smxMPU™ Manual

January 11, 2017

Contents

Introduction

The Cortex-M Memory Protection Unit (MPU) and its Supervisor Call (SVC) instruction can provide protection from bugs and malware for critical system resources. Although the Cortex-M MPU is difficult to use, it is the only means of hardware memory protection available for Cortex-M processors. Hence, it behooves us to use it effectively.

smxMPU has been developed to provide a simplified and effective platform upon which to build secure systems. It had two principal design goals:

- Ease of conversion of legacy applications and middleware to use the MPU and the SVC.
- To allow developers to focus on protection strategies without being tripped up by details.

Developing 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.

This manual assumes familiarity with the Cortex-M MPU architecture. Good sources for this information are:

1. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors* by Joseph Yiu. See the Memory Protection Unit chapter.
2. *ARM v7-M Architecture Reference Manual*, ARM Ltd. 2014, Chapter B3.5 Protected Memory System Architecture, PMSAv7.

terminology

Cortex-M processors have three modes of operation:

- **Handler Mode**: Privileged mode for ISRs, fault handlers, the SVC handler, and the PendSV handler. This mode can be entered only via an exception or interrupt.
- **Privileged Thread Mode**: Privileged tasks (*ptasks*) run in this mode. It can be entered only from handler mode, by setting CONTROL.nPRIV = 0.
- **Unprivileged Thread Mode**: Unprivileged tasks (*utasks*) run in this mode. It can be entered from either of the above two modes, by setting CONTROL.nPRIV = 1.

In the discussion that follows, the first two modes are collectively called *pmode* and the third mode is called *umode*. Similarly, discussion below refers to *pcode*, *ucode*, *psystem_services*, *usystem_services*, etc.

protection goals

The **main goal** of protection is to protect trusted, critical software and data from less-trusted, non-critical software. Examples of trusted software are: software that is essential to the main function(s) of the device, RTOS, ISRs, handlers, and low-level drivers. Examples of less-trusted software are: code that merely gathers and reports statistics on operation, code that is vulnerable to malware such as protocol stacks, third party software (SOUP), and insufficiently tested code. Protection is accomplished by running critical software in pmode and running all less-trusted software in umode. The MPU and the SVC handler are used to isolate pcode from ucode.

The **second goal** is to minimize trusted code since it is easier to carefully write small amounts of code. Running more code in umode increases isolation of sections of code, thus improving reliability.

The **third goal** is to detect intrusions and bugs and shut them down so that critical system operation is not imperiled.

Obviously, the degree of protection implemented will depend upon the security and safety requirements of the specific system. Note that protection for a system can be increased in future releases as it becomes more widely distributed and therefore more vulnerable. smxMPU is particularly designed to foster progressive protection improvement.

smxMPU contents

smxMPU is an extra-cost option enabled by setting SMX_CFG_MPU to 1 in xcfg.h and in xarmm*.inc. It adds Memory Protection Unit (MPU) support and a software interrupt (SWI) service call API to smx and to smx++ for Cortex-M processors. smxMPU consists of the following modules, plus conditional code in some smx modules:

mpu.c	MPA template and MPU init and load functions.
mpu.h	MPA constants and typedefs.
xmpu.c	MPU functions, unprivileged system service shell functions and system service jump table.
xapiu.h	Unprivileged system service prototype functions, starting with “smxu_” and system service to usystem service mapping macros. To be included ahead of ucode.

mpu.c,h are in the \BSP directory and are compiled and linked by the application. The rest of the files are in the \XSMX directory and are compiled and put into the smx library.

MPU basics

Cortex-M MPUs[1] have 8 *slots*. Each *active* slot defines a *memory region* with attributes such as size, alignment, read/write (RW), read only (RO), execute never (XN), etc. Slots in which the EN bit is 0 are *inactive* and have no effect upon memory accesses. Hence a user is not forced to use all slots. Unused slots usually are filled with 0’s to disable them. In the event that slot regions overlap, higher numbered slots prevail.

An unfortunate aspect of the Cortex-M MPU is that region sizes must be powers of 2 ranging from 32 bytes to 4 GB, and regions must start on multiples of their sizes. These requirements undermine the utility of the MPU by potentially wasting a substantial amount of memory. For example, if a protected stack increases from 256 bytes to 260 bytes, the region containing it must be increased from 256 bytes to 512 bytes and if it is not already on a 512-byte boundary it must be moved up 256 bytes to the next 512-byte boundary, resulting in a nearly 200% waste of memory! Later in this manual, methods are presented to reduce wasted memory, but using the MPU is nonetheless likely to significantly increase memory requirements.

MPU initialization is performed by sb_MPUInit() in mpu.c. It is called at the very beginning of the startup code. Following initialization, the MPU *background region* is normally enabled[2]. Low-level slots may be filled with privileged regions to enhance protection. This is discussed later. The background region flows around all other regions in the MPU so that all of memory is covered and accessible to pcode (not ucode). It is considered to have a slot number of -1. Hence any other MPU region takes precedence over it in its region of memory. In the background region, default memory attributes are in effect, as if no MPU were present. System initialization is performed in this mode, which is called the *baseline mode*.

[1] This manual is concerned only with the ARMv7-M MPU, which is used on Cortex-M3/4/7 devices. The ARMv8-M MPU fixes significant limitations of the v7 MPU. It is used in upcoming Cortex-M23/33 devices.
[2] This is actually controlled by the MPU_BKGND configuration constant in mpu.h. Not running in background mode is an advanced technique, which is discussed later in this manual. Until that point, background mode is assumed to be enabled.

converting legacy code

As previously noted, ease of legacy code conversion is a primary goal of smxMPU. Even with SMX_CFG_MPU set to 1, legacy code should run normally. In this case, all code is running in the baseline mode. This is a good starting point for improving the security of existing software. An effective methodology is to gradually convert less-trusted modules to ucode and less-trusted tasks to utasks, all the while verifying that the system still runs correctly. If a problem occurs, the process can be quickly reversed and the problem tracked down and fixed.

Trusted code and trusted tasks may be best left running in pmode. It is necessary to use the linker command file to isolate pcode and pdata from other code and data in order to assure that it cannot be accessed from utasks. Nonetheless, pcode and pdata will be in the background region and thus accessible by all ptasks, ISRs, LSRs. This assures that mission critical code, which may have been carefully crafted, need not be rewritten – it stays the same and runs the same. Furthermore, ptasks and utasks communicate the same as before, with the exception that utasks may not perform *restricted system services* (e.g. smx_SysPowerDown()!).

