MPU Security

by Ralph Moore

Introduction

MMUs vs. MPUs

For decades, large OSes have used isolated processes on microprocessors with MMUs to achieve separation and system security. Our recently enhanced product, MPU-Plus adds similar capabilities to SMX RTOS using a microcontroller with an MPU. This has been done for the Cortex-M architecture and will be extended to others in the future.

Memory Management Units (MMUs) are used with full Operating Systems (OSs) to provide isolated virtual memories to processes. Processes are independently compiled and linked, then individually loaded and run by OSs like Windows and Linux. Process to process isolation is good enough that if a process becomes infected by malware or starts malfunctioning for any other reason, the OS can usually shut it down with little or no damage to other processes. Hence, security is good. The use of MMUs is well-studied and well-understood. The downside to MMU usage is that it typically requires high-performance processors and very large memories.

Fast, power-hungry processors and huge memories are not compatible with the requirements for most embedded systems. In addition, full OSs do not have the real-time response times needed for many applications. Consequently, most embedded systems use low-power, moderate-performance Microcontroller Units (MCUs) with Real Time Operating Systems (RTOSs). These systems usually have small memories compared to full OS systems. For security many MCUs offer Memory Protection Units (MPUs), which are not capable of creating virtual address spaces as MMUs do.

In most embedded systems, we deal with *partitions* rather than *processes*. The idea is basically the same – partitions include one or more tasks and perform specific functions for a system. Just like processes, it is desirable to isolate partitions from each other so that if one partition is penetrated or begins malfunctioning, it can be stopped with minimal damage to the rest of the system. Unfortunately, this is not as easy to accomplish in embedded systems using MPUs as it is in full OS systems using MMUs. The biggest challenge here is that all MCU software is compiled and linked into a single executable. Also, MPUs impose limitations of their own. The use of MPUs for security is not well-studied, nor well-understood, and there are many complex tradeoffs involved, especially since embedded systems usually have small memories, moderate performance processors, power limitations, and the need for deterministic real-time performance.

Increasing Need for Security

If it is so hard, why bother? This seems to be approach adopted by the embedded systems industry over the past several decades. Unfortunately, it will not suffice for long because embedded systems are being connected into the Internet of Things (IoT) at a rapid rate. As a

consequence, once isolated, defenseless embedded systems are becoming accessible via the Hacker's Highway (AKA the Internet). This means TROUBLE.

Protection Goals

The **main goal** of protection is to protect trusted, critical software and data from less-trusted, non-critical software, which may become infected with malware or which is buggy. Examples of trusted software are the RTOS, exception handlers, security software (e.g. crypto, authentication, secure boot, secure update, etc.), and mission-critical software. Examples of less-trusted software are code vulnerable to malware attacks such as protocol stacks, device drivers, SOUP[1], and insufficiently tested code.

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

The **third goal** is to minimize the amount of trusted code since it is easier to carefully write small amounts of code. Running more code in unprivileged mode increases partition isolation, thus improving reliability.

The degree of protection that must be implemented depends upon the security and safety requirements of the specific system and the threats to which it is exposed. MPU-Plus provides a range of security services that enable achieving a level of protection appropriate for a given system. In addition, protection can be increased in future releases as a system becomes more widely distributed and therefore more vulnerable. MPU-Plus is designed to foster progressive protection improvement.

MPU-Plus Snapshot

The main purpose of MPU-Plus is to enhance the security of multitasking systems based upon the SMX® RTOS. The main things MPU-Plus does to help achieve better security are to:

1. allow defining different MPU templates for each task;
2. handle MPU switching during task switches;
3. provide an SVC API to allow unprivileged code to call privileged services;
4. limit which services can be called from unprivileged code and what they can do;
5. allow allocation of protected blocks and messages;
6. run the SMX RTOS and system code in privileged mode and middleware and application code in unprivileged mode;
7. allow tasks to be privileged or unprivileged.
8. serve as a platform upon which to build security.

---

[1] Software of Unknown Pedigree – typically third-party software.

MPU-Plus had two principal design goals:

1. Easier conversion of legacy code to use the MPU and the SVC API.

2. Allow developers to focus on protection strategies without being tripped up by MPU peculiarities and other hardware details.

Implementing a good protection strategy is difficult enough. Excessive complication at the detailed level is not only frustrating but may result in adopting less-than-ideal system protection.

Cortex-v7M

The Cortex-v7M processor architecture was introduced in 2005 and was intended for medium-size embedded systems. Since then, thousands of different Cortex-v7M Microcontroller Units (MCUs) have been developed by the semiconductor industry; they are used in tens of thousands of products developed by device manufacturers; and billions of chips have been shipped to date. It is by far the dominant MCU architecture (70% market share) and hence the one we support. If you are not familiar with it, the following references will be helpful:

1. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, Memory Protection Unit chapter, by Joseph Yiu, Elsevier Inc, 2014.

2. *ARM v7-M Architecture Reference Manual*, Arm Ltd. 2014, Chapter B3.5 Protected Memory System Architecture, PMSAv7.

3. *ARM Platform Security Architecture Overview,* Revision 1.2 Arm Ltd. 2018. See for security term definitions and security discussion.

The Cortex-v7M architecture offers the following security features:

1. Privileged Mode.

2. Supervisor Call (SVC) Instruction.

3. Memory Protection Unit (MPU).

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

1. Handler Mode: Privileged mode for ISRs, fault handlers, the SVC handler, and the PendSV handler. This mode can be entered only via an interrupt or an exception.

2. Privileged Thread Mode: Privileged tasks run in this mode. It can be entered only from handler mode, by setting CONTROL.nPRIV = 0.

3. Unprivileged Thread Mode: Unprivileged tasks run in this mode. It can be entered from either of the above two modes, by setting CONTROL.nPRIV = 1.

To reduce wordiness in the discussions that follow, we use an abbreviated terminology as follows: the first two modes are collectively called ***pmode*** and the third mode is called ***umode***. The p prefix can be interpreted either as *privileged* or *protected*; the u prefix can be interpreted as either *unprivileged* or *user*. Code and tasks that run in pmode are called *pcode* and *ptasks*; code and tasks that run in umode are called *ucode* and *utasks*

The critical aspect of the Cortex-M architecture is that pmode can be entered only via an interrupt or an exception. The SVC N instruction causes such an exception. It can be executed from umode with an 8-bit argument, N. This allows making a system call from umode where N specifies the function to call. The called function executes in pmode. SVC can also execute from pmode.

