. Its MPA should be the same as shown in the MPU display. The following is a partial ARMM7 MPA display (ARMM8 is similar):

```
    MPU Regions for this task
MPA[0]/MPU[0]  rbar 08030010  rasr 0602e017  "fsdp_code"
    Start               08030000
    End                 08030fff
    Subreg Dis          5,6,7 (Size 0x200)
    Sub Start           08030000 (Size 0xa00)
        End             080309ff
    Attributes          CODE
MPA[1]/MPU[1]  rbar 20034011  rasr 1302e019  "fsdp_data"
    Start               20034000
    End                 20035fff
    Subreg Dis          5,6,7 (Size 0x400)
    Sub Start           20034000 (Size 0x1400)
        End             200353ff
    Attributes          DATARW
MPA[2]/MPU[2]  rbar 0804a012  rasr 0602c019  "ucom_code"
    Start               0804a000
    End                 0804bfff
    Subreg Dis          6,7 (Size 0x400)
    Sub Start           0804a000 (Size 0x1800)
        End             0804b7ff
    Attributes          CODE
...
MPA[7]/MPU[7]  rbar 2000a017  rasr 1302c115  "stack"
    Start               2000a000
    End                 2000a7ff
    Subreg Dis          0,6,7 (Size 0x100)
    Sub Start           2000a100 (Size 0x500)
        End             2000a5ff
    Attributes          DATARW
```

The display is similar to the MPU display, except each region's MPA/MPU slot number is indicated, which will be different if the MPU has static slots. In that case, the static slots are displayed at the end of the MPA after a separation line.

9.3.3 Tasks Display

The Task display shows task information and its full TCB, including additional fields for SecureSMX. Clicking on the + for flags, reveals all task flags.

9.3.4 Memory Map Window

This window is helpful for debugging SecureSMX-based software. It displays a graphical memory map overview showing MPU regions in memory bars. It allows zooming in and out on any memory area to reveal what it is used for (e.g. stack t2s, free heap, MPU[1] sys_code) and if it is pmode or umode. In addition, region overlaps are flagged. A Detail window shows function,

starting address, ending address, and size. This window is helpful to visualize how memory is being used.

9.3.5 Portal Events

Portal calls are displayed in the Event Buffer in pink. The portal name is shown for the pch and psh parameters to make it easier to associate the calls made for a particular portal when studying the buffer. This results from the PortalCreate() and PortalOpen() functions both adding entries to the handle table for psh and pch, respectively. (The name will only show for pch while the portal is open since PortalClose() removes it from HT, and it will only show for psh while the portal exists since PortalDelete removes it from HT.) Studying the event buffer is very helpful to see how the portal works and what system calls it uses.

The Event Timelines graph shows portal operations as small pink rectangles within task bars. This helps to visualize portal activity. Putting the Detail cursor on a rectangle shows the portal function and its parameters.

9.4 Multitasking Issues

Converting to SecureSMX is likely to add more tasks to a system, due to partitioning and portals. Thus multitasking debug skills become even more important. A basic problem is that one cannot step from one task to another in the way that one steps from one function to another. It is necessary to anticipate where control will go, then set a breakpoint or run-to-cursor at that point. However, other tasks may intervene between the start and end points due to interrupts and preemptions. Even worse, the end point may have been reached by a different path than expected. This can be very confusing.

The smxAware Graphical Analysis Tool (select “Graph” on the smxAware pull-down menu) enables you to see what actually happened and thus clarify why things are not as expected. Using it helps to see where you can take actions so that the sequence of events goes as expected. Then you can step through the target task and fix the problems you are after.

Sometimes it necessary take small steps rather than big leaps. It often is helpful to put a breakpoint in the scheduler just past where smx_ctnew is started to see if it is the task you expect to run next. Don’t put a breakpoint in code used by multiple tasks. Instead put it at the place in the task code that calls that code.

9.5 Pay Attention to Errors

All SMX modules do extensive error monitoring and reporting. To benefit from this, keep a Watch window open showing smx_errno, sb_errno, as well as key variables such as smx_sched, smx_srnest, smx_ct, all tasks of interest to you, etc. When an error occurs that stops the system, look at the errno’s to see what happened. It also helps to have a terminal connected to the console output UART in your system to display error messages.

For example, all return values, especially from system calls should be tested before use. However, few programmers do this, so in the following case:

```
ut2a = smx_TaskCreate(tm06_ut2a, TP2, 300, 0, "ut2a");
smx_TaskStart(ut2a);
```

If smx_TaskCreate() fails, ut2a will be NULL. Then smx_TaskStart(ut2a) will abort, report SMXE_INV_TCB and branch to smx_EM(), which will output "smx INV TCB" to the console. If you are paying attention to the watch window or terminal, you will know right away that task ut2a is not running, and that is why your breakpoint in it is not triggering. Otherwise you will probably waste time trying to figure out what is wrong with task ut2a or some other task, not realizing that ut2a is not even running!

9.6 Debug Tips

Most of these are covered elsewhere, but it helps to have a brief list of what can go wrong when debugging:

- MMF first things to check using debugger register window and smxAware MPU display: Ensure the instruction pointer is in a code region of the MPU, a data reference is in a data region of the MPU, and the stack pointer is in the stack region of the MPU.
- MMF for ARMM7 can occur even though an address is in an MPU region, if attributes are wrong (e.g. PCODE from umode code) or if the region start is not properly aligned.
- MMF for ARMM8 when an access is made to an area of overlapping regions. A region overlap does not cause an MMF until access to an address in the overlap area is attempted. Hence the MMF appears to be due to an out-of-bounds access. However, the smxAware MPU window will indicate that there is a region overlap, which is the true cause of the MMF. Region overlap is most likely to occur if regions are defined with hard addresses rather than by the linker. Otherwise overlapping regions can occur due to loading dynamic regions such as stacks and pmsgs.
- MMF for ARMM8 when a region is accessed that is not a multiple of 32. To avoid this make sure that all size definitions in the linker command file are in hexadecimal and end in 0x00, 20, 40, 60, 80, A0, C0, or E0.
- MMF in C library code: A difference in optimization level (such as in Debug and Release targets) may cause the linker to use a different version of a C library function, such as memcpy. Then the object file for this version must be added to the appropriate region in the linker command file.
- MMF on task switch: The task stack may be outside of MPU regions. In this case, even though the task main code is in an MPU region, it cannot be reached.
- MMF with BR ON: Although BR ON allows accessing all installed memory, MPU region attributes prevail over default attributes.
- MMFs too distracting: MPU operation can be temporarily disabled by setting SMX_CFG_SSMX_ENABLE to 0 in xcfg.h and in xarmm_iar.inc and rebuilding everything. See section 9.1.2 SMX_CFG_SSMX_ENABLE.
- Hard Fault: Occurs due to calling an SVC instruction with interrupts disabled. Check that PRIMASK == 0 and that BASEPRI == 0.
- Hard Fault: Occurs due to an SVC from an SVC or an SVC from an IRQ since the SVC must run immediately, but it can only preempt lower priority exceptions.

- Hard Fault: ISRs must use direct system calls not SVC system calls because they have higher priority than the SVC Handler.
- Hard Fault: Occurs if an MMF occurs in an exception handler with the same or lower priority. This is called *fault escalation.* Note that sb_IntCtrlInit() sets priorities of MMF, BF, and UF to 0, which is the highest configurable priority, so an MMF occurring in BF or UF would escalate to HF. Likewise, exceptions with effective negative priorities are higher than 0, so an MMF in one of them would also escalate to HF. Effective priorities: Reset (-3), NMI (-2), or HF (-1). User exceptions should have lower priorities (> 0), so an MMF in one will not escalate.
- When invoked from the SVC_Handler(), the PendSV_Handler() cannot run until the SVC_Handler() exits, because it is lower priority.
- Critical sections cannot be protected in umode by disabling interrupts. See section 8.10.7 Critical Sections for more discussion of this.
- To determine if a system call is a SVC call rather than a direct call, look at it in the disassembly window to see if its prefix ends with u (e.g. smxu_ rather than smx_).
- Template change: If a region is being added or removed be sure to change the tmsk in mp_MPACreate()'s using the template. Failing to do so can cause bizzare behavior.
- CONTROL register is not valid in handler mode following an exception or interrupt. It only indicates the state of the interrupted utask or ptask in thread mode.
- Build change: If switching from one build target to another causes a problem, e.g. Debug to Release, check each overridden file (indicated with a check mark in the Workspace window) for each project to ensure that the Extra Options settings are the same.
- Portals: If a problem occurs in common code, use the call stack to see what portal it is. The portal's main function is at the bottom of the call stack. If it is a chained portal, look at the *caller* field in the service header (see section 9.2.10 Portal Debugging) and run back to that address where the call was made in the first portal's server, and then look at the call stack.
- Portals: If pmsg->pri is greater than the client priority, the server will preempt and the client will not wait at rxchg or csem. This can be confusing when debugging client/server operation.
- Portals: If pmsg->pri is equal to or less than the client priority, the server will wait for the client and possibly other tasks to finish running. In this case, a very long timeout for the client wait on csem may be necessary. However, do not use INF because then the client cannot recover if the server disconnects from it.
- Portals: If pmsg->pri is equal to or less than the client priority be sure to specify a timeout > 0 for mp_TPortalReceive() and mpTPortalSend(), else the server cannot run to perform the intended operation. This is particularly a problem with sending the OPEN command.
- Memset: Breaking on memset() is confusing. It is better to break after memset().
- Reset: If the application runs from the debugger but not from reset, it probably means that the reset vector (initial SP and IP) is not at the beginning of internal flash, as it must be. See section 8.10.5 Reset Vector.

- smx Control Block Fields: These cannot be accessed from umode – use the object's Peek() instead.
- smx Object Handles: If a handle in a service call is not in range, an MMF will occur. To avoid this either define the handle to be in range or create an alias handle that is in range and copy the handle contents into it.
- A variable is not being put where expected. Make sure that the section is included in the region block in the linker command file. For example, if tsx is put into .ut2a.bss, but .ut2a.bss is not put into ut2a_data, then accessing tsx in ut2a will cause an MMF.

9.7 C-SPY Tool Issues

- In a SVC system call, the call stack will not show the functions before the SVC handler. In order to see how you arrived at the call, return back to the SVC handler and step through its tail over the POP {PC} where it returns to the shell. (It is in all caps so it can be found easily.) Then the functions leading up to the SVC call will appear in the call stack.
- If there is a problem looking at MPU registers in C-SPY or smxAware after a fault occurs, put a breakpoint at the failure point, restart, and run to it.
- Stepping with the debugger in umode may cause the MPU it to run differently than when it is free running. Stepping over an instruction that should cause an MMF may not cause the MMF. If stepping in the disassembly window, the debugger refuses to step over the instruction causing the MMF. If behaviors like these occur, try running to a point past the problem, then observe if correct operation has occurred.
- Stepping over setting MPU_CTRL.PRIVDEFENA causes the change not to show in the register, but it does seem to take effect.
- ETM Trace: Unfortunately, SVC calls seem to confuse the trace navigation, so when stepping backward at the trace level, it will not step over calls that make an SVC call, requiring you to step backward through all the lower code, even the SVC handler.

Appendix A.1 SecureSMX Services

Except for portal client services all services in this section of Appendix A should be executed only in pmode or hmode. These services are not task-safe and must be protected against reentrancy due to task preemption.

mp_FPortalClose

bool mp_FPortalClose(FPCS* fpch, u8 xsn)

Parameters fpch fportal client handle
xsn Available slot in the MPA of task deleting portal.

Returns true fportal closed.
false fportal not closed.

Errors See smx_PMsgReceive(), smx_PMsgRel(), and smx_MsgXchgDelete().

Descr Called by a client to close a free message portal. Releases the number of pmsgs obtained by it when it was opened. Waits up to fpch->tmo ticks for each pmsg. If a timeout occurs, aborts and returns false. In this case, fpch->num is the number of messages remaining to be released. If all pmsgs are released, deletes rxchg and clears free message portal client structure fields, except portal name and handles. At this point, fpch->rxchg = NULL. fpch->pname and fpch->sxchg are cleared only if the portal is deleted, and fpch->pmsg is cleared only if pmsg is released.

Can be called by a task deleting a portal via mp_FPortalDelete() – see discussion there.

