

Achieving Device Security

A guide to partitioning connected device firmware
to achieve secure operation by Ralph Moore

© Copyright 2021-2024

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

All rights reserved.

smx and SecureSMX are Registered Trademarks of Micro Digital, Inc.

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

Contents

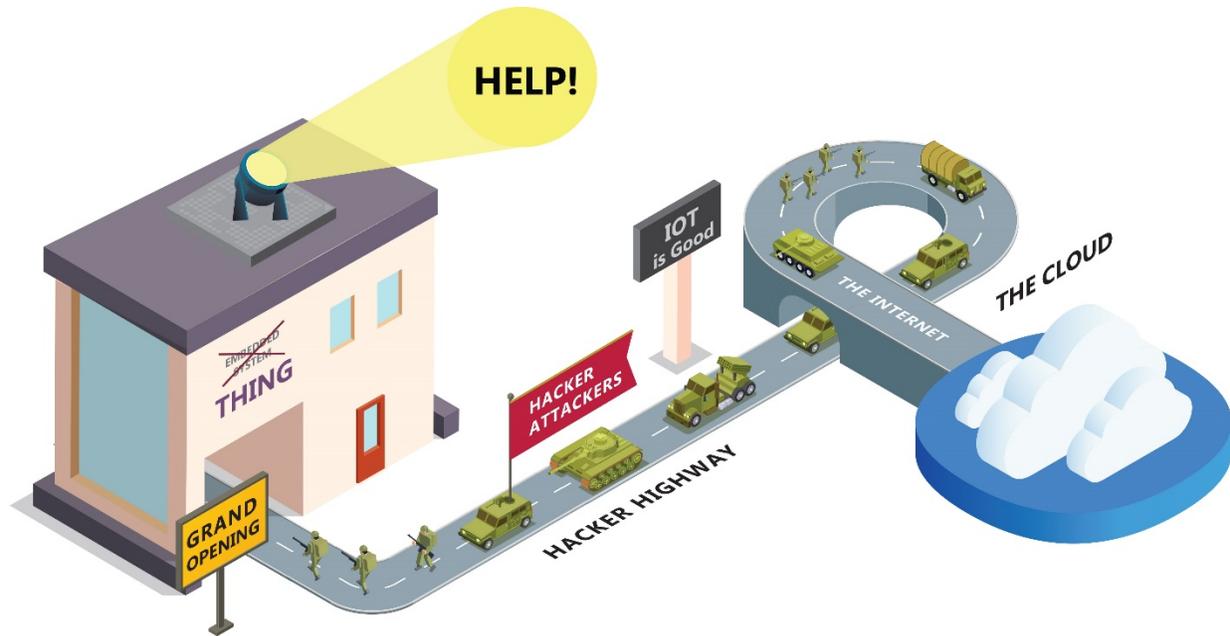
1. Introduction	1
The First Problem.....	1
The Second Problem	2
The Hardware	2
The Security Goals	3
2. Basics of Partition Isolation	4
Introduction	4
Let's Get Started	4
Advantages of Isolated Partitions.....	6
The Need to Isolate Code As Well As Data	7
Partition Definition	7
ptasks vs utasks.....	8
Summary	9
3. Partitioning Code	10
Introduction	10
MPU Operation	11
Typical MPU Regions	12
MPAs, Templates, and Tasks	13
Sections	14
Linker Command File	15
MPA Templates	16
<i>ptask template for v7M:</i>	16
<i>utask template for v7M</i>	17
<i>ptask template for v8M</i>	18
<i>utask template for v8M</i>	18
MPA Creation	18
<i>default MPA for v7</i>	19
MPU/MPA Relationship	19
Stack Regions	20
v7M Memory Gaps.....	21
v7M Memory Tails.....	22

Partition Demos	23
<i>pd0</i>	23
<i>pd1</i>	23
<i>pd2</i>	24
<i>pd3</i>	24
<i>pd4</i>	24
Summary.....	24
4. Creating Isolated Partitions.....	25
Introduction	25
System Services from umode	25
SVC Shell Functions	26
Custom SVC Shell Functions.....	28
Multiple Heaps	30
Protected Blocks and Messages.....	31
<i>pblocks</i>	31
<i>pmsgs</i>	31
Summary.....	32
5. The Need for Portals	33
Introduction	33
Free Message Portals.....	34
<i>Portal Operation</i>	34
<i>Portal Initialization</i>	36
Tunnel Portals.....	37
Shell Functions	38
Tunnel Portal Timeouts.....	44
Summary.....	44
6. Partition Limitations.....	45
Introduction	45
Runtime Limiting	45
Tokens	47
The ISR Problem	49
Summary.....	51
7. Advanced Features	52
Parent/Child Tasks.....	52
Auxiliary Slots	53
Dynamic Slots	54
Multi-task Partition Templates.....	55

Critical Code Sections.....	55
Scheduler Callbacks	56
smxAware	57
Event Monitoring.....	57
Porting Applications to SecureSMX	58
Frameworks	58
Debugging.....	59
MPUMapper.....	60
8. Conclusion	61
Appendix A eheap.....	63
Introduction	63
Doubly Linked Chunks	63
Heap Bins	64
Large Bin Sorting	65
Merge Control	65
Heap Recovery	65
Aligned Blocks	66
Region Blocks	67
Chunk Types.....	68
Integrated Block Pools	68
The Need for Mutexes	69
Summary.....	69
Appendix B smxAware	70
MPU Display	70
MPA Displays.....	73
Finding Memory Manage Faults (MMFs)	76
Event Display	77
Timeline Display.....	79
Summary.....	79

Figures

Figure 2.1 Typical Embedded System Structure	4
Figure 2.2 Secure Network Solution.....	5
Figure 3.1 MPU Operation	11
Figure 3.2 Typical MPU Regions.....	12
Figure 3.3 Templates, MPAs, and Tasks Relationship.....	13
Figure 3.4 MPU/MPA Relationship.....	20
Figure 3.5a Before MPUPacker	21
Figure 3.5b After MPUPacker	21
Figure 4.1 System Service Calls	26
Figure 4.2 Multiple Heaps	30
Figure 4.3 Protected Message	32
Figure 5.1a Server Function API	33
Figure 5.1b Portal API	33
Figure 5.2 Free Message Protocol Configurations	34
Figure 5.3 Free Message Portal Initialization	36
Figure 5.4 Tunnel Portal Operation	37
Figure 5.5 Portal Operation	38
Figure 6.1 Runtime Limiting	46
Figure 6.2 Tokens Controlling Semaphore Access	47
Figure 6.3 Safe LSR Operation	50
Figure 7.1 Parent/Child Tasks.....	52
Figure 7.2 Swapping IO Regions	54
Figure 7.3 Multiple MPAs from One Template	55



1. Introduction

The First Problem

Complacency is biggest problem in device security. So far, many devices have been hacked, but hackers have focused mainly on the low-hanging fruit of phishing and similar tactics to gain entry into computer networks. However, this low-hanging fruit is beginning to disappear as those companies with large networks adopt better security software and better security practices. Consequently, the next main target is likely to be the thousands of unprotected devices already connected to computer networks.

Furthermore, large governments around the world have large stockpiles of zero-days ready to activate without warning. These and insider attacks are unknown threats, but are likely to be more device-oriented.

OEM executives need to decide if they want to start taking prudent steps now or to hire an army later to deal with attacks on their devices. Courts could well decide that inadequate security provisions constitute negligence, and therefore, device manufactures are on the hook for damages due to their devices being hacked [Ref 5]. This could put many OEMs out of business. Even if this does not happen it is likely that provable security will become an important sales point for many types of devices.

Looking at the broad software picture, it is likely that only about 10-20% of the software in a device does the mission-critical work. This code is probably carefully written, thoroughly tested, field-proven, and thus unlikely to change. The remaining 80-90% of the code is a mixture of third-party, open source, and newly developed code, and it is

Achieving Device Security

probably filled with vulnerabilities. Without protection, a hack anywhere in this code exposes the entire system, thus facilitating ransomware attacks and theft of critical data.

Also, it should be anticipated that insider attacks will become more prevalent. Mission-critical code should be locked away and inaccessible to all but a few highly-trusted employees. Since it may need no security nor functional updates, this code should not be included in updates so it cannot be tampered with, which gives some hope of averting catastrophic insider attacks.

The Second Problem

Overconfidence is the second problem. The common image of a lone hacker working while sitting on his bed in pajamas is not useful. Hacker organizations are well-staffed, well-funded, and well-equipped. More likely the hacker is wearing tasteful business attire and working in a large, air-conditioned office, alongside an army of coworkers. Even large development teams are seriously out-gunned. For these teams to think that they are smarter, can hide the flaws in their code, or that there is safety in small probabilities is sheer folly. Hackers are not only smart, they use computers, even super-computers, to try billions of attacks until they find one that works. Governments and crime syndicates spend millions of dollars to find zero-days that can cause great damage.

No doubt your team is able to write small amounts of hacker-resistant code. But to write a whole system that way is clearly impractical – it takes too long and costs too much. We need a better methodology for writing embedded system code. What is needed is baked-in security that protects against all kinds of attacks. Neither Linux nor wimpy RTOSs are likely to achieve this objective. However, we believe that our new, high-security RTOS, SecureSMX[®], can do it. The purpose of this ebook is to present the features contained in SecureSMX that are necessary to achieve this objective.

The Hardware

SecureSMX supports Cortex-M because it is the most popular microcontroller architecture and it has some good security features:

- Three Processor Modes:
 - Handler (hmode)
 - Privileged (pmode)
 - Unprivileged (umode)
- Memory Protection Unit (MPU)
- Supervisor Call (SVC N)
- PSLIM & MSLIM (v8)
- TrustZone (v8) is not used
- Both v7 and v8 are supported

Cortex-M processors have three modes of operation: Handler mode, or hmode, can be entered only via an exception or an interrupt. Privileged mode, or pmode, can be entered from hmode or on startup. Unprivileged mode, or umode, can be entered only via an exception return from the other two modes. umode is most secure because it cannot do certain system functions, such as turn off the MPU. Cortex-M MPUs typically have 8 slots, each holding a *region*. A region enforces a memory range and attributes on all memory accesses. The supervisor call permits system service calls from umode. PSLIM and MSLIM provide overflow detection for task stacks and the main stack. The security methods implemented by SecureSMX do not require TrustZone.

The Security Goals

The following goals have guided the development of SecureSMX:

- Upgrade existing devices with minimal code changes
- Provide a security framework for new devices
- Achieve fully isolated partitions
- Hardware-based security
- Protect the system from compromised partitions
- Flexible set of tools
- Consistent solutions
- Wide applicability

Fixing the billions of devices already in use is of paramount importance, even if they cannot be updated in the field. Units in the field can be replaced when necessary and new units coming off the factory floor will be more secure. The process should support iterative improvement. Starting with a security framework that runs like the final system allows ironing out inter-module problems in advance and bakes in security. Fully isolated partitions are necessary to stop hackers. Security must be hardware-based, else hackers will just program around it. Malware can do considerable damage from inside of a partition, so limitations on code running it are necessary to protect the system. Flexibility in the application of security tools is very important. For example, we should not runtime limit the task that is preventing a crane from tipping over! Consistency and wide applicability are hallmarks of a good set of tools.

2. Basics of Partition Isolation

Introduction

To me, achieving *full partition isolation* is the Holy Grail of Microcontroller Unit (MCU) system security, because a hacker cannot access data and code outside of a partition from inside a partition that is fully isolated from the rest of the system. Achieving full isolation between *processes* using Memory Management Units (MMUs) is relatively easy but requires power-hungry processors to achieve acceptable process switching times, and it is not appropriate at the task-level, anyway. Achieving full partition isolation for MCUs using Memory Protection Units (MPUs) is possible, but comes with a higher level of difficulty.

This ebook discusses how to achieve full partition isolation in MCU-based systems. Many papers have been written concerning MPUs. Ref. 1 is a particularly well-written introduction to the subject by Jean Labrosse, and I recommend that you read it ahead of this ebook, as an introduction to MPU concepts. Refs. 2 and 3 are also helpful. The main problem with Ref. 1 is that it does not go far enough to achieve full partition isolation. However its contents are reviewed, in a few places here, in order to illustrate the consequences of different approaches to MPU usage and other aspects of partition isolation.

Let's Get Started

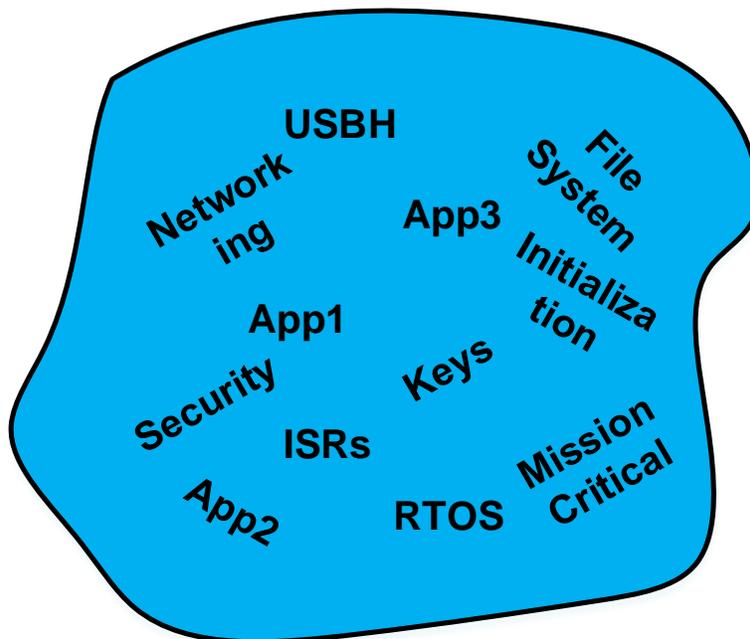


Figure 2.1 Typical Embedded System Structure

Figure 2.1 shows the structure of a typical embedded system. There is no structure, and there are no partitions. If a hacker breaks in anywhere he has access to everything – to the keys, to critical data, etc. This is not good.

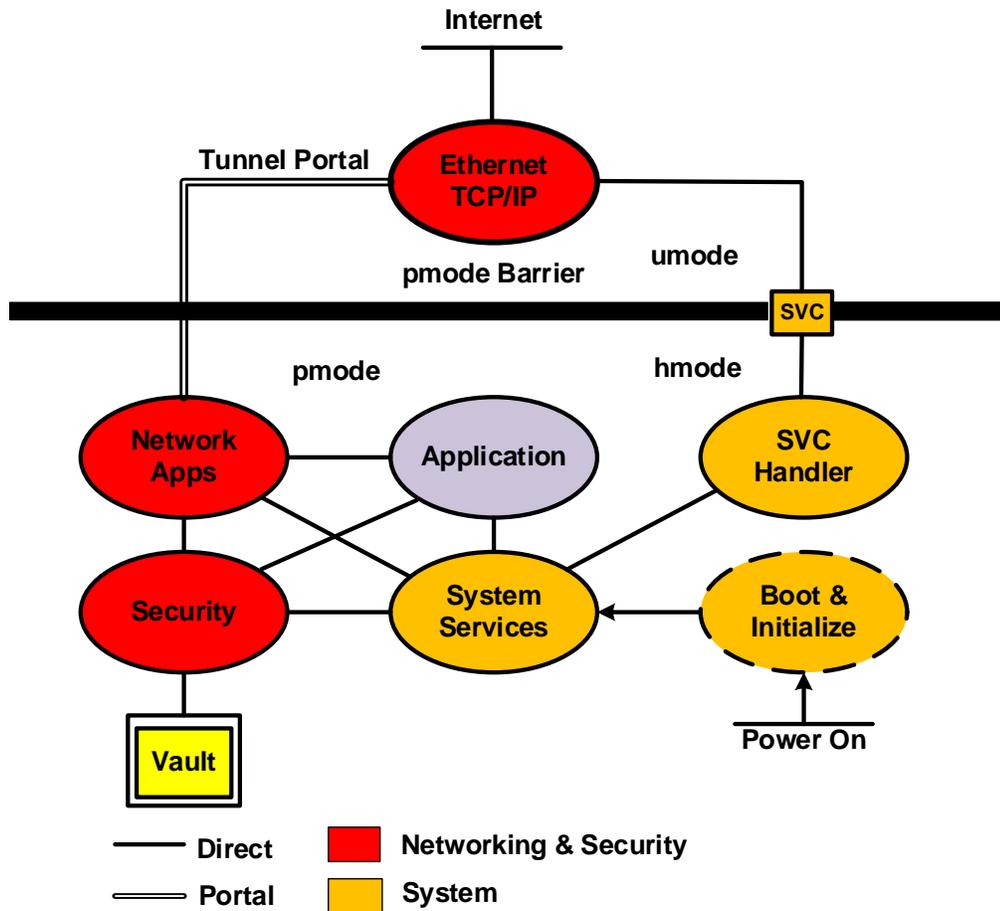


Figure 2.2 Secure Network Solution

Figure 2.2 shows a solution to the problem of safely adding networking to an existing, defenseless embedded system. It is shown here in order to discuss the basics of system partitioning – see Ref. 3 for more information on this solution.

Above the heavy line is umode. Below the heavy line is pmode and hmode. We call the heavy line the *pmode Barrier*, because it is enforced by the processor, and umode code cannot break through it, except via SVC exceptions. As shown, the Ethernet driver plus TCP/IP stack, both of which are vulnerable to hacking, are in a umode partition. This partition sees only encrypted data passing in or out. It connects to the Network Apps partition via a *tunnel portal*, and it obtains system services via the SVC exception, both of which strongly limit what can be done from inside this partition. Note that only the Security partition connects to the Vault, which contains the keys and other proprietary information of the system owner.

Achieving Device Security

In the above diagram, the entire embedded application code has been grouped into a single pmode partition and thus it is also protected from an intrusion via the Internet. Figure 2.2 is a solution for legacy systems and is not typical for new systems, which are the main subject of this ebook.

Advantages of Isolated Partitions

Dividing embedded system software into isolated partitions has many benefits:

- Protection against unpatched vulnerabilities.
- Protection against zero-days.
- Protection against insider attacks.
- Double protection of mission-critical code.
- Hardware enforced separation of privileged and unprivileged code.
- Hardware controlled access to system services, data, and I/O registers.
- Higher reliability and safety.
- Easier incorporation of legacy software.
- Hardware enforcement of the good programming practices of modular code with well-defined interfaces.
- Immediate detection of wild pointers and stack and buffer overflows.
- Strong development framework for new systems:
 - Protection from other people's bugs.
 - Well defined interfaces (portals).
 - Agile coding & CI/CD at the partition level.
 - Debugging in a full system environment.
- Partition-only reboot and shutdown to minimize system interruption.
- Partition-only updates to minimize code exposure and system interruption.
- Support for interchangeable modules.

Security and reliability are two sides of the same coin. In the first, hacks are deliberate; in the second, bugs and malfunctions are accidental. Both can damage property and cause injury. Measures that improve one tend to improve the other. Hardware enforcement of full isolation enables interchangeability of modules within a system and better module reusability in future systems. Hardware enforcement of better design practices also helps to improve system security and reliability. Partial reboots save time and may allow normal operation to not be interrupted. Partial updates save time and avoid exposing partitions not being updated. These are all good reasons for partitioning.

But it should be emphasized that partitioning does not replace other security measures such as:

- Encryption
- Authentication
- Least privilege

- Root of trust
- Firewalls
- Best coding practices
- Bounds checking of inputs
- Monitoring for anomalous behavior

Multiple layers are still necessary for effective security. But if a hacker penetrates these security layers, partitioning will limit his ability to go further into the system in order to steal sensitive data or to cause damage. Hence partitioning is an important part of a strong security strategy.

The Need to Isolate Code As Well As Data

There seems to be some controversy concerning whether or not the code of each partition should be isolated from the code of other partitions. For example Ref. 1 states:

“Because of the fairly limited number of regions available in an MPU, regions are generally set up to prevent access to data (in RAM) and not so much to prevent access to code (in flash).”

A competitive golfer must assume that his opponent will sink the putt, whatever the distance. An analogous situation exists here – you must assume that a hacker knows where each of your functions is and what it does. Using this knowledge he can wreak havoc upon your system simply by calling your functions at inappropriate times or with inappropriate parameters. If, on the other hand, he can only access code within the partition that he has penetrated, then he can only damage that partition. Therefore, code must be isolated in partitions, just like data.

Next we will cover how to define partitions and how to manage the task or tasks within them.

Partition Definition

Partitions typically are subsystems or modules that perform specific functions, e.g. file systems, networking stacks, data acquisition, etc. In a new system, we would like to see as much application, middleware, and driver code put into umode partitions, as possible. (As previously noted, this is not likely to be practical for legacy systems.)

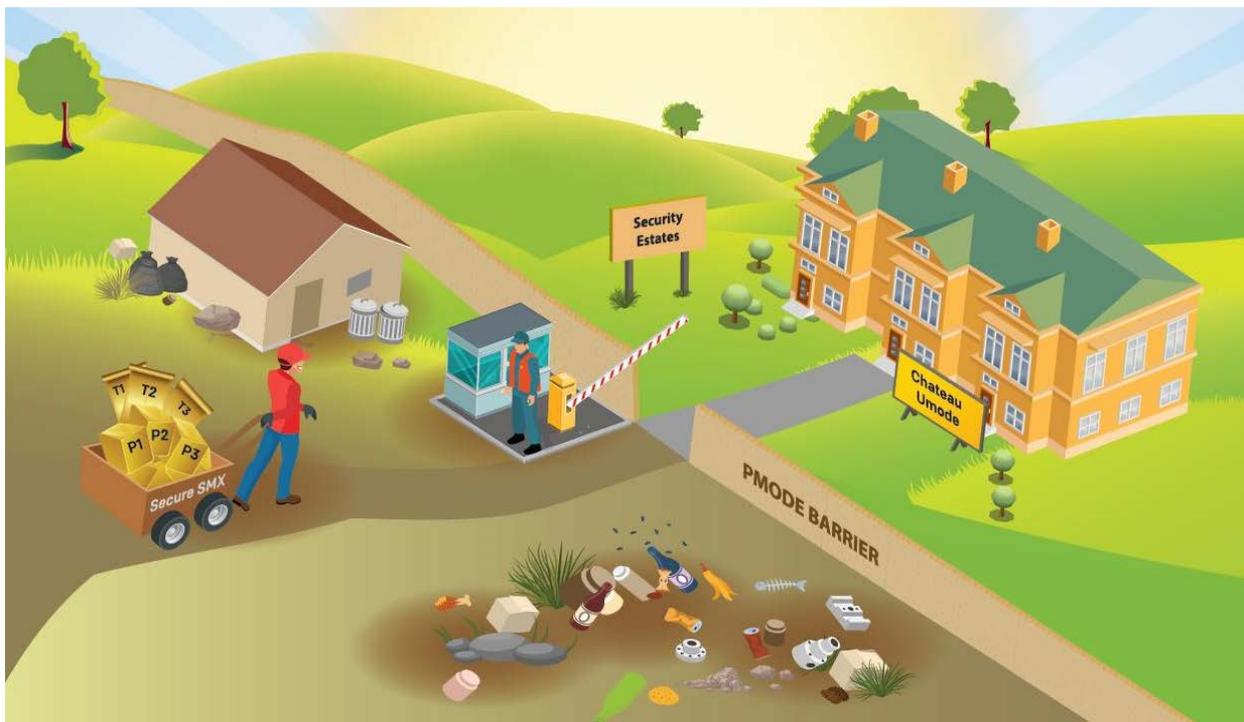
A partition must contain at least one *main task*. It may contain other *helper tasks*. When defining a new partition, it is a good idea to list all regions that the partition will need. This may reveal an MPU overflow problem. One technique to deal with MPU overflow is to divide the partition regions among the main task and one or more helper tasks. For example, an IO task might be created and assigned the IO regions along with a code region and a data region that it shares with the main task. Then the main task would not need the IO regions to be in the MPU when it runs.

Achieving Device Security

This might be the standard solution for the MPU overflow problem, except that tasks use a lot of memory. A typical task needs at least 100 bytes for its Task Control Block (TCB) and at least 500 bytes for its stack. Very simple tasks might have smaller stacks, but subroutine parameters and auto variables tend to eat up stack quickly, thus necessitating fairly large stacks for most tasks. Consequently, task stacks impose a significant memory requirement. For good performance, the TCB and the task stack should come from on-chip SRAM, but it is in short supply in most MCUs.