The MPU provides M slots for M *regions*. Each region has a starting address, a size, and access parameters, such as Read-Only (RO), Read/Write (RW), eXecute Never (XN), etc. If a memory access is not permitted by at least one region in the MPU, a Memory Manage Fault (MMF) is generated. The MMF is an exception that causes the MMF Handler to run, which normally stops the offending task and determines what to do in order to recover.
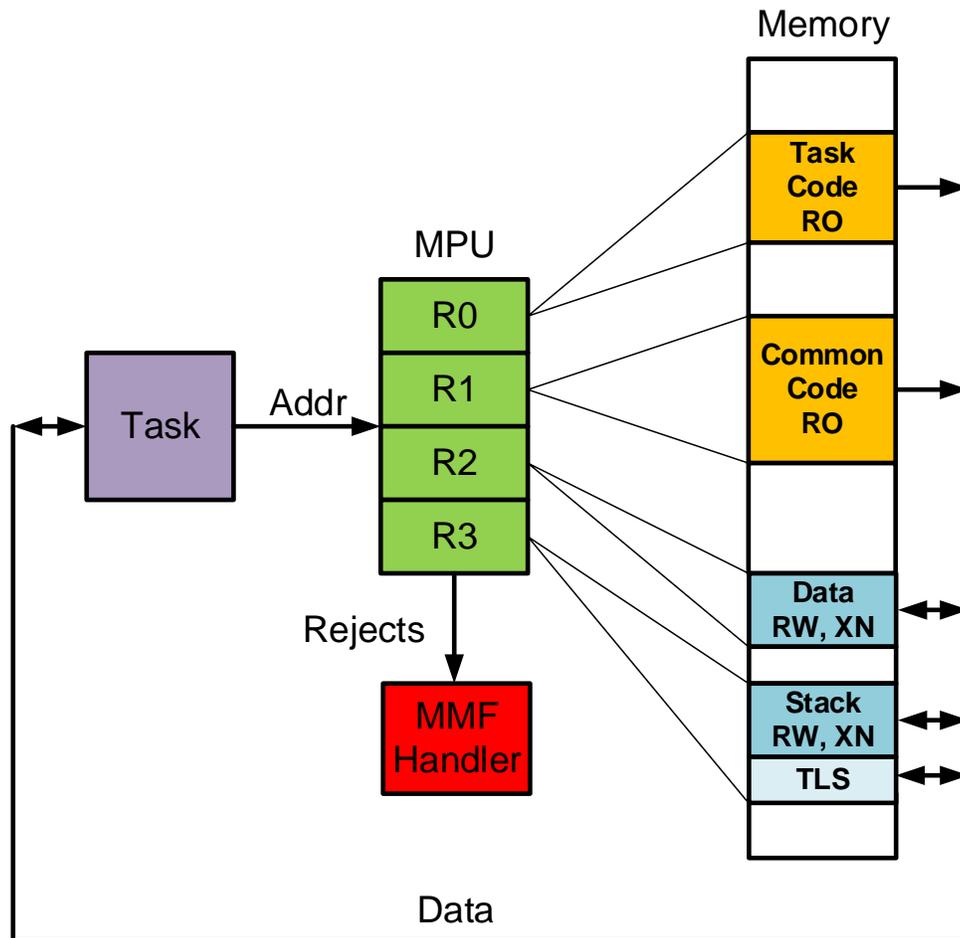

Figure 1 MPU Operation

Figure 1 shows an example MPU with only 4 slots. (Most MPUs have 8 slots, some have 16.) The region stored in MPU[0] is an RO region for task code. The MPU[1] region is an RO region for code that is common with other tasks. The region stored in MPU[2] is a RW and XN region for data. The final region is the task stack, which is also RW and XN. Shown below it is the optional Task Local Storage (TLS) that can be part of the task stack region and which can be used for local task data. Since only a pointer to the TLS is provided, the data

must be either a structure or an array. It has the same attributes as the task stack. The white regions of memory are inaccessible to this task.

A serious drawback of the v7 MPU is that region sizes must be powers-of-two, and regions must be aligned on their size boundaries. This requirement is probably the major reason for low usage of the v7 MPU, but it can be overcome. Actually, the biggest drawback of the MPU is insufficient slots. Nearly all Cortex-M processors have MPUs with 8 slots. When one gets into creating regions for real partitions, 8 just is not enough – 12 might be just right. A few processors have 16 slots; these are strongly recommended for new designs. We support both 8 and 16 slot MPUs, but our emphasis is on the former since they are, by far, the most common.

Although the Cortex-v7M MPU has significant limitations, which make it difficult to use, it is the only means of hardware memory protection available for Cortex-v7M processors, and hardware protection is the only protection that is effective against hacking. Therefore, it is important to find methods to use it effectively in order to achieve the reliability, security, and safety that modern embedded systems require.

Cortex-v8M

The Cortex-v8M architecture was announced a few years ago. To date, there are very few processors using it. The v8 MPU has greatly reduced the region size and alignment requirements to multiples of 32 bytes. This is helpful for limited-memory embedded systems. However, the number of MPU slots is unchanged in the v8 architecture. Hopefully semiconductor vendors will opt for 16 rather than 8 slots, at least in the non-secure state. We will be supporting the v8 MPU soon.

The Cortex-v8M architecture also offers a new feature called *TrustZone*, which permits secure and non-secure states. TrustZone secure state is good for storage of keys and other private data. However, running code in secure state may not be worth the extra code complexity and debug difficulty. Arm says otherwise, so we will have to see. In addition, we expect that most device manufactures will want a security solution that works with both v7 and v8 processors.
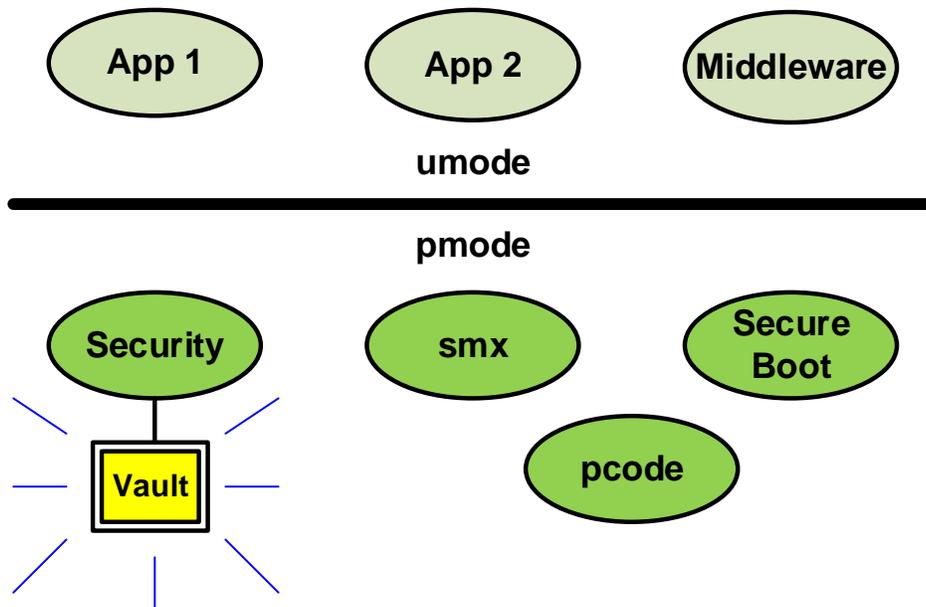
Partitions

Basics

Figure 2 Partitions

Figure 2 illustrates the software structure that we are trying to achieve for security. In this diagram, ovals represent isolated partitions. The partitions above the heavy line run in *umode* (*unprivileged or user mode*) and the partitions below the heavy line run in *pmode* (*privileged or protected mode*.) The heavy line represents the boundary between unprivileged operation and privileged operation. This isolation is enforced by the Cortex-M processor architecture. It is safe and dependable, unless we do something wrong.

Above the heavy line are two application partitions and one middleware partition. Of course, an actual system is likely to have many more umode partitions. The goal here is to achieve complete isolation of one umode partition from another. Then penetrating one partition does not enable a hacker to penetrate others and thus the breach is contained. Each umode partition is a group of one or more utasks. The utasks form the basis of isolation from tasks in other partitions, but not from tasks in their own partition. umode partitions are capable of strong isolation. Hence, vulnerable code such as drivers, middleware, and application code should be put into umode partitions, whenever possible.

Below the heavy line are Secure Boot, pcode, smx, the SMX RTOS kernel, and Security partitions. These are comprised of pcode. Of course, an actual system may have many more pmode partitions. The goal is to also isolate pmode partitions from each other. However, this isolation is not as strong as umode isolation, as discussed later.