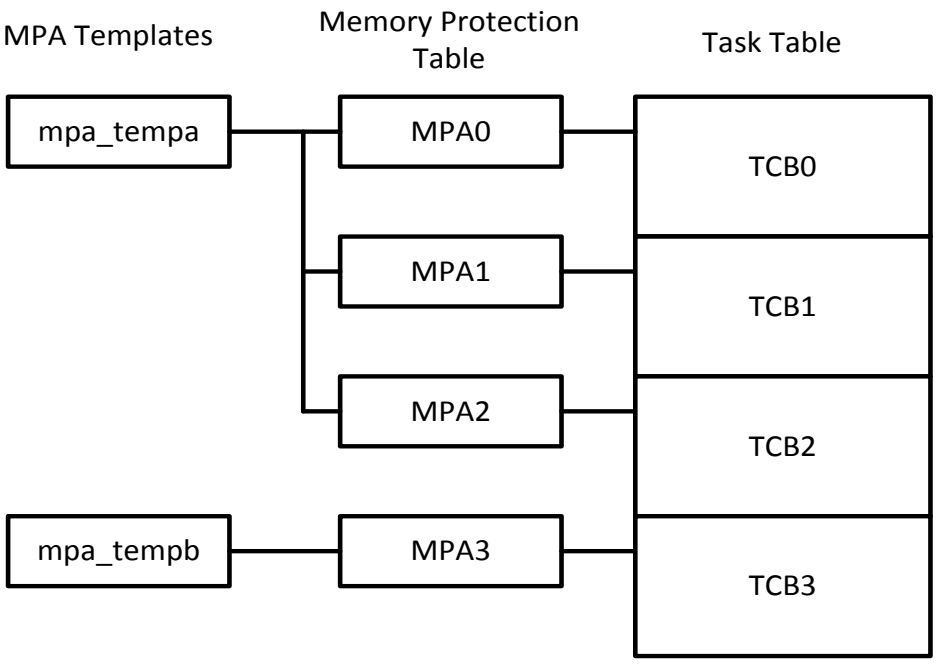
developing new code

Baseline mode may also be the best starting point for developing new code for new projects since debugging is simpler. However, once the code is operating reasonably well, moving it into umode will enable the power of the MPU and SVC to assist in debugging problems such as stack and buffer overflows, invalid operations, etc. Of course, this can only be done if the final code is intended to be ucode.

Structure

task control

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



The *Memory Protection Table* smx_mpt[], has a Memory Protection Array (MPA) for each task. The MPAs are in the same order as the TCBs in the task control table. Each MPA is a replica of the MPU when its associated task is running. The MPA is loaded into the MPU when its task is started or resumed. Hence, each task has its own regions in the MPU, when it is running. This applies to both ptasks and utasks.

Also shown in the figure, MPA Templates determine the contents of MPAs. A template may be shared between MPAs. This would be the case for a group of tasks associated with the same function (e.g. networking). Alternatively, a task may have its own template. An example of this would be a utask that is isolated from all other utasks (e.g. it has its own code and data regions).

Each MPA is an array of structures. The size of this array is governed by MPA_SIZE in mpu.h. The MPA array is loaded into the MPU starting at the MPA_START (in mpu.h) slot. This allows lower slots, not needed by tasks, to be used to enhance pmode protection, or to be left inactive. This is discussed later. Setting MPA_SIZE less than 8, if fewer regions are needed by tasks, reduces memory used and task switching time.

An MPA array element is a structure consisting of two 32-bit fields named *rbar* and *rasr*. These are exact copies of the MPU RBAR and RASR registers in each slot, except that the valid bit is set in rbar, but not in RBAR. When a task switch occurs, the new task's MPA is copied to the MPU by using the *alias regions* of the MPU. This permits loading up to four regions, at a time. This plus using a reduced MPA_SIZE results in minimal overhead on task switching time. MPA_SIZE allows a tradeoff to be made between security and performance.

MPA templates

After a task has been created, its MPA is loaded with a template by:

```
smx_TaskSet(task, SMX_ST_MPA, mpa_temp_xxx);
```

where, for example, mpa_temp_t2a is an MPA template for utask t2a:

```
const MPA mpa_temp_t2a = {
  /*0*/   {V | 0, RW_DATA},
  /*1*/   {RA("t2a_data")      | V | 1, RW_DATA | RSI("t2a_data")      | EN},
  /*2*/   {RA("t2a_code")     | V | 2, UCODE     | RSI("t2a_code")     | EN},
  /*3*/   {RA("ucom_code") | V | 3, UCODE     | RSI("ucom_code") | EN},
};
```

The macros used above are defined as follows:

```
#define RA(s)  ((u32)__section_begin(s))                    /* region address */
#define RSI(s) (30-__CLZ(__section_size(s))<<1) /* region size index */
```

This template has only four regions, because MPA_SIZE = 4. The MPA regions are relative to the MPA_START slot in the MPU (i.e. MPA region 0 is loaded into it). In the above array of struct's, the first field is *rbar* (region base address register). Looking at rbar for region 1, we see that it starts at t2a_data, the V flag is set, and region 1 is selected. The V flag is necessary to override the MPUs RNR register in order to perform rapid MPU loading using alias regions. The second structure field is *rasr* (region base attribute and size register). Looking at rasr for region 1, we see that it is a RW_DATA region, its *size index* follows, and EN means that the region is active. The size index is the power-of-two exponent – 1 for the region (e.g. 4 for a 32-byte region). The RSI() macro automatically calculates the size index, to avoid errors.

Region 2 is similar, except that it starts at t2a_code and is a code region. Region 3 is also a code region and it starts at ucom_code. This region contains the shell functions discussed in the SWI API section below and other common functions needed by utasks. Region 0 is the task stack region, which is discussed below. When task t2a is running, and the above template has been loaded into the MPU, t2a can access only the regions shown and only in the manner permitted (e.g. UCODE (see below)).

Region attributes are defined as follows;

```
#define RW_DATA (XN | RW)
#define UCODE     (RO)
```

where XN, RW, and RO are MPU attributes, which mean *execute never*, *read/write*, and *read-only*. These and other attributes are defined in mpu.h. Hence t2a_data cannot be executed, and t2a_code cannot be written by t2a. Nor can t2a access any system code or data, nor the code or data of any other task, unless it shares a region, such as ucom_code, with that other task. Hence, t2a is pretty well fenced in, as is desirable for untrusted or vulnerable code.

t2a_data, t2a_code, etc. are block names defined in the *linker command file*, which is discussed next.

creating and locating blocks

The linker plays a prominent role in assigning MPA regions to actual memory. In general, splitting control between code and a command file is error-prone because the two can get out of sync. This is particularly bad in this situation where the goal to increase, not decrease, reliability. The combined approach presented in this section and the above section is intended to make the process of creating MPU regions easier, thereby reducing errors.

The following discussion is based upon the IAR ILINK linker. Hopefully it will serve as a guide for other linkers having similar commands. For consistency with the above, we will work with ucom_code and ucom_data below.

Start by defining *sections* in the C source code modules. For example, in each C module containing code for a specific region, start the code with:

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

// Place all ucom functions here.

#pragma default_function_attributes =
```

".ucom_code" is a section name to identify a section containing common code between tasks. For code that is specific to taskA, use a section name such as ".taskA_code". The "." ahead of the name is not required. It is just a useful convention to indicate that the name is a section name. Several functions can be enclosed above. Also, the above structure can be repeated in other modules, and all ucom functions will be combined into the .ucom_code section.

For data:

```
#pragma default_variable_attributes = @ ".ucom_data"

// Place all ucom data here.