Example

```
FPCS*  fpch;
bool pass;

pass =sb_FPortalClose(fpch);
if(!pass)
{
    if (fpch->num > 0)
    {
        Delay(100);
        pass = sb_FPortalClose(fpch);
    }
}
```

The above closes the client portal pointed to by fpch. If close fails and some pmsgs have not been released, it tries again after a 100 tick delay.

mp_FPortalCreate

```
bool   mp_FPortalCreate(FPSS* fpsh, FPCS** pclp, u32 pclsz, u8 ssn, const char* pname=NULL,
                                                                    const char* sxname=NULL)
```

Parameters

fpsh	fportal server handle.
pclp	Permitted client handle list.
pclsz	Permitted client handle list size (number of clients).
pname	fportal name.
ssn	Server MPU slot number for pmsg region.
sxname	Server exchange name.

Returns

true	fportal created.
false	fportal not created.

Errors See smx_MsgXchgCreate()

Descr Creates sxchg, loads pname and sxchg handle into the free message portal client structures listed in pcl and into the portal server structure. Loads ssn into the portal server structure. Notes:

1. Must be called from pmode.
2. Portals are owned by servers.

Example

```
#pragma default_variable_attributes = @ ".ut2A.bss"
 FPSS    fpssa;
 TCP* ut2A;
#pragma default_variable_attributes = @ ".ut2A.data"
FPSS*   fpssah = &fpssa;
extern FPCS  fpcs1, fpcs2;
FPCS*   fpcla[] = {&fpcs1, &fpcs2};
 #pragma default_variable_attributes =

 ut2A = smx_TaskCreate(tpF1_ut2A, TP2, TS_SSZ, SMX_FL_UMODE, "ut2A");
 mp_MPACreate(ut2A, (MPA*)&mpa_tmplt_ut2A, 0x3F);
 fpssah->stask = ut2A;
 mp_FPortalCreate(fpssah, fpcla, sizeof(fpcla)/4, SV_SLOT, "portalA", "sxchgA");
 smx_TaskStart(ut2A);
```

fpssa, fpssah, and fpcla[] are defined for portal A in ut2A regions. fpcs1 and fpcs2 clients are permitted access to portal A. The portal task, ut2A, is created first and its handle is loaded into fpssa, then mp_FPortalCreate() is called to create the portal and task ut2A is started. The above normally occurs during partition initialization.

mp_FPortalDelete

bool mp_FPortalDelete(FPSS* fpsh, FPCS** pclp, u32 pclsz, u8 xsn)

Parameters

fpsh	fportal server handle
pclp	Permitted client handle list.
pclsz	Permitted client handle list size (number of clients).
xsn	Available slot in the MPA of the calling task.

Returns

true	fportal deleted.
false	fportal not deleted.

Errors See smx_TaskStop() and smx_MsgXchgDelete()

Descr Stops the portal server task, deletes sxchg, sets pname = "no portal" and sxchg = NULL in the portal client structures listed in pcl and in the portal server structure. Closes portal in all clients listed in pcl. Clears portal server structure after stask. At this point only stask is a valid field. Notes:

1. Must be called from pmode.
2. fportal cannot be reopened by a client.

Example

```
sb_FPortalDelete(&pssa, fpcla, sizeof(fpcla)/4, xsn);
```

Reverses portal creation in the mp_FPortalCreate() example.

mp_FPortalOpen

bool mp_FPortalOpen(FPCS* fpch, u8 csn, u32 msz, u32 nmsg, u8 pri, u32 tmo=0,
const char* rxname=NULL)

Parameters

fpch	fportal client handle.
csn	Client task's MPU slot for pmsg region.
msz	pmsg data block size.
nmsg	Number of pmsg's to create.
pri	Priority to send pmsgs.
tmo	rxchg timeout (ticks).
rxname	rxchg name.

Returns

true	fportal opened.
false	fportal not opened.

Errors PORTAL_NEXIST fportal does not exist.
See also smx_MsgXchgCreate(), smx_PMsgGetHeap(), and smx_PMsgSend().

Descr Opens a free message portal for a client. If nmsg > 0, creates rxchg, gets up to nmsg pmsgs from main heap, with msz block size, and sends them to rxchg. If nmsg == 0, rxchg must be created and assigned to the FPCS rxchg field, externally to this

function. Initializes FPCS fields, except name and sxchg, which were already set by mp_FPortalCreate(). Aborts if sxchg for the portal does not exist, reports PORTAL NEXIST, and returns false. Allows re-opening a portal closed by the client.

Example

```
FPCS  fpcsA_xxx;

sb_FPortalOpen(&fpcsA_xxx, CL_SLOT, MSIZE, 1, PRI, TMO, "rxchg_xxx")
```

Opens fportal A for client xxx with one pmsg of MSIZE bytes, and stores pmsg at rxchg_xxx.

mp_FPortalReceive

```
MCB*  mp_FPortalReceive(FPCS* fpch, u8** dpp=NULL)
```

Parameters

	fpch	fportal client handle.
	dpp	Data pointer.

Returns

pmsg	pmsg received.
NULL	pmsg not received.

Errors PORTAL_NOPEN fportal not open.
See also smx_PMsgReceive().

Descr Called by client to receive a pmsg from fpch->rxchg. If the portal is not open, reports PORTAL_NOPEN, sets fpch->errnum, and returns NULL. If a pmsg is not available, at fpch->rxchg waits for up to fpch->tmo ticks, then returns NULL if a timeout occurs. If a pmsg is received, loads the address of its data block into *dpp, unless dpp == NULL, loads the pmsg data region into the current task's MPA[fpch->csn] and into MPU[fpch->csn+fas], and returns the pmsg handle.

Example

```
u8*     dp;
FPCS    fpch;
MCB*    pmsg;

pmsg = mp_FPortalReceive(fpch, &dp);
```

Receives pmsg from fpch->rxchg and loads the address of its data block into dp.

mp_FPortalSend

```
bool   mp_FPortalSend(FPCS* fpch, MCB_PTR pmsg)
```

Parameters

	fpch	fportal client handle.
	pmsg	pmsg to send.

Returns

true	pmsg sent.
false	pmsg not sent.

Errors

PORTAL_NOPEN fportal not open.
See also smx_PMsgSend().

Descr Called by a client to send pmsg from the client to the server waiting at fpch->sxchg. If the portal is not open, reports PORTAL_NOPEN, sets fpch->errnum, and returns false. Otherwise, loads pmsg->rasr/rlar from MPA[fpch->csn], sets pmsg->pri = fpch->pri, sets pmsg->con.bnd = false, and if fpch->rxchg exists, loads its index into pmsg->rpx. Clears MPU[fpch->csn+fas] if it is an active slot. Clears fpch->pmsg if it equals pmsg.

Example 1

```
u8*     dp = NULL;
FPCS    fpch;
MCB*    pmsg;

pmsg = mp_FPortalReceive(fpch, &dp);
mp_FPortalSend(fpch, pmsg);
```

Gets pmsg from fpch->rxchg and sends it to fpch->sxchg.

Example 2

```
u8*     dp = NULL;
FPCS    fpch;
MCB*    pmsg

pmsg = smx_PMsgGetHeap(fpsize, &dp, fpch->csn, DATARW, 0);
mp_FPortalSend(fpch, pmsg);
```

Gets pmsg from heap 0 and sends it to fpch->sxchg.

mp_FTPortalSend

```
bool   mp_FTPortalSend(FPCS* fpch, u8* bp, MCB_PTR pmsg)
```

Parameters

fpch	fportal client handle.
bp	pmsg block pointer.
pmsg	pmsg to send.

Returns

true	pmsg sent.
false	pmsg not sent.

Errors

PORTAL_NOPEN fportal not open.
See also smx_PMsgSend().

Descr Operates the same as mp_FTPortalSend(), except that it is used to send a free message to a tunnel portal. In order to do so, this function fills in the type and cmd fields in the standard tunnel portal message header, as follows:

```
mhp->type = FREEMSG;
mhp->cmd = CONTROL;
```

This results in bypassing the normal tunnel protocol code in mp_TPortalServer() to mp_TPortalCallServerFunc().

Example

```
u8*     dp;
FPCS    fpch;
MCB*    pmsg;

pmsg = mp_FPortalReceive(fpch, &dp);
mp_FTPortalSend(fpch, dp, pmsg);
```

Gets a pmsg from fpch->rxchg and sends it to fpch->sxchg. dp points to the start of the pmsg data block.

mp_MPACreate

```
bool   mp_MPACreate(TCB_PTR task, MPA* tmp, u32 tmsk=MP_TMSK_DFLT, u32
                    mpasz=MP_MPU_ACTVSZ)
```

Parameters

task	Task for MPA
tmp	Template pointer.
tmsk	Template mask.
mpasz	MPA size (slots)

Returns

true	Template created and loaded.
false	Template not created.

Errors

SBE_INV_PAR	Invalid parameter
SBE_PRIV_VIOL	utask tried to change its own MPA
SMXE_INV_TCB	Invalid TCB

See also smx_HeapMalloc() and smx_HeapFree()

Descr Allocates a large enough MPA for regions in the template that are selected by 1 bits in tmsk. Must be at least large enough for MP_MPU_ACTVSZ slots. For a ptask, uses the template selected by tmp, but for a utask uses parent's template. Loads template regions into MPA that are selected by 1's in tmsk. For ARMM7, active slots must contain MPU slot numbers not MPA slot numbers. If tmsk 1 bits run out, remaining active slots are filled with (V+i, 0), where i is the MPU slot number. For ARMM8 there is no slot number nor valid flag and if tmsk 1 bits run out, remaining active slots are filled with (0, 0). Saves mp, msz, and tp in task's TCB. If task already has an MPA, its stack rbar and rasr is saved in task->rv and task->sv, respectively, and the MPA is freed. Then the new MPA is allocated and loaded. Freeing the old MPA is necessary because new MPA size may be different. The new MPA may have additional *auxiliary slots* above the active slots.

Example

```
t2a = smx_TaskCreate(tm15_t2a, TP2, 0, 0, "t2a");
smx_MPACreate(t2a, &mpa_tmplt_t2a, 0x3F, 6);
smx_TaskStart(t2a);
```

As shown here, MPACreate() is typically called immediately after a task is created and before it is started. In this case the active area is 6 slots and there are no auxiliary slots, so mpa_tmplt_t2a has 6 regions, and the mask loads them sequentially starting at MPA[0].

mp_MPACreateLSR

```
bool   mp_MPACreateLSR(LCB_PTR lsr, MPA* tmp, u32 tmsk=MP_TMSK_DFLT, u32
                mpasz=MP_MPU_ACTVSZ)
```

Parameters

task	LSR for MPA
tmp	Template pointer.
tmsk	Template mask.
mpasz	MPA size

Returns

true	Template created and loaded.
false	Template not created.

Errors

SBE_INV_PAR	Invalid parameter.
SMXE_INV_LCB	Invalid LCB

See also smx_HeapMalloc() and smx_HeapFree()

Descr Creates an MPA for a safe LSR. Allocates a large enough MPA for regions in the template that are selected by 1 bits in tmsk 1. Loads template regions into MPA that are selected by 1's in tmsk. For ARMM7, active slots must contain MPU slot numbers not MPA slot numbers. If tmsk 1 bits run out, remaining active slots are filled with (V+i, 0), where i is the MPU slot number. For ARMM8 there is no slot number nor valid flag and if tmsk 1 bits run out, remaining active slots are filled with (0, 0). Saves MPA pointer, mpasz, and stack memory region in LCB. If LSR already has an MPA, it is freed. Then the new MPA is allocated and loaded. Freeing the old MPA is necessary because the new MPA size may be different. The new MPA may have additional *auxiliary slots* above the active slots.

Example

```
lsra = smx_LSRCreate(tsch07_lsra, SMX_FL_TRUST, NULL, 100, "lsra", &lsra);
mp_MPACreateLSR(lsra, (MPA*)&mpa_tmplt_t2a, 0x3F, 6);
smx_LSRInvoke(lsra, 1);
```

As shown here, MPACreateLSR() is typically called immediately after a safe LSR is created and before the LSR is invoked. In this case, the active area is 6 slots and there are no auxiliary slots, so mpa_tmplt_t2a has 6 regions and the mask loads them sequentially starting at MPA[0]. Note that lsra is using the t2a MPA template. This is commonly done for safe LSRs so that they can share code and data with a task in the same partition. However, the LSR might have an IO slot that no task has.

mp_MPUSlotLoad

bool mp_MPUSlotLoad(u8 dn, u32* rp)

Parameters dn destination MPU slot number.
rp region pointer.

Returns true Slot loaded.
false Slot not loaded.

Errors SBE_INV_PAR Invalid parameter.

Descr Loads MPU[dn] slot from a template region pointed to by rp. Use in pmode only. This is intended to be used in MPU initialization code to load static MPU slots (see section 4.2.4 Static Slots).