Secure Boot

When the system powers up or is rebooted, the processor comes up in pmode and it is in the Secure Boot partition.
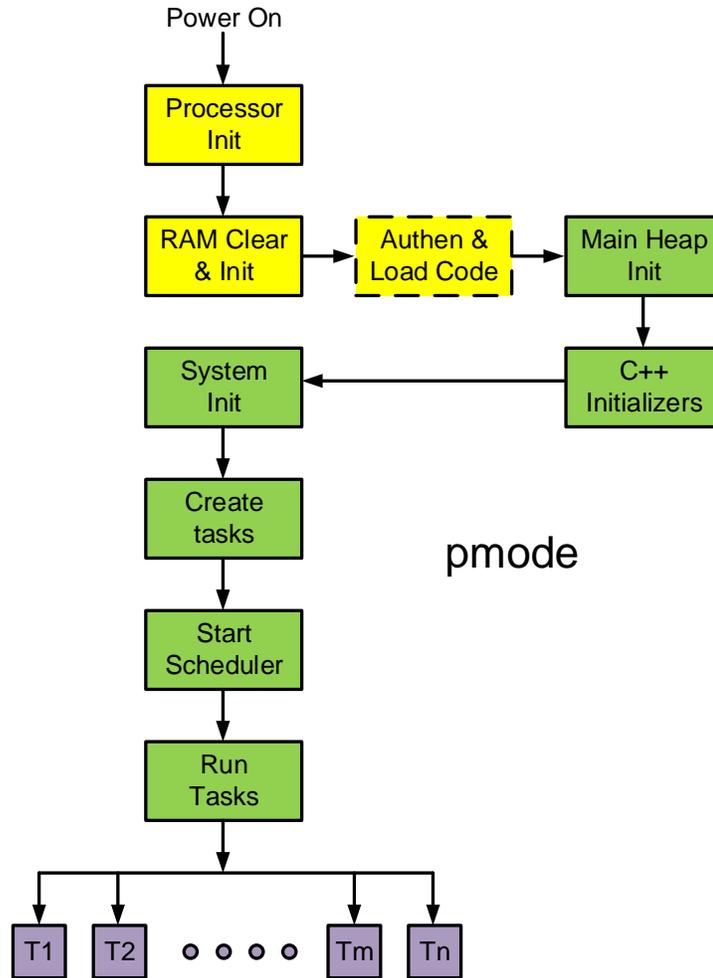
Figure 3 Secure Boot

As illustrated in Figure 3, secure boot software does basic hardware and software initialization, it loads code, if necessary, it creates the tasks necessary to start operation, and then it starts the scheduler. Prior to starting the scheduler no tasks are running. After starting the scheduler, the system is running in task mode and the first task scheduled at the highest priority is running. Other partitions do their own initializations. Both for structural and security reasons, it is best to minimize the code in the secure boot partition. In Figure 3, the secure boot loader is shown in yellow. These are available from many sources and are outside of the scope of this paper. Code shown in green is system and application code.

smx

The SMX RTOS consists of the smx kernel plus middleware. SMX is split such that the smx kernel runs in pmode, whereas the SMX middleware runs in umode. MPU-Plus, when bundled with SMX®, eheap™, and certain other products, constitutes SecureSMX™.

The smx partition contains the smx kernel and related software such as the SVC Handler and the PendSV Handler. It runs in pmode in order to strongly isolate it from umode partitions,

which could have been corrupted. MPU-Plus extends smx to add security functions. For more information on these, see:

smx User's Guide, by Ralph Moore, Micro Digital, Inc.

smx Reference Manual, by Ralph Moore, Micro Digital, Inc.

We have found that, in addition to adding MPU-Plus, a surprising amount of modification to smx, itself, has been necessary, even though smx has been in use as an embedded kernel for 30 years! Middleware products also have required significant modification. Security seems to be bringing a paradigm shift to embedded systems software.

Security

Finally, there is the Security partition and the Vault. The Vault is where we store the jewels (encryption keys, passwords, authentication codes, certificates, etc.) and the cash (private data). If pmode is breached, the Vault springs open and the Kingdom is lost. Therefore, protecting the Vault is of paramount importance, and only the security partition, which contains crypto, authentication, and other security tasks is allowed access to the vault.

Encryption and authentication software have been moved out of SMX middleware products into the security partition. Thus, only security software has access to the Vault.

pcode

The pcode partition contains Interrupt Service Routines (ISRs), Link Service Routines[2] (LSRs), and other code that must be in pmode. This is a mixture of system, middleware, and application code. In an actual system, this would probably be split into a system partition and an application partition. It may contain some ptasks as well as ISRs and LSRs. Likewise, the smx Error Manager, smx_EM() and error recovery code are not tasks. Hence, most of the code in this partition runs in the context of the current task.

Handling interrupts presents special security problems, which are discussed later, in Partition Problems.

utasks

utasks can provide high levels of isolation. This is primarily because they cannot access the MPU. The MPU is loaded with the regions that a task is permitted to access, including access permissions (e.g. read-only, execute never, etc.), but the task can do nothing to change them. If the Background Region is on, it has no effect in umode. However, all is not peaches and cream – there are heap problems and function call problems, which are discussed later.

ptasks

The isolation provided by ptasks is weak compared to utasks. This is because once a ptask is breached, only one step is required for malware to either turn off the MPU or to turn on its

---

[2] Link Service Routines are unique to smx. ISRs cannot make smx calls; they must invoke LSRs to do so. LSRs can make most smx calls but cannot wait. They operate at a higher priority than any task and provide a convenient method for deferred interrupt processing and to buffer peak interrupt loads.

Background Region (BR). Then the MPU regions have no effect. The MPU is defenseless in pmode, whereas it is impregnable in umode.

However, ptasks may help to thwart attackers by catching many hacking techniques (e.g. stack or buffer overflow, attempted execution from a stack or buffer, etc.) and triggering an MMF before the hacker gains actual control. The MMF handler can then delete the penetrated task and recreate it, hopefully with only a small hiccup in system operation. It can also report the incident, which is helpful to finding and reducing code vulnerabilities.

ptasks are also useful to catch programming errors and can be a useful step on the way to utasks.

Basic Operations

MPU Control

A Memory Protection Array (MPA) is a set of regions to be loaded into the MPU on a task switch; there is an MPA for each task. A task's index is used to find its MPA in the memory protection table, mpt[indexsmx_TaskCreate() copies the current (parent[3]) task's MPA to the task being created. If ct is a ptask, it can change the task's MPA via:

smx_TaskSet(task, SMX_ST_MPA, tp);

where tp points to the MPA template for the task.

The foregoing is illustrated in Figure 4. In this figure, MPA0, 1, and 2 share template mpa_tmplta. Hence the three corresponding tasks share the same regions. They are thus most likely in the same partition. Note that MPA3 uses template mpa_tmpltb. Hence, the corresponding task is most likely in a separate region. The fifth task has not yet been created, nor has its MPA been loaded.

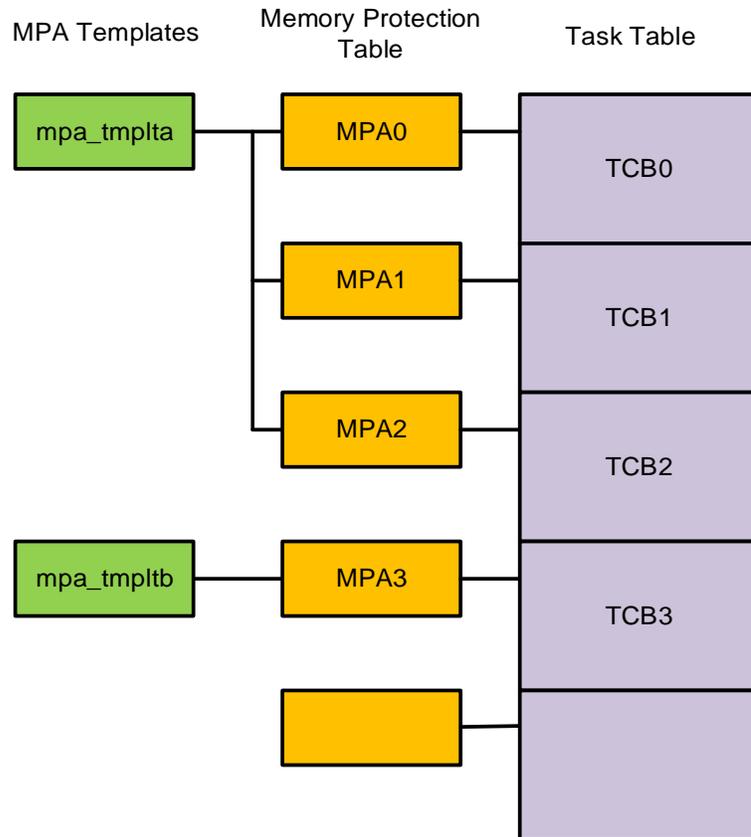[3] Parent and child tasks are discussed in Task Creation and Control, in Partition Problems.