#pragma default_variable_attributes =
```

As with code, many variables can be enclosed above, and the above structure can be repeated in other modules to create a single .ucom_data section containing all of the ucom variables. This, of course, can also be done for a section containing data for a single task, such as .taskA_data.

In the linker command file (.icf for ILINK), for the .ucom_code section:

```
define block ucom_code with size = 1024, alignment = 1024 {ro section .ucom_code};
…
place in ROM {block ucom_code, …};
```

If desired, additional sections can be placed in the ucom_code block and they can be placed in a fixed order, if desired. In this example, only the .ucom_code section is placed in the ucom_code block. (Note that the section and block names differ only by a ".". This is not required, but the names must be different.)

In the C file above where the template is being defined or in an included header file:

```
#pragma section="ucom_code"
```

Now, ucom_code can be used in the MPA template definition, as shown in the previous section. It is important to note that the block names, defined by the linker, not the section names defined

by the C code, are used to define regions in MPA templates (and hence in the MPU). It is an idiosyncrasy of IAR EWARM that the __section_begin() and __section_size() pragmas[3] work on blocks as well as on sections. It would be ok to use the section name in __section_begin(), if it is the first section in the block, but it would be erroneous to use it in _section_size() because the section will usually be smaller than the block which contains it. This is an unfortunate complexity of IAR EWARM that requires getting used to.

In effect, blocks in the linker command file are equivalent to regions in MPA templates and ultimately in the MPU. Using blocks mitigates against assigning wrong sizes and/or alignments. If a block is too small for the section(s) it contains, the linker will report an error. Make sure that the block size is a power of 2 greater than or equal to 5 (i.e. 32 bytes or larger) and that the block alignment is the same as its size. This will avoid pernicious errors, likely to result in Memory Management Faults (MMFs). The linker should optimize placement of blocks in memory regions.

As noted above, multiple sections can be placed in the same block and their order can be fixed. Also, smaller blocks can be placed in larger blocks and their order fixed. These capabilities are useful for final manual memory efficiency improvements, which are discussed near the end of this manual. (Of course, if you have plenty of memory, you should rely on using the linker only.)

more on MPA templates

As shown in the smx_mpt[] diagram, above, every MPA requires an MPA template. However, a template may be shared between MPAs (and hence, tasks). In this regard, it is important to note that MPA region 0 (loaded into the MPA_START slot of the MPU) is dedicated to the task stack. It is loaded dynamically by the scheduler, and thus only the attributes are part of the template for MPA region 0. See **task stacks**, below, for more discussion.

The creation of templates is dependent upon the security strategy. For example, the strategy might be to just isolate all utasks from all pcode, including ptasks. In this case, a single template for all utasks might suffice (and pcode could just use the MPU background mode). On the other hand, it might be desirable to isolate certain groups of utasks from each other or even individual utasks from each other as well as from pcode.

Note that the MPU is of value for ptasks also, in that it allows assigning attributes (e.g. XN) to regions for greater reliability. Although pcode and ptasks are trusted, they are still susceptible to bugs and to single event upsets (e.g. high energy neutrons from gamma rays). pmode protection is discussed more later.

Since a handler can interrupt any task, it is important to note that all handler code is subject to the MPU region permissions of the current task, which could be any utask or any ptask. For example, if a region of RAM is defined as XN, then a handler cannot execute code from that region even though background mode (i.e. the *default memory map*) would permit code execution from there. Problems resulting from this can be reduced by placing handler code and data outside of task regions as much as possible.

The foregoing illustrates that there is a need for some advance region planning in new designs, even though MPU support need not be initially implemented. It also illustrates potential

[3] These pragmas are used in the RA(s) and RSI(s) macros defined above. These macros are in turn used to define MPA templates.

problems for porting legacy code to the MPU. These problems may require some code restructuring or ingenuity in defining MPA regions.

task stacks

MPA region 0 is reserved for *protected task stacks*. Protected task stacks must come from the stack pool so that they can be correctly sized and aligned. Stack pool stacks are allocated to tasks when they are dispatched by the scheduler. Following stack allocation, MPA[0] for the task is loaded by the scheduler. These stacks are not only protected, but also, overflows are immediately detected and reported as Memory Management Faults (MMFs). In order to permit the latter, the register save area (RSA) has been moved from the top to the bottom of the stack block. Also, STACK_PAD_SIZE (acfg.h) must be 0. If not, stack overflow will not be caught until the stack pad is also exceeded. This may be desirable during early debugging in order to not be distracted by MMFs due to stack overflows. (Stack pads are normally used during debugging so that the system will keep running despite stack overflows.) Normal smx stack overflow detection is still enabled and reported as STK OVFL by the smx error manager. However, since the stack block has not been overflowed operation continues normally.

The stack pool block size is controlled by STACK_BLK_SIZE in main.h. If SMX_CFG_MPU is 1, it must be a power of two from 32 to 4GB, and stack alignment is the same. A stack block consists of the Register Save Area (RSA), the stack, and the stack pad. The RSA is typically 32 bytes, stack size is determined by STACK_SIZE in acfg.h, and stack pad size is determined by STACK_PAD_SIZE, also in acfg.h. If the sum of these exceeds STACK_BLK_SIZE a warning is issued to increase it to the next power of two. Normally, in released systems, the stack pad size is set to 0 to minimize memory and so that stack overflows will immediately generate MMFs.

If a task is using a heap stack, the MPU[0] slot is inactive. In this case the task stack will be located in the region containing the heap, and will be subject to that region’s attributes. Overflow of a heap stack will not result in an MMF, but will be reported by smx, when the task using it is suspended, stopped, or deleted.

System Operation

processor control

When an interrupt or an exception occurs, the processor automatically switches to pmode and to the system stack, SS (also called the main stack, MS, by ARM). It also automatically stacks r0-r4, r12, lr, pc, and xpsr on the task stack, TS (r0 is on top). The processor is now in handler mode. If PendSV was set, the processor is running smx_PendSV_Handler(), which is used for the smx task scheduler and LSRs.

If task->flags.umode in the task TCB is 1, task is a utask; if it is 0 task is a ptask. Following task creation, this flag is 0. It can be set to umode prior to starting the task with:

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

smx_TaskSet() is restricted to pmode. The umode flag is ignored unless SMX_CFG_MPU is 1. If so, when smx_PendSV_Handler() returns to thread mode, it sets the nPRIV bit in the CONTROL register = umode flag. This determines whether the task runs as a ptask or as a utask.

SWI API

ptasks can directly access all system functions and data, including smx system services, but utasks cannot do so, because they are outside of utask regions. Instead, utasks must use a software interrupt (SWI) API to access system services, and they can never access system data directly. In addition, only services permitted by smx to smxu macros in xapiu.h (see below and section *restricted services*) can be accessed by utasks. These barriers protect the operating system from untrusted code.

The SWI API is implemented via the Cortex-M SVC instruction:

```
SVC n
```

where n specifies the system service to perform. An application module can start with pcode followed by ucode, or it may be entirely pcode or ucode. Either way, preface its ucode with:

```
#if SMX_CFG_MPU
#include "xapiu.h"
#endif
```

xapiu.h consists of mapping macros for system services that are permitted in umode. For example:

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

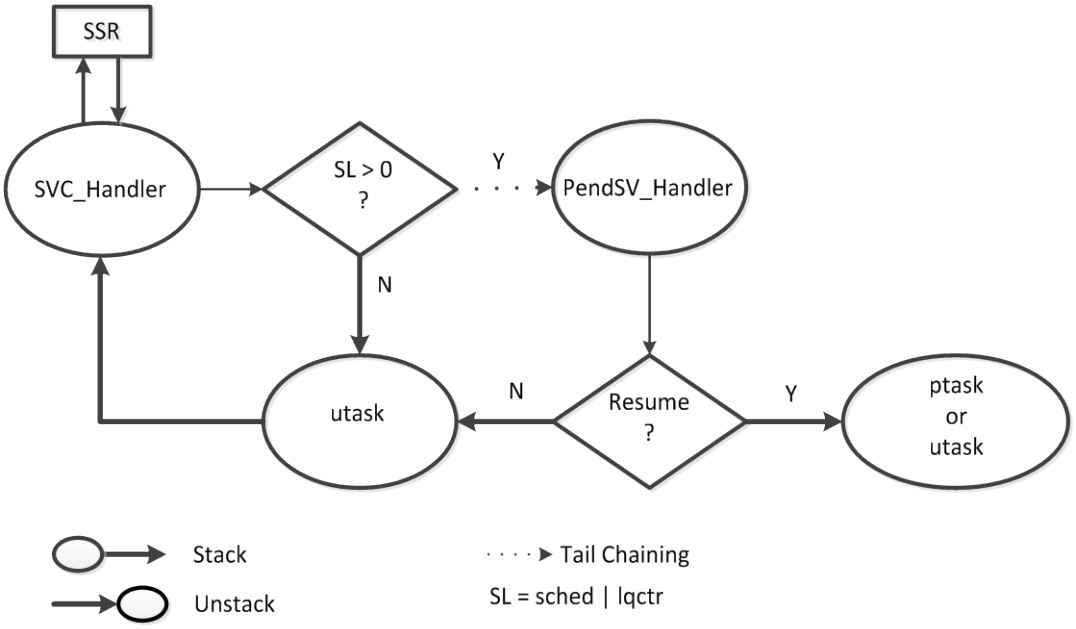
This, in effect, overrides the function prototype in xapi.h with this macro so that for the rest of the application module, rather than calling the service directly, it calls a shell function, instead:

```
NI BOOLEAN smxu_SemSignal(SCB_PTR sem)
{
  sb_SVC(SMX_ID_SEM_SIGNAL & 0xFF)
}
```

This shell function is in xmpu.c and it serves to call the SVC instruction with n == ID of the system service. (System service IDs are defined in xdef.h and are also used by smxAware.) NI (Not Inline) is a macro that blocks function inlining by the compiler.

Note: All of the above is done at compile time and thus becomes hard-coded and therefore resistant to malware and bugs, especially if the code is in ROM.

As shown in the following diagram, smx_SVC_Handler() is invoked by the SVC instruction and runs in handler mode:



When smx_SVC_Handler() starts running, the system service parameters are in the task stack, TS, due to *stacking* by the processor. It moves parameters 0 thru 3 into r0 thru r3 and it moves the 5th parameter, if any, into the top of the system stack, SS. (This is where the system service expects to find these parameters.) smx_SVC_Handler() then calls the system service via the smx_ssrt[] jump table in xmpu.c. The system service executes normally and returns to smx_SVC_Handler(), which moves the system service return value from r0 to its correct position in TS. The handler return operation, performed by the processor, *unstacks* all stacked registers in TS, thus the return value ends up in r0.

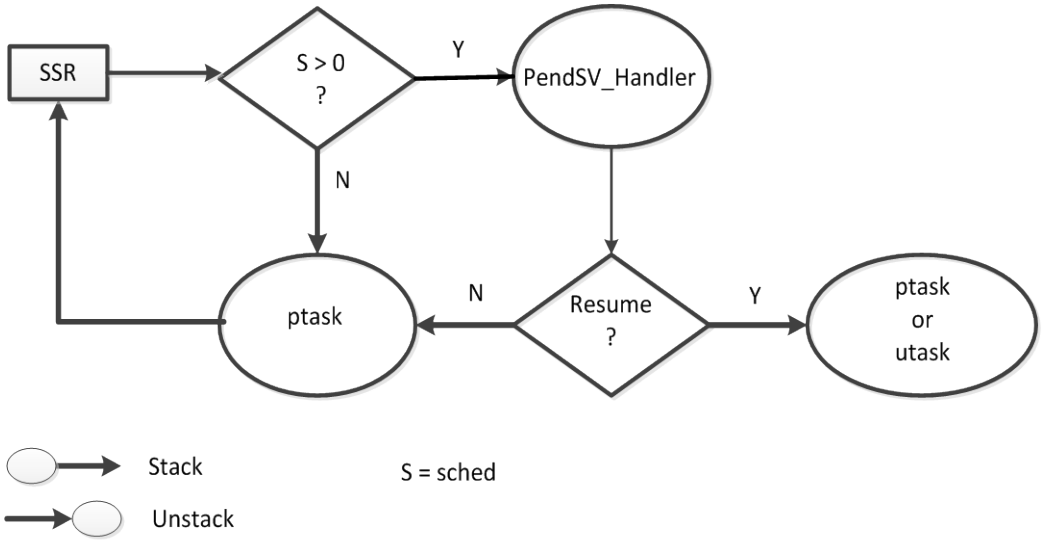
If the system service has resulted in a need for the task scheduler to run (sched > 0) or an interrupt has resulted in a need for the LSR scheduler to run (lqctr > 0), the smx_PendSV_Handler() will have been pended. In this case the processor tail-chains[4] from the smx_SVC_Handler() to the smx_PendSV_Handler, instead of returning to the utask.

In this case, control does not go back to the point of call in the utask yet, but rather to the smx task or LSR scheduler running in smx_PendSV_Handler(). This may result in the current task being suspended and another task being run, instead. The preempting task can be either a utask or a ptask. However, eventually the suspended utask will be resumed, unstacked, and continue running, assuming it was not stopped nor deleted by a preempting task..

Note: All of the above is done in privileged mode and is thus protected from malware.

[4] This term, used by ARM, means that normal unstacking and stacking are skipped when one handler immediately follows another. Hence, when the PendSV handler returns, it will not return to the SVC handler, but rather to the point from which the SVC handler was called, and it will use that task's stack to do so. The transition from one handler to the other is thus transparent to the application

By contrast, the following diagram shows operation when a system service is called from a ptask:



Note that this is quite a bit simpler: smx_SVC_Handler() is not involved. The ptask calls the system service directly, and if sched is set, smx_PendSV_Handler() is pended. From here, operation is identical to that for the utask. If an interrupt occurs during ptask, the system service, or smx_PendSV_Handler, it is handled normally (see smx User's Guide).

smx system services operate the same regardless of whether they are invoked from ptasks or utasks. For example, a utask may test a semaphore and become suspended upon it. A ptask may signal the semaphore and the utask will resume. Or vice-versa. A ptask may have higher or lower priority than a utask and the scheduler will dispatch the tasks according to their priorities. (Privilege has no priority.) What is different is that the ptask executes trusted code (pcode) and usually has full access to memory and peripherals, whereas the utask executes unprivileged code (ucode) and has access to only what the MPU permits, as dictated by its MPA, and MPAs are strictly controlled by pcode.

Lest there be concern that ptasks are completely unbridled, note that it is possible to prevent access to a region via the MPU. Hence, a region that is RW to one task could be RO to another and XN to both. On the other hand, ptasks do have direct access to all smx services. As security of a system is tightened, consideration should be given to limiting ptasks as well as utasks.

restricted services

For safety, some smx services should not be permitted in umode. For example, services that impact other tasks should not be allowed, such as:

```
smx_StopTask(task, INF)
```

Malware could use this to stop a critical task. Similarly, services that impact system operation should not be allowed, such as:

```
smx_SysPowerDown(power_mode)
```

No mappings, in xapiu.h, are given for restricted services, such as these. Hence, their use in ucode will result in compiler errors and thus be quickly corrected. However, this is not enough; the smx_ssrt[] jump table has a slot for every system service. It is conceivable that malware could try to jump through a restricted slot (even though smx_ssrt[] is in privileged memory). An additional protection is that all restricted slots are set to smx_PVEM() (rather than a system service), which generates a PRIVILEGE VIOLATION error and returns to the point of call with nothing done.

If ucode attempts to execute a restricted service directly (i.e. by knowing its address), an MMF violation error will occur because smx code is not accessible via the MPU when the utask is running. This results not only in reporting an MMF, but also in stopping the offending task, which is done to limit damage due to malware or bugs.

To add a system service to the permitted list, simply modify xapiu.h to map it to its smxu_system_service equivalent, create a shell for the smxu_system_service in xmpu.c, and add the system service to the smx_ssrt[] jump table in xmpu.c, in place of the smx_PVEM already there. Be sure to use the system_service ID defined in xdef.h. Alternatively, to remove a system service from the permitted list, reverse this process.

It may be desirable to have different permitted system service lists for different utasks. For example, for third-party SOUP, it may be desirable to have very minimal list, whereas for in-house code it may be desirable to have a wide range of RTOS services available. This can be easily accomplished by defining different xapiu.h's for different modules. Note that these are compile-time changes and are not something malware could do or undo. Basically, different lists implement different levels of trust.

pmode protection

The need for umode protection is obvious; the need for pmode protection is less obvious. pmode protection is an important consideration for legacy systems. When converting existing software to the MPU, most of the code may have been working reliably for years, and thus it may be desirable to leave that code in pmode and convert only a small amount of recently added or new code to umode. In a new project, there may be limited resources to convert pcode to ucode late in the project. Hence, in both cases, the amount of protection that can be achieved in pmode is important.

Although we hope that pmode cannot be penetrated by malware (else we are up to our armpits in alligators) there are other things that can go wrong. One, of course, is latent bugs. Even the best code is likely to have some of these and less good code even more. Of course, the less pcode there is in a system, the less likely these are. The other problem is environment disturbances. As noted previously, cosmic ray collisions in the upper atmosphere produce high-energy neutron streams that cause bit flips in semiconductor devices. These neutrons can penetrate up to 4 feet of concrete, so shielding small devices is unlikely. As IC feature sizes decrease and more unshielded devices are installed at higher altitudes and latitudes, they become more of a problem. (Think of autonomous vehicles driving in the mountains and Alaska – maybe you want to keep your hands on the steering wheel!) In addition, there are other natural phenomena to be concerned about, such as lightning, heat, cold, humidity, shock, etc.

Therefore, it is desirable for ptasks to have task-specific regions, just like utasks. Unfortunately, these regions cannot be used to detect overflows because the background region fills in the spaces between them. But they can be used to impose access restrictions such as RO, XN, etc.

In addition, it is desirable to augment protection for handlers and initialization code. Since handlers can interrupt any task, this requires permanent MPU regions that are present for all tasks, unless MPU switching is implemented for handlers, which is unlikely due to its overhead. So, it is desirable to define pmode regions for the lower slots that are not needed by tasks, rather than just leaving the slots empty. This is done by sb_MPUInit(), which is called at the beginning of startup.

For example, if five regions are sufficient for tasks, then regions below MPA_START (0, 1, and 2), can be used for permanent pmode regions. Since the background region has a priority of -1, these slots will prevail over it, except that overflows cannot be detected, as noted above. An exception to this is that by locating SS[5] first in SRAM an overflow will trigger a Bus Fault. Regions 0, 1, and 2 could be assigned to SRAM, ROM, and DRAM. Then assigning XN for SRAM and DRAM and RO for ROM is beneficial. If a data cache is present, dcache attributes can also be defined. Of course, these regions must be limited to privilege mode access, only, in order to prevent utasks from accessing them.

In view of the above limitations, it would be desirable to operate without the background region. Unfortunately, this may require up to six permanent MPU regions for handlers: SS, SRAM, ROM, DRAM, IO, and SYSTEM, leaving only two for tasks. But, tasks need at least five slots, so the Cortex-M MPU simply does not have enough slots to achieve both good pmode protection and good task-specific protection.

memory management fault (MMF)

The MMF is connected to the smx_EM() error manager. It produces an MMF_VIOL error, which is treated as being irrecoverable, and thus smx_EMExitHook() is called. By default, this function generates a TASK STOP warning and stops the current task. Deciding how to handle an MMF violation is an important security consideration. For example, if the violation occurred in a non-critical task, a better action than stopping the task might be to restart it, which should kill the malware virus and give the task a new lease on life. However, if the MMF occurred in a critical task, it may be better to reboot the system or to stop it altogether, pending operator intervention. What actions are desirable and feasible are clearly application-dependent and an important part of its security strategy.

[5] System Stack, also known as the Main Stack, MS, in ARM terminology.

Application Notes

size and performance

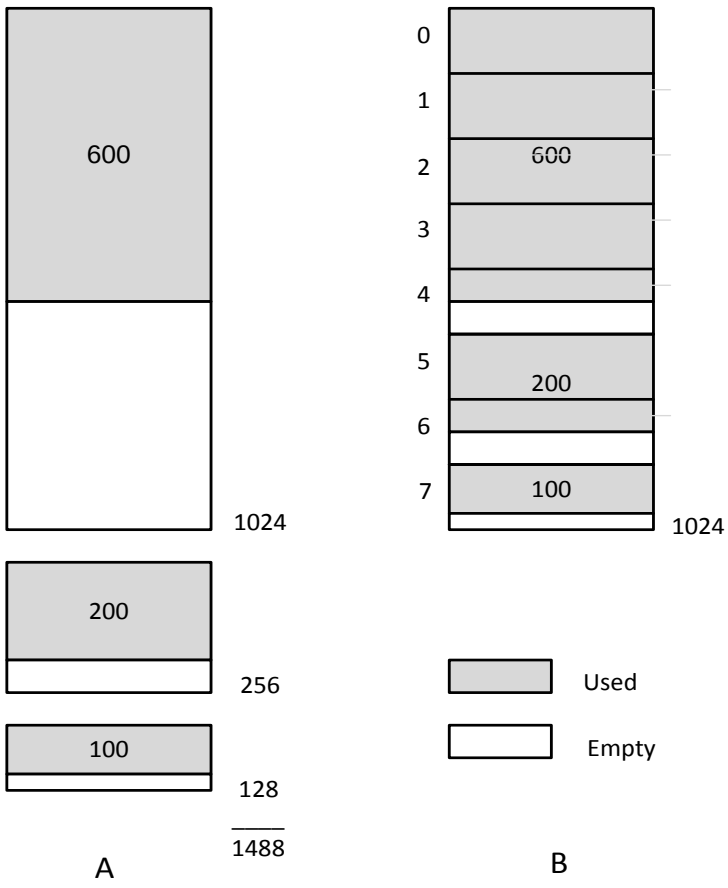
smxMPU adds about 1800 bytes of code to smx. It adds 8*MPA_SIZE*NUM_TASKS bytes of RAM and 8*MPA_SIZE bytes of ROM per MPA template to the application code. Execution overhead for typical system services, such as smx_SemTest() and smx_SemSignal() is about 100%. Percent overhead is less for longer system services, since the overhead is a fixed amount of time per system service. Overhead for task start is about 25%, for loading the task's MPA, then the MPU, and for other SMX_CFG_MPU conditional code. It is about half of this for task resume and negligible for task suspend or stop.

improving memory efficiency

Due to the ARMv7-M MPU requirements that regions be power-of-two sizes and that they be aligned on their sizes, wasted memory can be substantial. The following subsections discuss methods to reduce waste. However, this should be done late in a project after memory requirements have stabilized, else it probably will be wasted effort.

Avoiding alignment gaps can be done easily by ordering regions by decreasing size in memory. Then gaps due to alignment cannot occur. The linker should do this naturally.

Subregions can greatly reduce wasted memory. Every region has 8 equal-size subregions, which can be disabled by bits in the RASR field of the region. Three regions are shown in figure A, below:



They are 1024, 256, and 128 bytes. The actual memory used within each is 600, 200, and 100 bytes, respectively, as shown by the shaded areas, but each region must be the next power of two larger than its actual size. Hence total memory used by the regions is 1488 bytes, as shown in Figure A. Figure B shows the use of subregions within the 1024-byte region. There are 8 subregions, each 128-bytes in size. These regions are numbered on the left side. Hence subregions 0 to 4 = 5 * 128 = 640 bytes, which is large enough for the 600-byte block. Subregions 5 and 6 = 2 * 128 = 256 bytes, which is large enough for the 200-byte block, and subregion 7 = 128 bytes is large enough for the 100-byte block. This results in a 45% memory savings, or put another way, waste is decreased from 526 bytes to 62 bytes or from 55% to 6%.

The regions are defined such that the starting address in each RBAR is the same for all, as is the region size in each RASR. Each RASR can have different attributes (e.g. RO, RW, etc.). The subregion disables must be as follows: region 1: N57 (not 5, 6, & 7); region 2: N04 | N7 (not 0 thru 4 or 7); region 3: N06 (not 0 thru 6). Overflow will be detected if any region overflows into another region, as long as the regions are not co-resident in the MPU (i.e. they are used in separate MPAs).

The linker command file for the above would be as follows:

```
define block a600_code  with size = 640, alignment = 1024 {ro section .a600_code};
define block b200_code  with size = 256, alignment = 128 {ro section .b200_code};
define block c100_cons  with size = 128, alignment = 128 {ro section .c100_cons};
define block abc_ro  with fixed order {block a600_code, block b200_code ,block c100_cons};
```

```
place in ROM        {block abc_ro, ro}
```

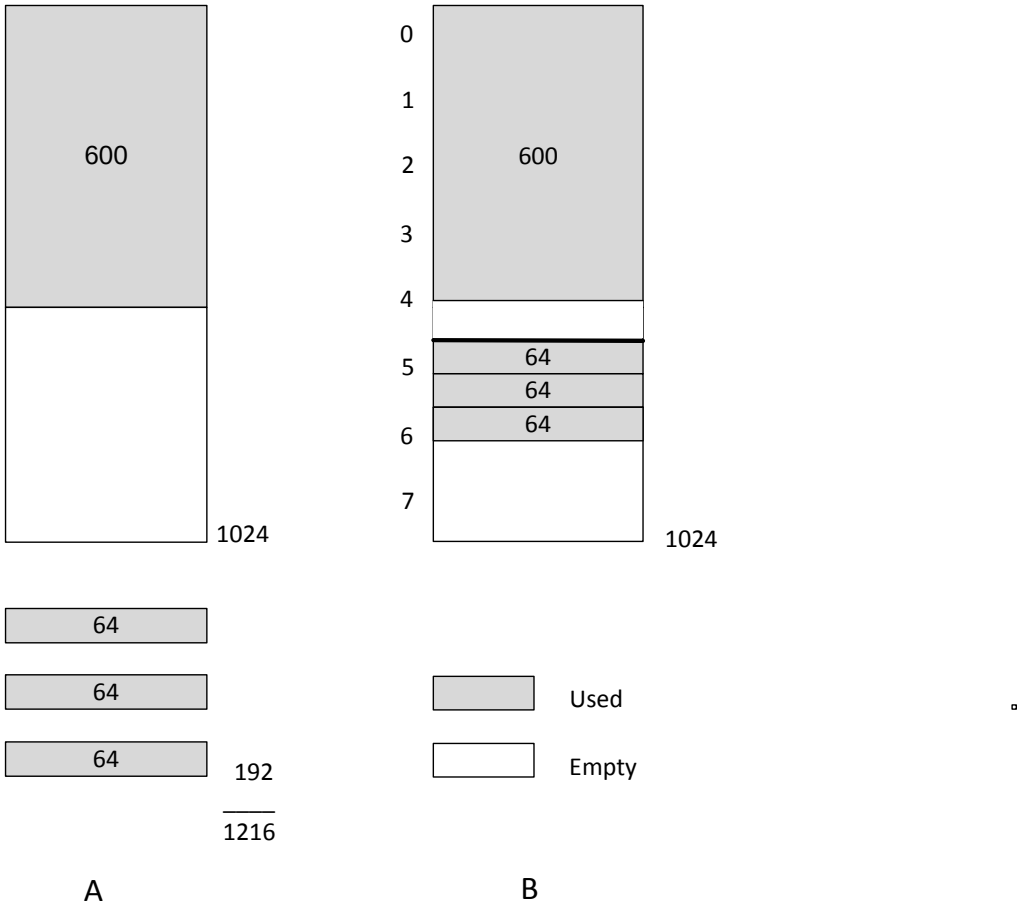
The result of the above linker commands is that the blocks are placed in the order and aligned as shown in figure B, and the combined block, abc_ro, is placed in ROM with other ro code and data.

Then, the regions are defined as follows, in their respective MPAs (a, b, and c):

```
{RA("abc_ro") | V | Sa, PCODE   | RSI("abc_ro") | N57 | EN},
{RA("abc_ro") | V | Sb, UCODE   | RSI("abc_ro") | N04 | N7 | EN},
{RA("abc_ro") | V | Sc, RO_DATA  | RSI("abc_ro") | N06 | EN},
```

Note that each region has the same region address (RA(“abc_ro”) and the same region size index (RSI("abc_ro")). However the MPA slot numbers (Sa, b, and c) may differ, the attributes differ, and the subregions disabled are different.

Overlaid Host Regions A similar result to that shown above can be achieved by overlaying unused space in a large *host* region with smaller *overlaid* regions, as shown below:



This approach can be used to pack many small regions into the space not used by the host region. In this case, the small regions are smaller than the host subregion size. To assure that the host

region cannot overwrite any of the the overlaid regions, its subregion disables must be N57 (not 5, 6, & 7). In this case, the RBAR starting addresses would vary from region to region, as would the RASR region size. Overflow will be detected if any overlaid region overflows, as long as the regions are not co-resident in the MPU.

The linker command file for the above would be as follows:

```
define block a600_code  with size = 640, alignment = 1024 {ro section .a600_code};
define block b64_code    with size = 64, alignment = 64 {ro section .b64_code};
define block c64_code    with size = 64, alignment = 64 {ro section .c64_code};
define block d64_cons    with size = 64, alignment = 64 {ro section .d64_cons};
define block abcd_ro      with fixed order {block a600_code, block b64_code block c64_code,
                                                                          block d64_cons};
place in ROM   {block abcd_ro, ro}
```

The result of the above linker commands is that the blocks are placed in the order and aligned as shown in figure B, and the combined block, abcd_ro, is placed in ROM with other ro code and data.

Then, the regions are defined as follows, in their respective MPAs (a, b, and c):

```
{RA("abcd_ro") | V | Sa, PCODE   | RSI("abcd_ro") | N57 | EN},
{RA("b_code")        | V | Sb, UCODE   | RSI("b_code") | EN},
{RA("c_code")    | V | Sc, UCODE   | RSI("c_code") | EN},
{RA("d_cons")    | V | Sd, RO       | RSI("d_cons") | EN},
```

Note that each region has a different region address (RA(“abcd_ro”), and the same region size indices correspond to Figure B, above. The MPA slot numbers (Sa, b, c, and d) may differ, the attributes may differ and only the host region has subregions disables.

security plan

The need for a security plan has been alluded to a few times, and a few suggestions have been offered regarding it. There are many aspects to security, hence such a plan is outside of the scope of this manual. Essential as it may be in this modern day, security adds yet another design dimension to projects, which probably already have too many dimensions. At the early stages of a project, it is pretty much in the same category as a root canal.

Undoubtedly some up-front security planning is necessary, such as which tasks will run in pmode and which will run in umode, how tasks will be grouped for security purposes, what regions will be created, how security breaches will be handled, etc. However, actual implementation of MPU support is probably best postponed until the late stages of the project to allow the code to stabilize. Otherwise, there is likely to be much wasted activity when regions are overflowed, structural changes become necessary, etc. Unfortunately, this may not be a good time either. Late in a project is often a time when panic is rampant and niceties such as safety, security, and reliability may not seem important. (However, implementing MPU support late in a project could help to find the elusive bugs that that are responsible for delaying delivery.)

Hence it may be even better to wait until after shipments have begun, then implement MPU support. The same techniques as used for converting legacy code to the MPU also apply here,

except that, assuming some initial security planning, they should be even easier to implement. Once the dust settles, it is possible to step back, look at the system security requirements, and start making the system more secure. During this time, manufacturing and installation problems are being slowly solved by other people, shipments are gradually increasing, and likewise, security can be gradually improving.

tips for avoiding errors

It is particularly galling to introduce bugs when trying to improve the reliability of a system. Using an MPU, while not particularly difficult, does require different thinking. The following tips are intended to help minimize MPU errors:

1. Prefix utask names and their main code with “u”, e.g. utaskA, utaskA_main(). This helps to keep the actors straight so you know who is supposed to be able to do what.
2. If a module has pcode and ucode, put a strong demarcation barrier between them, to help identify which world an object is in. Put pcode first, since ucode starts with #include “xapiu.h”, which applies for the rest of the module. It would be safer to put pcode and ucode into separate modules, but that may not be desirable for functional reasons – e.g. keeping ptasks and utasks together that perform a common function. See the example below.
3. Make blocks just large enough to contain their section(s). As code grows, the linker will let you know if a block overflows and you can increase the size and alignment of the block to the next power of two.
4. Unless you have the powers of 2 well-memorized, using hexadecimal numbers for sizes and alignments, e.g. 0x800 instead of 2048, may avoid problems. For a power of 2, only one digit is permitted, and the digit must be 1, 2, 4, or 8.
5. In a block definition, be sure to update the alignment with the size and it should be the same.
6. Placing blocks into large regions that correspond to physical memories (e.g. DRAM, SRAM, flash, etc.) allows the linker to order the blocks efficiently in order to avoid gaps that waste memory. Only when the project is essentially done and memory has overflowed should the subregion and overlay techniques described above be applied.
7. Templates should initially be broadly defined for groups of tasks. This avoids wasted time tracking down MMFs due to unreachable variables or subroutines. This aspect of the MPU takes some getting used to – it actually works! When code has stabilized, it is then practical to start isolating tasks more, if that is the security plan. Note that the .ucom_code segment, which is used for smxu shells, is a convenient place for common subroutines, since it must be present if a task is using smx services.

Example

The following is a complete example[6] showing how to create and start a utask from a ptask. In this example, the ptask is resumed from a semaphore when the utask signals the semaphore. This demonstrates how utasks and ptasks can interact via smx services. This example also shows how to define *sections* in the source code and how to put variables and functions into those sections. Below the pmode/umode boundary (--x--x--), #include "xapiu.h" causes smx service calls to be converted to smxu shell functions. These invoke the SVC handler in order to make smx service calls. For example:

```
NI BOOLEAN smxu_SemSignal(SCB_PTR sem)
{
  sb_SVC(SMX_ID_SEM_SIGNAL & 0xFF)
}
```

Following the source code example is the complete linker command file, which shows how blocks are defined from source code sections and how to locate them in memory. These linker-defined blocks are then used to define regions for utask MPA in the source code. In the region definitions, the following macros are used:

```
#define RA(s)  ((u32)__section_begin(s))       /* region address */
#define RSI(s) (30-__CLZ(__section_size(s))<<1)     /* region size index from section */
```

As the mpa_temp_ut1a table and linker command file below show, the ut1a utask can only access its own task stack, its own 64-byte data region in SRAM, its own 128-byte code region in flash, a common 64-byte data region in SRAM, and a common 1024-byte code region in flash. (The latter contains the smxu shell functions and any other common subroutines.) Any attempt to access memory outside of these regions will result in an immediate memory management fault (MMF), which will vector to the smx error manager (smx_EM). The latter will record the fault, like other smx errors, then stop ut1a. In addition, the code regions have UCODE attributes (unprivileged read-only) and the stack and data regions have RW_DATA attributes (unprivileged read/write and execute never. Violating these restrictions will also result in a MMF.

The result of the foregoing is that ut1a cannot access data nor code outside of its narrowly-defined regions, nor can it execute from RAM nor write to flash. Furthermore, it does not have direct access to smx services and data. Hence, critical resources in the system are well-protected from ut1a.

Example

```
/*
* empu.c                                Version 4.4.0
*
* smxBMPU Examples
*
* Copyright (c) 2016 Micro Digital Inc.
* All rights reserved. www.smxrtos.com
```

[6] This example is based upon the IAR EWARM tool suite. Different techniques may be required for other tool suites. It is available with the esmx example suite and can be run on supported processors.