Example

```
mp_MPUSlotLoad(0, (u32*)&mpa_tmplt_sys[1]);  /* MPU[1] = sys_code */
mp_MPUSlotLoad(1, (u32*)&mpa_tmplt_sys[0]);  /* MPU[0] = sys_data */
```

The above shows loading sys_code into MPU[0] and sys_data into MPU[1] static slots. Static slots do not change on task switches. Hence these regions will be available at all times. However, they are privileged, so utasks cannot access them.

mp_MPASlotMove

bool mp_MPASlotMove(u8 dn, u8 sn)

Parameters dn destination MPA slot number.
sn source MPA slot number.

Returns true Slot moved.
false Slot not moved.

Errors SBE_INV_PAR Invalid parameter (dn or sn > MPA size).

Descr Moves contents of MPA[sn] to MPA[dn] and to MPU[dn+fas], if dn < MP_MPU_ACTVSZ, where fas is the first active slot in the MPU (see section 4.2.3 Active Slots). Normally used to move an auxiliary region that is above the active slots to a shared slot (see section 4.2.5 Auxiliary Slots). Also can be used to save an active region in an auxiliary region, then restore it later. Since MPA[dn] is also loaded, a task switch will not cause loss of region. This function can be called in umode via the SVC exception or it can be called directly in pmode.

Example

```
#define IO    4
#define IO2   7
 #define IO1  8
```

```
/* access IO port 1 */
if (mp_MPASlotMove(IO, IO1)
…
/* access IO port 2 */
if (mp_MPASlotMove(IO, IO2)
…
```

The above shows moving the needed IO region into the common IO slot prior to accessing that region. This is a good way to access multiple IO regions with a single MPU IO slot.

mp_MPUSlotSwap

```
bool   mp_MPUSlotSwap(u8 dn, u32* rp)
```

Parameters

dn	destination MPU slot number.
rp	region pointer.

Returns

true	Slot swapped.
false	Slot not swapped.

Errors none

Descr Swaps content of region at rp with MPU[dn].
Notes:

1. For use in pmode, only.
2. dn and *rp must be correct. They are not tested.
3. For use in ISRs, but not interrupt safe.
4. Not for tasks, since MPA is not changed, and would be lost on task switch.

Example

```
MPR iad = RGN(1 | RA("intA_data")  | V, PDATARW | SRD("intA_data") | RSIC(sdsz)
                        | EN, "intA_data");   /* macros in mpatmplt.h; see note below */

void  ISRA(void)
{
    smx_ISR_ENTER();
    mp_MPUSlotSwap(1, (u32*)&iad);
    …
    mp_MPUSlotSwap(1, (u32*)&iad;
    smx_ISR_EXIT();
}
```

The above assumes sys_code is in the single static MPU[0] slot, and sys_data is in MPU[1] if this is an smx ISR (using smx_ISR_ENTER/EXIT() as shown). This permits BR to be off in utasks as well as ptasks. Then iad is swapped into MPU[1] and MPU[1] into iad. iad is a very small data region for ISRA only. This helps to harden ISRA against hacking.

Note that the definition of iad should be in mpa.c since it uses the same macros used by MPA definitions. Due to the ultra-short names of these macros, it is not recommended to include mpatmplt.h in any other files.

mp_TPortalCall

bool mp_TPortalCall(TPCS* tpch, u32 tmo=0)

Parameters

tpch	tportal client handle.
tmo	csem timeout (ticks).

Returns

true	Command sent.
false	Command not sent.

Errors

PORTAL_CLOSED	tportal closed.
CLIENT_TMO	csem timed out.

See also smx_SemSignal() and smx_SemTest().

Descr Called by client to send a command from client to server. This is a macro that calls mp_TPortalSend(). It is used to make an API call that doesn't transfer data blocks.

Example 1

```
SFSP_SH* shp = (SFSP_SH*)tpch->shp;
mp_SHL1(SFS_ID_FCLOSE, (u32)filehandle, SB_FAIL);
mp_TPortalCall(tpch, SFSP_CTMO);
```

This is an example of sending a command, from the client shell for sfs_fclose(). mp_SHL1() is one of several macros defined in portl.h to load a service header. This macro loads the fid and filehandle into the SFSP service header, and it preloads the ret field with SB_FAIL, in case the portal operation fails. The return value is the most appropriate error value for the particular API call.

Example 2

```
SFSP_SH* shp = (SFSP_SH*)tpch->shp;
char* mp1p = (char*)tpch->mdp;
char* mp2p = mp1p + strlen(filename)+1;
strcpy(mp1p, filename);
strcpy(mp2p, mode);
mp_SHL2(SFS_ID_FOPEN, (u32)mp1p, (u32)mp2p, NULL);
mp_TPortalCall(tpch, SFSP_CTMO);
```

This is an example of sending a command that has some data values to pass. This is from sfsp_fopen() client shell. It copies the filename and mode strings into the pbuf, and loads pointers to each into service header p1 and p2. The handler in the server passes these as p1 and p2 to the actual sfs_fopen() function.

Example 3

```
SUP_SH* shp = (SUP_SH*)tpch->shp;
mp_SHL3(SU_ID_FTDI_GET_STATUS, iID, 0, 0, -1);
mp_TPortalCall(tpch, SUP_FTDI_CTMO);
if (shp->ret != -1)
{
  *pModemStatus = *(u8*)(tpch->mdp);
  *pLineStatus = *((u8*)(tpch->mdp)+1);
}
```

This is an example of sending a command that gets data values via parameters. The two bytes being returned are written to the first two bytes of the portal buffer by the server handler, and they are retrieved above. Special handling is needed for par2 and par3 because they are pointers to locations to store returned data. The function being called is:

```
int sup_FTDIGetStatus(uint iID, u8 *pModemStatus, u8 *pLineStatus, TPCS* pch);
```

mp_TPortalClose

```
bool   mp_TPortalClose(TPCS* tpch, u32 tmo)
```

Parameters

tpch	tportal client handle
tmo	Timeout (ticks).

Returns

true	tportal closed.
false	tportal not closed.

Errors See smx_SemDelete(), smx_SemSignal(), and smx_SemTest().

Descr Called by a client. Returns true if tportal is already closed. Otherwise, sends CLOSE command to server and waits for ack. Continues if ack received or a timeout occurs. Clears pmsg hdr and client structure starting at shp, and deletes csem and ssem. Returns true unless a semaphore fails to delete.

Example

```
TPCS*  tpch;

if (!sb_TPortalClose(tpch, CTMO))
    tfailu();
```

Sends a CLOSE command to the server and waits CTMO ticks for acknowledgement from the server. Then closes the portal from the client side.

mp_TPortalCreate

```
bool   mp_TPortalCreate(TPSS* tpsh, TPCS** pclp, u32 pclsz, u8 dsn, const char* pname=NULL,
                                                                    const char* sxname=NULL)
```

Parameters

tpsh	tportal server handle.
pclp	Permitted client handle list.
pclsz	Permitted client handle list size (number of clients).
dsn	Dual server slot number for pmsg received.
pname	tportal name.
sxname	Server exchange name.

Returns

true	tportal created.
false	tportal not created.

Errors See smx_MsgXchgCreate()

Descr Creates sxchg and loads pname and sxchg handle into the portal client structures in pcl and into the portal server structure. sxchg is a pass exchange with priority inheritance. A permitted client list must be defined for every portal in the system. It is a list of handles of portal clients that are allowed to access the portal. Loads dsn into tpsh->dsn. Notes:

1. Must be called from pmode.
2. Portals are owned by servers.

Example

```
TPSS   pssa;
 TPSS* phs = &pssa;

phs->stask = ut2s;
phs->dsn = SSLOT;
mp_TPortalCreate(phs, pcla, sizeof(pcla)/4, SV_SLOT, "portalA", "sxchg");
```

Creates tportal, pssa, for task ut2s, which was previously created. Loads “portalA” and sxchg handle into all client structures in pcla so they can open and use this tportal.

mp_TPortalDelete

```
bool   mp_TPortalDelete(TPSS* tpsh, TPCS** pclp, u32 pclsz)
```

Parameters

tpsh	tportal server handle
pclp	Permitted client handle list.
pclsz	Permitted client handle list size (number of clients).

Returns

true	tportal deleted.
false	tportal not deleted.

Errors See smx_TaskStop() and smx_MsgXchgDelete()

Descr Stops the server task, deletes sxchg, and clears sxchg handles and portal names in permitted portal client structures. Marks the portal closed in the pmsg header and clears the portal server structure. Notes:

1. Must be called from pmode.
2. tportal cannot be reopened by a client.

Example

```
sb_TPortalDelete(&pssa, tpcla, sizeof(pcla)/4);
```

Reverses portal creation in the above mp_TPortalCreate() example.

mp_TportalOpen

```
bool   mp_TPortalOpen(TPCS* tpch, u32 msz, u32 thsz, u8 pri, u32 tmo=0,
                                        const char* ssname=NULL, const char* csname=NULL)
```

Parameters

tpch	tportal client handle.
msz	pmsg data block size.
thsz	Total header size.
tmo	csem timeout (ticks).
ssname	ssem name.
csname	csem name.

Returns

true	tportal opened.
false	tportal not opened.

Errors

NO_PMSG	pmsg does not exist.
PORTAL_NEXIST	tportal has not been created.

See also smx_SemCreate(), smx_SemTest(), smx_PMsgSendB().

Descr Called by a client to open an existing portal from the client side, so that the portal can be used by the client. Aborts if sxchg for the portal does not exist, reports PORTAL_NEXIST, and returns false. Aborts if pmsg for the portal does not exist, reports NO_PMSG, and returns false. Allows reopening a portal closed by either the client or the server. Creates csem and ssem for the portal, unless they already exist. Loads pmsg parameters into the portal client structure, creates an OPEN pmsg, and sends it to sxchg of the portal. Waits for the server to acknowledge. msz is the size of the pmsg data block. Total header size, thsz = pmsg header size + service header size. Must be preceded with:

1. Get pmsg of msz and load tpch->pmsg and tpch->mhp
2. Load tpch->name and tpch->sxchg by TPortalCreate().

Note: pmsg is bound to the client task. The server can receive the pmsg, but only the client can send or release it.

Example

```
TPCS*  tpch = &pcsA;

tpch->pmsg = smx_PMsgGetHeap(mdsz, (u8**)&tpch->mhp, clslot, DATARW, 0);
mp_TPortalOpen(tpch, mdsz, thsz, TP3, CTMO, "ssem", "csem");
```

Gets a pmsg from heap 0 with a data block of at least mdsz bytes, loads the data block address into tpch->mhp and loads the data block region into MPA[clslot] and MPU[clslot+fas]. Opens tportal using pmsg.

mp_TPortalReceive

```
bool   mp_TPortalReceive(TPCS* tpch, u8* dp, u32 rqsz, u32 tmo=0)
```

Parameters

tpch	tportal client handle.
dp	Data pointer.
rqsz	Total request size.
tmo	csem timeout (ticks).

Returns

true	Last data block has been received.
false	Last data block has not been received.

Errors

INV_SIZE	No-copy size exceeds size of pbuf.
PORTAL_NOPEN	tportal not open to start.
PORTAL_CLOSED	tportal closed during operation.
CLIENT_TMO	csem timed out.
TRANS_INC	Transfer incomplete.

See also smx_SemSignal(), smx_SemTest(), smx_PMsgSendB().

Descr

Called by a client to receive data from a server. Sets command to RECEIVE, passes rqsz via message header, and signals ssem. Waits on csem for a data block from server. dp != NULL means copy mode. In this case, copies each received data block from pbuf to the client buffer at dp. If not end of data, updates dp and signals ssem for the server to send the next block. If end of data, operation stops. dp == NULL means no-copy mode. In this case only one block is transferred from the server, and it is left in pbuf, which also serves as the client's the work buffer so no data copying is necessary at the client end.

Note: This function is used only for data block transfers from server to client. For all other calls by client shells, use mp_TPortalCall() or mp_TPortalSend().

Example

```
SFSP_SH* shp = (SFSP_SH*)pch->shp;
mp_SHL4(SFS_ID_FREAD, 0, 0, 0, (u32)filehandle, 0);

#if SFSP_NO_COPY
mp_TPortalReceive(pch, NULL, size*items, SFSP_CTMO)
#else
mp_TPortalReceive(pch, (u8*)wbuf, size*items, SFSP_CTMO)
#endif
```

Reads a file via the file system portal. SFSP_SH is the file system service header. It comes after the standard portal message header. It is filled by the mp_SHL4() macro, then mb_TPortalReceive() is called. This example shows both no-copy and copy operation. Note that dp is NULL in the no-copy case, but points at a work buffer, wbuf, in the copy case.

mp_TPortalSend

```
bool   mp_TPortalSend(TPCS* tpch, u8* dp=NULL, u32 rqsz=0, u32 tmo=0)
```

Parameters		
	tpch	tportal client handle.
	dp	Data pointer.
	rqsz	Total request size.
	tmo	csem timeout (ticks).