A solution to this is the *one-shot task*. This type of task does not have an infinite loop in its code. Instead it receives a stack from a stack pool, runs straight through its code, then stops and returns the stack to the stack pool. This is possible because it has no need to store information between runs. One-shot tasks are a good fit for helper tasks and server tasks. For example, the IO task referred to above needs to run only when its IO operation is required. For example, it might output infrequent messages to a console. While waiting for the next message, it does not require a stack. It often happens, due to the system design, that not all one-shot tasks can run at the same time, or that it is ok for some to wait for stacks. Consequently many one-shot tasks may be able to share just a few stacks, thus saving a large amount of SRAM.

ptasks vs utasks



utasks provide a higher level of security than ptasks because the MPU cannot be changed from umode. In pmode, a hacker is only one instruction away from turning off the MPU and taking control of the whole system. However, ptasks have equal reliability protection

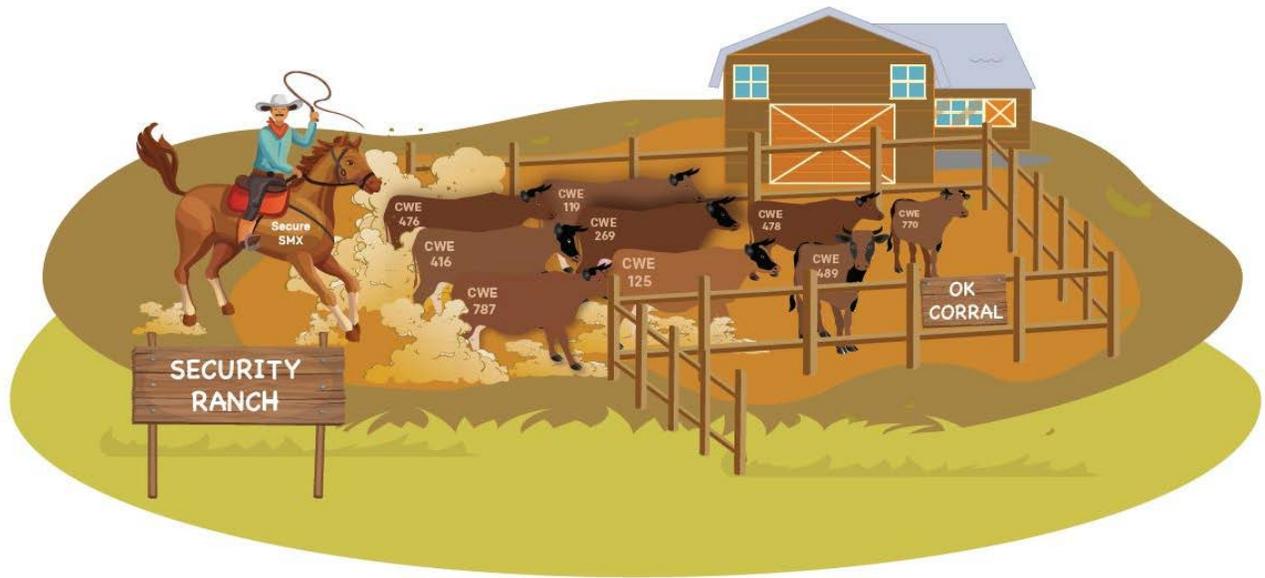
to utasks, and it is not always possible to implement mission-critical functions in umode due to its lower performance and restrictions. In particular, all ptasks have *sys_code* and *sys_data* regions, which allow them to make direct, unfiltered calls for system services.

As shown in Figure 2.2, utasks must use the SVC exception for system services, which is much slower and more restrictive (services that could cause system damage are not permitted). In addition, interrupts cannot be disabled nor enabled from umode, which is essential for drivers and some low-level code.

Therefore, the importance of ptasks should not be overlooked. In fact, if all vulnerable partitions have been put into umode, then the umode partitions and the pmode barrier will provide strong protection against hacking ptasks from umode.

Summary

In the foregoing we have examined the need for and the advantages of partitioning embedded system software. In addition, we have explored the uses of pmode and umode and the relationship of tasks to partitions.



3. Partitioning Code

Introduction

In this chapter we examine the strategies and techniques to effectively manage MPUs and solutions to the MPU overflow problem. The references at the end of this ebook are recommended if you are not familiar with the Cortex-M MPUs.

MPU Operation

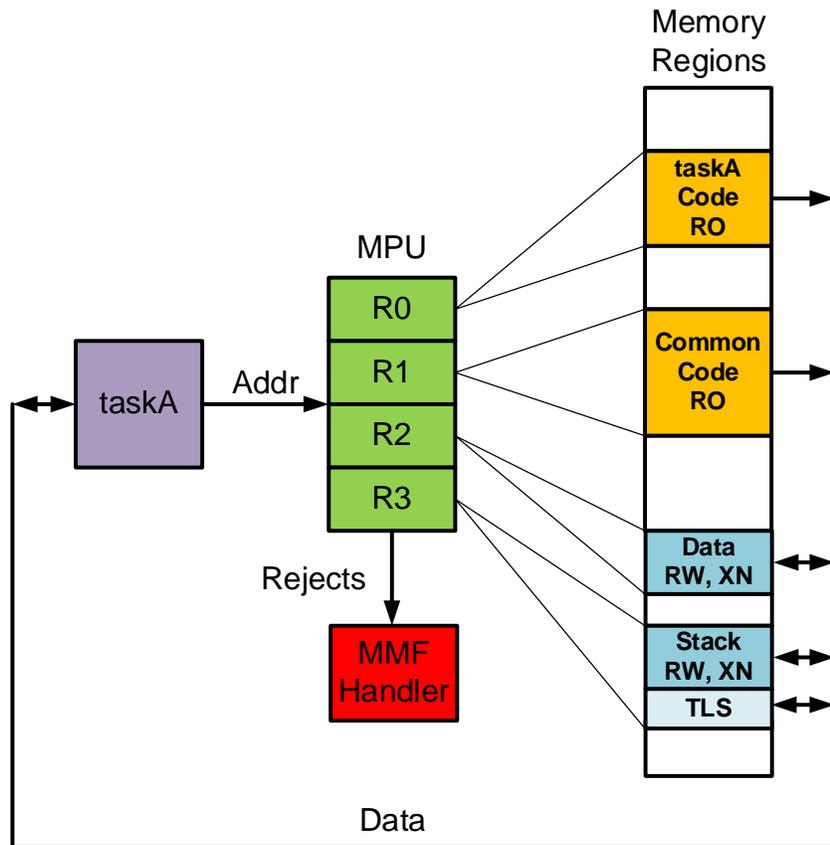


Figure 3.1 MPU Operation

Figure 3.1 shows how an MPU operates. taskA must access memory through the MPU. Region R0 allows taskA to access taskA read-only code. R1 allows taskA to access common read-only code. R2 allows taskA to access read/write, execute-never data. R3 allows taskA to access its read/write, execute-never stack and optional task local storage (TLS). The latter permits task-specific data storage without requiring another data region for it. If an address is not within the ranges shown or has the wrong attributes such as !XN for a data access, the Memory Manage Fault (MMF) handler is triggered and takes control of the system, expelling the hacker.

Typical MPU Regions

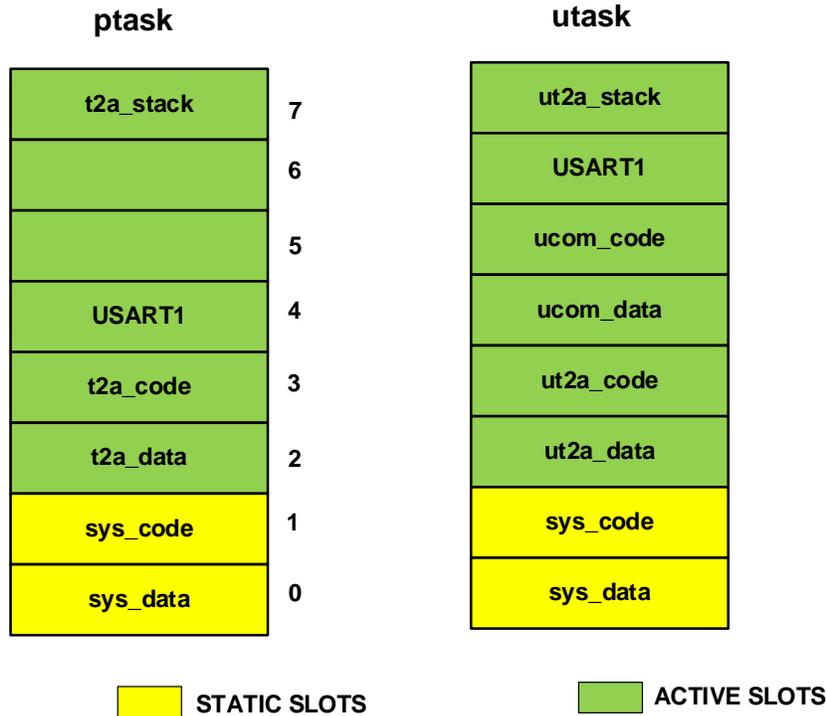


Figure 3.2 Typical MPU Regions

Figure 3.2 shows typical MPU regions for a ptask and a utask. *Static slots* are loaded once, during initialization. *Active slots* are loaded each time a task switch occurs. The combined slots determine what regions a task can access.

Starting with the ptask on the left, *sys_code* means system code. It contains the RTOS, system services, exception handlers, ISRs, and ucom_code. *sys_data* contains system objects such as TCBs, privileged data, the main heap, and ucom_data. Above these are the task t2a_code and t2a_data regions, a region for USART1, two empty regions, and the t2a_stack region.

The utask on the right, also has *sys_code* and *sys_data*. However, it cannot access them because they are privileged. They are needed by exceptions and interrupts, which switch the processor to hmode in order to run exception handlers and ISRs. Above these are the task ut2a_code and ut2a_data, ucom_code and ucom_data, and USART1 and ut2a stack regions. ucom code holds common code between utasks such as C library functions and system service shell functions. ucom_data contains common data, if any. Certain common code is acceptable, but it is preferable that there be no common data. The ucom regions are not necessary for ptasks because they are contained in *sys_code* and *sys_data*.

MPAs, Templates, and Tasks

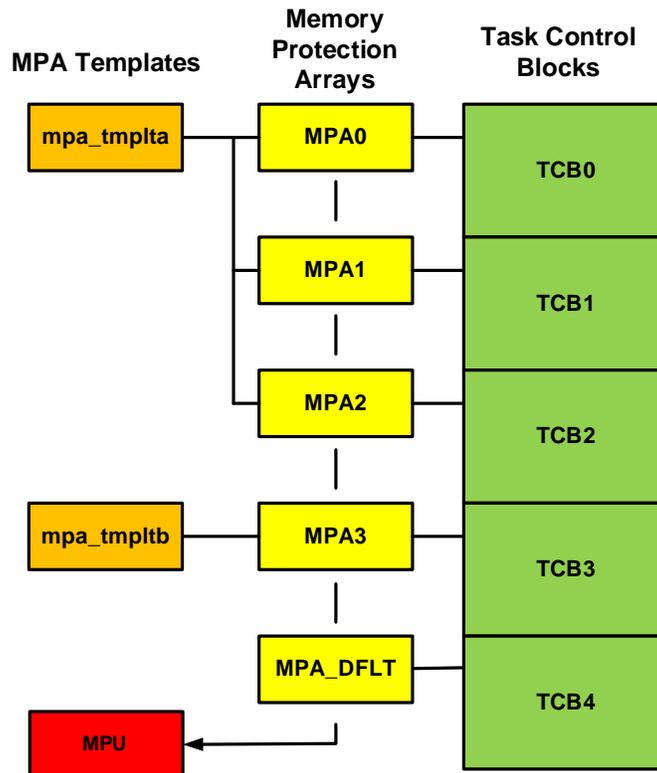


Figure 3.3 Templates, MPAs, and Tasks Relationship

Figure 3.3 shows the relationship between *MPA templates*, *Memory Protection Arrays (MPAs)*, and *Task Control Blocks (TCBs)*. Each task, represented here by its TCB, has an MPA. When the task starts running, the active regions in its MPA are loaded into the MPU, as shown.

Note that an MPA template can apply to a single MPA or it can be shared among multiple MPAs. Normally, in the latter case, the tasks would be in the same partition. There is also a *default MPA* that does not require a template. When a task is created, it is given a pointer to the default MPA. During debug, the default MPA might allow access to all memory, whereas during operation it might allow little or no memory access. The sections that follow discuss how to create MPA templates.

Sections

The first step in defining templates is to define *sections*, which compose *regions*. To do this it is not necessary to reorganize modules that include code or data from other partitions. For this, *pragmas* can be used, as follows¹:

```
#pragma default_function_attributes = @ ".t2a_text"
void t2a_main(void)
{
    ...
}
#pragma default_function_attributes =
```

Similarly for data:

```
#pragma default_variable_attributes = @ ".t2a_data"
u32      irq;
TCB_PTR  parent;
u32      tskctr;
#pragma default_variable_attributes =
```

Section pragma definitions for the same section that are scattered throughout the code are merged by the linker into a single section.

If modules get too messy with pragmas everywhere, code and data for a partition can be brought together into one or more C files for that partition. Then, *command line options* can be used, instead of pragmas. For example, in the project window for the file(s) select: Options, C/C++ Compiler, Extra Options, and enter the following:

```
--section .bss=.t2a_bss
--section .data=.t2a_data
--section .text=.t2a_text
--section .rodata=.t2a_rodata
--section .noinit=.t2a_noinit
```

This simply renames the standard section names assigned by the compiler, provided that *Override inherited settings* is checked. If several modules for a partition are grouped into a *project file node*, then the above need be put only into the options for that node. Or, instead of the above, put the following into Extra Options:

```
-f $PROJ_DIR$\\..\\..\\..\\CFG\\t2a.xcc
```

and put the above command lines into a new file, t2a.xcc.

¹ All code and directions are for the IAR EWARM tool suite.

Linker Command File

The second step in defining templates is to define *region blocks* in the linker command file. These are composed of the sections previously defined. For example, for v7M²:

```

/* region sizes (must be power of two) */
define exported symbol t2acsz    = 0x1000;
define exported symbol t2adsz    = 0x200;
define exported symbol romsz     = 0x80000;
define exported symbol sramsz    = 0x40000;

/* task code regions */
define block t2a_code with size = t2acsz*7/8, alignment = t2acsz
    {ro section .t2a_text, ro section .t2a_rodata};
/* task data regions */
define block t2a_data with size = t2adsz*6/8, alignment = t2adsz
    {rw section .t2a_bss, rw section .t2a_data};

/* optimized block order using MPUPacker */
define block rom_block with fixed order, size = romsz*5/8,
    alignment = romsz {block sys_code, block
    ut2c_code, block t2a_code, block ut2a_code,
    block ut2s_code, block t2b_code, block t2s_code,
    block ut2d_code, block ut2b_code, block t2c_code,
    block ut1a_code, ro};
define block sram_block with fixed order, size = sramsz*8/8,
    alignment = sramsz{block sys_data, block
    ut2c_data, block ut1a_data, block t2a_data,
    block ut2s_data, block ut2a_data, block t2b_data,
    block t2s_data, block ut2ax_data, block
    ut2d_data, block t2c_data, block ut2b_data, block
    tm23_data, block heap3x};

/* placements */
place in ROM          {block rom_block};
place in SRAM        {block sram_block};

```

Region sizes are defined at the top. For v7M, these must be powers of 2. If using hex numbers, only one digit can be non-zero and it must be 1, 2, 4, or 8, as shown above. Next the t2a_code and t2a_data region blocks are defined. These become MPU regions. Note that the actual region block sizes are 5/8, 6/8, 7/8, or 8/8 times the region sizes defined previously. This utilizes *subregion disables* for the last 3, 2, 1, or 0 subregions, respectively. The region blocks are aligned on the region sizes, then the sections for each region block are listed within the { }. The region blocks are placed in rom_block and

² v7M is shorthand for ARMv7-M architecture, and v8M is shorthand for ARMv8-M architecture.

Achieving Device Security

sram_block super blocks in order to control their locations in memory. The order of blocks is optimized to minimize memory waste by using our MPUPacker utility, which is discussed below. Finally, rom_block is placed in ROM and sram_block is placed in SRAM. The code shown above is just a small part of an actual linker command file.

For v8M, things are simpler and more efficient, as shown by this reduced example:

```
define exported symbol t2acsz = 0xB00;
define exported symbol t2adsz = 0xA0;
...
define block t2a_code with size = t2acsz, alignment = 32
    {ro section .t2a_text, ro section .t2a_rodata};
define block t2a_data with size = t2adsz, alignment = 32
    {rw section .t2a_bss, rw section .t2a_data};
```

In this case, the sizes need only be multiples of 32 (0x20) and the region blocks need only be aligned on 32, as shown. The rest of the linker command lines are the same.

Obviously, v8M is much more memory efficient than v7M. For example, t2a code is $0x1000 * 6/8 = 0xC00 = 3072$ bytes for v7M vs $0xB00 = 2816$ bytes for v8M. However, we have several other methods to improve v7M memory efficiency.

MPA Templates

The last step in defining MPA templates is shown in the following examples. These are consistent with Figure 3.2.

ptask template for v7M:

```
#pragma section = "sys_code"
#pragma section = "sys_data"
#pragma section = "t2a_code"
#pragma section = "t2a_data"

extern u32 scsz;
extern u32 sdsz;
extern u32 t2acsz;
extern u32 t2adsz;

MPA mpu_static =
{
    RGN(0 | RA("sys_data") | V, PDATARW | SRD("sys_data") | RSI("sys_data") | EN, "sys_data"),
    RGN(1 | RA("sys_code") | V, PCODE | SRD("sys_code") | RSI("sys_code") | EN, "sys_code"),
};

MPA mpa_tmplt_t2a =
{
    RGN(2 | RA("t2a_data") | V, DATARW | SRD("t2a_data") | RSIC(t2adsz) | EN, "t2a_data"),
    RGN(3 | RA("t2a_code") | V, CODE | SRD("t2a_code") | RSIC(t2acsz) | EN, "t2a_code"),
    RGN(4 | 0x40011000 | V, IOR | (0x9<<1) | EN, "USART1"),
};
```

In the above, #pragma section gives access in the code to section names, such as “t2a_code”, that were defined in the linker command file as region block names and exported from it. The region sizes, such as t2acsz, were also defined in the linker command file and exported from it.

RGN, *RA*, *SRD*, and *RSIC* are macros that generate the fields required for the MPU registers *RBAR* and *RASR*. The third field (e.g. "t2a_data") allows assigning a name to each region for use during debugging. It is not loaded into the MPU. Note: the fields are separated by commas that are hard to see.

t2a is a ptask, so the static regions, *sys_code* and *sys_data*, are necessary for it to obtain system services via direct function calls. *sys_code* contains the RTOS and other system service code, and *sys_data* contains the control blocks (e.g. TCBs) and the globals needed by the RTOS and other system services. These are static regions loaded into MPU slots 0 and 1 during initialization. Note that P in PDATARW and PCODE means that these regions are privileged.

Next is the t2a active region template, starting with t2a_data and t2a_code, which are defined above. Note that the IO region, USART1, is defined with numbers. This is because IO regions are at fixed locations in memory and are not defined in the linker command file. Regions above 4 are not being used and they are loaded with 0's by *smx_MPACreate()*. Region 7 is reserved for the t2a stack. If it is a permanent stack, its region is loaded from the TCB into the MPA when the MPA is created. If it is a temporary stack, its region is created and loaded into the MPA and the MPU when t2a is started.

utask template for v7M

```
MPA mpa_tmplt_ut2a =
{
    RGN(2 | RA("ut2a_data") | V, DATARW | SRD("ut2a_data" | RSI("ut2a_data") | EN, "ut2a_data"),
    RGN(3 | RA("ut2a_code") | V, CODE | SRD("ut2a_code") | RSI("ut2a_code") | EN, "ut2a_code"),
    RGN(4 | RA("ucom_data") | V, DATARW | SRD("ucom_data") | RSI("ucom_data") | EN, "ucom_data"),
    RGN(5 | RA("ucom_code") | V, CODE | SRD("ucom_code") | RSI("ucom_code") | EN, "ucom_code"),
    RGN(6 | 0x40011000 | V, IOR | (0x9 << 1) | EN, "USART1"),
};
```

The only differences from the t2a template are that *sys_data* has been replaced with *ucom_data*, and *sys_code* has been replaced with *ucom_code* and the task is ut2a instead of t2a. Note that *sys_data* contains *ucom_data*, and *sys_code* contains *ucom_code*, so ptasks do not need regions for them.

Achieving Device Security

ptask template for v8M

```
MPA mpa_tmplt_t2a =
{
    RGN(0, RA("sys_data") | PDATARW, RLA("sys_data") | AI(0) | EN, "sys_data"),
    RGN(1, RA("sys_code") | PCODE, 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, 0x40011000 | IOR, 0x40011FFF | AI(1) | EN, "USART1"),
};
```

The v8M MPU register structure is simpler. However the v8M constraint against overlapping regions makes static regions difficult to define, so the t2a template includes `sys_data` and `sys_code`. Hence, task switching will be a little slower for v8M. Note that P in PDATARW and PCODE means that these regions are privileged.

utask template for v8M

```
MPA mpa_tmplt_ut2a =
{
    RGN(0, RA("sys_data") | PDATARW, RLA("sys_data") | AI(0) | EN, "sys_data"),
    RGN(1, RA("sys_code") | PCODE, RLA("sys_code") | AI(0) | EN, "sys_code"),
    RGN(2, RA("ut2a_data") | DATARW, RLA("ut2a_data") | AI(0) | EN, "ut2a_data"),
    RGN(3, RA("ut2a_code") | CODE, RLA("ut2a_code") | AI(0) | EN, "ut2a_code"),
    RGN(4, 0x40011000 | IOR, 0x40011FFF | AI(1) | EN, "USART1"),
};
```

The only difference from the ptask template is that t2a is replaced with ut2a. `sys_data` and `sys_code` are still present, but ut2a cannot access them because they are privileged.

MPA Creation

```
t2a = smx_TaskCreate(t2a_main, PRI2, 300, HEAP0, NO_FLAGS, "t2a");
if (ARMM7)
{
    mp_MPACreate(t2a, (MPA*)&mpa_tmplt_t2a, 0x7, 6);
}
else
{
    mp_MPACreate(t2a, (MPA*)&mpa_tmplt_t2a, 0x1F, 8);
}
smx_TaskStart(t2a, 0);
```

Here t2a is created followed by creating its MPA with the `mpa_tmplt_t2a` template. In the ARMM7 case, 0x7 means to take the first 3 regions and that 6 is the MPA size. In the ARMM8 case, 0x1F means to take the first 5 regions and 8 is the MPA size.

default MPA for v7

```
MPA mpa_dflt =
{
  RGN(2 | RA("rom_block") | V, PCODE | SRD("rom_block") | RSI("rom_block") | EN, "rom_block"),
  RGN(3 | RA("sram_block") | V, PDATARW | SRD("sram_block") | RSI("sram_block") | EN, "sram_block"),
  RGN(4 | RA("ram_block") | V, PDATARW | SRD("ram_block") | RSI("ram_block") | EN, "ram_block"),
  RGN(5 | 0x40000000 | V, PIOR | RSIN(0x80000) | EN, "IO Regs"),
};
```

This is a default MPA for v7M. Note that the `rom_block`, `sram_block`, and `ram_block` cover all memory that is in use. All accesses outside of these regions will cause MMFs. Also, if the attributes are wrong (e.g. trying to execute from `sram_block`) an MMF will occur. So even this default MPA helps to catch errors.

The static `sys_data` and `sys_code` regions are still present in MPU slots 0 and 1, but are also included in `sram_block` and `rom_block`. v7 allows regions to overlap, but v8 does not. This MPA is useful when debugging of a partition starts. Later a more restrictive template will be assigned to the partition. For released code, `mpa_dflt` might be reduced to no accesses permitted. This would block a task without an MPA from running.