Figure 4: Templates, MPAs, and TCBs

There are as many slots in the MPA as dynamic slots in the MPU[4]. Most slots are filled with static regions defined in the linker command file (a tedious process). However, some slots have pointers to an array of regions that are dynamically created at run time. These are discussed more in Dynamic Regions. The top MPU in-use slot, which has highest priority, is reserved for the task stack region. The task stack is dynamically created from the main heap when a task created or obtained from the stack pool when the task is dispatched[5]. While the task is running, any updates to the MPU are also made to its MPA so it is not necessary to save the MPU contents during a task switch.

Creating static regions is a laborious process. For example, for a code region it is necessary to identify all functions needed by a particular partition or a particular task, including subroutines. Pragmas are inserted into the code to put all of these into a unique code section, for example:

---

[4] In rare cases, certain regions may be permanently assigned to top MPU slots and thus these slots do not appear in MPAs.

[5] Allowing *permanent* or *temporary* stacks for tasks is a unique feature of smx. Temporary stacks are used with one-shot tasks, which do not have infinite loops. Instead, they run once, stop, and release their stacks, thus permitting a few stacks to be shared among many one-shot tasks.

```
#pragma default_function_attributes = @ ".ut1a_text"
void tm05_ut1a(void)
{
  smx_SemSignal(sbr1);
}
#pragma default_function_attributes =
```

Then a block is defined in the linker command file to hold this and related sections, for example:

```
define block ut1a_code with size = 1024, alignment = 1024 {ro section .ut1a_text, ro section
                                    .ut1a_rodata};
```

Regions are defined in the linker command file and blocks are placed in them, for example:

```
define region ROM = mem:[from 0x00200000 to 0x002FFFFF];
```

```
place in ROM  {block t2a_code, ro section .tmplt, block ut1a_code, block ut2a_code, block
              ut2b_code};
```

Back in the code, a slot in the MPA is defined:

```
#pragma section = "ut1a_code"
```

```
MPA mpa_tmplt_ut1a =
{
...
  RGN(3 | RA("ut1a_code") | V, CODE | RSI("ut1a_code") | EN, "ut1a_code"),
...
};
```

All of this is done for one MPU region in one template – clearly a laborious process. Template macros (e.g. RGN()) are shown above that reduce the work and help to reduce errors. Because some of the statements are in the code and some are in the linker command file, the process is error prone. Not only that, it is very easy to leave out an obscure subroutine for a code region or variable for a data region, resulting in an annoying MMF during debug (a good reason to turn off MMFs during the early stages of debugging).

System Calls

ptasks can call all smx and system functions directly, but utasks cannot call them directly because they must execute in pmode. Instead, the SVC N instruction is used. For umode code, the xapi.h header file, which contains smx and system prototype functions, is replaced with the xapiu.h header file. The latter maps smx_ calls to smxu_ calls, which are shell functions that invoke SVC N, where N is the system call ID. However, *restricted calls*, which are prohibited for umode, generate Privilege Violation errors. Restricted calls can be made only by pcode. For example, smx_HeapInit() is not needed by utasks and could cause system harm, if called from malware, so there is no smxu_HeapInit(). A reasonable set of restricted calls is defined. However this set can be expanded or contracted, as necessary, for a specific application.
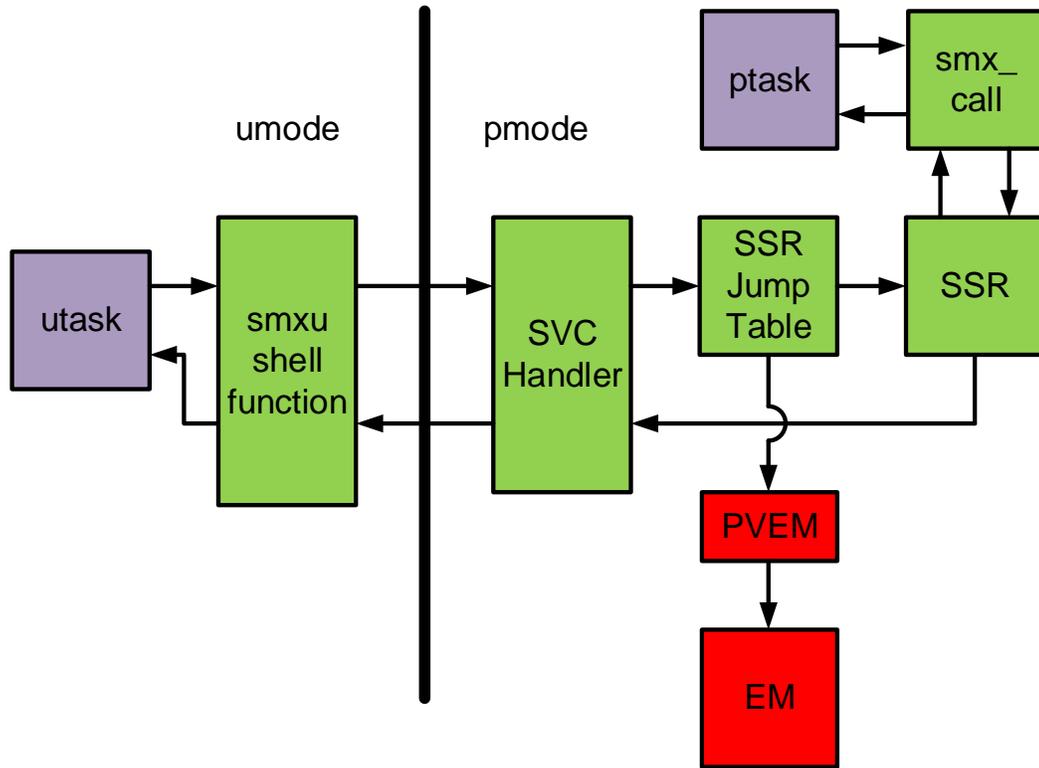
Figure 5 System Calls

Figure 5 illustrates the system call mechanisms for both utasks and ptasks. The SVC Handler, uses N as an index through the SSR jump table to the smx System Service Routine (SSR). The SSR executes in pmode, then returns the result to the SVC Handler. The SVC Handler returns this result to the smxu shell function, which returns it to the utask. All of this detail is hidden from the caller, and it appears as if a normal function call were made. A system call that is not allowed in umode results in a branch to the Privilege Violation Error Manager (PVEM), which, in turn, calls the smx Error Manager (EM).

Note that an smx call from a ptask goes directly to an SSR, and there are no disallowed service calls.

Partition Problems

Defining partitions is but one step in the security process. We must also be concerned about what a hacker will do after he penetrates a partition. In this regard, four major problem areas emerge:

1. Heap usage.
2. Function call APIs.
3. Interrupts.
4. Task creation and control.

There are others, but these will do for now. Solutions are discussed in what follows.