Returns		
	true	All data sent.
	false	All data not sent.

Errors		
	INV_SIZE	No-copy size exceeds pbuf size.
	PORTAL_NOPEN	tportal not open.
	CLIENT_TMO	csem timed out.
	TRANS_INC	Transfer incomplete.

See also smx_SemSignal() and smx_SemTest().

Descr Called by client to send data or command from client to server. dp != NULL means *copy mode*. Sets command to SEND, sets end of data flag if one block is being sent, else clears it, passes rqsz via message header, and copies up to sizeof(pbuf) bytes from user buffer at dp to pbuf and updates dp. Then signals ssem, and waits at csem for ack from server. When ack received, if all data has not been sent, copies the next data block from the user buffer at dp to pbuf and sends it. Repeats sending blocks until rqsz bytes have been sent. Sets end of data for last block sent.

dp == NULL means *no-copy* mode. Sets command to SEND, sets sod and eod, clears err, passes rqsz via message header, and signals ssem. Then waits at csem for ack from server. Only the data block already in pbuf is sent. pbuf is also the client work buffer so no data copying is necessary. This is the usual case when commands, are being sent.

Note: This function is used for all client shell operations, except block data transfers from server to client. For those, use mp_TPortalReceive(). Even for a client shell that implements an API call that gets some data value(s), this function is used. For all but data block transfer, set dp = NULL and rqsz = 0. mp_TPortalCall() is provided as a simpler way to call this function for API calls that don't send data blocks. It passes NULL and 0 for dp and rqsz.

Example

```
SFSP_SH* tpch = (SFSP_SH)tpch->shp;

mp_SHL4(SFS_ID_FWRITE, 0, 0, 0, (u32)filehandle, 0);
#if SFSP_NO_COPY
mp_TPortalSend(tpch, NULL, size*items, SFSP_CTMO)
#else
mp_TPortalSend(tpch, (u8*)wbuf, size*items, SFSP_CTMO)
#endif
```

This shows writing a file via the file system portal. SFSP_SH is the file system service header. It comes after the standard portal message header. It is filled by the mp_SHL4() macro, then sb_TPortalSend() is called. This example shows both no-copy and copy operation. Note that dp is NULL in the no-copy case, whereas dp points at a work buffer, wbuf, and rqsz is the size of the buffer in the copy case.

Note: See mp_TPortalCall() examples for sending commands that do not send data blocks.

mp_TPortalServer

```
void    mp_TPortalServer(TPSS* tpsh, u32 stmo)
```

Parameters

tpsh	tportal server handle.
stmo	ssem timeout (ticks).

Errors

INV_CMD	Invalid command.
INV_SID	Invalid server ID.
SERVER_TMO	ssem timeout.

See also smx_PMsgReceive(), smx_SemSignal(), and smx_SemTest().

Descr Called by the portal task to perform portal server functions. Waits at sxchg for a portal OPEN pmsg to open the tunnel portal for the server or for a FREEMSG pmsg. Implements the tunnel portal protocol at the server end if mhp->type == TUNNEL. Signals csem when commands are done, and waits at ssem for more commands. When the portal is closed by a client or a free pmsg is processed, waits at sxchg for the next OPEN or FREEMSG pmsg. This also occurs if an invalid pmsg type or command is received. If ssem times out, the portal is closed at the sender's end and the portal server structure is cleared. When done or error, gets next pmsg from sxchg. To stop the portal server, stop or delete the portal task.

Example

```
void sfsp_main(void)
{
  mp_TPortalServer(tpsh, SFSP_STMO);
}

static void mp_TPortalCallServerFunc(TPSS* tpsh)
{
   switch (psh->sid)
   {
      …
      case SFSP:
        sfsp_server(psh);
        break;
      …
      default:
        mp_PortalEM((PS*)psh, INV_SID, &psh->mhp->errno);
   }
}
```

As shown above, sb_TPortalServer() is the main function for the file system portal task. It calls mp_TPortalCallServerFunc(tpsh), which is partially shown above for the file system case SFSP, the sid for the file system. This case calls sfsp_server(tpsh), which is located in fpsvr.c. This function interprets the fhp->fid and calls the appropriate file system function, as follows:

```
switch (fhp->fid)
{
     …

     case SFS_ID_FWRITE:
          fhp->ret = sfs_fwrite((void*)ph->mdp, mhp->mdsz, 1, (FILEHANDLE)fhp->p4);
          break;
     …
}
```

Creating a new server portal, requires defining a new server id, sid, and adding a case for it to mp_TPortalCallServerFunc(). Then defining function ids, fids, for the server functions accessible via the portal and defining a new server function that interprets them. Normally, this function is located with other server code for the file system.

mp_RegionGetHeapR

u8* mp_RegionGetHeapR(MPR_PTR rp, u32 sz, u8 sn, u32 attr, const char* name=NULL, u32 hn=0)

Parameters

rp	Region pointer.
sz	Size of block (minimum).
sn	Slot number in task MPA.
attr	Attributes for region.
name	Name for region.
hn	Heap number.

Returns

bp	Block pointer.
NULL	No block available or error.

Errors See smx_HeapMalloc()

Descr Used to create a dynamic region that might be used by one or more tasks. Gets a block of at least sz bytes from heap hn, creates rbar and rasr or rlar for it, and loads them into the region at rp. If MP_MPA_DEV also loads name into region defined as for ARMM7:

```
typedef struct {     /* MEMORY PROTECTION REGION */
  u32  rbar;         /* region address and slot number */
  u32  rasr;         /* region attributes and size */
  const char *name;    /* region name (for debugging, if MP_MPA_DEV) */
} MPR, *MPR_PTR;
```

Example

```
u8*  bp;
MPR* dpr[3];
bp = mp_RegionGetHeapR(&dpr[0], 140, 4, DATARW, "dblock", h0);
mpa_tmplt_t2a[4] = MP_DYN_RGN(dpr[0]);

t2a = smx_TaskCreate(t2a_main, TP2, 0, 0, "t2a");
mp_MPACreate(t2a, &mpa_tmplt_t2a);
```

Gets a 140-byte data block from heap 0, loads its region information into dpr[0]. mpa_tmplt_t2a[4] is loaded with the address of dpr[0] and its dynamic flag is set. t2a is created and its MPA is created from mpa_tmplt_t2a. In this process, the region information for block bp will be loaded into t2a MPA[4]. This block is fully protected and can be accessed only by t2a for read/write, only.

mp_RegionGetHeapT

u8* mp_RegionGetHeapT(TCB_PTR task, u32 sz, u8 sn, u32 attr, const char* name=NULL,u32 hn=0)

Parameters

task	Task for region.
sz	Size of block (minimum).
sn	Slot number in task MPA.
attr	Attributes for region.
name	Name for region.
hn	Heap number.

Returns

bp	Block pointer.
NULL	No block available or error.

Errors See smx_HeapMalloc()

Descr Used to directly create a dynamic region for a task. Gets a block of at least sz bytes from heap hn, creates rbar and rasr or rlar for it, and loads them into task MPA[sn]. If MP_MPA_DEV also loads name into MPA[sn]. Used to get a protected block for a task. Called from smx_TaskCreate(), smx_PBlockGetHeap(), smx_PMsgGetHeap(), and directly.

Example

```
u8*  bp;
bp = mp_RegionGetHeapT(taskA, 100, 3, DATARW, "pblock", h0);
memset(bp, 0, 100);
```

Gets a 100-byte data block from heap 0, loads its region information into taskA MPA[3], then clears the block. The block is fully protected and can be accessed only by taskA for read/write, only.

mp_RegionGetPoolR

u8* mp_RegionGetPoolR(MPR_PTR rp, PCB_PTR pool, u8 sn, u32 attr, const char* name=NULL)

Parameters

rp	Region pointer.
pool	Pool to get block from.
sn	Slot number in task MPA.
attr	Attributes for region.
name	Name for region.

Returns

bp	Block pointer.
NULL	No block available or error.

Errors See sb_BlockGet()

Descr Basically the same as mp_RegionGetHeapR(), except that the block comes from a block pool. For ARMM7, the pool block size must be a power of two and the pool must be aligned on the block size. For ARMM8 the block size must be a multiple of

32 and the pool must be aligned on 32 bytes. This function is useful if several tasks require blocks of the same size, in which case, creating a block pool is likely to be more efficient than using a heap. Or it might be used in the case where it is not desirable to create a heap for the partition.

Example

```
u8*  bp;
MPR* dpr[3];
PCB_PTR  pool_256;

bp = mp_RegionGetPoolR(&dpr[1], &pool256, 4, DATARW, "block2");
mpa_tmplt_t2a[4] = MP_DYN_RGN(dpr[1]);
t2a = smx_TaskCreate(t2a_main, TP2, 0, 0, "t2a");
mp_MPACreate(t2a, &mpa_tmplt_t2a);
```

Gets a 256-byte data block from pool_256 and loads its region information into dpr[1]. mpa_tmple_t2a[4] is loaded with the address of dpr[1] and its dynamic flag is set. Then t2a is created and its MPA is created from mpa_tmplt_t2a. As a result, the region information for block bp is loaded into t2a MPA[4]. Assuming pool_t2a is directly accessible only in pmode, this block is now fully protected and cannot be accessed by any other task. Also t2a can access it only for read/write.

mp_RegionGetPoolT

```
u8*  mp_RegionGetPoolT(TCB_PTR task, PCB_PTR pool, u8 sn, u32 attr, const char* name=NULL)
```

Parameters

task	Task for region.
pool	Pool to get block from.
sn	Slot number in task MPA.
attr	Attributes for region.
name	Name for region.

Returns

bp	Block pointer.
NULL	No block available or error.

Errors See sb_BlockGet()

Descr Basically the same as mp_RegionGetHeapT(), except that the block comes from a block pool. For ARMM7, the pool block size must be a power of two, and the pool must be aligned on the block size. For ARMM8, the block size must be a multiple of 32 and the pool must be aligned on 32 bytes. This function is useful if several tasks require blocks of the same size, in which case, creating a block pool is likely to be more efficient than using a heap. Or it might be used in the case where it is not desirable to create a heap for the partition.

Example

```
u8*  bp;
PCB_PTR  pool_256;

bp = mp_RegionGetPoolT(taskA, &pool_256, 3, DATARW, "pblock");
memset(bp, 0, 256);
```

Gets a 256-byte data block from pool_256 and loads its region information into taskA MPA[3], then clears the block. Assuming pool_t2a is directly accessible only in pmode, this block is now fully protected and cannot be accessed by any other task. Also taskA can access it only for read/write, not to execute code.

mp_RegionMakeR

```
bool   mp_RegionMakeR(MPR_PTR rp, u8* bp, u32 sz, u8 sn, u32 attr, const char* name=NULL)
```

Parameters

rp	Region pointer.
bp	Block pointer.
sz	Size of block.
sn	Slot number.
attr	Attributes for region.
name	Name for region.

Returns

true	Region made.
false	Region not made.

Errors SMXE_INV_PAR

Descr Makes a region for a block from any source given a block pointer, bp, and size, sz, creates rbar and rasr or rlar, and loads them into region at rp. Used to create a dynamic region for slot sn with attributes, attr. Block must be at least 32 bytes and of proper size and alignment.

For umode, the block must be fully within the MPU. If not, the make fails and SMXE_INV_PAR is reported. This is to prevent unauthorized memory accesses from umode. This limitation does not apply for pmode. For ARMM8, if block overlaps an MPU region, sn must be an auxiliary slot in the current task's MPA because ARMM8 does not permit overlapped MPU regions. If not, make fails and SMXE_INV_PAR is reported.

Example

```
MPR* dpr[2];
#pragma data_alignment=128
u8 ablk[128];

mp_RegionMakeR(&dpr[2], ablk, 128, 4, DATARW, "ablk")
mpa_tmplt_taskA[4] = MP_DYN_RGN(dpr[2]);
…
taskA = smx_TaskCreate(taskA_main, TP2, 0, 0, "taskA");
mp_MPACreate(taskA, &mpa_tmplt_taskA);
```

Creates region information for ablk and loads it into dpr[2]. mpa_tmplt_taskA[4] is loaded with the address of dpr[1] and its dynamic flag is set. At some later time, taskA is created and its MPA is loaded from mpa_tmplt_taskA, so the ablk region is loaded into MPA[4] of taskA. ablk is now accessible by taskA for read/write. If the dpr[2] dynamic region is also used in mpa_tmplt_taskB then taskA and taskB can both access ablk and can communicate through it. The attributes need not be the same: taskA could have read/write access, and task B could be read only.

mp_RegionMakeT

```
bool   mp_RegionMakeT(u8* bp, u32 sz, u8 sn, u32 attr, const char* name=NULL)
```

Parameters

bp	Block pointer.
sz	Size of block.
sn	Slot number.
attr	Attributes for region.
name	Name for region.