MPU/MPA Relationship

Figure 3.4 shows the relationship between the MPU and an MPA. The *static slots* are loaded once during system initialization. These are likely to contain privileged regions such as `sys_code` and `sys_data`, in order to allow ISRs and exception handlers to run without Background Region, BR,³ on. Static slots are quite likely in 16-slot MPUs, but not in 8-slot MPUs, due to the need for more active slots.

³ Background Region, BR, enables access to all memory in hmode and pmode. If `sys_code` and `sys_data` are not in the MPU, BR must be on in umode in order for ISRs and exception handlers to run when they interrupt. BR has no effect in umode.

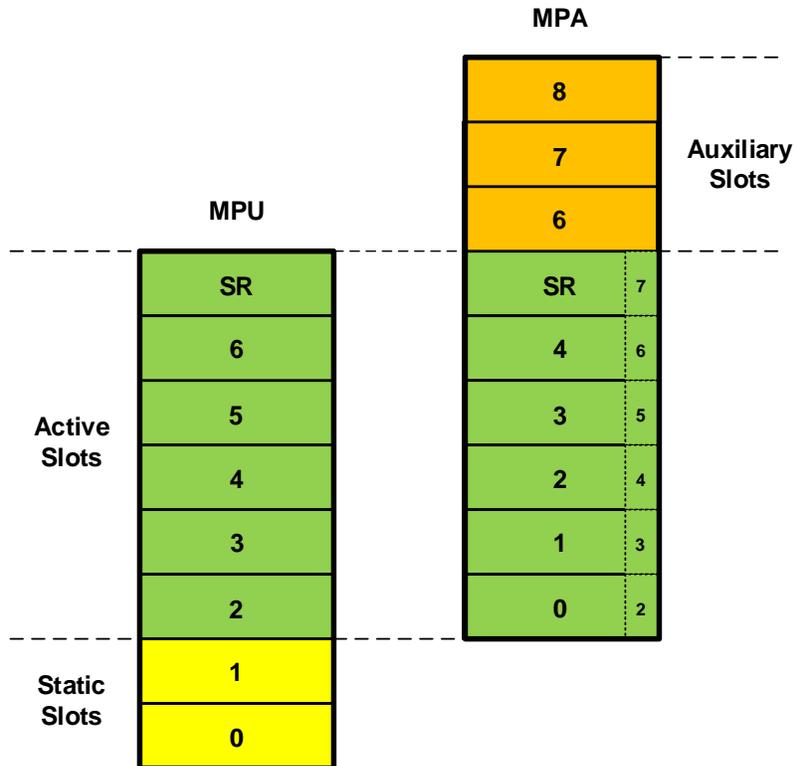


Figure 3.4 MPU/MPA Relationship

The active MPU slots are loaded when a task is dispatched. The active region of each slot 7 is the task stack region. This may be loaded into the MPA and MPU when a task is dispatched, or loaded into the MPA when the MPA is created, then loaded into the MPU when the task is dispatched. Finally the auxiliary slots are only in the MPA, and their number varies from one MPA to another. (MPAs are allocated from the main heap and thus can vary in size.)

Stack Regions

Ref. 1 suggests an interesting idea of using *red zones* to protect against *stack overflows*. This has the advantage that it permits stacks to be right-sized for their tasks and to be located adjacent to each other in a single region that may contain other common variables. This can save a great deal of RAM in a v7M system. The red zone is a small region (e.g. 32 bytes) loaded into the top MPU slot on a task switch; it overlays the top of the task's stack. The red zone prohibits all accesses so that a stack overflow (from below the red zone) will cause an MMF.

Whereas this might be useful to catch some stack usage errors, it is not very useful for security. A hacker can easily jump over the red zone by defining a large local array in his first malware function being currently run by the task. You must always assume that a hacker knows as much or more than you do about your own code. Hence he knows which stack goes with which task and he will be able to place a false return to his second

malware function in whatever stack he wishes. When that task runs, he will gain control of it via a return from its stack. In addition, the red zone uses an MPU slot, so it gains nothing over a stack region, which also uses an MPU slot, as far as MPU usage is concerned.

The ideal approach is to allocate a stack from a heap when the task is created. Of course, this requires a heap that can allocate a block of the right size and alignment for the MPU. In addition, for v7M, it must set subregion disables to achieve the best greater-than-or-equal fit. *ehheap*, which is discussed in Appendix A can do this. An alternative approach is to use static stacks or stack pools in which the stacks are already properly sized and aligned. The heap approach offers more flexibility and efficiency, but either works fine for security. In both cases, the stack region has RW (Read/Write) and XN (eXecute Never) attributes. XN defeats a number of hacker tricks.

As a consequence of using a stack region, stack overflows and attempts to execute from the stack cause immediate MMFs. The worst a hacker can do is to wreck the stack and possibly the Task Local Storage (TLS) below it. He cannot damage any other stacks. For best security, the task stack region is put into the top slot. This is because for v7M, the attributes of the higher number slot prevails when regions overlap. This assures that the XN will not be overridden. v8M does not permit region overlap (more on this later), so this is not a factor for it, but the top slot is still used for consistency.

v7M Memory Gaps

Gaps are wasted memory between region blocks. The first line of defense against gaps is MPUPacker:

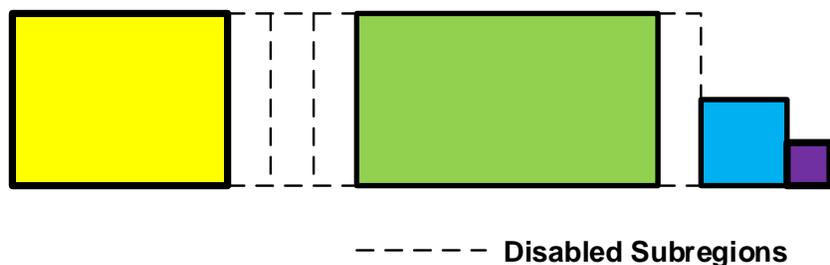


Figure 3.5a Before MPUPacker



Figure 3.5b After MPUPacker

Achieving Device Security

MPUPacker is a utility provided with SecureSMX. It is used to determine optimum ordering of region blocks. Figure 3.4a shows the linker output with wasted space due region block alignment on regions boundaries, which results in gaps where there are disabled subregions. Figure 3.4b shows how MPUPacker fills the first gap with small region blocks that are still on their own region boundaries.

If the reduction by MPUPacker is still not enough, *plug blocks* can be created from memory used for boot, initialization, and shutdown that are not in run-time regions, and these can be used to provide more gap fill. This code runs either before the MPU is enabled or when it is using MPA regions that span all memory.

Gaps can also occur inside of region blocks if other aligned region blocks are being included, such as:

```
define block sys_code with fixed order, size=scsz*6/8,alignment=scsz
    {block ucom_code, block cp_code, ro section
      .sys_text, ro section .sys_rodata};
```

The first step is to put the blocks in order by size. In this case `ucom_code = 0x4000*6/8` and `cp_code = 0x4000*5/8`, so there is a gap of `0x1000` between them. `.sys_text = 0xF680`, which is too big for the gap; `.sys_rodata = 0xE16`, which will fit into the gap, as follows:

```
define block sys_code with fixed order, size=scsz*6/8,alignment=scsz
    {block ucom_code, ro section .sys_rodata,
      block cp_code, ro section .sys_text};
```

Now the gap is reducing the gap to `0x1EA`, however, `sys_code` size remains at `0x18000`.

v7M Memory Tails

Tails are wasted memory inside of region blocks. In the above example, the tail has grown to `0x234a`, which is still much less than the subregion size = `0x4000`, so we cannot do a subregion disable to get rid of it. A tail can be as large as the subregion size minus one byte. Tails can be much harder to reduce or eliminate than gaps. However, if available memory is not being exceeded, spare memory might as well be distributed among tails. That is because they provide expansion memory for partition updates, without impacting other partitions, thus allowing smaller and faster updates.

The following techniques can be used to reduce tails:

1. If a tail is greater than 1/2 the region size for the block, reduce the region size.
2. If a tail is greater than or equal to the subregion size for the block, disable the subregion(s) occupied by the tail.
3. If code or data slightly exceeds a subregion boundary, improve code or data efficiency in order to reduce it below the subregion boundary, then disable that subregion or reduce the region size if the last 3 subregions are already disabled.

4. 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 the tails is smaller than the original tail. This may require some experimentation.
5. Use auxiliary slots to free up active slots and/or to reduce region block sizes in order to apply the above methods.
6. 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 likely to require adding new tasks. However, smaller partitions do enhance security.

Obviously, the foregoing techniques can take a lot of work and thus it is likely to be feasible only after all development is done. During development, we recommend using a pin-compatible MCU with much larger internal memory, if one is available. If not, then use *stub partitions* for partitions not being debugged. These either have less-important functionality removed or simply return constants for service requests. See the Framework section for more information on this.

Partition Demos

A partition demo package consisting of five separate demos plus instructions can be downloaded from www.smxrtos.com/securemx. These demos illustrate the process of putting vulnerable code, such as a file system in this case, into a pmode partition, then moving it into a umode partition without changing mission-critical code. This illustrates the process for upgrading the security of existing devices.

The demos are in source code form, so they can be modified to try out ideas. They are as follows:

pd0

pd0 represents a typical, unprotected embedded system running in handler mode (hmode) and privileged mode (pmode). It contains three tasks: *idle*, *mctask*, and *ffdemo*. *mctask* is intended to represent a mission-critical task. 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 with no significant change to the *mctask* code.

pd1

The first step is to turn on `SMX_CFG_MPU`. This enables the MPU. When it is on, `smx_TaskCreate()` assigns the default Memory Protection Array (MPA), *mpa_dflt*, to each task it creates. When a task is dispatched, its MPA is loaded into the MPU. So this step is to define *mpa_dflt*, which involves defining *region blocks*, *mpa_dflt*, and task stack regions.

pd2

In this step we put FatFs into a *pmode partition*, called *fs*, and create a unique MPA for it. Since a partition must have at least one task, we add *ffdemo* to the *fs* partition. Next, *sections* are defined for the *fs* partition using pragmas in the code and compiler section switches. The sections are used to define *region blocks* in the linker command file. These, in turn, are used to define an *MPA template* for the *ffdemo* task, which is used at run time to define the MPA for it. This MPA is created and loaded immediately after the *ffdemo* task is created, and now when it is dispatched, it will be limited to the regions in its MPA. Functions and variables that have been left out cause *Memory Management Faults (MMFs)*, which are tracked down and fixed. In the end, FatFs and *ffdemo* are running only in *fs* partition regions and cannot access code nor data used by *idle* and *mctask*.

pd3

In this step we prepare to move the *fs* partition to *umode*. First we must get it to make system calls via the *SVC exception*, because when running in *umode*, it will not be able to access *hmode* directly from *umode*. The *SVC* exception is triggered by the *SVC N* instruction, where *N* represents one of 256 possible system services. For each service, a value of *N* is assigned using an enum. Then, using *N* as the index, a jump table to the system service is created. For each service, a shell function is created to call *SVC N*, with its *N*. Also, a mapping header file is created that converts system calls to shell function calls. Then the standard API header file is replaced by the mapping header file for all *fs* C files, and the demo now runs using *SVC* exceptions, instead of direct service calls.

pd4

In this step we move to *umode*. The first step is to define the *ucom_code* and *ucom_data* regions, which are common to all *utasks*. The shell functions are put into *.ucom_text*. Next, the *fs* MPA template is modified to replace the *sys_code* and *sys_data* regions with the *ucom* regions. Since *fs* requires heap access, a custom heap is created for the *fs* partition. The final step is to set the *umode* flag in the *ffdemo* task create function. Now the *fs* partition runs in *umode*, and it cannot access any code or data in *hmode* or *pmode*, so *mctask* and *idle* are protected from any malware that might be put into *fs*.

Summary

In the foregoing we have examined methods to create static and dynamic regions. In addition, we have covered creating partition templates and using them to initialize task MPAs. We have also covered the MPU/MPA relationship and the definition of static, active, and auxiliary regions. Using auxiliary regions to create precise IO regions, and task stack region tradeoffs have been examined, as well. Finally the security improvement process for existing devices is illustrated with a set of five demos that can be downloaded and run. This is a pretty complete review of the current state of MPU management.



4. Creating Isolated Partitions

Introduction

The quest for security is not over, but we are closer. In this chapter we cover methods to achieve isolated partitions.

System Services from umode

ptasks can directly call system services in pmode, but utasks require a *software interrupt interface (SWI)* for system services such as signaling a semaphore. This API is implemented using the *SVC N* instruction, which causes an SVC exception that results in switching to pmode and executing the desired system service. The parameter N selects the system service to be performed. The SVC instruction is the only way a utask can penetrate the pmode barrier, and then only to run a *permitted system service*. When the system service completes, the utask is resumed in umode with the return value and data, if any, from the service. Figure 4.1 illustrates this process.

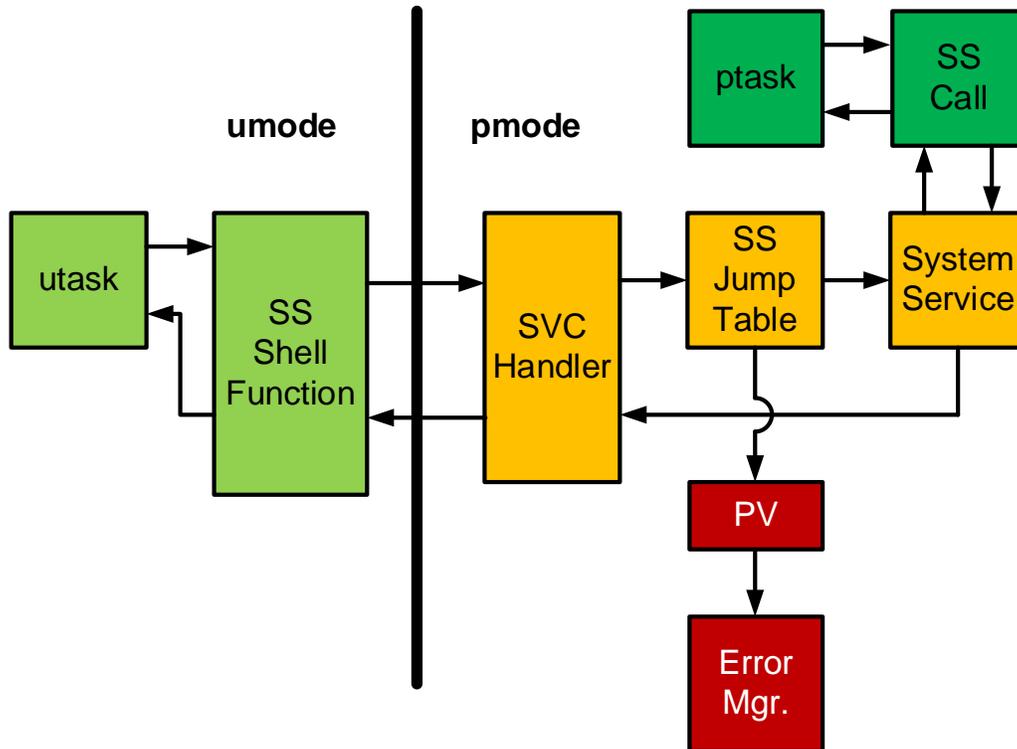


Figure 4.1 System Service Calls

Not only the code for system services but also the structures they use (e.g. Task Control Blocks) reside in pmode and thus are not accessible to a hacker from umode. In addition, services that could cause system damage are not permitted from utasks. Attempted use of a *restricted system service* results in a *Privilege Violation (PV)*, causing the Error Manager to run, the utask to be stopped, and recovery software to take control, thus stopping a hacker dead in his tracks.

SVC Shell Functions

The following code shows how the SVC shell functions work. The xapiu.h header function is included in all umode C files. It defines the shell functions, which differ in name from standard smx functions with a prefix of smxu_ instead of smx_. The second part of xapiu.h maps standard smx functions to the shell functions.

xapiu.h

```

u32      smxu_BlockPeek(BCB_PTR blk, SMX_PK_PAR par);
BOOLEAN  smxu_BlockRel(BCB_PTR blk, u16 clrslz=0);
u32      smxu_BlockRelAll(TCB_PTR task);
...
#define smx_BlockPeek(blk, par)      smxu_BlockPeek(blk, par)
#define smx_BlockRel(blk, clrslz)    smxu_BlockRel(blk, clrslz)
#define smx_BlockRelAll(task)       smxu_BlockRelAll(task)

```

SVC.C

```

/* main system service table indices */
enum ssndx {LIM, BP, BR, BRA, ... }

/* system service jump table */
u32 smx_sst[] = {
    (u32)END,
    (u32)smx_BlockPeek,
    (u32)smx_BlockRel,
    (u32)smx_BlockRelAll,
    ...
}

```

The `svc.c` file starts with the `ssndx` enum, which assigns `n` to abbreviated service call names, such as `BP`, for `BlockPeek`. Below is the system service jump table, `smx_sst`, which has the addresses of system services in the same order as `ssndx`. As shown in Figure 4.1 this table is in `pmode`.

Next are the system service shell functions. These are put into `.svc_text`, which is in `ucom_code`. The shell functions are very simple – they just call one of the macros shown below in `svc.h`.

```

/* system service shell functions */
#include "xapiu.h"
#pragma default_function_attributes = @ ".svc_text"
NI u32 smxu_BlockPeek(BCB_PTR blk, SMX_PK_PAR par)
{
    sb_SVC(BP)
}
NI BOOLEAN smxu_BlockRel(BCB_PTR blk, u16 clrslz)
{
    sb_SVC(BR)
}

```

Achieving Device Security

```
NI u32 smxu_BlockRelAll(TCB_PTR task)
{
    sb_SVC(BRA)
}
```

The first macro in `svc.h` is for four parameters or less, which are passed in registers 0 to 3. The second macro is for more than four parameters, in which case those over 4 are passed in the task stack.

`svc.h`

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

#define sb_SVCG4(id) \
{ \
    __asm("mov r12, #1"); \
    __asm("push {r4} \n\t"); \
    __asm("svc %0" : : "i" (id)); \
    __asm("pop {r4} \n\t"); \
}
```

All system service calls that are deemed safe for use from `umode` are included in the main system shell functions in `svc.c`. Excluded service calls are those that could cause system damage such as initialization functions. These should only be used from `pmode` during initialization and not while running.

Custom SVC Shell Functions

Most partitions need only a dozen or so system services. It is possible to define a custom set of shell functions and a custom jump table for a partition, as shown below. Notice that the header file `xapipa.h` is different. It is included in all C files in the partition. The prefixes of services are now `pa_` where `pa` is the partition name. (`smxpa_` could be used instead, if preferred.)

xapipa.h

```

u32      pa_BlockPeek(BCB_PTR blk, SMX_PK_PAR par);
BOOLEAN  pa_BlockRel(BCB_PTR blk, u16 clrslz=0);
...
#define smx_BlockPeek(blk, par)    pa_BlockPeek(blk, par)
#define smx_BlockRel(blk, clrslz)  pa_BlockRel(blk, clrslz)

```

The `svcpa.c` file is used in the `pa` partition instead of the `svc.c` file. It starts with its own enum, `pa_ssndx`, which assigns `n` to a lesser number of abbreviated service call names, such as BP and BR. Below it is the `pa` system service jump table, `pa_sst`, which has the addresses of system services in the same order as `pa_ssndx`. Even though this table is for partition `pa`, it is located in `sys_code`, where it is used.

svcpa.c

```

/* main system service table indices */
enum pa_ssndx {LIM, BP, BR,... }

/* main system service jump table */
u32 smx_sstp[] = {
    (u32)END,
    (u32)smx_BlockPeek,
    (u32)smx_BlockRel,
    ...
}

```

Next are the system service shell functions. These are put into `.pa_text`, which is included in `pa_code`. The shell functions are very simple – they just call one of the macros shown in `svc.h`.