Heaps

Especially with object-oriented languages, using heaps in modern application code is popular. This is a growing trend as embedded systems become more complex and are expected to do more operations – especially in IoT systems. Also, some middleware uses heaps.

It is obviously unacceptable for utasks to have direct access to the main heap. A hacker could easily bring down the whole system simply by exhausting or corrupting it. Hence, dedicated heaps must be used in umode partitions, and possibly pmode partitions, that require heaps. To solve this problem, *eheap*™ has recently been upgraded to support multiple heaps. These heaps will generally be small, but not necessarily so. Given its proclivity for small heap support, this is a good solution. For more information see:

> *eheap User's Guide,* by Ralph Moore, Micro Digital, Inc.

Figure 6 illustrates allocating a small dedicated heap from the main heap. Heap calls from TaskA operate only on this heap and cannot go outside of it. TaskA can access the main heap only for a *protected block* or a *protected message*, as shown by the dashed line, and cannot go outside of the protected block. (Protected blocks and messages are discussed later.) Hence the main heap is protected from TaskA, which may be a utask or a ptask.
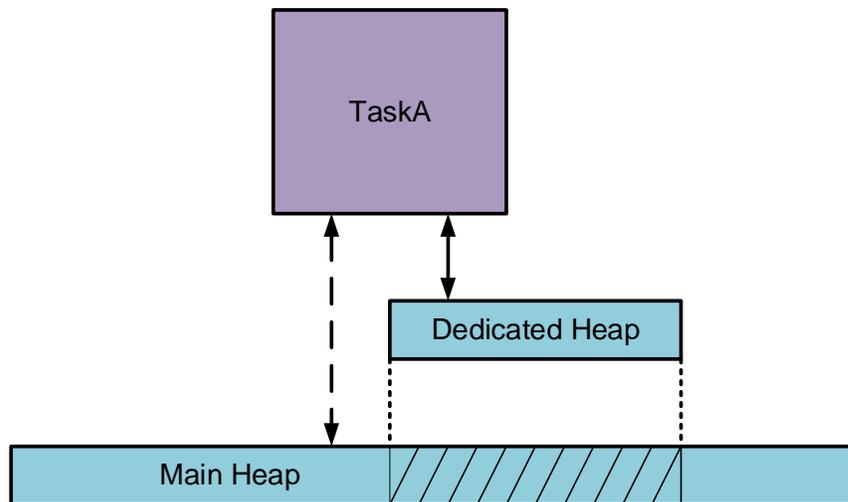

Figure 6 Dedicated Heap from the Main Heap

Memory for a dedicated heap could also be a static block of memory allocated by the linker.

Function Call APIs

Function calls are the predominant API between sections of software. This creates a problem. For example, an application partition may need file system services. As a consequence, the file system API functions must be accessible to it. Subroutines in the file system must be accessible to the file system API functions and driver functions must be accessible to the

subroutines. Also, file buffers and global variables must be accessible to all of these functions. So, the whole ball of wax – file system and driver(s) – ends up in a code region of the application partition, and file buffers and globals end up in a data region of the application partition. Worse, if other partitions need file I/O then these regions become common regions between those partitions.

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

Interrupts

Interrupts cause an immediate switch to pmode and thus expose pmode to the outside. Recalling that any pmode function is but one step away from opening the Vault, this is a serious security problem. In many cases, as illustrated in Figure 7, only a few lines of carefully written code in an ISR or in an ISR + LSR are necessary. (LSRs provide deferred interrupt processing.)
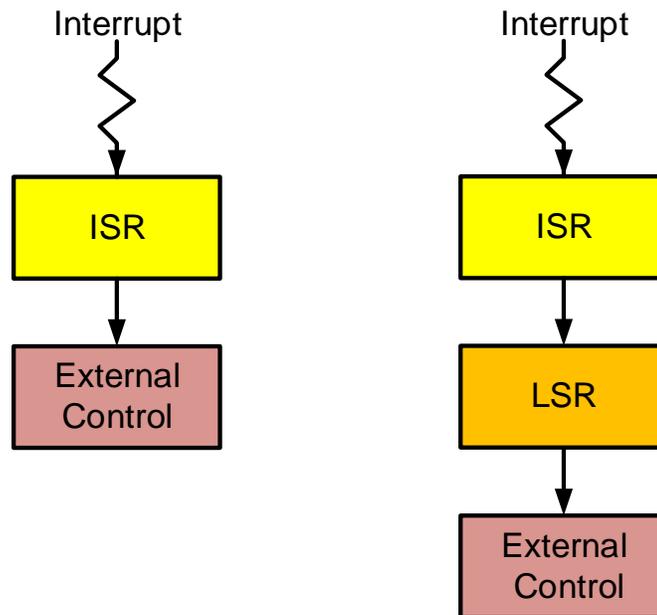

Figure 7 Minimal Interrupt Processing

Unfortunately, the limited number of MPU slots aggravates the interrupt problem. Initially, we defined a *sys_code* region to contain ISRs and other system code needed by interrupts and a *sys_data* region for needed data. The Vault, Security, and other sensitive partitions were excluded from these regions. sys_code and sys_data were present in every task MPA. Hence, when an interrupt occurred, ISRs and LSRs could run, but had limited access to other pcode and pdata. This is still our preferred solution, if the MPU has enough slots.

The sys_code and sys_data regions were privileged regions and thus not usable by utasks. Unfortunately, we found that for 8-slot MPUs we could not afford to waste two slots for every utask. Hence, standard MPU-Plus turns on Background Region (BR) whenever

switching to a utask. BR has no effect in umode, but when an interrupt occurs it allows the ISR and LSR to run. Unfortunately, BR on in pmode also allows accessing everything – hence the Vault is open!

Where practical, it is recommended that ISRs immediately load minimal sys_code and sys_data regions into the MPU and switch BR off. This at least closes the vault and makes accessing it a little more difficult. When exiting, the ISRs must, of course, restore the replaced regions.

For ptasks, the sys_code and sys_data regions are present and usable. They are somewhat enlarged to include other system functions. Hence, the two regions do not pose a problem and BR is turned off whenever a ptask runs in order to protect the Vault, among other things.

Figure 8 illustrates the approach adopted. Note the sys_code and sys_data regions for the ptask. The utask does not have these regions because BR is on, instead. As a consequence, it has been possible to expand the utask MPA by two slots. This has enabled splitting the peripheral region in MPU slot 4 into separate USB host and UART1 regions in slots 4 and 5, respectively. This provides better security because there are several peripherals in memory between the USB host and UART1, which are now excluded from access by the utask. Also slot 6 is available for a dynamic region (see Dynamic Regions). Note that for both tasks, there are task code and task data regions and common code and common data regions. The latter regions are not the same for the ptask and the utask – even if the ptask eventually becomes the utask.
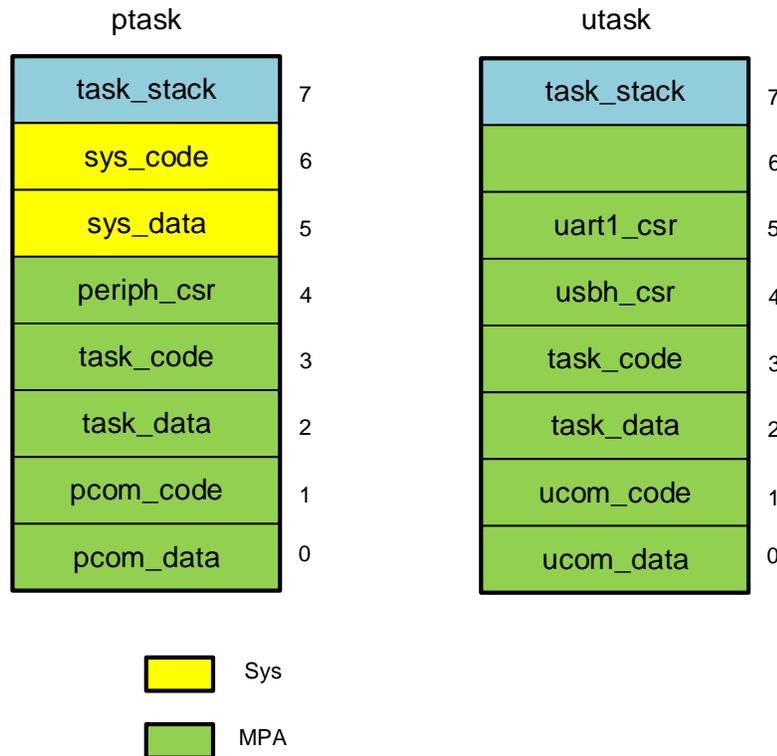