Returns

true	Region made.
false	Region not made.

Errors SMXE_INV_PAR

Descr Makes a region for a block from any source given a block pointer, bp, and size, sz, creates rbar and rasr or rlar for it, and loads them into the current task's MPA[sn]. Block must be at least 32 bytes and of proper size and alignment.

For umode, the block must be fully within the MPU. If not, the make fails and SMXE_INV_PAR is reported. This is to prevent unauthorized memory accesses from umode. This limitation does not apply for pmode. For ARMM8, if block overlaps an MPU region, sn must be an auxiliary slot in the current task's MPA because ARMM8 does not permit overlapped MPU regions. If not, make fails and SMXE_INV_PAR is reported.

Example

```
#pragma data_alignment=128
u8 ablk[128];

bp = mp_RegionMakeT(ablk, 128, 3, DATARW, "ablk");
memset(bp, 0, 128);
```

Creates a region for ablk and loads it into MPA[3] of taskA, and then clears the block. ablk is now accessible by taskA for read/write, only. If this function is repeated for ablk for taskB, then taskA and taskB can both access ablk and can communicate through it. The attributes need not be the same: taskA could have read/write access, but task B might be read only.

Appendix A.2 smx Protected Block & Message Services

These services manage protected blocks and protected messages for use in SecureSMX systems. Although they are smx services, they are available only with SecureSMX and they are intended to be used only with it. Most of these services are SSRs and thus are task-safe. For ARMM8, if a protected block overlaps another MPU region, an MMF will occur when the overlap area is first accessed. This is not the case for ARMM7.

smx_PBlockGetHeap

```
u8*   smx_PBlockGetHeap(u32 sz, u8 sn, u32 attr, const char* name=NULL, u32 hn=0)
```

Type SSR.

Summary Gets a protected block from a heap.

Compl smx_PBlockRel().

Parameters

	sz	minimum block sz required.
	sn	MPA slot number.
	attr	block attributes (e.g. RW_DATA).
	name	region name.
	hn	heap number

Returns

bp	block pointer.
NULL	no block available or error.

Errors SMXE_INV_PAR invalid sn.
See also smx_HeapMalloc().

Descr Gets an aligned block of at least sz bytes from heap hn, creates a region for the block, and loads the region into MPA[sn] of the current task. Then loads MPA[sn] into MPU[sn+fas]. Block alignment meets MPU requirements for the block size. Actual memory allocation is minimized by sizing to the nearest multiple of subregion size, and the proper subregion disable bits are set in the region created. *name* is loaded into the region name field in MPA[sn] if MP_MPA_DEV. The block is a bare block.

Example

```
u8*     bp;
bp = smx_PBlockGetHeap(80, 4, DATARW, “work_buf”, 3);
```

Gets an 80-byte data pblock, named “work_buf” from heap 3, puts its region into MPA[4] and MPU[4+fas], and returns a pointer, bp, to the block.

smx_PBlockGetPool

u8* smx_PBlockGetPool(PCB_PTR pool, u8 sn, u32 attr, const char* name=NULL)

Type SSR.

Summary Gets a protected block from a block pool.

Compl smx_PBlockRel().

Parameters

pool	block pool.
sn	MPA slot number.
attr	block attributes (e.g. RW_DATA).
name	region name.

Returns

bp	block pointer.
NULL	No block available or error.

Errors

SMXE_INV_PAR	invalid sn.
SMXE_WRONG_POOL	invalid block size or alignment.
SMXE_POOL_EMPTY	pool empty.

Descr Gets a block, from *pool*, creates a region for the block, and loads the region into MPA[sn] of the current task. Then loads MPA[sn] into MPU[sn+fas]. Block must meet MPU alignment requirements for its size. *name* is loaded into the region name field in MPA[sn] if MP_MPA_DEV. The block is a bare block.

Example

```
u8*   bp;
PCB   pool128;

bp = smx_PBlockGetPool(&pool128, 4, DATARW, “out_buf”);
```

Gets an 128-byte data pblock, named “out_buf” from block pool128 and puts its region into MPA[4] and MPU[4+fas], and returns a pointer, bp, to the block.

smx_PBlockMake

```
bool   smx_PBlockMake(u8* bp, u32 sz, u8 sn, u32 attr, const char* name=NULL)
```

Type SSR.

Summary Makes a pblock from a standalone block.

Compl smx_PBlockRel().

Parameters

bp	block pointer.
sz	block size**.**
sn	MPA slot number.
attr	block attributes (e.g. RW_DATA).
name	region name.

Returns

true	Conversion succeeded.
false	Conversion failed.

Errors

SMXE_INV_PAR	invalid sn.
SMXE_WRONG_POOL	invalid block size or alignment.

Descr Creates a region for a standalone block, and loads the region into MPA[sn] of the current task. Then loads MPA[sn] into MPU[sn+fas]. Block must meet MPU alignment and size requirements. *name* is loaded into the region name field in MPA[sn] if MP_MPA_DEV. The block is a bare block.

For umode, the block must be fully within the MPU. If not, the make fails and SMXE_INV_PAR is reported. This is to prevent unauthorized memory accesses from umode. This limitation does not apply for pmode. For ARMM8, if block overlaps an MPU region, sn must be an auxiliary slot in the current task's MPA because ARMM8 does not permit overlapped MPU regions. If not, make fails and SMXE_INV_PAR is reported.

Example

```
#pragma data_alignment = 32
u8      cb[32];

bp = smx_PBlockMake(&cb, sizeof(cb), 4, DATARW, "char_buf");
```

Makes a 32-byte pblock from cb[] and puts its region into MPA[4] and MPU[4+fas], names it "char_buf", and returns a pointer to the block, bp.

smx_PBlockRelHeap

```
bool   smx_PBlockRelHeap (u8* bp, u8 sn, u32 hn=0)
```

Type SSR.

Summary Releases a protected block to heap hn.

Compl smx_PBlockGetHeap().

Parameters

bp	block pointer.
sn	MPA slot number.
hn	Heap number.

Returns

true	Block released.
false	Block not released due to error.

Errors SMXE_INV_PAR invalid sn is or bp is not in heap hn range.

Descr Releases a block obtained by smx_PBlockGetHeap(). Aborts if parameters are incorrect. Otherwise, releases the block back to heap hn and clears[35] MPU[sn+fas] and MPA[sn]. Thus, once released, a block can no longer be accessed by the current task.

Example

```
u8*     bp;

smx_PBlockRelHeap(bp, 4, 2);
```

Releases the block pointed at by bp to heap 2 and clears MPA[4] and MPU[4+fas]

smx_PBlockRelPool

```
bool   smx_PBlockRelPool (u8* bp, u8 sn, PCB* pool=NULL, u32 clrsz=0)
```

Type SSR.

Summary Releases a protected block to block pool.

Compl smx_PBlockGetPool() and smx_PBlockMake().

Parameters

bp	block pointer.
sn	MPA slot number.
pool	Pool handle.
clrsz	Number of bytes to clear in block, after byte 4.

[35] Clearing MPA and MPU slots is a complex process. See SecureSMX User's Guide, Clearing Slots for details.

Returns true Block released.
false Block not released due to error.

Errors SMXE_INV_PAR invalid sn or pool.

Descr Releases a protected block obtained by smx_PBlockGetPool() or smx_PBlockMake(). Aborts if parameters are incorrect. Otherwise, releases the block back to its block pool and clears clrsz bytes from byte 4 up to the end of the block and clears MPU[sn+fas] and MPA[sn]. Thus, once released, a block can no longer be accessed by the current utask. If pool == NULL, the block is a standalone block and it is not released nor cleared.

Example1

```
u8*   bp;
PCB*  poolA;

smx_PBlockRelPool(bp, 4, poolA, PA_BLK_SZ);
```

Releases block pointed to by bp to poolA, clears MPA[4] and MPU[4+fas] and clears PA_BLK_SZ bytes in it from byte 4.

Example2

```
smx_PBlockRelPool(bp, 4);
```

Clears MPA[4] and MPU[4+fas] for the block pointed to by &cb. Does not release the block, nor clear it (standalone block).

smx_PMsgGetHeap

```
MCB_PTR  smx_PMsgGetHeap(u32 sz, u8** bpp, u8 sn, u32 attr, u32 hn=0 , MCB_PTR* mhp=NULL)
```

Type SSR.

Summary Gets a protected message from heap hn.

Compl smx_PMsgRel().

Parameters

sz	minimum block sz required.
	bpplace to put pmsg block pointer.
sn	MPA slot number.
attr	block attributes (e.g. RW_DATA).
hn	heap number.
mhp	message handle pointer (used for tokens)

Returns

pmsg	pmsg handle.
NULL	no block available or error.

Errors SMXE_INV_PAR invalid sn.
See also smx_HeapMalloc().

Descr Gets an aligned block of at least sz bytes from heap hn, creates a region for the block, and loads the region into MPA[sn] of the current task. Block alignment meets MPU requirements. Actual memory allocation is minimized by sizing to the nearest multiple of subregion size, and the proper subregion disable bits are set in the region created. Then loads MPA[sn] into MPU[sn+fas], gets an MCB, and makes the protected block into a protected message, pmsg. Returns the pmsg block pointer via bpp, unless bpp == NULL. The pmsg can be sent to and received from a normal smx message exchange.

Example

```
u8*     bp;
MCB_PTR pmsg;

pmsg = smx_PMsgGetHeap(80, &bp, 4, DATARW);
```

Gets an 80-byte data block from heap 0, puts its region into MPA[4] and MPU[4+fas], makes the pblock into a pmsg, returns the pmsg handle, and loads the address of the data block into bp.

smx_PMsgGetPool

```
MCB_PTR smx_PMsgGetPool(PCB_PTR pool, u8** bpp, u8 sn, u32 attr , MCB_PTR* mhp=NULL)
```

Type SSR.

Summary Gets a protected message from a block pool.

Compl smx_PMsgRel().

Parameters

pool	block pool.
bpp	place to put pmsg block pointer.
sn	MPA slot number.
attr	block attributes (e.g. RW_DATA).
mhp	message handle pointer (used for tokens).

Returns

pmsg	pmsg handle.
NULL	No pmsg available or error.

Errors

SMXE_INV_PAR	invalid sn.
SMXE_OUT_OF_MCBS	no MCBs available.
SMXE_WRONG_POOL	invalid block size or alignment.
SMXE_POOL_EMPTY	pool empty.

Descr Gets a block, from *pool*, creates a region for the block, and loads the region into MPA[sn] of the current task. Block must meet MPU alignment requirements for its size. Then loads MPA[sn] into MPU[sn+fas], gets an MCB, and makes the protected

block into a protected message, pmsg. The pmsg can be sent to and received from a normal smx message exchange.

Example

```
u8*         bp;
PCB         p128;
MCB_PTR     pmsg;

pmsg = smx_PMsgGetPool(&p128, &bp, 4, DATARW);
```

Gets an 128-byte data block from block pool p128, puts its region into MPA[4] and MPU[4+fas], makes the pblock into a pmsg, returns the pmsg handle, and loads the address of the data block into bp.

smx_PMsgMake

MCB_PTR smx_PMsgMake(u8* bp, u32 sz, u8 sn, u32 attr, const char* name=NULL , MCB_PTR* mhp=NULL)

Type SSR.

Summary Makes a protected message from a standalone block.

Compl smx_PMsgRel().

Parameters

bp	block pointer.
sz	block size**.**
sn	MPA slot number.
attr	block attributes (e.g. RW_DATA).
name	region name.
mhp	message handle pointer (used for tokens)

Returns

true	Conversion succeeded.
false	Conversion failed.

Errors SMXE_INV_PAR invalid sn, sz, or alignment.

Descr Creates a region for the standalone block and loads the region into MPA[sn] of the current task. Block must meet MPU alignment requirements for its size. Then loads MPA[sn] into MPU[sn+fas], gets an MCB, and makes the protected block into a protected message, pmsg. Also sets pmsg->bs to -1 to indicate that the pmsg block is standalone and thus need not be released by smx_PMsgRel(). The pmsg can be sent to and received from a normal smx message exchange.