```

/* paSSR shell functions */
#include "xapipa.h"
#pragma default_function_attributes = @ ".pa_text"
NI u32 pa_BlockPeek(BCB_PTR blk, SMX_PK_PAR par)
{
    sb_SVC(BP)
}
NI BOOLEAN pa_BlockRel(BCB_PTR blk, u16 clrslz)
{
    sb_SVC(BR)
}

```

When a switch to a task in `pa` occurs, `smx_sstp = &pa_sst`, where `smx_sstp` is used to access the current jump table. This is done in the `START` and `ENTER` cases of the task callback function. The `EXIT` case does `smx_sstp = &smx_sst` to restore the normal jump

table. Thus only partitions with custom jump tables need to have callback functions for this reason.

Multiple Heaps

There was a day when embedded systems were so simple that heaps were seldom used. However complexity has grown so much since then that most embedded systems, even RTOSs, now use heaps. In a typical partitioned system there may be several heaps. Multiple heaps are necessary because using a common heap between partitions destroys the isolation between them. SecureSMX uses *ehheap*.

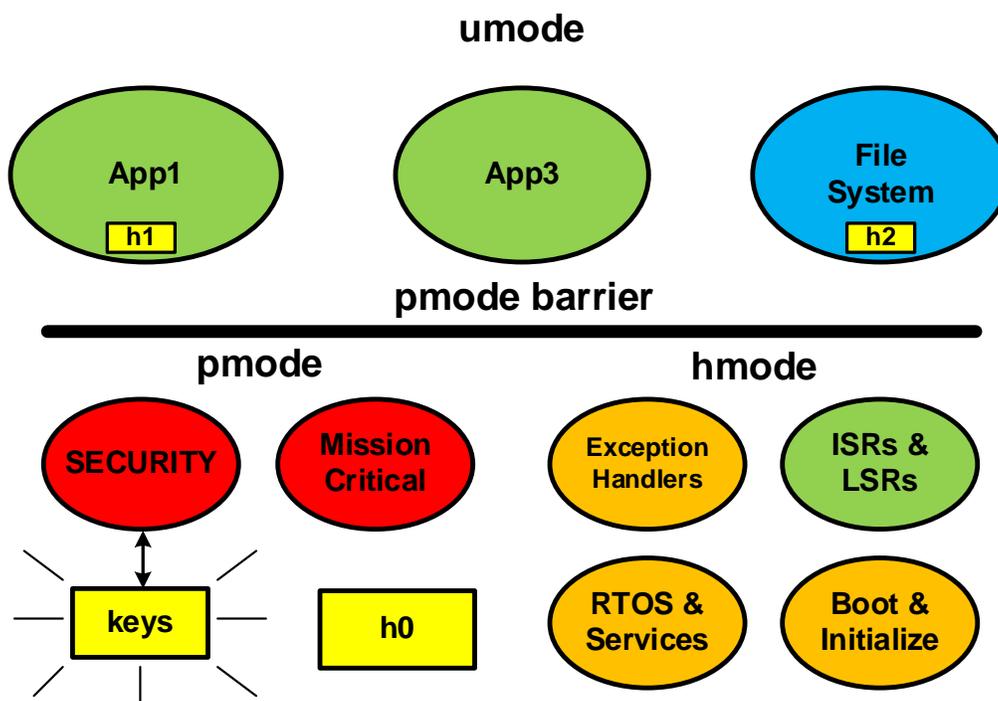


Figure 4.2 Multiple Heaps

Figure 4.2 shows a partitioned system requiring multiple heaps. Heap h0 is the main heap. It is used by pmode and hmode code. It is also used for task stacks and pmsgs, which can be used in umode. These are special cases. Task stacks have been discussed previously, and pmsgs are discussed next. The App1 partition needs heap h1 because its code is written in C++. *ehheap* incorporates small block pools to allow fast object creation and deletion. The file system partition requires heap h2 because it was written to get variable size buffers from a heap for files. If App1 or the File System were allowed to use h0, then the entire h0 would need to be accessible to them. Thus a hacker in either partition could wreak havoc on the system.

ehheap is similar to *dlmalloc* [Ref. 4] in that it uses bins. However, it is different in that the number and sizes of bins is configurable. Due to its extensive use, h0 is likely to have

many different bins, whereas h1 and h2 may require just a few bins and thus be much simpler. The following are features of eheap:

- Developed for Embedded Systems
- Permits Multiple Heaps Per System
- Aligned Block Allocations
- MPU Region Allocations
- Integrated Small Block Pools for C++
- 1 to 32 Configurable Bins
- Merge Control
- Heap Extension
- Auto Allocation Recovery
- Heap and Bin Self-Test and Recovery

For more information on eheap see Appendix A and the eheap User's Guide.

Protected Blocks and Messages

Protected blocks (pblocks) and *protected messages (pmsgs)* are unique to SecureSMX. pblocks provide secure buffers and pmsgs provide secure messages for portals, which are discussed next.

pblocks

If a buffer is needed and there is a spare slot in a task's MPA, it is desirable to allocate a pblock for the buffer. This can be done from an outside heap (normally the *main heap*), an outside block pool, or a statically defined block (e.g. `block[100]`) inside an existing MPA region⁴. A region for the pblock is automatically created and loaded into the MPU and into the task's MPA. pblocks can be used for partition heaps, temporary buffers, work areas, etc.

Such a block would normally have RW and XN attributes. It could also have *cache*, *strongly ordered*, and *shareable* attributes, which might be useful for DMA buffers, for example. An outside pblock immediately detects overflow and underflow and triggers an MMF. This defeats hacker buffer overflows. Additionally, if allocation of pblocks is performed in pmode, attributes such as XN cannot be changed in umode, which provides additional protection.

pmsgs

A normal smx message consists of a *Message Control Block (MCB)* linked to a *data block*, which contains the actual data. Tasks exchange messages via *exchanges*. An

⁴ An outside block cannot be permitted because this would give a hacker access to any memory he desired. An inside pblock can also be used for pmsgs.

Achieving Device Security

exchange can enqueue tasks waiting for messages, or messages waiting for tasks. Both tasks and messages can be enqueued by priority or by order of arrival.

A *protected message (pmsg)* is a normal smx message for which the data block is a pblock and the MCB contains the corresponding region information (e.g. *RBAR* and *RASR/RLAR*), as shown in Figure 4.3.

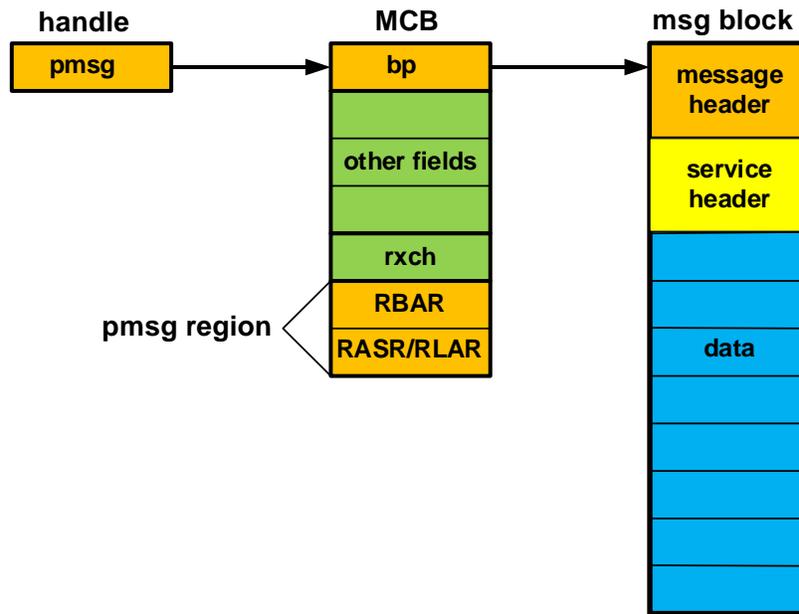


Figure 4.3 Protected Message

In the above figure, *pmsg* is the handle of the protected message. When a pmsg is received by a task, its region information is loaded into a free slot in the MPU and into the same slot in the task's MPA. pmsg attributes are normally RW and XN, but they could be RO and XN. pmsgs are, in effect, *self-contained, traveling regions*.

Summary

In the foregoing we have examined two of the essential requirements for isolated partitions: SWI system calls and multiple heaps. We also examined pblocks and pmsgs, which provide additional protection. Another essential requirement, partition portals is covered in the next section.

5. The Need for Portals

Introduction

The normal API between clients and servers is a *function call API*. Unfortunately as shown in Figure 5.1a, this destroys isolation between the client partitions and between the client and server partitions as illustrated by the large ellipses, because clients must have access to the server API functions.

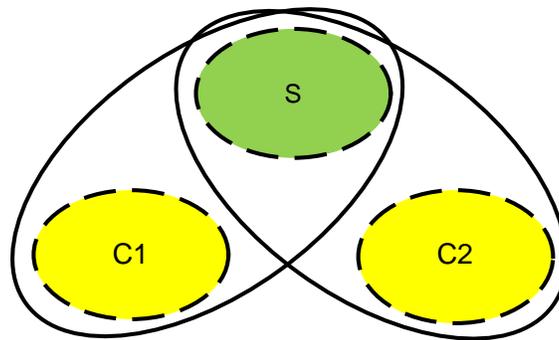


Figure 5.1a Server Function API

A *partition portal* provides a message API. As shown in Figure 5.1b, using a portal, P, preserves partition isolation.

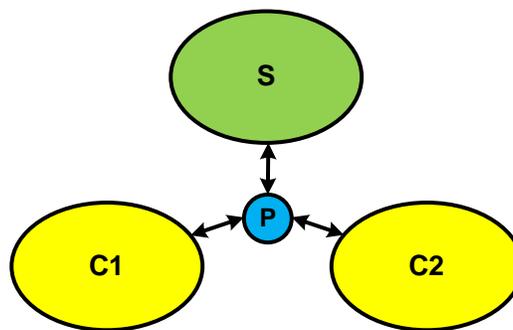


Figure 5.1b Portal API

Basically, function calls are converted to pmsgs in the *clients*, C1 and C2, and converted back to function calls in the *server*, S. In this chapter, two types of portals are discussed: *free-message portals* and *tunnel portals*.

Free Message Portals

In a free message portal, a pmsg is used once. It is sent from a client to a server and when the server is done with it, it is recycled or deleted. After being sent, the client can no longer access it. After being released, the server can no longer access it nor determine its fate. Hence, free message portals provide strong isolation. They are generally used for commands and small amounts of data.

Portal Operation

The free message portal gets its name from the fact that the pmsgs are not bound to the client as they are for a tunnel portal; hence they are free. The free message protocol is a *connectionless protocol* similar to UDP. It makes use of protected messaging capabilities, as discussed previously. Figure 5.2 illustrates three of many possible free message protocol configurations. small circles indicate waiting messages at the message exchanges shown.

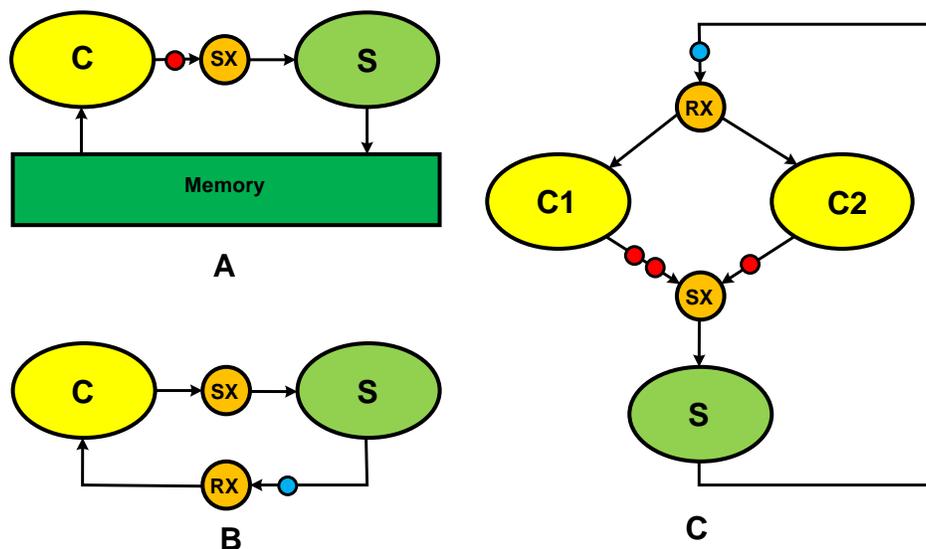


Figure 5.2 Free Message Protocol Configurations

Figure 5.2A is the simplest configuration: Client C gets a pmsg from memory, fills it, and sends it to Server S via server exchange SX. Server S processes the pmsg and returns it to memory. Figure 5.2B is a more efficient configuration. In this case, Server S sends pmsgs to reply exchange RX from which they are obtained by Client C. In the case where a single pmsg is being circulated, the return pmsg probably has ACK/NAK status or data, and RX serves as a *reply exchange*. In the case where several pmsgs are circulating, RX serves as a *resource exchange*. Figure 5.2C is an extension of 5.2B where two clients are using the same server. In this case S is serving multiple clients and RX is a resource exchange.

Normally pmsgs have priorities. In 5.2C, for example, messages from C2 might have a higher priority than pmsgs from C1 and thus be serviced ahead of C1 pmsgs. In addition, if SX is a *pass exchange*, the pmsg priority is passed on to server task S while it is processing the pmsg. If $S \leq C$, the pmsg will be processed some indeterminate time in the future after C and any other tasks $\geq S$ that are ahead of it have been suspended or stopped. This would probably be satisfactory for logging, for example. If $S > C$, C will be preempted for S to run. The latter mimics a direct call. Thus pmsg priority can be adjusted to achieve different results.

Note that the client is the *master*, and that it initiates transactions. Should the server need to initiate a transaction, such as for a *client callback*, a separate portal is required. If a pmsg block comes from a client data region, the client can use an auxiliary slot for its region (see Figure 3.4). Otherwise the client must use an active slot. Similarly for the server. v8M introduces a complexity for pmsgs because it does not permit overlapping regions. Therefore, if a pmsg comes from a local region in a client or server, its region must be put into an auxiliary slot.

The task in a client that was making direct calls to the server is now making and sending pmsgs. The task in the server which is handling the portal is serving as a *proxy* making API calls for the client task.

Portal Initialization

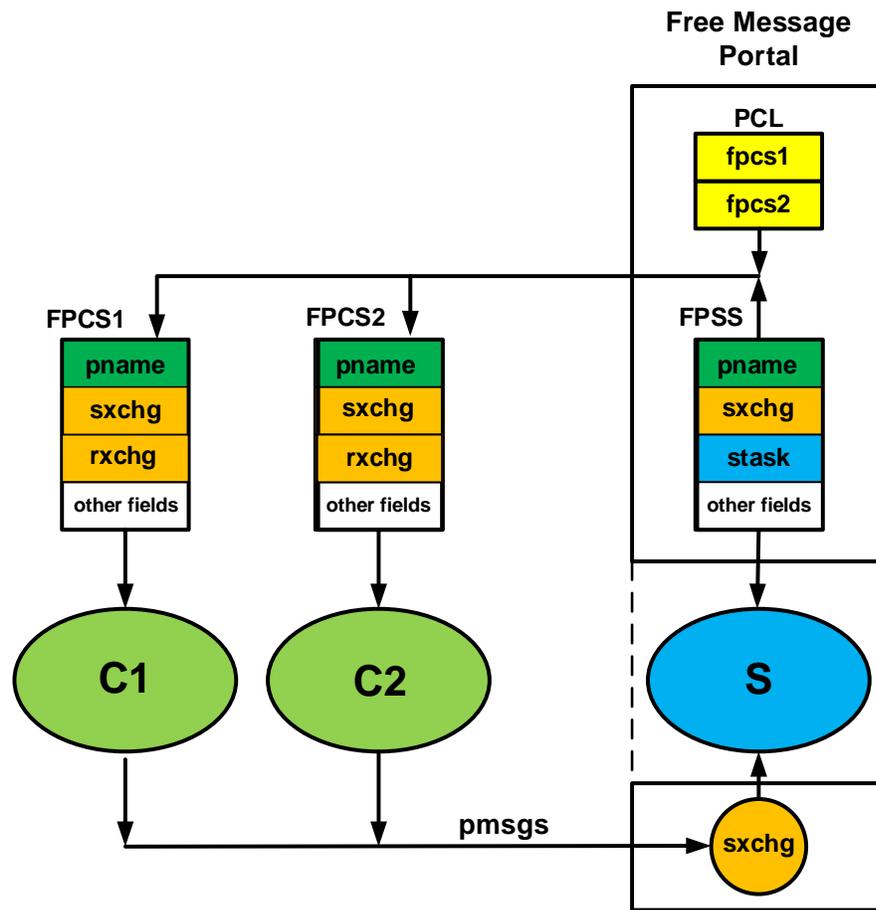


Figure 5.3 Free Message Portal Initialization

Figure 5.3 illustrates how a free message portal is created and initialized. There is a *Free Portal Server Structure (FPSS)* for each free portal in a server. (A server has a portal for each interface that it presents to the rest of the system.) There is a *Free Portal Client Structure (FPCS)* for each server that a client can access. In addition, for each FPSS, there is a *Permitted Client List (PCL)* array of pointers to all FPCSs of clients permitted to access the server's portal. All structures and arrays are defined at compile time and cannot be changed during run time. Also they are in pmode and not accessible from umode. Server portal initialization code, which must be called in pmode does the following:

- Creates the server exchange, **sxchg**.
- Loads the **sxchg** handle and the portal name into the FPCSs, using the portal client list, **PCL**, pointers (e.g. **fpcs1** and **fpcs2**).
- Creates the server task, **stask**.
- Loads the **sxchg** handle, portal name, and **stask** handle into the **FPSS**.
- Causes **stask** to wait at **sxchg** for the first **pmsg**.

A client calls the *portal open* function, which creates a *resource exchange*, *rxchg*, gets N pmsgs, sends the N pmsgs to *rxchg*, and loads FPCS fields, except name and *sxchg*, which were already loaded. This is consistent with Figure 5.2B. If N = 0, *rxchg* is not created, and no pmsgs are obtained. This is consistent with Figure 5.2A. For Figure 5.2C, C1 gets N pmsgs and *rxchg* and C2 gets no pmsgs and shares *rxchg* with C1.

This protocol may seem overly elaborate, but it is not. C1 and C2 are isolated from S. Hence, the only way C1 and C2 can get the *sxchg* handle and the portal name is if they are copied into their regions by initialization code running in *pmode*. The portal name is optional, but it helps to avoid confusion during debugging if there is more than one portal structure in a partition. It is important that the portal structures and the PCL array cannot be altered from *umode* in order to prevent tampering by a hacker.

Tunnel Portals

Tunnel portals are used for multi-block data transfers, such as files or network streams. A tunnel portal is similar to a free message portal, except that the pmsg sent by a client remains in use until the multi-block transmission is complete and the portal is closed by both ends. Hence the tunnel portal is a *connection protocol* similar to TCP. Also, the client remains the *owner* of the pmsg and server is just its *host*, but both ends have the pmsg region in their MPAs.

As shown below, the tunnel has a door at each end, but only one door, at a time, can be open. When the client's door is open, it puts data into the pmsg in the tunnel. The door closes; then the door on the server end opens and the server takes data out of the pmsg. Sending data from the server to the client is the opposite. The pmsg data block acts as an *alternately shared region* between the client and the server, similar to an airlock.

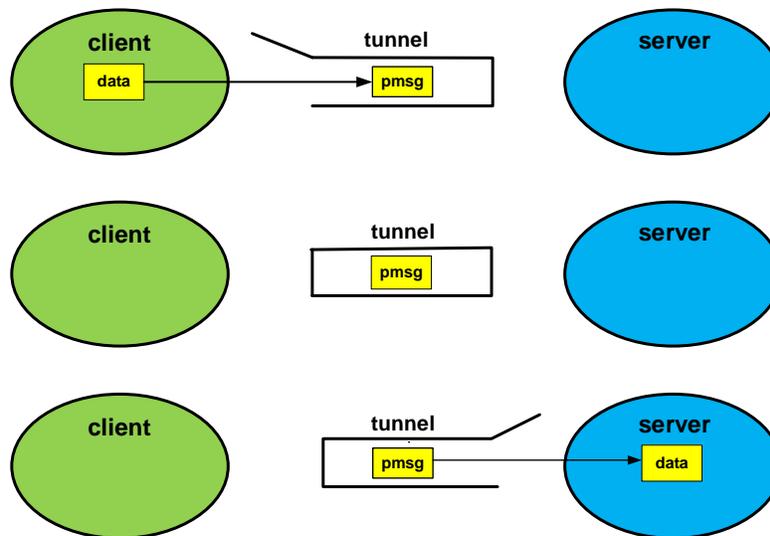


Figure 5.4 Tunnel Portal Operation

Achieving Device Security

The half-duplex operation, is controlled by two semaphores, *csem* (*client semaphore*) and *ssem* (*server semaphore*). For example, the client can load data into the pmsg block, signal *ssem*, and then wait at *csem*. This closes the client's door and opens the server's door. The server can read the data, load a return value or data into the pmsg block, then signal *csem* and wait at *ssem*, which reverses the doors. When the tunnel portal is closed by both ends, the pmsg is recycled or deleted.

Tunnel portal initialization is similar to free message portal initialization, except that the structures are different: *Tunnel Portal Server Structure (TPSS)* and *Tunnel Portal Client Structure (TPCS)*. Portal creation is similar. A single pmsg is obtained by the client and sent to the sxchg. Then the client initiates a multi-block send or receive, where the blocks are pmsg block size with the last block usually smaller. Naturally the larger the pmsg block is, the better the performance. If the pmsg block is as big as the data being sent, *no-copy* operation can be implemented, where the pmsg block, itself, is used as the client working buffer.

Shell Functions

Both types of portals use shell functions. Here we examine shell functions for tunnel portals since they are slightly more complicated. Each portal has protocol functions such as *Create()*, *Open()*, *Close()*, *Send()*, and *Receive()* for using the portal. Figure 5.5 shows how a portal converts an API function call to a pmsg in the client, then uses a switch statement to convert the pmsg back to the function call in the server. The server performs the function then sends the return value and data, if any, back to the client via the pmsg.

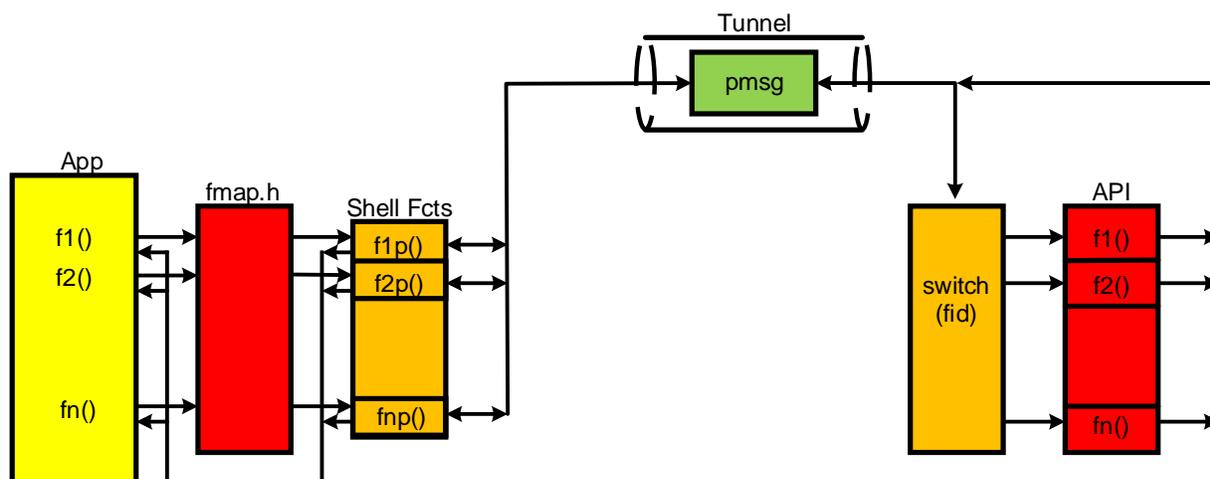


Figure 5.5 Portal Operation

The following code is for a umode client, ut2c, and a umode server, ut2s. First are pragmas to put cbuf and tp_pcsa into ut2c_data and to put tp_pss and tp_pcl into ut2s_data.

```
#pragma default_variable_attributes = @ ".ut2c_bss"
u8      cbuf[LBSZ];      /* client buffer */
TPCS    tp_pcsA;        /* tunnel portal client structure A */
#pragma default_variable_attributes = @ ".ut2s_bss"
TPSS    tp_pss;         /* tunnel portal server structure */
#pragma default_variable_attributes = @ ".ut2s_data"
TPCS*   tp_pcl[] = {&tp_pcsA, &tp_pcsB}; /* permitted client list*/
#pragma default_variable_attributes =
```

tportal_init() creates ut2c and its MPA, then creates ut2s and its MPA. Following this the TPSS for ut2s is initialized, then ut2s and ut2c are started. tportal_init() runs during initialization in pmode.

```
void tportal_init()
{
    TPSS* psh; /* portal server handle */
    /* create client and server tasks */
    ut2c = smx_TaskCreate(func, TP2, TS_SSZ, SMX_FL_UMODE, "ut2c");
    mp_MPACreate(ut2c, (MPA*)&mpa_tmplt_ut2c, tmskc, 9);
    ut2s = smx_TaskCreate(funs, TP2, TS_SSZ, SMX_FL_UMODE, "ut2s");
    mp_MPACreate(ut2s, (MPA*)&mpa_tmplt_ut2s, tmsks);
    /* initialize portal server structure */
    psh = &tp_pss;
    psh->stask = ut2s;
    psh->sid = sid;
    psh->ssid = 0;
    mp_TPPortalCreate(&psh, tp_pcl, sizeof(tp_pcl)/4, SV_SLOT,
                    "tp_uport1", "tp_sxchg");

    /* start server and client */
    smx_TaskStart(ut2s);
    smx_TaskStart(ut2c);
}
```

Achieving Device Security

The following is the server code, which is in the `ut2s_code` region. `ut2s_main()` simply calls `mp_TPPortalServer()`, which is a tunnel portal protocol function.

```
#include "xapiu.h"
#pragma default_function_attributes = @ ".ut2s_text"

/* umode server task */
void ut2s_main(u32)
{
    mp_TPPortalServer(&tp_pss, STMO);
}
```

Next is the client task code, which is in the `ut2c_code` region. The first step is to get a `pmsg` block from the main heap of size `SMSZ`. The block address is loaded into `tpch->mhp` (message header pointer), the `pmsg` region is loaded into `CL_SLOT` in the MPU and in the `ut2c` MPA so that `ut2c` can access it, and the region has a `DATARW` attribute. Next, `tportal` is opened with its tunnel portal control structure = `tp_pcsA`, message size = `SMSZ`, total header size = `THSZ`, priority = `TP3`, timeout = `COTMO`, and `ssem` and `csem` names. `mp_TPPortalOpen()` creates the `ssem` and `csem` semaphores, initializes `tpch`, then creates an `OPEN` `pmsg`, which is sent to the server with `TP3` priority, then causes `ut2c` to wait at `csem`.

If `OPEN` succeeds, the server returns `TRUE` and the client continues. If `COTMO` ticks elapse, the open fails and operation is aborted, as shown. For reliability, there must be a timeout, but `COTOMO` must allow sufficient time for the server to first serve other clients that are ahead.