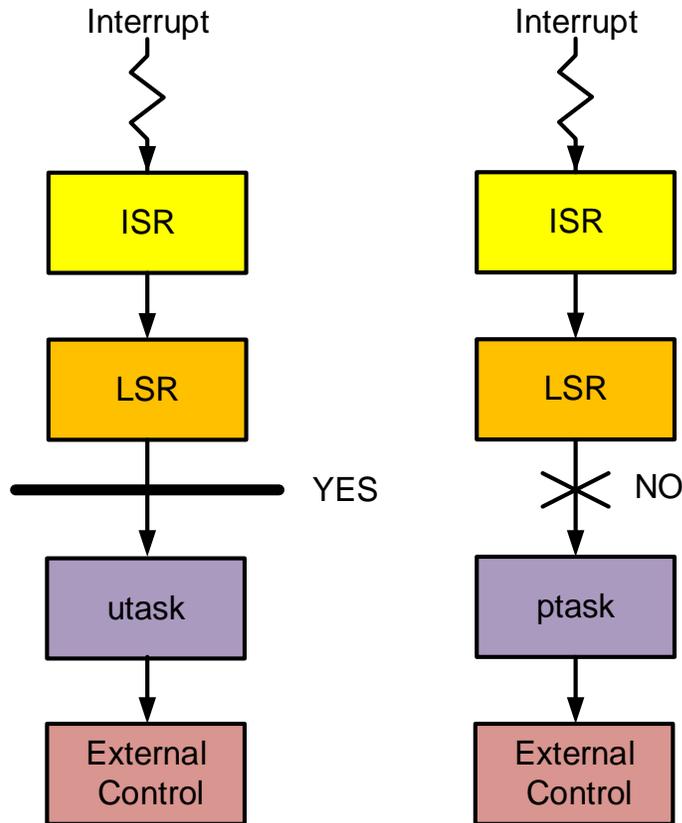
Figure 8 MPU for pmode vs. umode

Figure 9 Task Interrupt Processing

When more than minimal interrupt processing is required, Figure 9 illustrates what to do on the left and what not to do on the right. The objective is to move as much processing as possible into a utask where hacking can be better contained. Here, as for simple interrupts, the goal is minimal code in ISRs and LSRs. Also, this code must be carefully written -- it must employ extensive range checks and other tests intended to defeat hacking. This is challenging if high performance is also required.

Despite the foregoing caution, it may be necessary to do full interrupt processing in pmode (i.e. the right-hand side of Figure 9). This is definitely faster and simpler, especially if there are critical sections of code and if system services are being called. In this case, it is preferable to do the processing in the ptask rather than the ISR or LSR since a ptask offers more protection, since BR is disabled, so it is restricted to the MPU regions.

More Interrupt Problems

Interrupt problems just won't go away. Another set of problems revolves around disabling and enabling interrupts. In umode these two operations are no-ops. So, if interrupts are being disabled to protect a critical section in your umode code, guess what? They aren't disabled and you have a hidden problem! This can be a headache when converting legacy code to ucode, since interrupt disabling is commonly used to protect critical sections of code. It is just as well that interrupts cannot be disabled from umode. If they could, it would be a field

day for hackers. Note that this is not a problem in pmode since all privileged instructions can be accessed in pmode.

The solution to this problem is to allow specific interrupts to be masked and unmasked by utasks, using the smx functions sb_IRQMask(irq_num) and sb_IRQUnmask(irq_num). The range of IRQs that a task is allowed to mask and unmask is stored in its TCB. Hence, damage that can be done by a hacker is limited only to interrupts that are used by the task. For legacy code, it is necessary to track down all places that interrupts are disabled and enabled, replace them with masking and unmasking, and then load permitted IRQ ranges into the task TCBs.

To help find uses of interrupt disabling and enabling in umode, alternate versions of interrupt disable and enable macros or functions can be used that trap if called in umode. These are helpful to find misuses from macros and wrapper functions or code that was expected to run in pmode.

Task Creation and Control

Clearly a hacker could really cause trouble if he could create, delete, start, and stop tasks from a umode partition that he had penetrated. Hence, task functions should not be permitted in umode. One would think that all task creation and control should be performed only in pmode.

Unfortunately, this does not work well, especially if converting legacy code. To require that all tasks be created during pmode initialization results in unexpected limitations and complexities. In many situations, tasks need to be created as they are needed in order to deal with events as they occur. For example, tasks may be created as USB devices are plugged in and the tasks may be deleted when the USB devices are unplugged. As another example, some USB controllers can be switched between host and device modes, thus requiring one USB stack to be disabled and the other to be enabled. To save resources, this is likely to be implemented by deleting one set of tasks and creating another set of tasks.
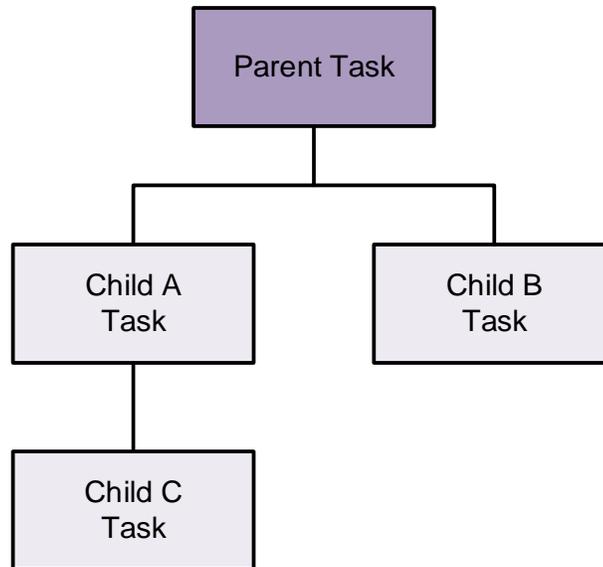
Figure 10 Parent and Child Tasks

The solution to this problem is task families, as shown in Figure 10. Typically, a partition will have one *parent* or *root* task, which is created in pmode and which runs in pmode to perform certain partition initializations. The latter may include creating or spawning some child tasks. The parent task then switches itself into umode, where it can start its child utasks and possibly create and start others. This provides the necessary flexibility for dynamic task control. Figure 8 illustrates a task family. Note that child tasks can create other child tasks and thus become parents for those children.

A parent task can create, start, stop, delete and perform other functions on its child tasks. It cannot perform these task functions on its parent or siblings, nor on their children. In addition to this limitation, a child inherits its parent's MPA template and all other parent limitations. Hence a child cannot do anything that its parent cannot do. (Otherwise, a hacker could breed monsters.)