For umode, the pmsg block must be fully within the MPU. If not, the make fails and SMXE_INV_PAR is reported. This is to prevent unauthorized memory accesses from umode. This limitation does not apply for pmode. For ARMM8, if pmsg block overlaps an MPU region, sn must be an auxiliary slot in the current task's MPA

because ARMM8 does not permit overlapped MPU regions. If not, make fails and SMXE_INV_PAR is reported.

Example

```
#pragma data_alignment = 32

u8          cb[32];
MCB_PTR     pmsg;

pmsg = smx_PMsgMake(&cb, 32, 4, DATARW, "smsg");
```

Makes a 32-byte pblock from cb[] and puts its region into MPA[4] and MPU[4+fas], makes the pblock into a pmsg, names it "smsg", and returns its handle.

smx_PMsgReceive

MCB_PTR smx_PMsgReceive (XCB_PTR xchg, u8** bpp, u8 dsn, u32 tmo=0, MCB_PTR* mhp=NULL)

Type SSR.

Summary Gets a pmsg from an exchange.

Compl smx_PMsgSend().

Parameters

xchg	exchange to get message from.
bpp	pointer to message block pointer. NULL if none.
dsn	MPA dual slot number for message region.
timeout	timeout in ticks.
mhp	Message handle pointer (used for tokens).

Returns

pmsg	Protected message handle.
NULL	Error or timeout.

Errors

SMXE_INV_PAR	invalid sn.
SMXE_INV_PRI	message priority is invalid for a pass exchange.
SMXE_INV_XCB	invalid exchange handle.
SMXE_WAIT_NOT_ALLOWED	called from LSR with nonzero timeout.

Descr Gets a pmsg from *xchg*. If no pmsg is waiting at xchg, suspends the current task for *tmo* ticks. Fails if tmo ticks elapse before a pmsg is received. Operates the same as smx_MsgReceive(), except that it obtains rbar and rasr for the pmsg block from the pmsg MCB and loads them into MPA[sn] and MPU[sn+fas], which allows the current task to access the pmsg block.

For ARMM7, sn = dsn. For ARMM8, dsn = xsn << 4 + asn, where xsn = auxiliary slot number and asn = active slot number. When a pmsg is received, if pmsg->con.sb = 1 and the current task is a ptask, sn = xsn, otherwise sn = asn. If no pmsg is waiting at xchg, dsn is saved in the current task's TCB.

Note dsn is necessitated because the ARMM8 architecture does not permit overlapping regions. Receive() and ReceiveStop() are the only services requiring dsn and they require it only when called from ptasks. If it is known that no pmsgs having data blocks in sys_data will be received by a ptask, then dsn is not required for it. See section 5.2.4 for more information.

Example

```
u8*     pbp;
MCB_PTR pmsg;
XCB_PTR uxr;

pmsg = smx_PMsgReceive(uxr, &pbp, 4, 5);
```

Receives pmsg from exchange uxr, puts pmsg block region into MPA[4] and MPU[4+fas], loads pmsg block address into pbp, if a pmsg is waiting at uxr. If not, will wait up to 5 ticks for a pmsg. If none received, returns NULL.

smx_PMsgReceiveStop

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

Type Limited SSR (tasks only).

Summary Same as smx_PMsgReceive() except that the current task is stopped then restarted when a pmsg is received or a timeout occurs.

Compl smx_PMsgSend().

Parameters

xchg	exchange to get message from.
bpp	pointer to message block pointer. NULL if none.
dsn	MPA dual slot number for message region.
timeout	timeout in ticks.
mhp	Message handle pointer (used for tokens).

Errors

SMXE_INV_PAR	invalid sn or bpp points into the current stack.
SMXE_INV_PRI	message priority is invalid for a pass exchange.
SMXE_INV_XCB	invalid exchange handle.
SMXE_OP_NOT_ALLOWED	called from an LSR.

Descr See smx_PMsgReceive() for operational description. The current task always stops, then restarts instead of resuming, if a pmsg is received or a timeout occurs. As a consequence, the task starts over from the beginning of its main function. The message handle is returned via the parameter in taskMain(u32 par) – see example.

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

Example

```
smx_TaskStartPar(taskM, -1);

void taskM(u32 par)
{
    static u8*     bp;
    MCB_PTR    pmsg = (MCB_PTR)par;
    XCB_PTR    uxr;

    if (pmsg ! = NULL && pmsg != -1)
        /* process pmsg using bp */

    smx_PMsgReceiveStop(uxr, &bp, 4, 500);
}
```

The first time taskM is started, par == -1 and processing is skipped. If a pmsg is waiting at uxr, smx_PMsgReceiveStop() stops the current task, receives pmsg from uxr, puts pmsg block region into MPA[4], loads pmsg block address into pbp, then restarts taskM, passes the pmsg handle in as par, and loads MPA[4] into MPU[4+fas]. If no pmsg is waiting at uxr, waits up to 500 ticks for one to arrive. If a pmsg is received, performs the previous steps, else restarts taskM with pmsg == NULL.

Although the code above seems convoluted, the advantage of using smx_PMsgReceiveStop() instead of smx_PMsgReceive(), is that taskM releases its stack while waiting for the next pmsg. This allows other tasks to share the stack, which is desirable if pmsgs are received very seldom,

smx_PMsgRel

```
bool   smx_PMsgRel(MCB_PTR *mhp, u16 clrsz=0)
```

Type SSR.

Summary Releases a protected message.

Compl smx_PMsgGetHeap().

Parameters

mhp	message handle pointer.
clrz	Number of bytes to clear from 4th to end.

Returns

true	pmsg released.
false	pmsg not released due to error.

Errors SMXE_INV_PAR invalid sn is or bp is not in heap hn range.

Descr Releases a pmsg obtained by smx_PMsgGetHeap(), smx_PMsgGetPool(), or smx_PMsgMake(). Dequeues pmsg if it is at an exchange. Clears MPA[osn] and MPU[osn+fas] and releases the pmsg data block back to its heap or pool, unless it is a

standalone block, where osn is the pmsg owner slot number, stored in the pmsg. Only the pmsg owner can release a pmsg. Usually this is the task that got or received the pmsg.

When a pmsg is bound to the sender (as in a tunnel portal), the sender (portal client) remains its owner and only it can release the pmsg. In this case, the receiver (portal server) is called the *host*. If the contents of the host slot number, hsn, stored in the pmsg, match the contents of the owner slot number, osn, MPA[hsn] is cleared. Otherwise MPA[hsn] is not changed. The reason for this is that the host may have received another pmsg before the client releases its pmsg. Either way, the host can no longer access the pmsg data block.

Example

```
MCB_PTR    pmsg;

smx_PMsgRel(pmsg);
```

Releases pmsg and clears MPA[osn] and MPU[osn+fas], where osn is stored in the pmsg MCB. If the pmsg is bound, MPA[hsn] is also cleared if it is the same, where hsn and the bnd flag are stored in the pmsg MCB. In this case, the pmsg data block is not cleared.

smx_PMsgReply

```
bool   smx_PMsgReply(MCB_PTR pmsg)
```

Type Function calls SSR.

Summary Restricted version of smx_PMsgSend.

Parameters

pmsg	pmsg handle

Returns

true	pmsg sent or released.
false	pmsg not sent nor released due to error.

Descr Determines rxchg from pmsg->rpx, pri from pmsg->pri, and calls smx_PMsgSend(). Used in the free message protocol to limit a server's actions and to reduce errors. When smx_PMsgReply() is used, the server has no knowledge of nor control of the rxchg to which pmsg is returned. If pmsg->rpx = 0xFF (no reply exchange) releases pmsg. See smx_PMsgSend() for details.

Example

```
MCB_PTR  pmsg;

smx_PMsgReply(pmsg);
```

Sends pmsg to its resource exchange or releases it if it has no resource exchange.

smx_PMsgSend

bool smx_PMsgSend(MCB_PTR pmsg, XCB_PTR xchg, u8 pri=0, void *reply=NULL)

Type SSR.

Summary Sends pmsg to an exchange, or delivers pmsg to the top waiting task, if any.

Compl smx_PMsgReceive(), smx_PMsgReceiveStop().

Parameters

pmsg	protected message handle to send.
xchg	exchange handle to send message to.
pri	priority of msg, unless == SMX_PRI_NOCHG.
reply	where to send reply. NULL if no reply is expected.

Returns

true	Message sent.
false	Message not sent due to error.

Errors

SMXE_INV_MCB	invalid pmsg or pmsg not owned by ct nor clsr.
SMXE_INV_PAR	invalid sn, pri, or reply.
SMXE_INV_XCB	invalid exchange.

Descr Operates the same as smx_MsgSend(), except for the following:

1. For ARMM7, pmsg->rasr = MPA[osn] and for ARMM8, pmsg->rlar = MPA[osn], where osn = pmsg->con.osn. osn is the pmsg *owner slot number*, which was loaded when the pmsg was first obtained or when it was received from another task. osn is the pmsg block region slot number in the MPA of the task that owns the pmsg. Only this task can send or release the pmsg.

2. If the pmsg bound flag is not set (pmsg->con.bnd == 0), clears MPU[osn+fas] and MPA[osn]. In this case, the sender can no longer access the pmsg data block. This is used for *free message portals*. If the bound flag is set, MPU[osn+fas] and MPA[osn] are not cleared. In this case, the sender remains the pmsg owner and it can still access the pmsg data block. This is used for *tunnel portals*.

3a. For ARMM7: if a receiving task, rtask, is waiting at xchg, its MPA[rsn] is loaded with the pmsg data region from the pmsg MCB, where rsn = (rtask->dsn & 0xF). In this case rsn is an active slot and MPA[rsn] will be loaded into MPU[rsn+fas] when rtask is resumed or started.

3b. For ARMM8: if (rtask is a utask || pmsg->con.sb == 0), rtask MPA[rsn] is loaded with the pmsg data region from the pmsg MCB, where rsn = (rtask->dsn & 0xF). In this case rsn is an active slot and MPA[rsn] will be loaded into MPU[rsn+fas] when rtask is resumed or started.

3c. For ARMM8: if (rtask is a ptask && pmsg->con.sb == 1), rtask MPA[rsn] is loaded with the pmsg data region from the pmsg MCB where rsn = rtask->dsn >> 4. In this case rsn must be an auxiliary slot and MPA[rsn] will not be loaded into MPU[rsn+fas] when rtask is resumed or started. The pmsg block region must be loaded into an auxiliary MPA slot because the sys_data region is loaded into the MPU whenever a ptask runs, else an MMF would occur due to overlapping regions.

Note dsn = dual slot number, was loaded into rtask->dsn by smx_PMsgReceive() or by smx_PMsgReceiveStop() and sb = system block, which, if set, means that the pmsg block came from the sys_data region.

Example

```
MCB_PTR    pmsg;
XCB_PTR    uxa;
XCB_PTR    uxr;

smx_PMsgSend(pmsg, uxa, 2, (void*)uxr);
```

Sends pmsg to exchange uxa with priority 2 and reply exchange uxr.

smx_PMsgSendB

```
bool   smx_PMsgSendB(MCB_PTR pmsg, XCB_PTR xchg, u8 pri=0, void *reply=NULL)
```

Type Function calls SSR.

Summary Sends a bound pmsg to an exchange.

Descr Same as smx_PMsgSend() except it sets the bound flag in pmsg MCB so that the pmsg continues to be owned by the sender. Used for tunnel portals.

Examples

```
MCB_PTR    pmsg;
XCB_PTR    uxa;

smx_PMsgSendB(pmsg, uxa, 2);
```

Sends a bound pmsg to exchange uxa with priority 2 and no reply exchange.