```

#pragma default_function_attributes = @ ".ut2c_text"
/* umode client utask. */
void ut2c_main(u32)
{
    u8*   cbp = (u8*)&cbuf; /* client buffer ptr */
    TPMH* mhp;                /* pmsg header pointer */
    TPCS* tpch = &tp_pcsA; /* tunnel portal client handle */

    /* get pmsg from heap */
    tpch->pmsg = smx_PMsgGetHeap(SMSZ, (u8*)&tpch->mhp, CL_SLOT,
                                DATARW);

    /* open tportal */
    if (mp_TPortalOpen(tpch, SMSZ, THSZ, TP3, COTMO, "ssem", "csem"))
    {
        /* send data in cbuf to server */
        tp_Send(tpch, cbp, LBSZ, TP_WRITE) /* shell function */

        /* receive data from server into cbuf */
        tp_Receive(tpch, cbp, LBSZ, TP_READ); /* shell function */

        /* close tunnel portal and release pmsg */
        mp_TPortalClose(tpch, CCTMO);
    }
    smx_PMsgRel(&tpch->pmsg);
}

```

If OPEN succeeds, the client then sends the data in `cbuf` to the server. If `LBSZ > SMSZ`, this will result in a multi-block transfer, which is transparent to the client. Then the client receives the data back, closes the portal, and releases the `pmsg`. To close the portal, it sends a CLOSE command to the server, then waits at `csem` for a response. When received or if `CCTMO` ticks elapse, it clears the client part of `tp_pcsA` and deletes `csem` and `ssem`. Care must be taken that `CCTMO` is long enough for the server to complete whatever it might be doing (e.g. an `fwrite()` operation.)

The two shell functions, `tp_Receive()` and `tp_Send()`, which are called above, are shown below. These are put into `ucom_code`, assuming there will be more than one client for this server. If there is only one client for this server, they would be put into `ut2c_code`. Basically, these just call the macro `mp_SHL1()` to load the service header in the `pmsg` block (see Figure 4.3) with the function id, parameter `n`, and a default return value, `0 = fail`. Then `tp_Receive()` calls `mp_TPortalReceive()` and `tp_Send()` calls `mp_TPortalSend()`, which are tunnel portal functions. These and the open and close functions implement the tunnel portal API on the client side.

Achieving Device Security

```
#pragma default_function_attributes = @ ".ucom_text"
/* client send data to server shell function */
BOOLEAN tp_Send(TPCS* tpch, u8* dp, u32 msz, TP_FID fid, u32 n)
{
    mp_PTL_CALLER_SAV();
    TPSH* shp = (TPSH*)tpch->shp; /* service header pointer */
    mp_SHL1(fid, n, 0);
    return (mp_TPortalSend(tpch, dp, msz, CTMO));
}
/* client receive data from server shell function */
BOOLEAN tp_Receive(TPCS* tpch, u8* dp, u32 msz, TP_FID fid, u32 n)
{
    mp_PTL_CALLER_SAV();
    TPSH* shp = (TPSH*)tpch->shp;
    mp_SHL1(fid, n, 0);
    return (mp_TPortalReceive(tpch, dp, msz, CTMO));
}
#define mp_SHL1(fid, par1, e) \
{ \
    shp->fid = fid; \
    shp->p1 = par1; \
    shp->ret = e; \
    mp_PTL_CALLER_SHL(); \
}
```

On the server side we see `mp_TPortalServer()`, which is a tunnel portal function. Note that it is in the `ucom_code` region. Basically, it waits passively at `sxchg` for the next `pmsg`. When it receives the `pmsg`, it switches on the command in the message header (see Figure 4.3). Shown here are just the `SEND` and `RECEIVE` cases. There are also `OPEN`, `CLOSE`, and default cases. `SEND` calls the user function `tp_server()`, then does send-related protocol functions. `RECEIVE` also call `tp_server()`, then does receive-related protocol functions.

```

#pragma default_function_attributes = @ ".ucom_text"
void mp_TPortalServer(TPSS* psh, u32 stmo)
{
    while (psh->pmsg = smx_PMsgReceive(psh->sxchg, ...))
    {
        switch (mhp->cmd)
        {
            case SEND: /* from client */
                tp_server(tpsh);
                ...
                break;
            case RECEIVE: /* to client */
                tp_server(tpsh);
                ...
                break;
        }
    }
}

```

tp_server interprets the service header to call the services provided by the server. In this case there are only two services, TP_WRITE and TP_READ. Note that tp_server is in ut2s_code.

```

#pragma default_function_attributes = @ ".ut2s_text"
void tp_server(TPSS* tpsh)
{
    thp = (TPSH*)tpsh->shp;
    switch (thp->fid)
    {
        case TP_WRITE:
            ...
            break;
        case TP_READ:
            ...
            break;
    }
}

```

Tunnel Portal Timeouts

The difficulty of choosing timeouts is a direct consequence of enforcing isolation. The best approach is to pick the longest timeouts that are tolerable for the system. Then design the client code to retry a few times, before resorting to more severe measures. When the client csem times out, it marks the portal as being closed so that subsequent read and write operations will be aborted. To retry, the portal should be closed, then reopened. The server treats an ssem timeout as a portal close, and it goes back to sxchg for the next pmsg.

Using semaphores is the bare minimum necessary to implement multiblock transfers. These are event semaphores so multiple signaling has no effect. However, a hacker probably can mess them up somehow. This is why tunnel portals are less secure than free message portals. Of course, application software can implement multi-block transfers with free message portals, but they may not be fast enough.

Summary

In the foregoing we have examined why portals are necessary to achieve fully isolated partitions. Two types of portals have been discussed, free message portals and tunnel portals, both of which utilize pmsgs. In addition we have looked into how a function call API can be converted to a message API. As with system services, only server functions that cannot cause system damage should be available via portals. For example, server initialization functions should only be called directly from pmode during initialization. Basically, portals should implement only the minimum set of API functions to utilize the server in the manner necessary for clients. If a potentially dangerous function must be available to clients, it should be carefully monitored and limited in the server to avoid hacker misuse.



6. Partition Limitations

Introduction

Unfortunately, fully isolated partitions are not enough. A hacker can do considerable system damage from within a partition. For example, putting a task into an infinite loop results in blocking all tasks of equal or lower priority from running. Repetitively creating the same object can exhaust the supply of that object. Hacking into an ISR puts the hacker into handler mode where he can turn off the MPU and access or do whatever he wants. The features discussed in this section are designed to help thwart these attacks.

Runtime Limiting

To prevent the first attack, it is necessary to implement task *runtime limiting*. Unfortunately, in real time systems, this can be a curse as well as a blessing – for example, we do not want mission-critical tasks to be runtime limited during emergencies, such as when a crane is about to topple over! But it is not possible for a developer to estimate how much run time may be required to avert such a catastrophe. So important tasks must be allowed to run without runtime limits. This, plus their high priorities should assure that they can run as long as necessary to never fail, even in extreme situations.

Less-trusted tasks are assigned runtime limits and counters. At the start of a frame, all counters are cleared. It turns out that tick resolution is too coarse, so CPU clocks are used, instead. Each time a task runs, the number of clocks it used is determined and

Achieving Device Security

added to its counter. If this exceeds the task's runtime limit, the task is suspended on a *gate semaphore* and can no longer run. At each tick, the current task's counter is updated, and if it is above its runtime limit, the task is suspended upon the gate semaphore.

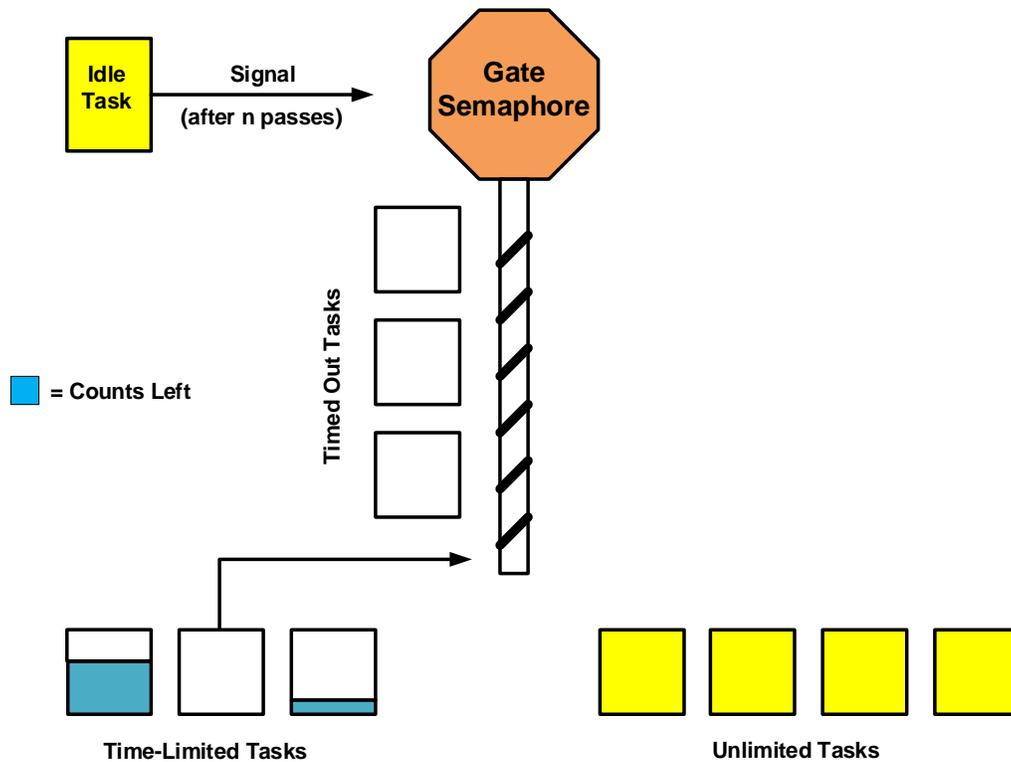


Figure 6.1 Runtime Limiting

It is hard enough to get a good balance between task priorities, which allows tasks to meet their deadlines. Adding a fixed runtime frame can only increase this difficulty. Instead, we end the runtime frame when the idle task has had sufficient passes to do its work. Since the idle task has the lowest priority, this assures that all tasks have run sufficiently to do their work. Then the gate semaphore is signaled, and all waiting tasks are resumed with their counters cleared. Tasks of the same priority resume in the same order as they were suspended. This avoids excessive task delays, which could cause problems.

A child task shares its top parent's⁵ runtime limit and counter. If the limit is exceeded, the child task is suspended immediately. The parent and its other children are suspended only if they attempt to run afterward. Assigning runtime limits to task families is much easier than trying to assign a runtime limit to each task in a family, as it is spawned.

⁵ SecureSMX permits child tasks to spawn other child tasks. So, the "top parent" is the task which is not a child of any other task.

Tokens

During World War II, families received red tokens for meat, blue tokens for fish, and silver tokens for the trolley. In this way our government controlled consumption. Similarly, we need to govern how much of each resource a partition can use and how the partition can use that resource. SecureSMX uses tokens for this purpose.

A *handle* is a memory location that stores the address of an *object control block*, such as a task's TCB – i.e. it is a *control block pointer*. A *token* is the address of a handle. Handles are defined at compile time, and assigned addresses at link time. There are two types of tokens: a HI token allows creating, deleting, modifying, and accessing an object, and a LO token permits only accessing the object (e.g. to test or signal a semaphore). A token list is compiled for each task, by the programmer, and assigned to the task after it is created. If no token list is assigned, the task does not require tokens to access or modify objects. The latter is necessary for tasks such as recovery tasks, and it makes things simpler for trusted tasks.

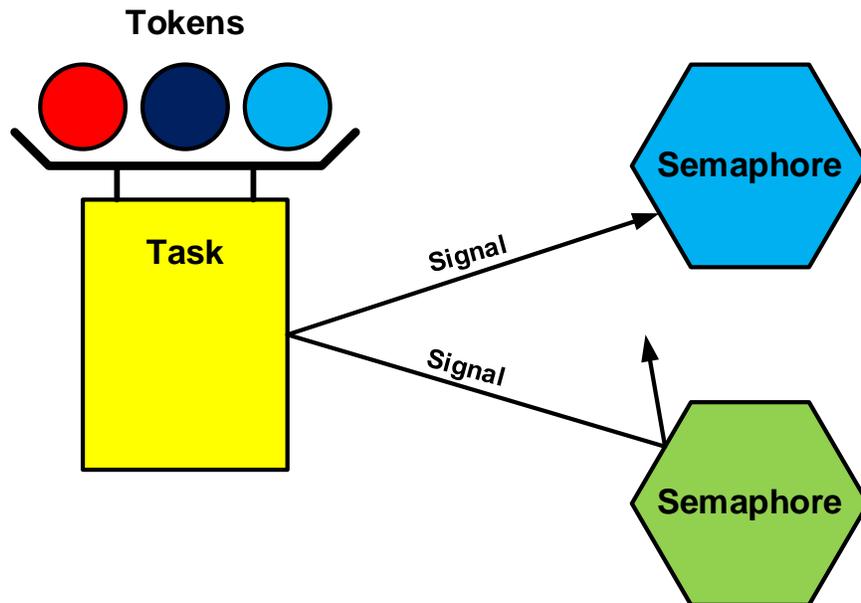


Figure 6.2 Tokens Controlling Semaphore Access

One of the insidious things a hacker can do from inside a partition is to create the same object over and over until the pool of object control blocks is exhausted. Then no other task can create an object of that type. This is blocked as follows: First, the task being used by the hacker must have a HI token for the object. Second, once created, the object cannot be re-created, as shown in `t2a_main()` until it has been deleted.

Another possible hacker attack is to guess a handle and use this to cause trouble. For example, a semaphore in another partition could be signaled, thus causing a task in that partition to run when it should not run. This is blocked by requiring a token for that semaphore.

Achieving Device Security

In addition to tokens, all handle parameters are verified to be valid handles before using them in system services. Each handle is range-checked, and its cbtype field is checked. This prevents a hacker from using invalid handles in system service calls.

The following code shows token operation for two utasks. First the token lists are defined. In this case each list has only one token: a LO sbe token and a HI sbe token for semaphore sbe. Token lists normally have more tokens.

```
u32 ut2a_ta1[] = {(u32)&sbe, 0}; /* ut2a token list with LO sbe token */
u32 ut2b_ta1[] = {(u32)&sbe+1, 0}; /* ut2b token list with HI sbe token */
```

This is followed by tsemTK01() which creates tasks ut2a and ut2b and assigns the token lists to them. Then the tasks are started.

```
void tsemTK01(void)
{
    /* create ut2a and attach token list */
    ut2a = smx_TaskCreate(tsemTK01_ut2a, TP2, TS_SSZ, UTASK, "ut2a");
    smx_TaskSet(ut2a, SMX_ST_TAP, (u32)&ut2a_ta1);

    /* create ut2b and attach token list */
    ut2b = smx_TaskCreate(tsemTK01_ut2b, TP2, TS_SSZ, UTASK, "ut2b");
    smx_TaskSet(ut2b, SMX_ST_TAP, (u32)&ut2b_ta1);

    /* start tasks */
    smx_TaskStart(ut2a);
    smx_TaskStart(ut2b);
}
```

Task ut2a attempts to create sbe, but it fails because it has only a LO sbe token.

```
void tsemTK01_ut2a(u32)
{
    /* fail to create sbe */
    sbe = smx_SemCreate(SMX_SEM_EVENT, 1, "sbe", &sbe);
    if (sbe != NULL)
        tfail();
}
```

Task ut2b successfully creates sbe because it has a HI sbe token, but it cannot create sbe twice.

```

void tsemTK01_ut2b(u32 mode)
{
    /* create sbe */
    sbe = smx_SemCreate(SMX_SEM_EVENT, 1, "sbe", &sbe);
    if (sbe == NULL)
        tfail();

    /* fail to create sbe twice */
    SEM_PTR sbe_sav = sbe
    sbe = smx_SemCreate(SMX_SEM_EVENT, 1, "sbe", &sbe);
    if (sbe != sbe_sav)
        tfail();
}

```

Nor can a hacker define a fake semaphore handle then create a semaphore for it because `ut2b` would not have a token for its handle.

The ISR Problem

Unfortunately, ISRs execute in `hmode` due to the Cortex-M architecture and thus they provide an attack surface into `hmode`. Many ISRs are not written to have minimal code. This problem is exacerbated by RTOSs that allow kernel services to be called from ISRs. Together these can create a large target for a hacker. This target is particularly tempting because hacking it puts him into `hmode` where he can turn off the MPU and access everything.

SMX supports a different design philosophy, wherein ISRs are minimized and most interrupt processing is deferred to *link service routines (LSRs)*. LSRs run in the order they were invoked and they run ahead of all tasks. Hence, they are immune to priority inversion problems and are thus more suitable for deferred interrupt processing than are tasks. Also LSR overhead is less than task overhead. Minimizing ISR size reduces the target size and allows the ISR programmer to focus on making the ISR code more difficult to hack.

SecureSMX supports three types of LSRs: trusted `tLSRs`, `pmode pLSRs`, and `umode uLSRs`. Each `pLSR` and `uLSR` has its own stack and its own MPA, and it behaves like a mini task.

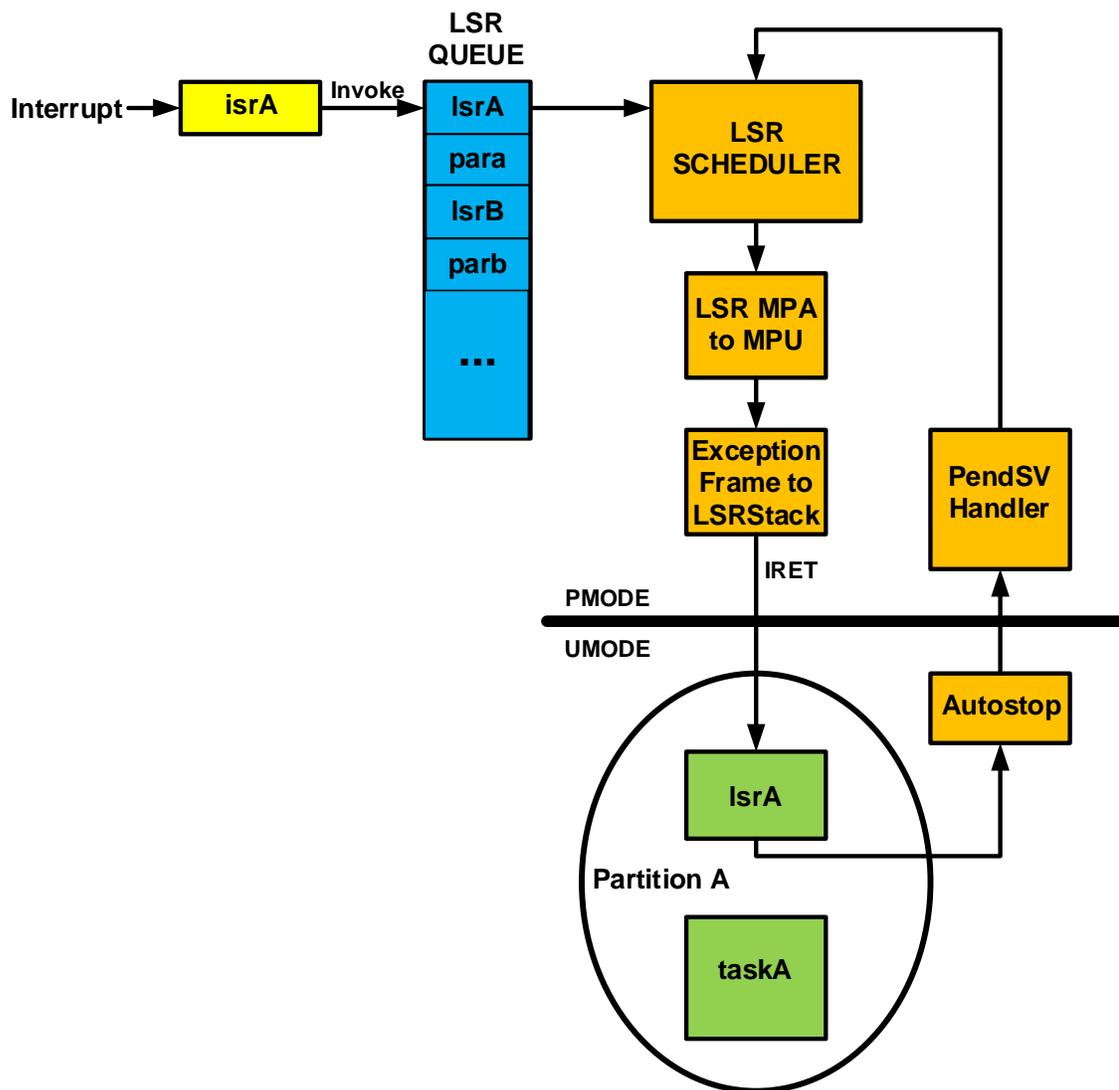


Figure 6.3 Safe LSR Operation

Figure 6.3 illustrates uLSR operation. isrA invokes IsrA and re-enables the interrupt. Invoke puts the IsrA handle and one parameter into the LSR queue. When all outstanding ISRs have run, the LSR scheduler runs and gets the handle and parameter of IsrA, then dispatches IsrA. This consists of loading its MPA into the MPU and a fake exception frame into the IsrA stack, then doing an iret to IsrA main code in umode. As a consequence, most original isrA code now runs in umode in partitionA along with taskA and shares regions with taskA.

If the hacker hacks into IsrA code, he is no better off than hacking into taskA code. When done, IsrA autostops. This triggers the PendSV Handler, PSVH, which calls the LSR scheduler for IsrB.

If overhead for a lsrA is too great as a uLSR, it can run as a tLSR. A tLSR runs in hmode and has very low overhead. However, it is not secure. To deal with this, lsrA could have minimal code, such as to just signal a semaphore at which taskA waits. taskA would then do the deferred processing. However, this is likely to be slower than a uLSR and is subject to priority inversion. So for fastest operation all of the code must remain in a tLSR.

pLSRs run in pmode and thus are not secure, either. They are provided to support developing code in a pmode partition, then moving it to a umode partition.

An interesting thing to note is that a uLSR may act like a child task and have some shared regions and some unique regions (e.g. IO). uLSRs have very small control blocks and typically require very small stacks. In addition, they run at a priority below ISRs and above tasks. Thus they present an interesting way to do IO processing that is much more secure than the usual ISR approach, yet nearly as fast.

Summary

SecureSMX offers the following methods to limit malware operations from inside of partitions:

- Only permitted clients can access a portal
- Limited access to server functions via portals
- Limited access to system services from umode
- Selective runtime limits for tasks
- Tokens to limit access to and control of system objects
- Prevention of creating an object more than once or of creating fake objects in order to exhaust object control blocks
- Moving hmode ISR code to a safe umode LSR
- Interrupt control permission tables
- Using precise IO regions by swapping auxiliary regions
- Extensive event monitoring to spot anomalous behavior
- Thorough system service parameter checking

Undoubtedly as time goes on, more limitations will be needed, but we think these are a good start.

7. Advanced Features

Parent/Child Tasks

There is a belief that all tasks should be created during system initialization. This might improve security but it may not be feasible in practice. For example, USB devices may be inserted or removed dynamically, and each requires a task for its *class driver*. Also there are USB controllers that can operate in either host mode or device mode. Each requires its own set of tasks. Since tasks are SRAM-hungry, it may be necessary while running to delete tasks for one device or mode and to create tasks for the other device or mode. Otherwise, the system would have to be stopped and rebooted to change modes.

There are many other cases where not allowing dynamic task creation and deletion would create implementation problems. It is not even acceptable to limit task creation to pmode, since USB stacks and similar code should be running in umode. On the other hand, allowing umode partitions to create, delete, and otherwise manipulate tasks does not seem like a good idea either.

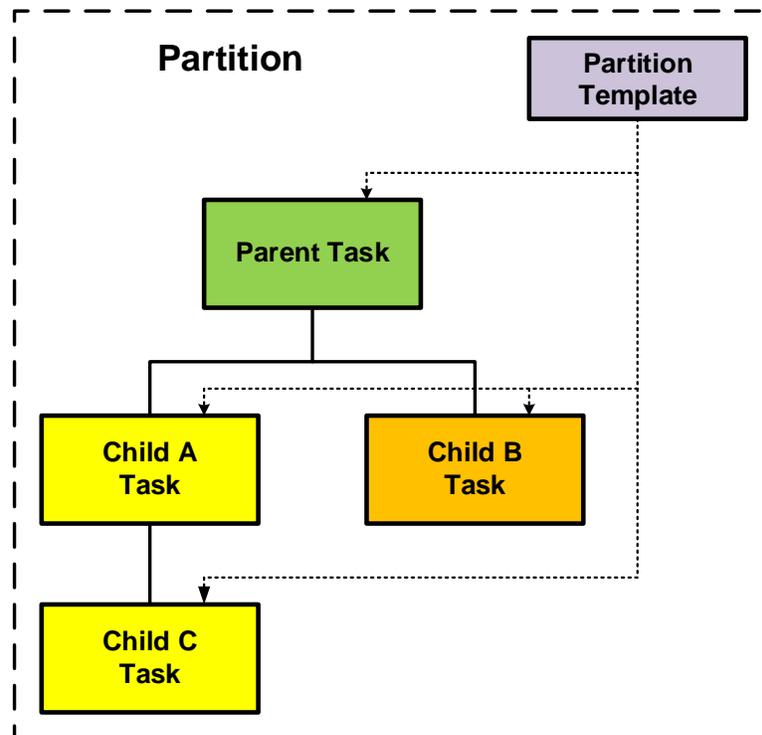


Figure 7.1 Parent/Child Tasks

The parent/child task concept shown in Fig. 7.1 provides a solution to this conundrum. The basic principle is that a child task cannot do anything that its parent cannot do. Hence, the child task inherits all limitations (e.g. interrupt access, service call permissions, runtime limits, and tokens) from its parent. In addition, it is limited to

drawing its regions from the partition regions (shown as *Partition Template* in Fig 7.1), and it will likely have only a subset of these regions. A parent task can create or delete a child task; it can start it, stop it, and perform certain other task operations on it. A child task can also be a parent of its own child tasks. However, it cannot perform task operations on its parent, its siblings, nor their children. From a security point of view, child tasks are simply extensions of their parents.

It should be noted that partition main tasks are normally created in pmode and initially run in pmode because it is easier that way to initialize their partitions. Then main tasks go into umode and possibly spawn child tasks and to complete the partition initialization in umode. So a partition main task may or may not be a parent task.

Auxiliary Slots

Auxiliary slots effectively increase the number of MPU slots. They are used in two ways: for *protected messages* (*pmsgs*) and for *slot swapping*. The former was discussed in Chapter 5. The latter is particularly useful for IO regions. For example, a USB host task might require access to the USB OTG, DMA, and USART controllers. For the STM32F746 processor, these are located at: 0x40040000 to 0x4007FFFF, 0x40026000 to 63FF, and 0x40011000 to 13FF, respectively. The first requires a 0x40000 = 256KB region starting on a 256KB boundary, the second and third require 0x400 = 1KB regions starting on 1KB boundaries.

If only one active IO slot is available, as shown, it is necessary to define a single region from 0x40000000 to 0x4007FFFF in order to include the three regions above — a whopping 0x8000 = 512KB region. It would have 0x10000 size subregions, so the first subregion (0x40000000 to 0x4000FFFF) could be disabled. Thus the actual region would cover from 0x40010000 to 0x4007FFFF. This includes about 20 other peripheral regions such as Ethernet, GPIO, SPI, Timers, and ADCs. The USB host task should not have access to these — it would be a field day for a hacker who infected the USB host partition!

Alternatively, these regions can be loaded into 3 auxiliary MPA slots and, when needed, a region can be swapped into the active IO slot in the current task's MPA and in the MPU. This is illustrated in Figure 7.2.

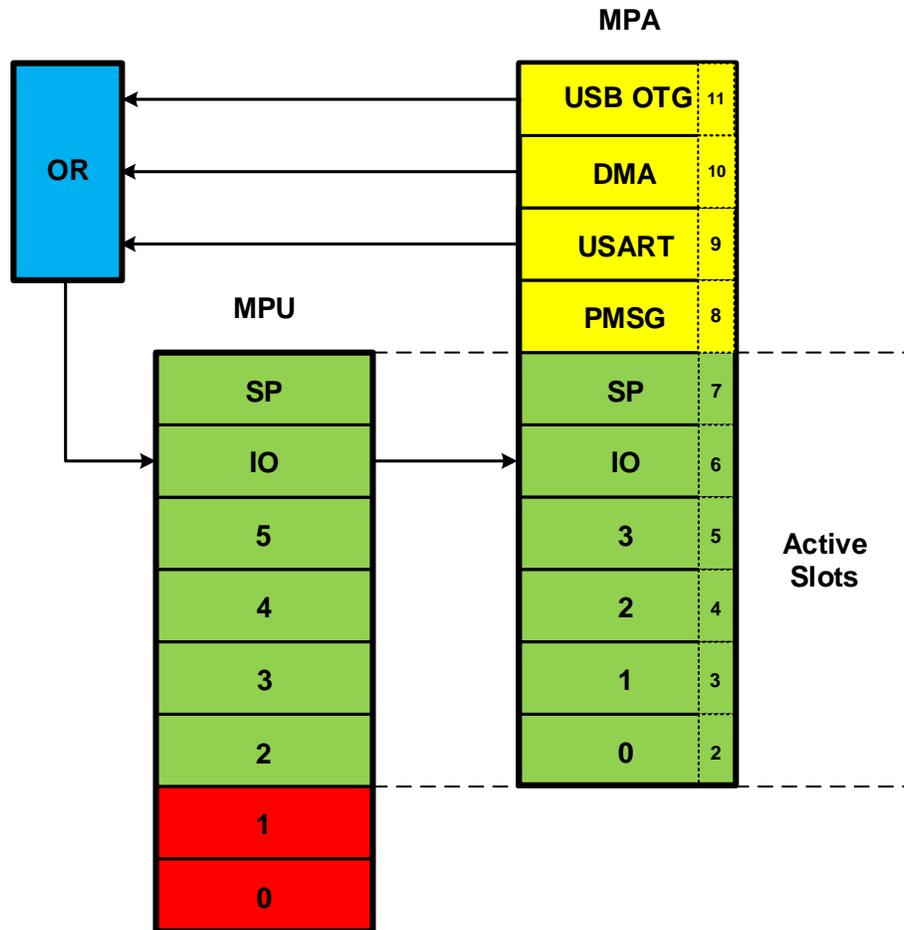


Figure 7.2 Swapping IO Regions

The small amount of time required to swap IO regions is well worth the increase in partition isolation and thus security. Example code is as follows:

```

mp_MPASlotMove(6, 9);
/* USART code */
mp_MPASlotMove(6, 11);
/* USB OTG code */
...

```

The places to put slot moves can be found by running the code and finding where MMFs occur, then putting slot moves there. `mp_MPASlotMove()` is fast, so this should not seriously impact performance.

Dynamic Slots

Templates and MPAs can contain *dynamic slots*. This is useful if templates are stored in ROM, for security, and thus cannot be changed during run time. A dynamic slot has a pointer to where its region is stored. When the MPA is created, the region is loaded into

the dynamic region of the MPA. The advantage of dynamic slots is that regions of needed sizes can be created on-the-fly. For example, they can be allocated from a heap. This allows adjusting to different installation or operational requirements at run time. It is important to note that MPAs, and hence dynamic slots, can be created only in pmode and thus only by trusted software. A hacker who has penetrated a umode partition cannot create dynamic regions other than via system calls to create *pblocks* and *pmsgs*, which were discussed in Chapter 4.

Multi-task Partition Templates

In many cases, a partition will have only one task. However, as discussed above, it may be desirable have child tasks in addition to the main task. This allows offloading some regions to child tasks, thus staying within the MPU slot limit. The method for assigning partition regions to the MPAs of partition tasks is illustrated in Fig. 7.3.

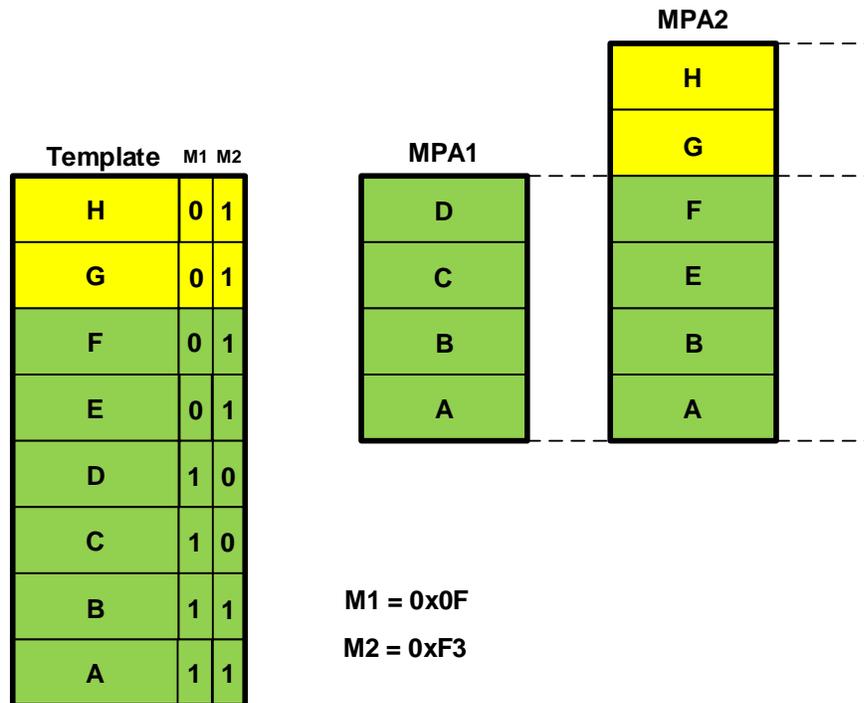


Figure 7.3 Multiple MPAs from One Template

For simplicity, this figure assumes a 4 slot MPU. When MPA1 is created, mask M1 selects active regions A, B, C, and D. When MPA2 is created, mask M2 selects active regions A, B, E, and F and auxiliary regions H and G. Thus the 4 slot MPU limit is met for the partition, even though the partition template has 6 active slots.

Critical Code Sections

Critical code sections, particularly in low-level driver code, are generally protected by *interrupt disable* before and *interrupt enable* after them. When moving a partition from

Achieving Device Security

pmode to umode, both of these instructions become NOPs and thus lose protection. One might think that SVC shell functions could be created for them. This is a Catch-22 situation – whereas the interrupt disable function would work, the interrupt enable function cannot work because interrupts and exceptions, including SVC, are disabled! Additionally, it would be bad to allow umode code to disable all interrupts indefinitely.

Instead, SVC functions are provided to mask and unmask interrupts. Permitted interrupts are specified on a task basis so that a hacker cannot mask or unmask interrupts used outside of the partition he has infected.

Scheduler Callbacks

Most RTOSs provide EXIT and ENTER scheduler callbacks. The former can be used to save an extended state (e.g. coprocessor registers) when a task suspends, and the latter can be used to restore the extended state when the task resumes. SMX also provides START and DELETE callbacks. When a task first starts running, the former can be used to do task initialization and to get resources that the task needs. When the task is deleted, the latter can be used to release the resources and do task cleanup. By placing the DELETE case below the START case in the callback function switch statement, it is easy to see if anything has been missed. This facilitates implementing partition-only stop and reboot, without leaking resources. The following code shows how these cases are used:

```
SEM_PTR sbr;
u8*      bp3;

void ut2a_CBF(u32 mode, u32 task)
{
    switch (mode)
    {
        case SMX_CBF_START:
            sbr = smx_SemCreate(SMX_SEM_RSRC, 1, "sbr");
            bp3 = (u8*)smx_HeapMalloc(128, 0, 3);
            break;
        case SMX_CBF_DELETE:
            smx_HeapFree(bp3, 3);
            smx_SemDelete(sbr);
            break;
        default:
            smx_EM(SMXE_INV_PAR, SMX_ERRH_UNK);
    }
}

void ut2a_main(u32)
{
    ...
}
```

```

void cbf_demo(void)
{
    ut2a = smx_TaskCreate(ut2a_main, TP2, TS_SSZ, 0, "t2a");
    smx_TaskSet(ut2a, SMX_ST_CBFUN, (u32)ut2a_CBF);
    smx_TaskStart(ut2a);
    smx_TaskDelete(&ut2a);
}

```

In the above example ut2a is created with the ut2a_CBF callback function. When ut2a starts, the sbr semaphore is created for it. When ut2a stops or is deleted, the ut2a semaphore is deleted. Similarly bp3 is allocated from heap0 on start, then freed back to it on delete. ut2a is a utask, but ut2a_CBF executes in pmode. So this is also a convenient way to initialize a utask in pmode before it starts running.

smxAware

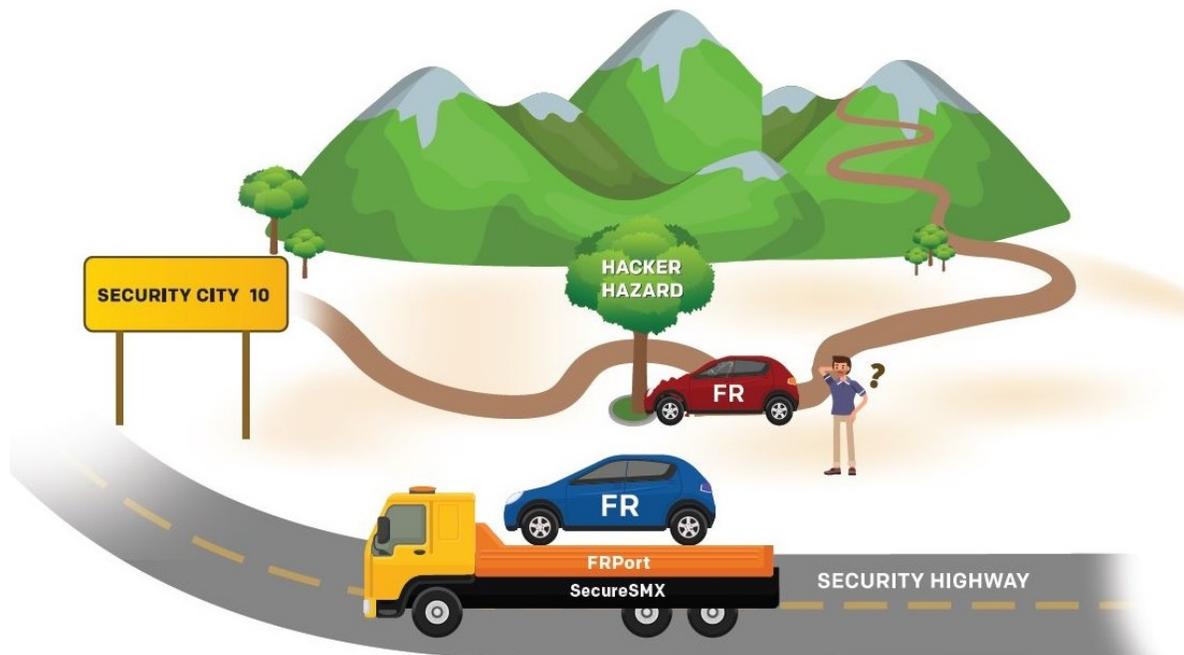
smxAware provides kernel awareness for the IAR C-SPY debugger. When used with SecureSMX, the MPU and MPA displays make it much easier to see region sizes and attributes when tracking down MMFs. Also, portal operations are shown in the event timelines graph, and MPU regions are shown in the memory map overview graph. See Appendix B for more information on smxAware.

Event Monitoring

A large number of events are monitored, such as service calls, ISR runs, LSR runs, task operations, errors, and user events. Relevant information for each event is stored in the event buffer, EVB. In addition, user events can be defined and logged. Logging can be filtered by event group so the EVB does not fill up too quickly. Periodically uploading EVB to a security monitoring site allows special software to look for anomalous behavior that might indicate an attack is in progress. If so, Security Control can take appropriate action, such as shutting down the partition.

Monitoring operation of all elements of a large system can be the only way to stop highly sophisticated attacks that evade security mechanisms and slowly penetrate computer networks.

Porting Applications to SecureSMX



It is not possible to run SecureSMX on other RTOSs, because it depends upon the rich features of SMX that are missing in other RTOSs. Thus applications must be ported to SMX in order to use SecureSMX. However, this is not as hard as it sounds. In order to ease this operation, FRPort and TXPort are included with SecureSMX. They provide porting functions that port nearly all FreeRTOS and ThreadX service calls used in applications to equivalent smx service calls. Moving applications to SMX should result in better operation as well as permitting access to all of the security features in SecureSMX. More ports are planned.

Frameworks

The ideal way to start a new project is to create a *framework* consisting of all partitions that will be needed in the final product. To do this, start with a diagram that shows the partitions or modules needed and the connections between them. In addition, the pmode barrier should be shown to indicate where each partition resides. Then create a task per partition and assign it main code that initially simply loops to approximate the expected execution time for the task. It is desirable to also assign a callback function with a START case to initialize the task and a DELETE case to delete the task cleanly. Also create an MPA template for each partition with expected regions, each containing expected amounts of ROM and RAM for them. Create an MPA for each task. If there are too many regions, this is a good time to define child tasks to take some of the regions.

Add system boot and initialization code, as well as dummy ISRs and LSRs that activate tasks, as expected.

For each connection between partitions, determine which type of portal is necessary, then create the portal. Portals should implement the expected APIs for their respective partitions. Initially, these might have just a few functions. The servers should have stubs that perform some action or return something. Add code to the clients to make portal calls.

When the initial framework is done, it should compile, link, and actually run. Now legacy code can be added to its partitions and the framework should start running in a manner similar to the final system. From here the remaining partitions can be coded using agile and CI/CD techniques while using the framework for testing. As coding progresses the framework will behave more and more like the final system. The main advantages of the framework approach are:

- Familiarity with security concepts and tools is developed early.
- Interface problems are worked out before a lot of code is written.
- Code is developed iteratively using agile techniques.
- CI/CD means delivering code to the framework for testing with it.
- Third party code can be contained in isolated partitions.
- When the last code is delivered and tested, no integration is left to do.
- Security is baked into the final product.

Because of the advantages above, building in security may not increase development time and cost -- it might actually reduce them, and the result will be a superior product.

Debugging

An ordinary debugger such as IAR C-SPY works fine for partition debugging. However, there are some differences from normal code debugging, as follows:

1. **MMFs.** When moving a partition from pmode to umode, a large number of MMFs are likely to be encountered. All one can do is fix the current problem, run the code to the next MMF, and repeat the process. The call stack is helpful tool. Clicking on the top entry takes you to the exact point in the code that caused the MMF. Usually this will be a function or variable outside of the partition, so it is easy to fix. Sometimes, however, it will be a parameter of the function. The parameter may be outside of the partition or it may be a handle (see below). Really tough problems are best solved by breaking at the point of MMF, then stepping through the code in the disassembly window to see the actual instruction (usually an LDR) causing the MMF.
2. **Handles.** We become so accustomed to using handles that we forget that they are the addresses of pointers to RTOS objects. The compiler dereferences the handle in order to pass the value of the pointer as an argument. It is easy to forget that an

Achieving Device Security

object was created outside of the current partition, hence the address of the handle is outside of the current partition and triggers an MMF. One can struggle with this problem for a long time, without seeing it. The solution is to step through the disassembly code. You will see an LDR into a register, then an LDR using that register, and then an MMF.

3. **Broken Call Stacks.** We are accustomed to using call stacks to trace a problem back to its origin. For example, a file system function may be failing due to a wrong parameter in the file system service call. Going back to the origin of the call makes fixing a problem like this, easy. When a direct call API is replaced with a portal, the call stack is broken because the file system server and the file system client are implemented in different tasks. To counter this problem, the caller of the API is saved in the caller field of the service header (SH) structure. This can be entered into the disassembly window, and a breakpoint can be set there to run to the point of call. When reached, the call stack window will be refreshed, and it will now show the sequence of calls leading to that point.
4. **Wild Pointers.** Uninitialized and corrupted pointers will usually trigger MMFs, making it easy to find them.

MPUMapper

MPUMapper is another utility shipped with SecureSMX. It runs after the ILINK linker and modifies the .map file so it is possible to see what is where. For example:

```
.ucom_text      ro code    0x20'1732      0xdc  bcc.o
  _itoa          0x20'1733
  _ltoa          0x20'173b
  _ultoa        0x20'17a1
.ucom_text      ro code    0x20'180e      0x24e  cpcli.o
  sbp_ConClearScreen 0x20'180f
  sbp_ConClearScreenUnp 0x20'183b
.ucom_rodata    const      0x20'1bd4      0x4    bspm.o
  sbu_ticktmr_cntpt 0x20'1bd4
```

The above shows what functions are in .ucom_text and what data is in .ucom_rodata. Therefore, searching on a function or variable reveals where it is, thus making it easier to put where it should be.

8. Conclusion

SecureSMX is intended to provide a flexible set of tools and a structure to improve the security of existing and new MCU-based systems. It allows doing so with minimal modification of trusted, legacy code. Its inherent flexibility permits fixing the most important problems first and gradually improving total system security.

If SecureSMX is used as the foundation for a new system, it is likely that strong security can be implemented with little or no schedule nor development cost increase. This is because it provides hardware-enforcement of design practices proven to reduce integration and debug time. And the downstream payoff, in terms of security protection, is huge. For all of the features presented above, the code size of SecureSMX is about 10 KB (excluding SMX size). Of course, a large amount of application code must be added, so the total size increase can be much larger.

In an ideal system, as much code as possible has been moved from pmode to isolated umode partitions, portals are used for inter-partition communication, all untrusted tasks are runtime and token limited, deferred interrupt processing is done in safe LSRs, and mission-critical code and data are doubly protected by the pmode barrier.

While this goal is achievable for new designs, it is not likely to be practical for existing designs. It may be that moving all untrusted code into a single umode partition and applying some limitations to it is an adequate solution for an existing design. SecureSMX is specifically designed to provide the flexibility to implement partial solutions. It also is designed to permit incremental improvements, wherein the security team makes one improvement at a time, thus achieving gradual security improvement over time.

It is important to recognize that other security measures are still needed, such as root-of-trust, secure boot, secure update, encryption, and code improvement. In this context SecureSMX provides a firm security foundation and offers many new options to deal with security problems.

For the project manager, partitioning allows putting the best programmers on the most important partitions and other programmers on the other partitions. Isolation guarantees protection of important partitions from less important partitions. Although a hacker might be able to hack one of the weaker partitions, he will not be able to get to the good stuff in the important partitions. Hence your device is safe from serious damage due to being hacked.

For more information on SecureSMX, please visit www.smxrtos.com/securesmx. Please email me at ralph@smxrtos.com if you have questions. I will be happy to answer them. Please put “RTOS” or “SecureSMX” in your title so your email will not be filtered out.

References

1. Jean Labrosse, [*Using A Memory Protection Unit With An RTOS*](#), 5/18.
2. Ralph Moore, [*Is Your Thing in Danger?*](#), 3/24.
3. Ralph Moore, [*Where's The Gold?*](#), 3/24.
4. Doug Lea, [*A Memory Allocator*](#), 12/96.
5. Paul R. Wilson, [*Dynamic Storage Allocation: A Survey and Critical Review*](#), 9/95.



Ralph Moore is a graduate of Caltech. He and a partner started Micro Digital Inc. in 1975 as one of the first microprocessor design services. In 1989 Ralph decided to get into the RTOS business and he architected the *smx* RTOS kernel. After 20 years of selling Micro Digital products and managing the business, he went back into product development. Recent projects include eheap, SecureSMX, FRPort and TXPort. Ralph has three children and six grandchildren.

Appendix A eheap

Introduction

eheap has been developed specifically for embedded systems. It has numerous features that are useful for embedded systems, especially partitioned systems. These features are discussed below.

Doubly Linked Chunks

Heaps consist of linked *chunks*. In each chunk is *metadata* used by the heap and the *data block* returned to the user. In *dmalloc* (see Ref. 4) and many of its derivatives, the block size is at the start of each chunk and also at the end of each free chunk. This achieves a very high memory efficiency – only 8 bytes per inuse chunk. However, it is only possible to trace forward (by adding sizes to the heap start), and not backward through the heap.

A better design for embedded systems is a forward link and a backward link in every chunk. This permits continual forward heap tracing during idle periods to find broken links, and when a broken link is found, backward tracing to fix it. Backward tracing is also necessary to check each link when tracing forward. Heaps contain vital information such as task stacks, memory protection arrays, and other system and application control structures. Hence, *heap self-testing and self-healing* is important for embedded systems exposed to background radiation, electrical disturbances, heat, and other environmental phenomena, not to mention hackers.

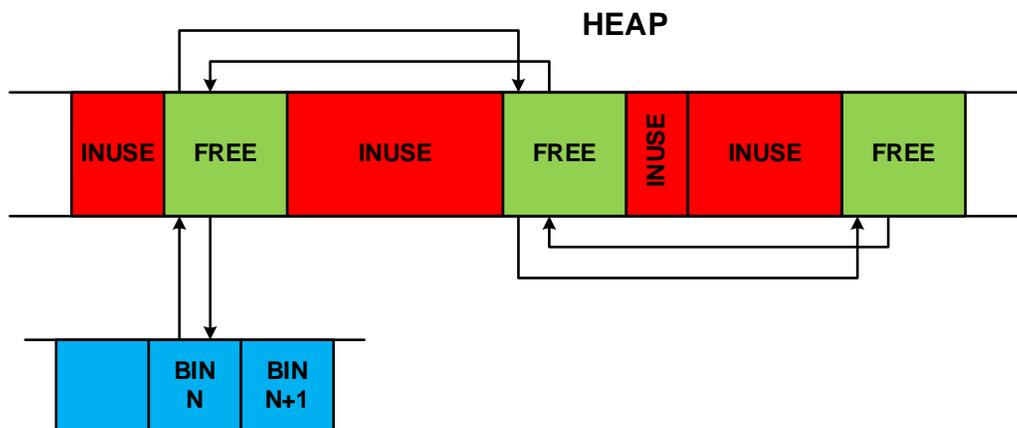


Figure A.1: Heap Bins

Heap Bins

dlmalloc was one of the first heaps to use *heap bins*, and most other heaps developed since dmalloc also use them. A heap bin consists of doubly linked free chunks, in a certain size range, linked to a bin header. Normally the smallest-size bins have only one size and are organized by successive chunk sizes (e.g. 24, 32, ..., 56) up to a maximum size. This is referred to as a *Small Bin Array (SBA)*. Allocating or freeing an SBA chunk is very fast – the size is used as an index into the SBA, and the first chunk is taken. Above the SBA are the *upper bins*⁶, which consist of a mixture of *small bins* and *large bins*. Small bins have one size; large bins have a range of sizes (e.g. 128 to 248, 256 to 504, etc.). The last upper bin is called the *top bin*, and it has all of the remaining sizes up to the *chunk size limit*.

Accessing the correct upper bin requires size comparisons, then searching the bin for the best match. If a big-enough chunk is not found, the first chunk in the next larger occupied bin is taken. This can be time-consuming, but not as bad as searching a serial heap with no bins. Each upper bin is, in effect, a small heap.

The problem with dmalloc, and its variants, is that the number of bins and trees is fixed. This may be ok for the main heap, but it is not a good match for partition heaps. A given partition may use only a few block sizes, thus it needs only a small SBA, if any, and only a few large block sizes. Thus a *configurable bin structure* is highly desirable for partition heaps, both for memory efficiency and for performance. For example, the following bin structure might work well for a network partition:

```
u32 const binsz2[] =
/*bin 0    1    2    3    4    end */
    {24, 512, 1024, 1526, 1534, -1};
```

In this case, there is no SBA. The first bin covers all chunk sizes from 24 to 504⁷. The next two bins cover all chunk sizes up to 1518. Bin 3 has a single chunk size of 1526, which can hold the maximum Ethernet frame size of 1518 bytes. Like an SBA bin, bin 3 requires no searching – the first chunk is taken. Bin 4 is the top bin and it contains all chunk sizes from 1534 on up to the chunk size limit.

An even simpler partition might contain only one bin:

```
u32 const binsz1[] =
/*bin 0    end */
    {24, -1};
```

⁶ Not applicable to dmalloc, which uses *trees*; trees are much more complicated than large bins.

⁷ Block sizes are 8 bytes less than chunk sizes, and chunk sizes are spaced 8 bytes apart.

This could be a partition that allocates mostly permanent blocks. In this case, the chunks would come primarily from the *top chunk*⁸ and seldom from bin0.

Large Bin Sorting

Generally, embedded systems have significant idle time in order to be able to handle peak loads. Large bin sorting can be done during idle times. If a large bin has been sorted, the first *big-enough chunk* is also the *best-fit chunk* in the bin. Assuming the rule is to always take the first big-enough chunk, sorting reduces unnecessary *chunk splitting* and saves bigger chunks for larger requests. (Note: Finding the best-fit chunk in an unsorted bin usually takes too much time.)

Merge Control

When a chunk is freed it may be merged with *adjacent free chunks*. dlmalloc has no merge control – freed chunks are always merged with adjacent free chunks. This tends to deplete bins because it results in removing a free chunk from a lower bin, merging it with the freed chunk, and placing the resulting free chunk into a higher bin. The next time this chunk size is needed, its bin may be empty. Then it becomes necessary to get a larger chunk from a higher bin, split it, and store the resulting free chunk in a lower bin. Thus two unnecessary operations are necessitated: merge and split, which hurts performance, and the allocation time is longer. If a chunk is in a bin, it will probably be needed again.

A better policy for embedded systems is not to merge until a specified *upper threshold* is reached, such as HEAP_USE_MAX, and then merge all freed blocks until a *lower threshold* is reached, such as HEAP_USE_MIN. This operates like a thermostat. Other criteria could be used, such as number of free blocks, total free block bytes, etc. In defense of dlmalloc, research has shown (See Ref. 5) that there is no universal solution to avoiding allocation failures due to fragmentation. Hence, merge control may be dangerous and possibly should be turned off. However, embedded systems, particularly individual partitions, tend to have regular behaviors, and merge control probably will not cause allocation failures for them.

Heap Recovery

In the event of an allocation failure, a heap recovery function is automatically called. It traces through the heap to find enough adjacent free blocks to satisfy the request, merges them together, and returns control to the allocation function. Thus the allocation takes longer but does not fail.

⁸ An eheap starts out as a *donor chunk* for the SBA and a *top chunk* for upper bin chunk sizes. If there is no SBA, there is no donor chunk and the whole heap starts out as a single top chunk.

Aligned Blocks

Dynamically allocated *protected blocks*, *pblocks*, and *protected messages*, *pmsgs*, must be aligned on power-of-two boundaries in order to be used as MPU regions. An efficient process for doing this is as follows:

1. Find the first large-enough free chunk for the desired block size, *sz*.
2. Find the first 2^n alignment boundary inside the chunk's data block, where 2^n is the next power of two $\geq sz$.
3. Test if the remainder of the chunk is $\geq sz$.
4. If not, go on to the next large-enough chunk.
5. When a big enough chunk has been found, put its *Inuse Chunk Control Block (ICCB)* below the boundary – i.e. just below the aligned data block.
6. The resulting space below the ICCB is called *free space*, and it is handled as follows:
 - a. If the preceding chunk is free, combine the free space with it.
 - b. Else, if the free space is large enough, make it into a free chunk.
 - c. Else, combine the free space with free space at the end of the preceding inuse chunk, and if the result is big enough, make it into a new free chunk.
7. Split off space after the block, if large enough for a free chunk, else make it free space.

This process is illustrated in Figure A.2. In this figure, ICCB = Inuse CCB, FCCB = Free CCB. In this case, option 6b has been taken and a small free chunk has been formed below the new data block. Some spare space is left at the top of this chunk because the space is not large enough to form a free chunk.

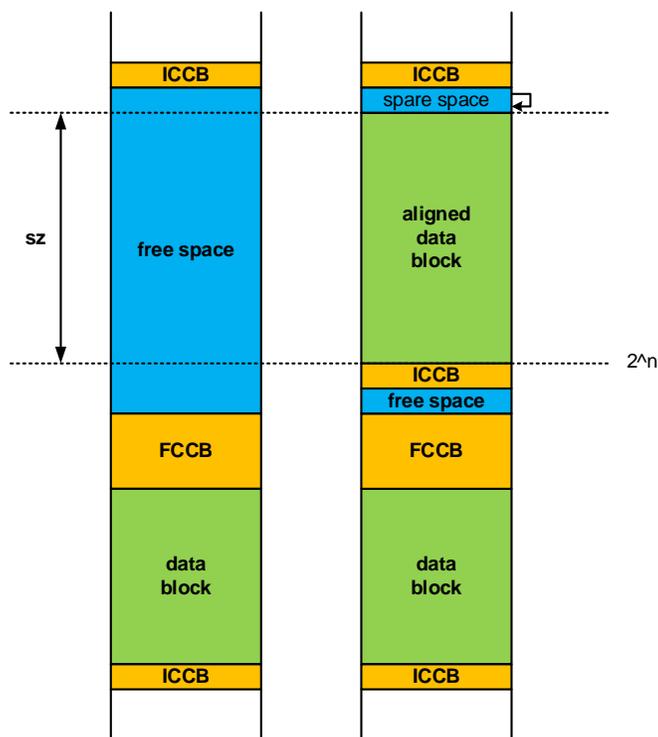


Figure A.2 Aligned Allocation

Over a period of time, the heap will start to be organized into chunks having aligned data blocks, and aligned allocations will become faster.

Region Blocks

The foregoing is adequate for v8M MPU regions when specifying alignment and size as multiples of 32, but v7M MPUs require additional steps:

1. Determine the *region size* as the next larger power of two. For example, if $sz = 630$, then the region size = 1024.
2. Determine the subregion size and the number of contiguous subregions needed. In the example, subregion size = 128, and $5 * 128 = 640 > 630$, so $N = 5$.
3. Do aligned search steps 1 - 3 with alignment = 128 and size = 640.
4. After step 3: Verify that all N subregions are in the same region – i.e. find the next region boundary (e.g. multiple of 1024) and verify that the last subregion ends before it.
5. Do steps 5 -7.

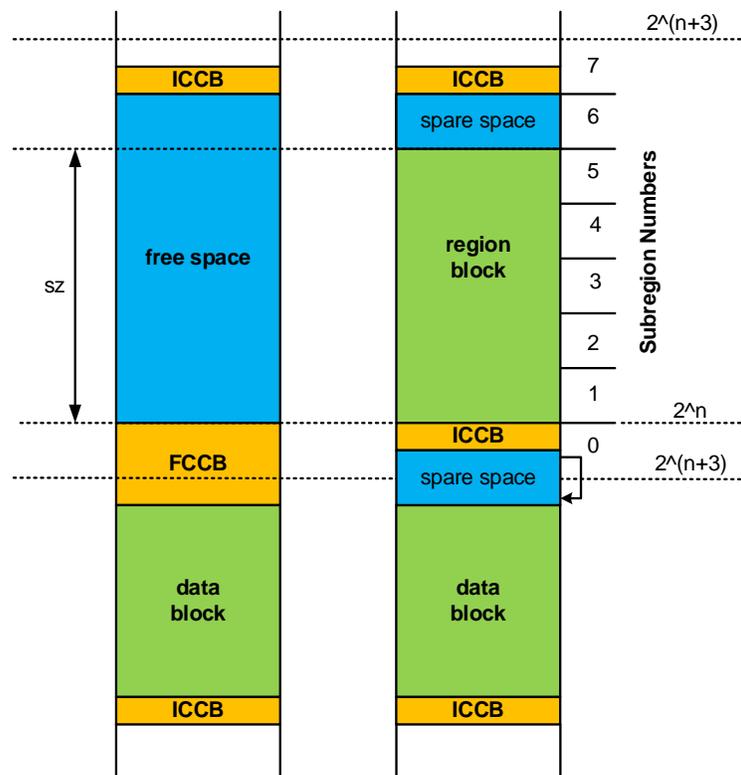


Figure A.3 Region Allocation

This results in a subregion-aligned v7M region block that is contained in contiguous subregions within a region, as shown in Figure A.3: In this example, subregions 1-5 will be enabled, and subregions 0, 6, and 7 will be disabled. Note how the disables protect the

Achieving Device Security

surrounding heap CCBs and spare space. Because the region block is subregion-aligned, it is easier to find than if it were region-aligned and it causes less heap disruption.

Chunk Types

Figure A.4 shows three types of chunks, each with a different *Chunk Control Block (CCB)* (Orange). All three CCBs have a *forward link* to the next CCB and a *backward link* to the previous CCB. In addition, the free CCB has chunk size, forward and backward bin links, and bin number * 8. This requires 24 bytes, so that is the minimum chunk size. The inuse CCB requires 8 bytes for links, so the smallest data block is $24 - 8 = 16$ bytes.

The third type of chunk is a *debug chunk*. The debug CCB is an inuse CCB with chunk size, time of allocation, owner, and a fence added. In addition the data block has N fences above and below it. These things are useful during debugging to find memory leaks, block overflows, and other problems. Whether an inuse chunk or a debug chunk is generated by an allocation depends upon whether the heap's debug mode is off or on, respectively. This permits limiting debug chunks to code of interest, which is useful since they can be much larger than inuse chunks (e.g. $N = 32$).

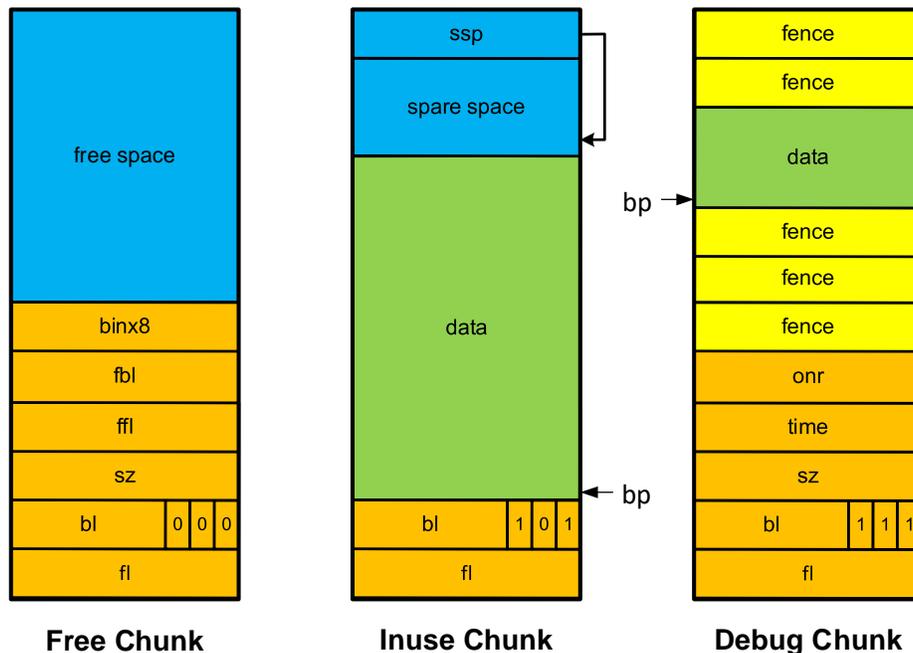


Figure A.4 Heap Chunk Types

Integrated Block Pools

16 bytes + 8 for the CCB is rather large for many C++ objects. To alleviate this problem, smaller block pools can be integrated into the heap. Then allocations of less than 16 bytes

are taken from the block pools in the heap. This is fast and commensurate with the needs of object-oriented code. A useful by-product of integrating block pools into a heap is that if a block pool runs out of blocks, the block comes from the heap, instead. This may result in slower performance, but the system does not break. When freed, blocks go back to their respective sources.

It is likely in modern embedded systems that some partitions (especially third party software) will be written in C++. A heap with integrated block pools may result in much better performance for the partition and may use less memory.

The Need for Mutexes

All RTOSs use some mechanism to protect critical sections of system service routines. Whatever method is used, the net result is that no other task can run during these critical sections. That approach is not workable for multiple heaps that are managed by the same heap code. Instead, we use a mutex per heap in order to limit one task at a time to access a heap, while allowing higher priority tasks to preempt and access other heaps.

Summary

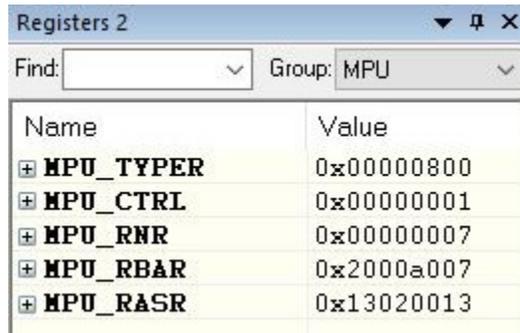
As can be seen from the foregoing, the heap choice is an important part of partitioning a system. The right heap can improve performance and reduce code rewriting. Providing dynamic regions is also an important heap requirement. *ehp* provides all of the features discussed above. For more information see the [ehp web page and the eheap User's Guide there](#).

Appendix B smxAware

smxAware has been enhanced with new features to help debug issues related to the Memory Protection Unit in SecureSMX systems. These are covered here. See the smxAware User's Guide for full documentation of them.

MPU Display

The IAR debugger displays the MPU registers like this:



The screenshot shows a window titled 'Registers 2' with a search bar and a 'Group: MPU' dropdown. Below is a table of registers:

Name	Value
MPU_TYPER	0x00000800
MPU_CTRL	0x00000001
MPU_RNR	0x00000007
MPU_RBAR	0x2000a007
MPU_RASR	0x13020013

RNR can be manually patched to the slot number to view, which then shows the RBAR and RASR values for that slot. A separate display shows RBAR and RASR for the 4 slots starting at slot RNR. smxAware provides a much better view of the MPU and also provides similar views for all MPAs, as shown below.

This is a huge help in determining the cause of a Memory Manage Fault (MMF). The MPU display shows the regions currently in the MPU. The MPA display shows the MPA images for all tasks in the system. Each shows the regions that will be loaded into the MPU on a switch to that task.

The following shows the MPU when LED_task is running:

```

smx Objects
├─ Tasks
├─ Stacks
├─ MPA
└─ MPU
    Current Task: (LED_task)          0x20002b3c
    MPU ON      BR ON      CPU: PRIV MODE MSP
    MPU[0] Enabled rbar 08000000 rasr 0602c01d "ucom_code"
    Start      08000000
    End        08007fff
    Subreg Dis 6,7 (Size 0x1000)
    Sub Start  08000000 (Size 0x6000)
    End        08005fff
    Attributes CODE
    MPU[1] Enabled rbar 20015801 rasr 13020015 "ucom_data"
    Start      20015800
    End        20015fff
    Size      00000800
    Attributes DATARW
    MPU[2] Enabled rbar 08035c02 rasr 06028013 "led_code"
    Start      08035c00
    End        08035fff
    Subreg Dis 7 (Size 0x80)
    Sub Start  08035c00 (Size 0x380)
    End        08035f7f
    Attributes CODE
    MPU[3] Enabled rbar 40020003 rasr 1300dd19 "GPIOBF"
    Start      40020000
    End        40021fff
    Subreg Dis 0,2,3,4,6,7 (Size 0x400)
    Sub Start  40020400 (Size 0x400)
    End        400207ff
    Sub Start  40021400 (Size 0x400)
    End        400217ff
    Attributes PIO
    MPU[4] Enabled rbar 40005404 rasr 13000013 "I2C1"
    Start      40005400
    End        400057ff
    Size      00000400
    Attributes PIO
    MPU[5] Enabled rbar 40023805 rasr 13000013 "RCC"
    Start      40023800
    End        40023bff
    Size      00000400
    Attributes PIO
    MPU[6] Disabled rbar 00000006 rasr 00000000 ""
    Start      00000000
    End        00000000
    Size      00000000
    Attributes
    MPU[7] Enabled rbar 20005007 rasr 13020013 "stack"
    Start      20005000
    End        200053ff
    Size      00000400
    Attributes DATARW
    └─ Heap
    └─ Semaphores
  
```

Achieving Device Security

A few things to discuss:

- A simple slot such as MPU[1] shows the Start, End, Size, Attributes, and name, as well as the actual MPU RBAR and RASR register values.
- A more complex slot such as MPU[0] also shows disabled subregions. Usually disabled subregions are at the end, but as shown for MPU[3] (GPIOBF), they can be any of the 8 subregions. Sub Start/End indicate the area mapped by the enabled regions. In this case, 1 and 5 (missing from the disabled list) are enabled, and the Start/End show their extents.

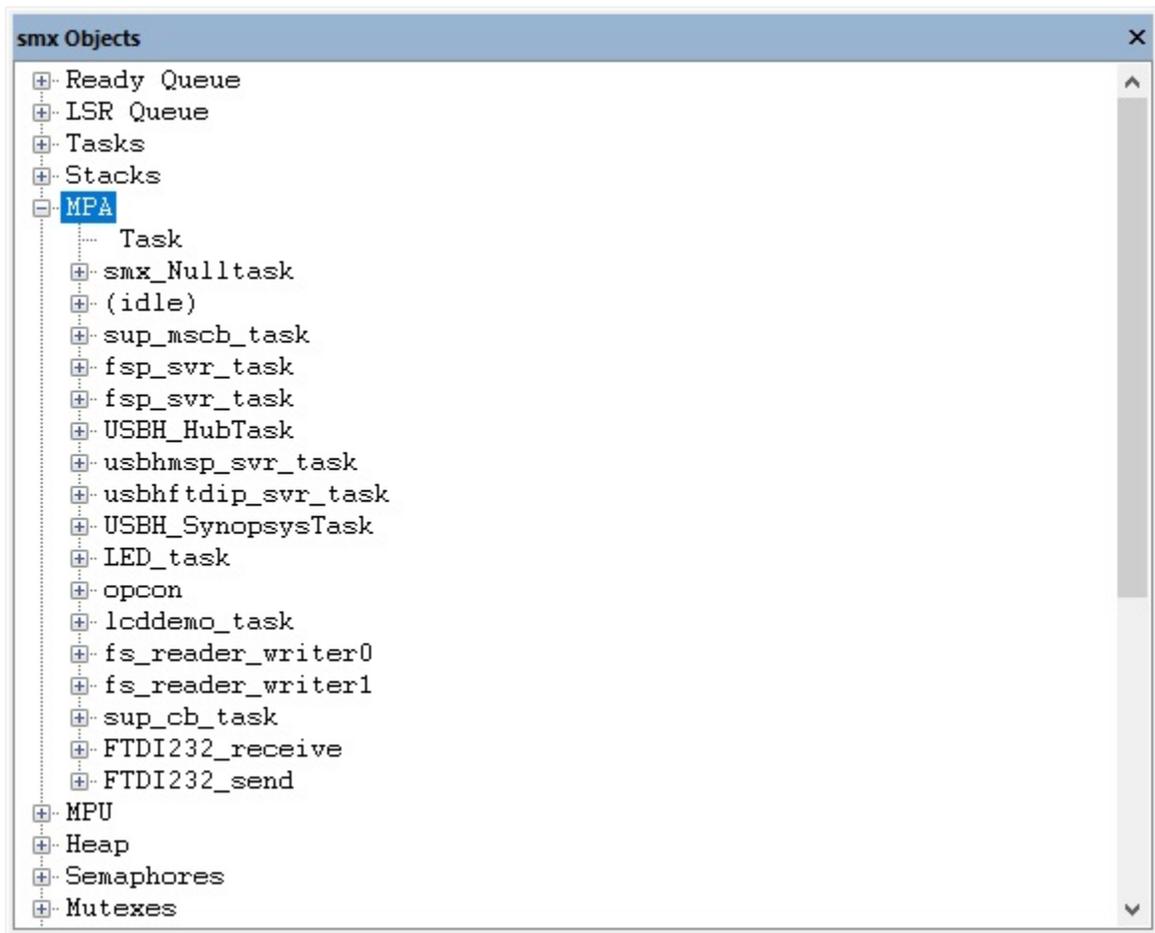
```
MPU[3] Enabled rbar 40020003 rasr 1300dd19 "GPIOBF"
Start          40020000
End            40021fff
Subreg Dis     0,2,3,4,6,7 (Size 0x400)
Sub Start      40020400 (Size 0x400)
End            400207ff
Sub Start      40021400 (Size 0x400)
End            400217ff
Attributes     PIO
```

- The name is assigned when built for debug. The MPU itself has no provision to name regions. For release, names can be turned off with a single switch, which makes the code use the fast MPU loading feature of the Cortex-M MPU.
- ucom_code contains common code such as SVC and portal client shell functions, and some common C library functions.
- Slot 6 is unused.
- This task's MPA template is defined in the code like this: (A template is loaded into an MPA when the MPA is created.)