Task Local Storage (TLS)

The task create function allows creating a TLS area that follows the Register Save Area (RSA) that follows the task stack:

    TCB_PTR smx_TaskCreate(fun, pri, tlssz_ssz, fl_hn, name)

tlssz_ssz is a split parameter: the upper 16 bits define the TLS size, tlssz, and the lower 16 bits define the stack size, ssz. Both can be up to 64 KB. TLS is available only if ssz > 0 – i.e. the task stack must be a permanent stack from heap, hn. TLS is a bonus protected task data block that does not cost an additional MPU region. It can be used in the same manner as a protected data block (see below).

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

    dp = (u8*)smx_TaskPeek(ut2a, SMX_PK_TLSP);

This operation is permitted for utasks as well as ptasks. The TLS can contain only structures and arrays (i.e. buffers). If all task static variables are defined as fields of structures and task buffers are defined as arrays, then the TLS can replace the task_data region, as long as no other task is attempting to access any of the variables (if so, put them into the com_data region). This frees up the task data slot to be used for another region in order to create smaller, more secure regions. Figure 8 is a primary example of this benefit.

Dynamic Regions

As noted previously, creating static regions is a time-consuming, tedious, and error-prone process. The *dynamic data regions* discussed below help to relieve some of this burden.

Dynamic Data Regions

The following functions allow dynamically creating data regions during initialization or from ptasks:

```
u8*        mp_RegionGetHeapR(rp, sz, sn, attr, name, u32 hn);
u8*        mp_RegionGetPoolR(rp, pool, sn, attr, name);
BOOLEAN    mp_RegionMakeR(rp, bp, sz, sn, attr, name);
```

where rp is a pointer to the created region, sz is the region size, sn is the slot number, attr are the attributes, name is an optional name for the region, hn is the heap number, pool is a block pool handle, and bp is a block pointer. The above can be used to make a data region from a heap, a block pool, or a static block (e.g. stat_blk[100]), respectively.

Normally rp points to an entry in the dynamic protection region array, dpr[n]. Then, the MPA template slot for the dynamic region is set as follows:

```
mpa_tmplt_t2a[sn] = MP_DYN_RGN(dpr[n]);
```

where MP_DYN_RGN() loads the address of dpr[n] and sets the dynamic region flag in the template slot.

These functions should normally be called during system initialization before tasks start running. However, they can also be called by ptasks, which are creating and initializing other tasks.

Dynamic data regions can be used to store a mixture of static arrays and structures, and they can be shared between tasks. Although they cannot be used for global variables, they do save the complexities of defining sections in the code, blocks in the linker command file, and static regions in templates. Hence, they are simpler and less error prone to use. Given that sz is likely to be a sum of sizeof()'s they may also be more flexible during development.

Protected Data Blocks

The following protected block functions permit creating *protected data blocks* from either utasks or ptasks and also releasing them, while running:

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

where the parameters are the same as for dynamic regions, except that for the release functions, bp is the block pointer returned by one of the Get functions, and clrsz specifies how many bytes to clear after the free block link in the first word of the block. Basically, a block is obtained from a heap or a pool or made from a static block. A region is created for it and loaded into MPU[sn] and into MPA[sn] of the current task. The heap can be any heap, including the main heap. This is safe because if a hacker penetrates a task, the MPU prevents him from accessing heap memory outside of the protected block.

Dynamically allocated blocks can be used for buffers, work areas, messages (see below), or structures. If a task is written such that all of its static variables are in a structure, for example:

```
u8* vp;
vp->var1 = vp->var2 + vp->var3;
```

Then a dynamic block can be used to store its static variables. In the above, vp is the block pointer returned by the BlockGet() function. (Note that vp is an auto variable and thus is stored in the task stack, not in the structure). If a function is not written this way, it is not difficult to convert it -- just insert "vp->" ahead of every static variable reference, define a VP structure with the variable names as fields, and define vp as a pointer to VP.

The difference between a protected data block and a dynamic data region is that a protected data block can be obtained by a task while it is running, whereas a dynamic data region is created during initialization, and a pointer to it is loaded into a task's template. Protected data blocks are especially useful for utasks to create temporary buffers and protected messages, as discussed below.

Using Dynamic Regions

Using a dynamic data region, a protected data block, or a TLS to replace the task_data static region requires redefining all task global variables as fields in one or more structures. If the structure name is very short, this does not appreciably complicate the code. For example, here is some code from eheap:

```
hvp[hn]->errno = EH_OK;
bsmap = hvp[hn]->bsmap;
csbin = hvp[hn]->csbin;
```

In order to support multiple heaps, it was necessary to change discrete globals to an array of structures, hvp[hn]. In this case, hvp[hn]-> was pasted to the start of every global variable name in the code. The Cortex-M architecture allows accessing structures as fast or faster than

discrete globals – one LDM instruction in a function loads the structure base address and fields are accessed via constant offsets, thereafter. The compiler may not be able to do this for all discrete globals used by a function, hence access to them may be slower. Using a structure also allows grouping fields together that are used together, which improves performance if the processor has an instruction cache. Multiple structures and arrays can be handled automatically by using sizeof()'s to determine pointer offsets.

Protected Messages

An smx message consists of a Message Control Block (MCB) linked to a data block. The smx_MsgMake() function can be used to make a protected data block into a protected message and the current task becomes the message owner. Messages are sent to message exchanges and received from message exchanges. While at a message exchange, the message's MCB is linked to the exchange's control block into a queue of waiting messages.
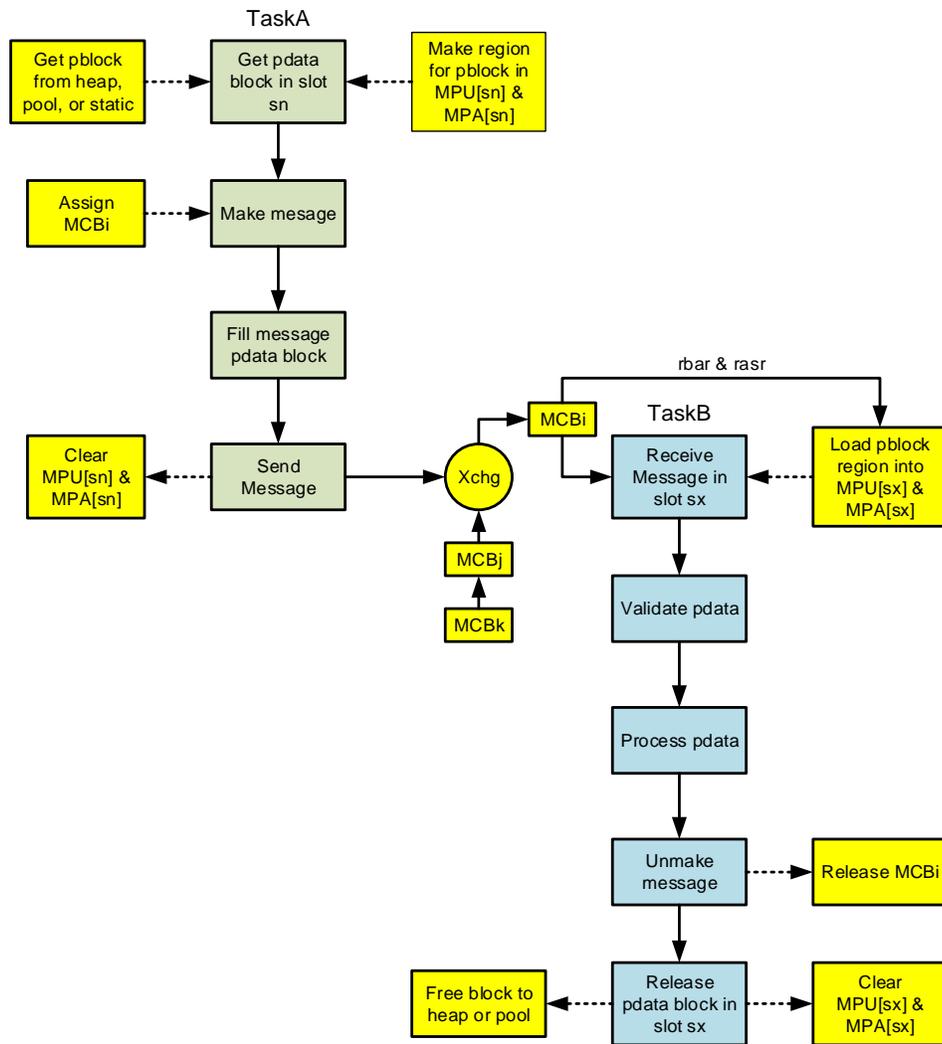