Appendix B: Linker Command Files

ARMM7

```
/*****************************************************************************
*  stm32746g_tsmx.icf                                          Version 5.1.0
*
*  ILINK Command File for IAR EWARM, TSMX, and STMicro STM32746G-EVAL board.
*
*  ARMM7 Internal ROM Internal SRAM Version (ROM build targets)
*
*  Memory Layout:
*
*   ITCMRAM   [0x00000000--0x00003FFF]  (16KB)  Unused (ITCM RAM)
*   ROM       [0x08000000--0x080FFFFF]   (1MB)  Int Flash via AXIM interface
*   ROM       [0x00200000--0x002FFFFF]   (1MB)  Int Flash via ITCM interface
*   TCRAM     [0x20000000--0x2000FFFF]  (64KB)  EVT, Main Stack (DTCM RAM)
*   SRAM1     [0x20010000--0x2004BFFF] (240KB)  Data (Int SRAM)
*   SRAM2     [0x2004C000--0x2004FFFF]  (16KB)  Joined with SRAM1 (Auxiliary Int SRAM)
*   RAM       [0xC0000000--0xC1FFFFFF]  (32MB)  LCD buffer (Ext SDRAM)
*
*****************************************************************************/

define symbol EVT_size = 4*(16+98); /* SB_IRQ_MAX+1 <1> */
define memory mem with size = 4G;

/* Memory region definitions */
define region ROM       = mem:[from 0x00200000 to 0x002FFFFF]; /* ITCM */
define region SRAM      = mem:[from 0x20000000 to 0x2004FFFF];
define region RAM       = mem:[from 0xC0000000 to 0xC1FFFFFF];

/* MPU region sizes (must be power of 2) <2> */
define exported symbol cpcsz    = 0x2000;    /* console partition */
define exported symbol cpdsz    = 0x1000;
define exported symbol cphsz    = 0x400;     /* console partition heap */
define exported symbol scsz     = 0x20000;   /* system */
define exported symbol sdsz     = 0x40000;
define exported symbol svccsz   = 0x2000;    /* svc */
define exported symbol t2acsz   = 0x1000;
define exported symbol t2adsz   = 0x100;
define exported symbol t2bcsz   = 0x100;
define exported symbol t2bdsz   = 0x20;
define exported symbol t2ccsz   = 0x100;     /* t2 client */
define exported symbol t2cdsz   = 0x40;
define exported symbol t2scsz   = 0x400;     /* t2 server */
define exported symbol t2sdsz   = 0x80;
```

```
define exported symbol tm23dsz  = 0x20;      /* tm23 swap region */
define exported symbol ut1acsz  = 0x20;
define exported symbol ut1adsz  = 0x80;
define exported symbol ut2acsz  = 0x1000;
define exported symbol ut2adsz  = 0x80;
define exported symbol ut2axdsz = 0x20;
define exported symbol ut2bcsz  = 0x200;
define exported symbol ut2bdsz  = 0x20;
define exported symbol ut2ccsz  = 0x2000;    /* ut2 client */
define exported symbol ut2cdsz  = 0x400;
define exported symbol ut2dcsz  = 0x100;     /* ut2 second client */
define exported symbol ut2ddsz  = 0x40;
define exported symbol ut2scsz  = 0x800;     /* ut2 server */
define exported symbol ut2sdsz  = 0x100;
define exported symbol ucsz     = 0x2000;    /* ucom */
define exported symbol udsz     = 0x100;

/* Empty block definitions (not initialized) */
define block CSTACK     with size = 0x200,    alignment = 8   { }; /* Main Stack */
define block EVT        with size = EVT_size, alignment = 512 { }; /* EVT <1> */
define block mheap      with size = 0x4000,   alignment = 16  { }; /* Mheap */
define block heap1      with size = 0x6144,   alignment = 16  { }; /* Heap1 ttCD17 */
define block heap2      with size = 0x2000,   alignment = 16  { }; /* Heap2 thMH05 */
define block heap3      with size = 0x2f60,   alignment = 16  { }; /* Heap3 thR */
define block cp_heap    with size = cphsz,    alignment = 16  { }; /* Cons part heap */
define block LCD_BUF    with size = 0x200000, alignment = 8   { }; /* LCD buffer */
keep  {block EVT, block LCD_BUF};

/* Console partition and SVC regions <3> */
define block cp_code    with size = cpcsz*7/8, alignment = cpcsz
                                {ro section .cp.text, ro section .cp.rodata};
define block cp_data    with size = cpdsz*5/8, alignment = cpdsz {block cp_heap,
                                 rw section .cp.data, rw section .cp.bss};
define block svc_code  with size = svccsz*5/8, alignment = svccsz
                               {ro section .svc.text, ro section .svc.rodata,
                                ro object ABImemset.o, ro object ABImemcpy.o,
                                ro object strcat.o, ro object strcpy.o,
                                ro object strncpy.o, ro object strlen.o};

/* Common umode regions <3> */
define block ucom_code  with fixed order, size = ucsz*7/8, alignment = ucsz
               {block svc_code, ro section .ucom.text, ro section .ucom.rodata};
define block ucom_data  with size = udsz, alignment = udsz
               {rw section .ucom.data, rw section .ucom.bss};

/* System regions <3> */
define block sys_code   with size = scsz*6/8, alignment = scsz {ro section .intvec,
               ro section .sys.text, ro section .sys.rodata, block cp_code,
               block ucom_code}; /*<4>*/
define block sys_data   with size = sdsz*5/8, alignment = sdsz, fixed order
   {block EVT, block CSTACK, block mheap, block ucom_data, block cp_data, /*<5>*/
    rw section .sys.bss, rw section .sys.data, rw section .sys.noinit,
    block heap1, block heap2, block heap3, rw};
```

```
/* Test task regions <3> */
define block t2a_code   with size = t2acsz*6/8, alignment = t2acsz
                              {ro section .t2a.text, ro section .t2a.rodata};
define block t2b_code   with size = t2bcsz, alignment = t2bcsz
                              {ro section .t2b.text, ro section .t2b.rodata};
define block t2c_code   with size = t2ccsz*5/8, alignment = t2ccsz
                              {ro section .t2c.text, ro section .t2c.rodata};
define block t2s_code   with size = t2scsz*5/8, alignment = t2scsz
                              {ro section .t2s.text, ro section .t2s.rodata};
define block ut1a_code  with size = ut1acsz, alignment = ut1acsz
                              {ro section .ut1a.text, ro section .ut1a.rodata};
define block ut2a_code  with size = ut2acsz*5/8, alignment = ut2acsz
                              {ro section .ut2a.text, ro section .ut2a.rodata};
define block ut2b_code  with size = ut2bcsz,  alignment = ut2bcsz
                              {ro section .ut2b.text, ro section .ut2b.rodata};
define block ut2c_code  with size = ut2ccsz*5/8, alignment = ut2ccsz
                              {ro section .ut2c.text, ro section .ut2c.rodata};
define block ut2d_code  with size = ut2dcsz, alignment = ut2dcsz
                              {ro section .ut2d.text, ro section .ut2d.rodata};
define block ut2s_code  with size = ut2scsz*5/8, alignment = ut2scsz
                              {ro section .ut2s.text, ro section .ut2s.rodata};
define block t2a_data   with size = t2adsz,  alignment = t2adsz {rw section .t2a.bss,
                                                                 rw section .t2a.data};
define block t2b_data   with size = t2bdsz,  alignment = t2bdsz {rw section .t2b.data};
define block t2c_data   with size = t2cdsz*5/8, alignment = t2cdsz {rw section .t2c.bss,
                                                                 rw section .t2c.data};
define block t2s_data   with size = t2sdsz*6/8,  alignment = t2sdsz {rw section
                                                        .t2s.bss,rw section .t2s.data};
define block ut1a_data  with size = ut1adsz, alignment = ut1adsz {rw section
                                                                    .ut1a.data};
define block ut2a_data  with size = ut2adsz*6/8, alignment = ut2adsz {rw section
                                                                    .ut2a.data};
define block ut2ax_data with size = ut2axdsz, alignment = ut2axdsz {rw section
                                                                    .ut2ax.data};
define block ut2b_data  with size = ut2bdsz, alignment = ut2bdsz {rw section
                                                                    .ut2b.data};
define block ut2c_data  with size = ut2cdsz*5/8, alignment = ut2cdsz {rw section
                                                                    .ut2c.bss};
define block ut2d_data  with size = ut2ddsz*5/8, alignment = ut2ddsz {rw section
                                                                    .ut2d.bss};
define block ut2s_data  with size = ut2sdsz, alignment = ut2sdsz {rw section .ut2s.bss,
                                                                 rw section .ut2s.data};
define block tm23_data  with size = tm23dsz, alignment = tm23dsz {rw section
                                                                    .tm23.data};

/* MPU region sizes for initialization (must be power of 2) */
define exported symbol romsz    = 0x100000;
define exported symbol sramsz   = 0x40000;
define exported symbol ramsz    = 0x200000;

/* Block ordering for best memory efficiency. MPUPACKER */
define block rom_block with fixed order, size = romsz*5/8, alignment = romsz
                        {block sys_code, block ut1a_code, block ut2b_code,
```

```
                         block ut2a_code, block t2a_code, block t2b_code,
                         block ut2c_code, block ut2d_code, block ut2s_code,
                         block t2c_code, block t2s_code, ro};
define block sram_block with fixed order, size = sramsz*6/8, alignment = sramsz
                        {block sys_data, block t2a_data, block t2b_data,
                         block ut1a_data, block ut2a_data, block ut2ax_data,
                         block ut2b_data, block ut2c_data, block ut2d_data,
                         block ut2s_data, block t2c_data, block t2s_data,
                         block tm23_data};
define block ram_block with fixed order, size = ramsz, alignment = ramsz
                        {block LCD_BUF};

/* Initialization */
initialize by copy with packing = none {rw};
do not initialize  {section .noinit};

/* Placements */
place in ROM            {block rom_block};
place in SRAM           {block sram_block};
place in RAM            {block ram_block};

/* Notes:
   1. size = 4*(16 + SB_IRQ_MAX + 1). Must be aligned to power of 2 >= EVT size.
   2. Ensures CSTACK is first to detect main stack overflows. Also places
      critical blocks in TCRAM for speed.
   3. Alignment must = size.
   4. Copied to EVT by startup code in case EVT is needed before the linker
      copy code runs.
   5. mheap must start at 0x20000400 or thAA16-24 will fail.
*/
```