```
*
* This software is confidential and proprietary to Micro Digital Inc.
* It has been furnished under a license and may be used, copied, or
* disclosed only in accordance with the terms of that license and with
* the inclusion of this header. No title to nor ownership of this
* software is hereby transferred.
*
* Author: Ralph Moore
*
*******************************************************************************/

#include "smx.h"
#include "esmx.h"
#include "mpu.h"

/* ut1a and ucom sections (defined as blocks in .icf, but sections here) */
#pragma section="ut1a_data"
#pragma section="ut1a_code"
#pragma section="ucom_data"
#pragma section="ucom_code"

/* MPA template for ut1a. This limits memory accessible to ut1a utask. */
const MPA mpa_temp_ut1a =
{
  /*0*/   {V | S0, RW_DATA},
  /*1*/   {RA("ut1a_data") | V | S1, RW_DATA | RSI("ut1a_data") | EN},
  /*2*/   {RA("ut1a_code") | V | S2, UCODE   | RSI("ut1a_code") | EN},
  /*3*/   {RA("ucom_data") | V | S3, RW_DATA | RSI("ucom_data") | EN},
  /*4*/   {RA("ucom_code") | V | S4, UCODE   | RSI("ucom_code") | EN},
};

#pragma default_variable_attributes = @ ".ut1a_data"
SCB_PTR sbr1; /* accessible by ut1a */
TCB_PTR ut1a;
#pragma default_variable_attributes =

void empu1(void);
void empu1_ut1a(void);

void empu(void)
{
  empu1();
  sb_MsgConstDisplay(SB_MSG_INFO, "EMPU RAN");
}

/*******************************************************************************
  empu1: Example of a ptask resumed by a utask. The current task is esmx, which
  is a ptask. ut1a and sbr1 resource semaphore are created here. The umode
```

```
   flag is set in ut1a and its MPA is loaded with the mpu_temp_ut1a template
   defined above. The locations and sizes of MPA regions are defined as
   "MPU protected blocks" in the .icf linker command file.

   sbr1 is cleared and ut1a is started, but does not run because it has
   the same priority as esmx. esmx then suspends on sbr1 and ut1a is started.
   The smx scheduler gives it a stack from the stack pool and a region for it
   is created and loaded into MPA slot 0. Then the ut1a MPA is loaded into
   the MPU staring at the MPA_START slot. Slots below MPA_START are used for
   pregions loaded by sb_MPUInit(). They do not change and are accessible only
   in pmode. See mpu.c for definitions of these regions.

   When ut1a signals sbr1, it autostops and esmx resumes and cleans up.
*******************************************************************************/

void empu1(void)
{
   sbr1 = smx_SemCreate(SMX_SEM_RSRC, 1, "sbr1");
   ut1a = smx_TaskCreate(empu1_ut1a, 1, 0, SMX_FL_NONE, "ut1a");
   smx_TaskSet(ut1a, SMX_ST_UMODE, 1);
   smx_TaskSet(ut1a, SMX_ST_MPA, (u32)&mpa_temp_ut1a);
   smx_SemTest(sbr1, NO_WAIT); /* clear sbr1 count */
   smx_TaskStart(ut1a);
   smx_SemTest(sbr1, INF);  /* wait for signal from ut1a */

   /* cleanup */
   smx_TaskDelete(&ut1a);
   smx_SemDelete(&sbr1);
}

//                          PRIVILEGED ABOVE
//x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x---x
//                         UNPRIVILEGED BELOW

#if SMX_CFG_MPU
#include "xapiu.h"
#endif

/*******************************************************************************
   ut1a is a utask started by the above esmx ptask. It signals sbr1 to allow
   the esmx task waiting at sbr1 to resume. smx_SemSignal() is converted to
   smxu_SemSignal() by the above xapiu.h header file. smxu_SemSignal() is
   a system service shell defined in xmpu.c. It invokes an SVC n instruction, where n is
   an index for the system service. This causes a switch the SVC Handler in pmode, which
   calls the real system service via the smx_ssrt[n] jump table in xmpu.c.

   The shell system services are in the .ucom_code region, which is accessible by this
   task. Note that the ut1a main function is in ut1a_code region and sbr1 is in
```

```
  the ut1a_data region, which are accessible here. The ucom_data region is not
  used in this example.

  Only code or data inside of the regions defined for ut1a are accessible by
  it and these regions are very small, as can be seen from the .icf file.
  Furthermore, these regions are accessible only as specified (i.e. as UCODE or
  RWDATA -- see mpu.h).
*******************************************************************************/

#pragma default_function_attributes = @ ".ut1a_code"
void empu1_ut1a(void)
{
  smx_SemSignal(sbr1);
}
#pragma default_function_attributes =
```

ILINK Command File

```
//*****************************************************************************
// ILINK Command File for IAR EWARM, APP, and STMicro STM32746G-EVAL board.
//
// --- Internal ROM Internal RAM Version (ROM build targets) ---
//
// STM32F746G-EVAL memory layout:
//
//   ITCMRAM   [0x00000000--0x00003FFF]  (16KB)  Unused (ITCM RAM)
//   ROM       [0x08000000--0x080FFFFF]  (1MB)  Int Flash via AXIM interface
//   ROM       [0x00200000--0x002FFFFF]  (1MB)  Int Flash via ITCM interface
//   TCRAM     [0x20000000--0x2000FFFF]  (64KB)  EVT, SDAR, System Stack (DTCM RAM)
//   SRAM1     [0x20010000--0x2004BFFF] (240KB)  Data, ADAR, EMAC buffers (Int. SRAM)
//   SRAM2     [0x2004C000--0x2004FFFF]  (16KB)  Joined with SRAM1 (Auxiliary Int. SRAM)
//   RAM       [0xC0000000--0xC1FFFFFF]  (32MB)  LCD buffer (Ext SDRAM)
//*****************************************************************************

define memory mem with size = 4G;

/* region definitions */
define region ROM       = mem:[from 0x00200000 to 0x002FFFFF];
define region SRAM      = mem:[from 0x20000000 to 0x2004FFFF];
define region RAM       = mem:[from 0xC0000000 to 0xC1FFFFFF];

/* empty block definitions */
define block CSTACK     with size = 0x200,    alignment = 8 { }; /* SS */
define block EMAC_BUF   with size = 0x7000,   alignment = 8 { }; /* EMAC buffer */
define block EVT        with size = 0x1C8,    alignment = 64 { }; /*<1>*/
define block HEAP       with size = 0,        alignment = 8 { };
```

```
define block LCD_BUF    with size = 0x200000, alignment = 8 { }; /* LCD buffer */

/* fast RAM block to order fast blocks <2> */
define block FRAM       with fixed order {block CSTACK, block EVT, section .smx_sdar};

/* MPU protected block definitions <3> */
define block ucom_code  with size = 1024, alignment = 1024 {ro section .ucom_code};
define block ut1a_code  with size = 128,  alignment = 128 {ro section .ut1a_code};
define block ucom_data  with size = 64,   alignment = 64  {rw section .ucom_data};
define block ut1a_data  with size = 64,   alignment = 64  {rw section .ut1a_data};

/* operations */
keep              {block EMAC_BUF, block EVT, block LCD_BUF};
initialize by copy {rw};
do not initialize  {section .noinit, section .smx_adar, section .smx_sdar};

/* placements */
place at start of ROM   {ro section .intvec};  /*<4>*/
place in ROM            {block ucom_code, block ut1a_code, ro};
place at start of SRAM  {block FRAM};
place in SRAM           {block ucom_data, block ut1a_data, rw, block HEAP,
                         section .smx_adar, block EMAC_BUF};
place at start of RAM   {block LCD_BUF}; /*<5>*/

/* Notes:
  1. size = 4*(16 + SB_IRQ_MAX + 1). Must be on a 64-byte boundary.
  2. Assures CSTACK is first to detect system stack overflows. Also places
     critical blocks in TCRAM for speed.
  3. Alignment must = size.
  4. Copied to EVT by startup code in case EVT is needed before the linker copy code runs.
  5. Putting at start allows smaller MPU RAM region.
*/
```