Figure 11: Protected messaging between tasks

Figure 11 illustrates transferring protected messages between tasks. TaskA is shown in green and TaskB is shown in blue. Yellow represents pcode and pdata, which are protected from either task. As shown, TaskA gets a pdata block in slot sn, makes it into a message, loads it, and sends it to Xchg. As part of the send operation, slot sn in the MPU and in TaskA's MPA are cleared. Note that other messages are waiting at Xchg and that MCBi is at the top of the message queue. TaskB receives it in slot sx. Note that rbar and rasr are obtained from MCBi and used to create the region for the message's pdata block in slot sx. TaskB validates the message, which is application-dependent and might consist of doing range and consistency checks on the data. It then processes the pdata, unmakes the message, releases the pdata block in slot sx back to its heap or pool, and clears slot sx in the MPU and in TaskB's MPA.

Two protected message functions have been added to smx:

```
MCB_PTR    smx_PMsgReceive(xp, bpp, sn, timeout);
BOOLEAN    smx_PMsgSend(mp, xp, sn, pri, rp);
```

Where xp is the exchange pointer, bpp is a pointer to the message block pointer, sn is the MPU/MPA slot number, timeout is in ticks, mp is the message pointer, pri is the message priority, and rp is a reply pointer (e.g. to the exchange to send a reply message.)

As shown in Figure 11, when a protected message is sent, its slot, sn, in the MPU and in the current task's MPA are cleared. Thus, even if the sending task retains a pointer to the message block (e.g. bpp), it cannot access the message block. This foils the hacking technique of changing a message after it has been validated by a receiving task in another partition. It also foils reading the message after it has been updated by a receiving task in another partition.

While at an exchange, the message block region information is stored in the message's MCB, and the exchange is the owner of the message. When the message is received by a receiving task, its message block region information is loaded into the specified slot, sx, of the MPU and of the receiver's MPA and the receiving task becomes the message owner. (sx need not be the same slot as was used by the sending task.)

Now, the receiving task can read and modify the message and possibly send it on to another exchange. Hence, a message could be created, loaded with data, passed on to a task to check the data and encrypt it, and then passed on to a third task to send it over a network. Note that there is complete isolation between the sending and receiving tasks. Of course, the sender could send some kind of disruptive message. Hence the receiver must perform validation before accepting a message. This level of security is application specific.

Partition Portals

Partition portals enable isolating client partitions from server partitions and are necessary to achieve 100% partition isolation, which is critical to achieve strong security. They are built upon the smx protected messages described above. Protected messages satisfy the Arm PSA Secure IPC requirement (see reference 3 in Cortex-v7M in Introduction) without the need for message copying. Hence, introducing portals versus normal function call APIs may not degrade performance appreciably.
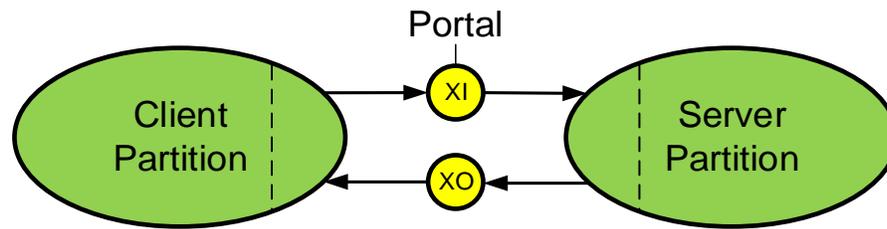
Portal Implementation

Figure 12 Partition Portal

As shown in Figure 12, a partition portal consists of exchanges, marked XI and XO — one for each direction. Code added to the client partition (represented by the area to the right of the dashed line) converts function calls and their parameters into messages that are sent to the XI exchange. The client task then waits at the XO exchange for a reply message.

A server task, in code added to the server side (represented by the area to the left of the dashed line), is waiting at the XI exchange for a message. When it receives a message, it converts it to a function call using parameters in the message. It then puts return information from the function call into a message that it sends to the XO exchange. The client task receives the message from the XO exchange and returns information to the caller in the client partition.

Clearly there is quite a bit of work to convert function call interfaces between partitions to partition portals. However, the result is strong partition isolation. This, of course, comes at a reduction in performance. Data buffers are passed in messages. If existing code is being modified, this may require copying data from buffers to messages and vice versa. If new code is being created, it can be designed to work with messages in no-copy mode. In the latter case, the performance hit may be minor.

Note that the data copy problem also exists in large MMU-based systems. In fact, in that case there is not a no-copy solution because of virtual address spaces. Hence, inter-partition communication via portals in MPU-based systems can be more efficient than inter-process communication in MMU-based systems. This favors smaller partitions, each of which do less work, and thus system security is potentially better. Having smaller partitions also makes redundant paths more practical – e.g. two independent paths to report suspicious activity back to headquarters.

Debug Support

Debugging code is much more challenging when security features are enabled. For this reason, SecureSMX allows overriding most security features during early code development and debugging to help speed up these phases. Security features can be reenabled during later-stage debugging where they actually become helpful by detecting stack and buffer overflows and other problems. Also, it is advisable to start addressing security issues before development has gone too far.

smxAware™ includes many security-related features to help debug MPU-Plus based software. It displays the current MPU and all task MPAs, with named regions. The graphical

Memory Map Overview shows MPU regions in the memory bars. Start and end addresses, as well as excluded subregions are shown. In all displays, errors such as alignment and overlap are flagged. For more information, see:

smxAware User's Guide, by Marty Cochran and David Moore, Micro Digital Inc.

Conclusions

Software engineering has lost its naivete – we are now designing for a hostile world. It is not possible to achieve perfect security, but it can be pretty good. A determined hacker will no doubt find some weakness in even the best security you can devise. It therefore is necessary to analyze all of your code versus probable threats.

In some cases a body of code may be so poorly designed and implemented and thus so vulnerable, that reworking it is a hopeless proposition. In that case it may be more cost-effective to leave the code as is and to put it into a fully isolated umode partition, using the methods described in this article. Then it is necessary to devise a strategy to deal with the inevitable break in and to implement the necessary code to handle it. In the process, some latent bugs may be found.

If upgrading legacy code to improve the security of an existing product or if developing a new product using legacy code, the main job is generally to restructure the legacy code. The amount of recoding that is required can be small, depending upon how well the code is structured to begin with. New code, of course, should be structured for security from the outset.

Security adds another dimension to product development. It is necessary to think not only about how to implement a function but also to think about how a hacker might gain access to the function in order to cause damage or to steal private data. MMFs are annoying during debugging, but they prove that the hardware security mechanisms actually work! The goal of MPU-Plus is to provide a path that will accomplish good security without excessive anguish.

For more information see www.smxrtos.com/mpu and check back for future updates.

Ralph Moore is President of Micro Digital. A graduate of Caltech, he has served many roles at Micro Digital since founding it in 1975. Currently he is lead architect and programmer for MPU-Plus, eheap, and smx.