```
MPA mpa_tmplt_led =
{
  RGN(0 | RA("ucom_code") | V, CODE | SRD("ucom_code") | RSIC(ucomcsz) | EN, "ucom_code"),
  RGN(1 | RA("ucom_data") | V, DATARW | SRD("ucom_data") | RSIC(ucomdsz) | EN, "ucom_data"),
  RGN(2 | RA("led_code") | V, CODE | SRD("led_code") | RSIC(ledcsz) | EN, "led_code"),
  RGN(3 | 0x40020000 | V, IOR | NO|N2|N3|N4|N67 | (12 << 1) | EN, "GPIOBF"),
  RGN(4 | 0x40005400 | V, IOR | ( 9 << 1) | EN, "I2C1"),
  RGN(5 | 0x40023800 | V, IOR | ( 9 << 1) | EN, "RCC"),
  RGN(6 | V, 0, 0),
  RGN(7 | V, 0, "stack"),
};
```

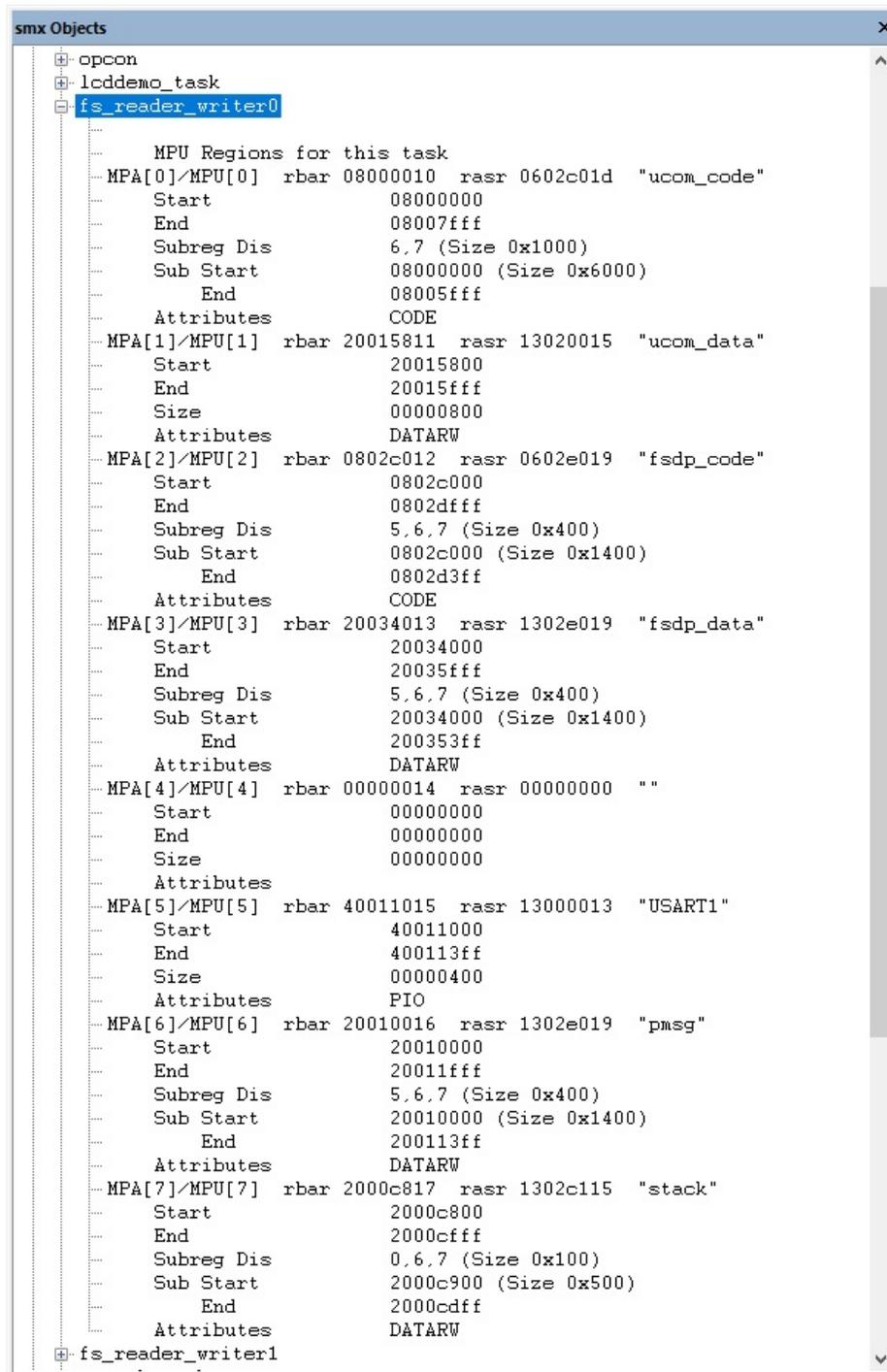
MPA Displays

The MPA tab lists all of the tasks:



Expanding one produces an MPA display similar to the MPU display. The following are example MPAs for the fs_reader_writer0 and the first fsp_svr tasks:

Achieving Device Security



```
smx Objects
├─ opcon
├─ lctdemo_task
└─ fs_reader_writer0
    ┌─── MPU Regions for this task
    │   MPA[0]/MPU[0] rbar 08000010 rasr 0602c01d "ucom_code"
    │   Start          08000000
    │   End            08007fff
    │   Subreg Dis     6,7 (Size 0x1000)
    │   Sub Start     08000000 (Size 0x6000)
    │   End            08005fff
    │   Attributes     CODE
    │   MPA[1]/MPU[1] rbar 20015811 rasr 13020015 "ucom_data"
    │   Start          20015800
    │   End            20015fff
    │   Size           00000800
    │   Attributes     DATARW
    │   MPA[2]/MPU[2] rbar 0802c012 rasr 0602e019 "fsdp_code"
    │   Start          0802c000
    │   End            0802dfff
    │   Subreg Dis     5,6,7 (Size 0x400)
    │   Sub Start     0802c000 (Size 0x1400)
    │   End            0802d3ff
    │   Attributes     CODE
    │   MPA[3]/MPU[3] rbar 20034013 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[4]/MPU[4] rbar 00000014 rasr 00000000 ""
    │   Start          00000000
    │   End            00000000
    │   Size           00000000
    │   Attributes
    │   MPA[5]/MPU[5] rbar 40011015 rasr 13000013 "USART1"
    │   Start          40011000
    │   End            400113ff
    │   Size           00000400
    │   Attributes     PIO
    │   MPA[6]/MPU[6] rbar 20010016 rasr 1302e019 "pmsg"
    │   Start          20010000
    │   End            20011fff
    │   Subreg Dis     5,6,7 (Size 0x400)
    │   Sub Start     20010000 (Size 0x1400)
    │   End            200113ff
    │   Attributes     DATARW
    │   MPA[7]/MPU[7] rbar 2000c817 rasr 1302c115 "stack"
    │   Start          2000c800
    │   End            2000cfff
    │   Subreg Dis     0,6,7 (Size 0x100)
    │   Sub Start     2000c900 (Size 0x500)
    │   End            2000cdfd
    │   Attributes     DATARW
    └─ fs_reader_writer1
```

fsdp_code and fsdp_data contain smxFs demo code that accesses smxFs via a portal. These do not contain smxFs library code and data. pmsg is a region for a tunnel portal message used by the client tasks to make indirect calls to the smxFs API.

By contrast, the smxFS portal server task has the smxFS library code and data regions, plus the IO regions to access SDMMC and DMA registers, as well as its own copy of the pmsg region:

```

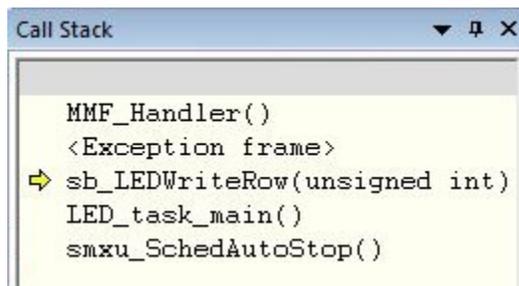
smx Objects
├── MPA
│   ├── Task
│   │   ├── smx_Nulltask
│   │   ├── (idle)
│   │   ├── sup_msch_task
│   │   └── fsp_svr_task
│       └── MPU Regions for this task
│           ├── MPA[0]/MPU[0] rbar 08000010 rasr 0602c01d "ucom_code"
│           │   ├── Start 08000000
│           │   ├── End 08007fff
│           │   ├── Subreg Dis 6,7 (Size 0x1000)
│           │   ├── Sub Start 08000000 (Size 0x6000)
│           │   ├── End 08005fff
│           │   └── Attributes CODE
│           ├── MPA[1]/MPU[1] rbar 20015811 rasr 13020015 "ucom_data"
│           │   ├── Start 20015800
│           │   ├── End 20015fff
│           │   ├── Size 00000800
│           │   └── Attributes DATARW
│           ├── MPA[2]/MPU[2] rbar 08020012 rasr 0602c01f "fs_code"
│           │   ├── Start 08020000
│           │   ├── End 0802ffff
│           │   ├── Subreg Dis 6,7 (Size 0x2000)
│           │   ├── Sub Start 08020000 (Size 0xc000)
│           │   ├── End 0802bfff
│           │   └── Attributes CODE
│           ├── MPA[3]/MPU[3] rbar 20020013 rasr 1302e021 "fs_data"
│           │   ├── Start 20020000
│           │   ├── End 2003ffff
│           │   ├── Subreg Dis 5,6,7 (Size 0x4000)
│           │   ├── Sub Start 20020000 (Size 0x14000)
│           │   ├── End 20033fff
│           │   └── Attributes DATARW
│           ├── MPA[4]/MPU[4] rbar 40012c14 rasr 13000013 "SDMMC1"
│           │   ├── Start 40012c00
│           │   ├── End 40012fff
│           │   ├── Size 00000400
│           │   └── Attributes PIO
│           ├── MPA[5]/MPU[5] rbar 40026415 rasr 13000013 "DMA2"
│           │   ├── Start 40026400
│           │   ├── End 400267ff
│           │   ├── Size 00000400
│           │   └── Attributes PIO
│           ├── MPA[6]/MPU[6] rbar 20010016 rasr 1302e019 "pmsg"
│           │   ├── Start 20010000
│           │   ├── End 20011fff
│           │   ├── Subreg Dis 5,6,7 (Size 0x400)
│           │   ├── Sub Start 20010000 (Size 0x1400)
│           │   ├── End 200113ff
│           │   └── Attributes DATARW
│           └── MPA[7]/MPU[7] rbar 2000a817 rasr 13020015 "stack"
│               ├── Start 2000a800
│               ├── End 2000afff
│               ├── Size 00000800
│               └── Attributes DATARW
└── fsp_svr_task

```

Using the map file produced by the linker, especially the modified version created by our MpuMapper utility, you can see what functions and variables are in each region.

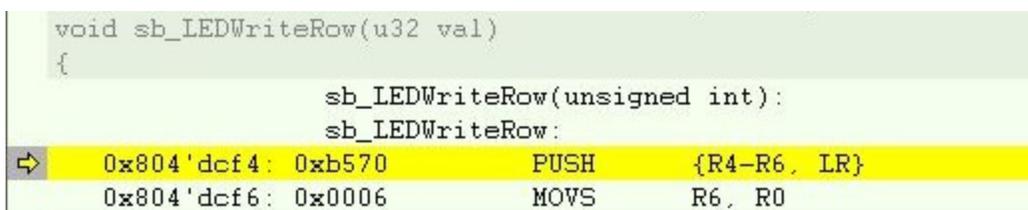
Finding Memory Manage Faults (MMFs)

The MPU/MPA display makes it possible to diagnose Memory Manage Faults because it lets you see all of the regions accessible to the current task and the extent of each region. To demonstrate this, an MMF is forced in our LED task. When it occurs, the debugger stops execution and the call stack looks like this:



```
Call Stack
MMF_Handler()
<Exception frame>
⇒ sb_LEDWriteRow(unsigned int)
LED_task_main()
smxu_SchedAutoStop()
```

The last call in the call stack before the handler is `sb_LEDWriteRow()`, and when double-clicked the disassembly shows it was on the first instruction.



```
void sb_LEDWriteRow(u32 val)
{
    sb_LEDWriteRow(unsigned int):
    sb_LEDWriteRow:
⇒ 0x804'dcf4: 0xb570    PUSH    {R4-R6, LR}
   0x804'dcf6: 0x0006    MOVS   R6, R0
```

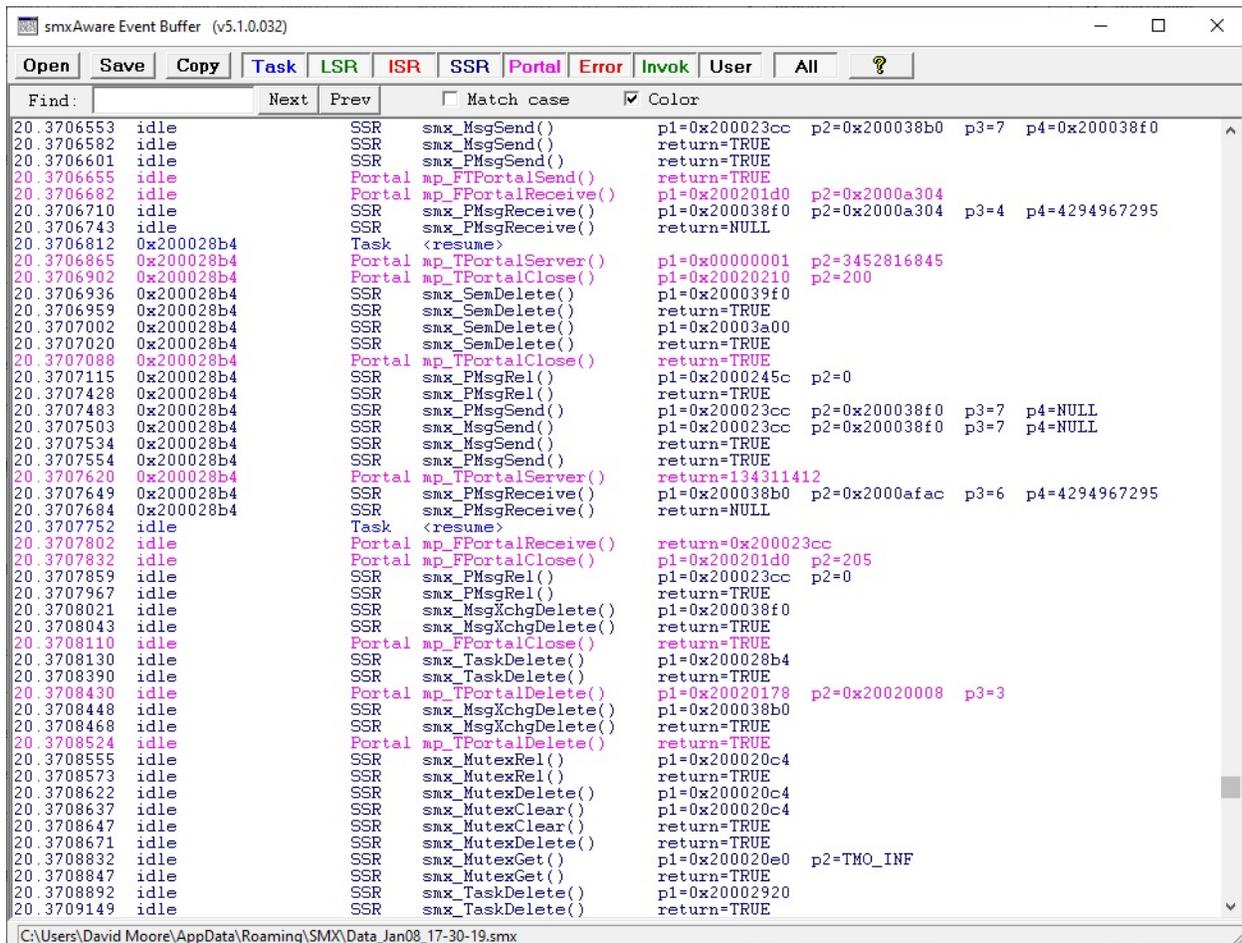
The address of the first instruction (0x0804dcf4) is outside all of the regions in the current MPU (see starts/ends in the MPU shown above), so the fault was caused by calling this function. It needs to be located in a code section that is in one of the regions shown, e.g. `ucom_code` or `led_code`. We normally put it in `ucom_code` for use by multiple tasks, but it could also be put in `led_code` if used only by `LED_task`. It is achieved by this pragma at the top of `led.c`:

```
#pragma default_function_attributes = @ ".ucom_text"
```

If the call stack and disassembly showed the fault occurred on a move instruction or other data access, then look at the registers to see what the attempted source and destination addresses were and whether either is out of range of all slots. Sometimes MMFs can occur due to unusual situations (such as region overlap on `v8`), so see the Debug Tips section of the Debug chapter in the SecureSMX manual for suggestions.

Event Display

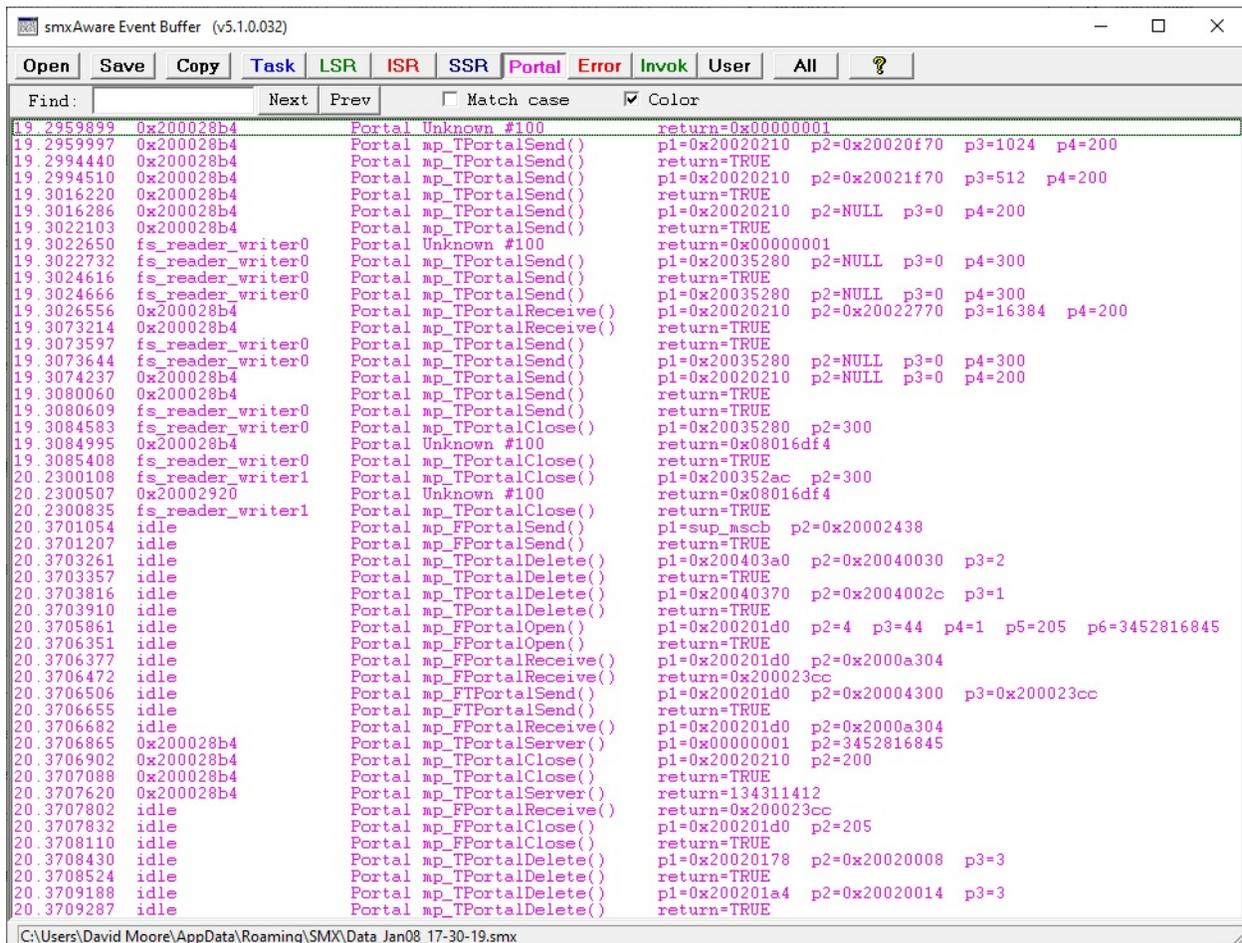
smxAware shows portal operations recorded in the Event Buffer like this:



In this example, all filter buttons at the top are enabled.

Achieving Device Security

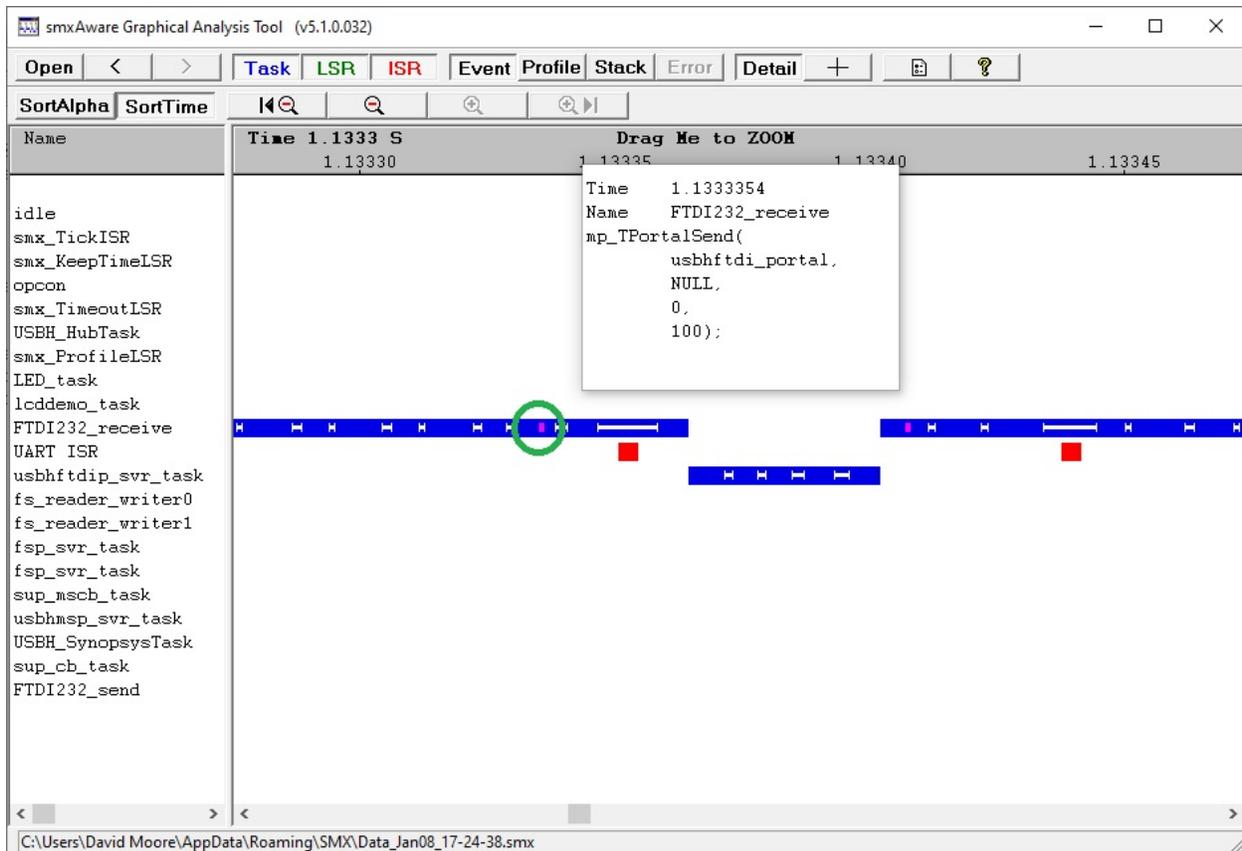
In the following example, only the Portal filter button at the top is enabled:



This allows seeing more portal operations in one screen. Similarly Task, LSR, etc. operations or any combination of operations shown by the buttons can be enabled.

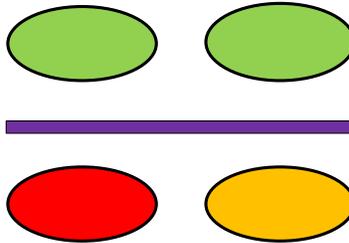
Timeline Display

Portal events are also shown on the timelines graph as purple dots. See the circled one below and the details about it that appear when the Details button is clicked.



Summary

smxAware is vital to see what the MPU currently looks like as well as the images for each task, in order to diagnose Memory Manage Faults. It is also helpful to debug portal issues by showing the events in the event buffer and the graphical event timelines display. We co-developed these new features right along with SecureSMX because we needed them to do the work. In addition, we have given detailed debugging tips in the manual to help solve problems you may encounter.



RTOS Security Specialists
www.smxrtos.com/seuresmx