ARMM8

```
/*****************************************************************************
*  lpc55s69_tsmx.icf                                            Version 5.1.0
*
*  ILINK Command File for IAR EWARM, TSMX, and STMicro NXP LPC55S69-EVK board.
*
*  ARMM8 Internal ROM Internal SRAM Version (ROM build targets)
*
*  Non-Secure Memory Layout:
*     ROM       [0x00000000--0x0009D7FF] (630KB)  Code (last 10KB reserved)
*     BOOT ROM  [0x03000000--0x0301FFFF] (128KB)  Boot code
*     SRAM X    [0x04000000--0x04007FFF]  (32KB)  Unused (Casper & power down)
*     SRAM0-3   [0x20000000--0x2003FFFF] (256KB)  EVT, Data, Main Stack
*     SRAM4     [0x20040000--0x20043FFF]  (16KB)  Unused (PowerQuad)
*     APB & AHB [0x40000000--0x4010FFFF] (1888KB) Peripherals
*  Secure Memory Layout:
*     Add 0x10000000 to all addresses
*
*****************************************************************************/
define symbol EVT_size = 4*(16+60);  /* 60=SB_IRQ_MAX+1 */
```

```
define memory mem with size = 4G;

/* Memory region definitions */
define region ROM       = mem:[from 0x00000000 to 0x0009D7FF];
define region SRAM      = mem:[from 0x20000000 to 0x2003FFFF];

/* MPU region sizes  */
define exported symbol cpcsz    = 0x13a0;    /* console partition */
define exported symbol cpdsz    = 0x8a0;
define exported symbol cphsz    = 0x400;     /* console partition heap */
define exported symbol scsz     = 0x12100;   /* system */
define exported symbol sdsz     = 0x1c080;
define exported symbol svccsz   = 0x10c0;    /* svc */
define exported symbol t2acsz   = 0x760;
define exported symbol t2adsz   = 0xa0;
define exported symbol t2bcsz   = 0x40;
define exported symbol t2bdsz   = 0x20;
define exported symbol t2ccsz   = 0xa0;      /* t2 client */
define exported symbol t2cdsz   = 0x20;
define exported symbol t2scsz   = 0x240;     /* t2 server */
define exported symbol t2sdsz   = 0x80;
define exported symbol tm23dsz  = 0x20;      /* tm23 swap region */
define exported symbol ut1acsz  = 0x20;
define exported symbol ut1adsz  = 0x20;
define exported symbol ut2acsz  = 0x640;
define exported symbol ut2adsz  = 0x40;
define exported symbol ut2axdsz = 0x20;
define exported symbol ut2bcsz  = 0x200;
define exported symbol ut2bdsz  = 0x20;
define exported symbol ut2ccsz  = 0x1240;    /* ut2 client */
define exported symbol ut2cdsz  = 0x2e0;
define exported symbol ut2dcsz  = 0x100;     /* ut2 second client */
define exported symbol ut2ddsz  = 0x20;
define exported symbol ut2scsz  = 0x420;     /* ut2 server */
define exported symbol ut2sdsz  = 0xe0;
define exported symbol ucsz     = 0x12a0;    /* ucom */
define exported symbol udsz     = 0x80;

/* Empty block definitions (not initialized) */
define block CSTACK     with size = 0x200,    alignment = 8   { }; /* System (Main)
Stack */
define block EVT        with size = 0x1c8,    alignment = 512 { }; /* Exception Vector
Table <1> */
define block mheap      with size = 0x4000,   alignment = 16  { }; /* mheap tsmx */
define block heap1      with size = 0x2000,   alignment = 16  { }; /* heap1 ttCD17 */
define block heap2      with size = 0x2000,   alignment = 16  { }; /* heap2 thMH05 */
define block heap3      with size = 0x2f60,   alignment = 16  { }; /* heap3 thR */
define block cp_heap    with size = cphsz,    alignment = 16  { }; /* heap for cp */
keep  {block EVT};

/* Console partition and SVC regions */
define block cp_code    with size = cpcsz, alignment = 32
                              {ro section .cp.text, ro section .cp.rodata};
```

```
define block cp_data    with size = cpdsz, alignment = 32 {block cp_heap,
                                rw section .cp.data, rw section .cp.bss};
define block svc_code   with size = svccsz, alignment = 32
                               {ro section .svc.text, ro section .svc.rodata,
                                ro object ABImemset.o, ro object ABImemcpy.o,
                                ro object strcat.o, ro object strcpy.o,
                                ro object strncpy.o, ro object strlen.o};

/* Common umode regions <3> */
define block ucom_code  with fixed order, size = ucsz, alignment = 32
               {block svc_code, ro section .ucom.text, ro section .ucom.rodata};
define block ucom_data  with size = udsz, alignment = 32
                               {rw section .ucom.data, rw section .ucom.bss};

/* System regions <3> */
define block sys_code   with size = scsz, alignment = 32 {ro section .intvec, /*<4>*/
               ro section .sys.text, ro section .sys.rodata, block cp_code,
               block ucom_code};
define block sys_data   with size = sdsz, alignment = 32, fixed order
   {block EVT, block CSTACK, block mheap, block ucom_data, block cp_data,
    rw section .sys.bss, rw section .sys.data, rw section .sys.noinit,
    block heap1, block heap2, rw};

/* Test task regions <3> */
define block t2a_code   with size = t2acsz, alignment = 32
                             {ro section .t2a.text, ro section .t2a.rodata};
define block t2b_code   with size = t2bcsz, alignment = 32
                             {ro section .t2b.text, ro section .t2b.rodata};
define block t2c_code   with size = t2ccsz, alignment = 32
                             {ro section .t2c.text, ro section .t2c.rodata};
define block t2s_code   with size = t2scsz, alignment = 32
                             {ro section .t2s.text, ro section .t2s.rodata};
define block ut1a_code  with size = ut1acsz, alignment = 32
                             {ro section .ut1a.text, ro section .ut1a.rodata};
define block ut2a_code  with size = ut2acsz, alignment = 32
                             {ro section .ut2a.text, ro section .ut2a.rodata};
define block ut2b_code  with size = ut2bcsz,  alignment = 32
                             {ro section .ut2b.text, ro section .ut2b.rodata};
define block ut2c_code  with size = ut2ccsz, alignment = 32
                             {ro section .ut2c.text, ro section .ut2c.rodata};
define block ut2d_code  with size = ut2dcsz, alignment = 32
                             {ro section .ut2d.text, ro section .ut2d.rodata};
define block ut2s_code  with size = ut2scsz, alignment = 32
                             {ro section .ut2s.text, ro section .ut2s.rodata};
define block t2a_data   with size = t2adsz,  alignment = 32 {rw section .t2a.bss,
                                                             rw section .t2a.data};
define block t2b_data   with size = t2bdsz,  alignment = 32 {rw section .t2b.data};
define block t2c_data   with size = t2cdsz,  alignment = 32 {rw section .t2c.bss,
                                                             rw section .t2c.data};
define block t2s_data   with size = t2sdsz,  alignment = 32 {rw section .t2s.bss,
                                                              rw section .t2s.data};
define block ut1a_data  with size = ut1adsz, alignment = 32 {rw section .ut1a.data};
define block ut2a_data  with size = ut2adsz, alignment = 32 {rw section .ut2a.data};
```

```
define block ut2ax_data with size = ut2axdsz, alignment = 32 {rw section .ut2ax.data};
define block ut2b_data  with size = ut2bdsz, alignment = 32 {rw section .ut2b.data};
define block ut2c_data  with size = ut2cdsz, alignment = 32 {rw section .ut2c.bss};
define block ut2d_data  with size = ut2ddsz, alignment = 32 {rw section .ut2d.bss};
define block ut2s_data  with size = ut2sdsz, alignment = 32 {rw section .ut2s.bss,
                                                            rw section .ut2s.data};
define block tm23_data  with size = tm23dsz, alignment = 32 {rw section .tm23.data};

/* rom_block and sram_block definitions */
define block rom_block with fixed order, alignment = 32
                       {block ut1a_code, block ut2b_code,
                        block ut2a_code, block t2a_code, block t2b_code,
                        block ut2c_code, block ut2d_code, block ut2s_code,
                        block t2c_code, block t2s_code, ro};
define block sram_block with fixed order, alignment = 32
                       {block t2a_data, block t2b_data, block ut1a_data,
                        block ut2a_data, block ut2ax_data, block ut2b_data,
                        block ut2c_data, block ut2d_data, block ut2s_data,
                        block t2c_data, block t2s_data, block tm23_data};
/* Initialization */
initialize by copy with packing = none {rw};
do not initialize  {section .noinit};

/* Placements */
place at start of ROM   {block sys_code};
place in ROM            {block rom_block};
place at start of SRAM  {block sys_data};
place in SRAM           {block sram_block, block heap3};

/* Notes:
   1. size = 0x1c8 so mheap will start at 0x3d0 like STM32F746.
   2. Ensures CSTACK is first to detect main stack overflows. Also places
      critical blocks in TCRAM for speed.
   3. Alignment and size must be multiples of 32 bytes.
   4. Copied to EVT by startup code in case EVT is needed before the linker
      copy code runs.
*/
```

Appendix C: Glossary

auxiliary slot	An extra MPA slot that is swapped with an active slot or one that is stored and never put into the MPU. See section 4.2.5 Auxiliary Slots.
active slot	An MPU slot that is replaced upon a task switch.
alias handle	Secondary pointer to a control block or structure.
attributes	Access permission with share, buffer, and cache control for a region.
bound stack	A permanent task stack that is retained by a task when it is stopped.
BR	Background Region flag of ARMM7 MPU. When on, allows access to all memory.
current task	The task that is running.
fas	First active slot in MPU. Slots below this are static and are not changed by task switches.
free message portal	A portal that uses free pmsgs to exchange data between client and server.
handle	Pointer to a control block or control structure.
head	Portion of a function before calling a callee (e.g. SVCHh()).
hmode	Privileged, main stack, no tasks.
MCB	smx Message Control Block.
MCU	Micro Controller Unit: A system on a chip including processor, memory, and peripherals.
message exchange	An smx object that allows exchanging messages between tasks and between tasks and LSRs.
MMF	Memory Manage Fault.
MMU	Memory Management Unit.
MPU	Memory Protection Unit.
MpuPacker	Micro Digital utility that suggests the best linker block order to minimize gaps for ARMM7.
MS	Main Stack of processor used in hmode.
non-volatile registers	R4-R11.
one-shot task	Runs once, stops, and releases its stack, while waiting.
partition	Abstract object consisting of the union all MPU regions that the partition's tasks are allowed to access.
pblock	Protected block – i.e. the block is an MPU region.
plug block	A non-region block used to fill a gap.
pmode	Protected or privileged mode.
pmsg	Protected message – i.e. the message block is an MPU region.
portal	Allows transfer of data between isolated partitions.
ptask	A task that runs in pmode with the MPU on and BR off.
RASR	Region Attribute and Size Register in ARMM7 MPU.
RBAR	Region Base Address Register in MPU.
RLAR	Region Limit Address Register in ARMM8 MPU.
region	Area of memory defined by a starting address, size, and attributes.
region block	A linker block that becomes an MPU region.

Appendix C

region size	Next power of two large enough to contain region for ARMM7 or next multiple of 32 for ARMM8.
restricted services	System services not allowed in umode.
RO	Read-Only.
RTOS	Real Time Operating System.
RW	Read/Write.
runtime frame	Ends when idle task has run a specified number of times.
runtime limit	The number of clocks per runtime frame that a task is allowed to run.
safe LSR	An LSR that has its own stack and MPA and runs in umode (uLSR) or pmode (pLSR). Functions like a mini task, except it cannot wait.
size boundary	A multiple of size.
SOUP	Software of Unknown Pedigree.
SRD	SubRegion Disable field in RASR.
SSR	smx System Service Routine. Performs smx services such as smx_TaskCreate().
stacking	Processor saves volatile registers on the task stack due to an exception.
subregion	1/8 of a region for ARMM7. Can be individually disabled.
super region	MPU region that maps an entire memory, e.g. SRAM. An MPA with super regions can be helpful initially for a task being developed, to avoid MMFs.
system service	A service performed by system software, such as smx_SemSignal() or sb_IRQMask().
tail	Portion of a function after return from callee (e.g. SVCHt()), or the unused portion at the end of a linker block (e.g. cp_code 0x1b92 <Block tail>).
TLS	Task Local Storage.
token	Assigned to tasks to limit access to smx objects. A HI token permits creating, changing, and deleting an object. A LO token permits object access, only.
top task	Longest waiting task at the highest priority level – i.e. next to run
trusted LSR	An LSR that runs in hmode, uses the main stack, and does not have an MPA.
TS	Task Stack.
tunnel portal	Uses a pmsg, bound to the client, as the portal buffer for data and command transfers in either direction and remains in place until the portal is closed.
umode	Unprotected or unprivileged mode.
unbound stack	A temporary task stack allocated from a stack pool. Released when task is stopped.
unstacking	Processor restores registers on return from an exception.
utask	A task the runs in umode with the MPU on.
ARMM7	ARMv7-M architecture.
ARMM8	ARMv8-M architecture**.**
volatile registers	R0-3, R12, LR, and PSR.
window portal	Uses a common dynamic region as a portal.
XN	eXecute Never.

Appendix D: SMX API Limitations

Most API services are permitted in umode. Some are prohibited or limited. This appendix summarizes the limitations.

Note: These limitations could change (by us or you), so please refer to the code to be sure.

Following are **prohibited in umode**:

```
smx_EventGroupSet()
smx_Go()
smx_HeapBinSeed()
smx_HeapExtend()
smx_HeapInit()
smx_HeapRecover()
smx_HeapScan()
smx_HeapSet()
smx_HTInit()
smx_LSRCreate()
smx_LSRDelete()
smx_LSRInvoke()
smx_LSRsOff()
smx_LSRsOn()
smx_MsgXchgSet()
smx_MutexGetFast()
smx_PipeSet()
smx_SemSet()
smx_SysPseudoHandleCreate()
smx_SysPowerDown()
smx_TaskLock()
smx_TaskLockClear()
smx_TaskSet()
smx_TaskUnlock()
smx_TaskUnlockQuick()
smx_TimerSetLSR()
```

Following are **limited by relationship**. A utask may operate upon its children, its siblings, and itself, but not upon its parent or other tasks.

```
smx_BlockRelAll()
smx_MsgRelAll()
smx_TaskBump()
smx_TaskDelete()
smx_TaskLocate()
smx_TaskPeek()
smx_TaskResume()
smx_TaskStart()
```

```
smx_TaskStartNew()
smx_TaskStop()
smx_TaskSuspend()
```

Following have **indicated limitations**:

```
sb_INT_DISABLE/ENABLE(): Ignored by processor. See section 6.2.2 Interrupt Disabling and
Masking in Tasks

sb_IRQMask/Unmask(): Permitted for IRQs specified in table assigned to task’s TCB by
smx_TaskSet(,SMX_ST_IRQ,). See section 6.2.2 Interrupt Disabling and Masking in Tasks.

smx_TaskCreate(): Child inherits parent's MPA, umode flag, and IRQ permissions.

smx_TaskSet(): pmode only and:
  SMX_ST_PRIV:    only top parent privilege can be set
  SMX_ST_TAP:     only top parent tap can be set
  SMX_ST_UMODE:   only top parent can be put into umode
```

These behaviors can be changed easily.

1. To prohibit calls, change #defines in xapiu.h to use _Pragma("error"), following examples there.
2. To change limitations by relationship, modify smx_TaskOpPermit() in xtask.c.
3. To change limitations for IRQs, modify sb_IRQPermCheck() in bspm.c.
4. To change others, edit the functions.