Chapter 7 Partition Demos

From SecureSMX User's Guide v5.2, February 2024
By Ralph Moore and David Moore

This chapter presents a series of demos that demonstrate how to create an isolated partition in pmode and then move it to umode. It is intended to provide a quick introduction to SecureSMX and how to use it.

7.1 Getting Started

The partition demos are in a single file at www.smxrtos.com/securesmx/demo. After you have downloaded and unzipped this file, you will observe five complete demos labeled pd0 thru pd4. Each of these can be made and run using the IAR EWARM tool suite. If you do not have EWARM, you can download a free evaluation copy from www.iar.com. The demos run on the STM32F746G-Discovery board, which is very low cost and widely available from online distributors.

The sections that follow contain instructions for stepping through each demo and observing how it works. This is the best way to learn. Each demo is derived from its preceding demo and then new code is added and changes are made. For example pd1 is derived from pd0. Hence, if you prefer, you can use a comparison tool such as Beyond Compare to see what is changed from one demo to the next.

smxAware provides insight into the demos. The smxAware files follow the demos in ssdemos. To install smxAware see the installation section of its user's guide.

Note: IAR EWARM v8.50.5 was used when writing the following sections, so if a newer version is used, addresses and sizes are likely to be a little different. We tested with v9.40.2 and verified it works (and addresses and sizes differ a little). Also, it is possible we may have made fixes or adjustments to the demos since this was written causing these to differ.

7.2 Creating an Isolated Umode Partition Demo

pd0 thru pd4 illustrate how to take a typical embedded system, identify a vulnerable partition, then move the partition from pmode to umode, where it is fully isolated.

pd0 is intended to represent a typical, unprotected, embedded system running in hmode and pmode. It contains three tasks: idle, mctask, and ffdemo. The mctask is intended to represent a mission-critical task, which must be changed very little. The ffdemo task uses FatFs, which is third party code and thus may be considered to be vulnerable. Our goal is to move ffdemo and FatFs into an isolated umode partition from which mctask is protected. This is done in a sequence of steps represented by pd1, pd2, etc.

Running the Demos

The demos write a simple file to SD card repeatedly. Use a card that is blank or has nothing important on it, and ensure it is inserted before starting the demo. In the IAR debugger, add passcnt and failcnt to the Live Watch window, and you should see passcnt increasing as it runs. If a terminal emulator is connected to the eval board, the top line tells what demo is running and the bottom line shows % Idle, % Work, % Overhead, and Seconds running. Note that % Work is quite high and overhead is very low. This line is presented so that you can see that the demo is running.

7.2.0 pd0

We recommend that you trace through pd0 just to see what is there and how it works. Starting at main(), certain startup code has already run. (A breakpoint can be put at __low_level_init() in startup.c and restart, if you wish to trace it to main()). When tracing main() and following code, note that SMX_CFG_MPU is off as are configuration constants dependent upon it (see xcfg.h). main() initializes a few things, then calls smx_Go(), which initializes smx. smx_Go() initializes the error manager, event buffer, LSR[1] queue, ready queue, timer queue, timeout array, several system LSRs, the idle task and a few other things. Then idle is started at PRI_MAX with ainit() as its main function. The system LSRs perform time functions, profiling, timeouts, and task self-delete.

ainit() does application initialization, including tick enable, (portals are not enabled because SMX_CFG_MPU is 0), profiling is enabled as is event monitoring. At this point the mctask and ffdemo tasks have been created and started. However, their priorities are less than PRI_MAX, so they do not run. Put breakpoints at ffdemo_main(), mctask_main(), and smx_IdleMain(). Then ainit restarts idle at PRI_MIN with smx_IdleMain() as its code, and the other tasks run.

All the tasks run in while loops. Since mctask has PRI_HI, it runs first. It just loops for a msec, then suspends itself for 2 ticks. This allows ffdemo to run. It performs file operations, then suspends itself for 25 ticks so idle can run. Idle performs a stack scan, a profile display, a heap manager function, and bumps another task at PRI_MIN, if any is there. Since

[1] An LSR is an smx object that is used to perform deferred interrupt processing. It runs after all ISRs complete and before any tasks resume. Hence it is immune to problems such as task priority inversion.

SMX_CFG_RTLIM is 0, runtime limiting is not performed. Power down is inhibited. Idle runs when the other tasks do not run.

The smxAware Event Timeline window provides a picture of the above tasks running. It can be very helpful during debugging to see exactly in what order tasks, LSRs, and ISRs are running. Let the system run for a few seconds, click the debugger pause button, then click on smxAware in the task bar. Click on Graph in the pulldown window to see the timeline window. Other smxAware windows present a great deal of useful information that may interest you. Clicking on Event shows the entire Event Buffer. Clicking on Memory Map shows how read/write memory is structured.

7.2.1 pd1

At this point we have three tasks: *idle*, *ffdemo*, and *mcon*. The first step is to turn on SMX_CFG_MPU[2]. This enables the MPU. When it is on, smx_TaskCreate() assigns the default memory protection array (MPA), *mpa_dflt*, to the task it creates (i.e. task->mpap = mpa_dflt). This is used for ffdemo and mcon. (idle is discussed later). So the next step is to define mpa_dflt.

Defining Region Blocks

From the map file (see "Unused Ranges, From" in it), we see that the memory sizes are as follows when using the linker command file from pd0 (sizes may vary slightly):

ROM	0x136eb
SRAM	0x6cac
DRAM	0x8000

(Do not try to run it yet until we change to pd1.icf below. Using pd0.icf was just to see these sizes in the map.)

The first step is to define *region blocks* for these. For v7M, a region block must have a size that is a power of two and it must be aligned on its size. The first step is to determine the size. Then we use subregion disables (by n/8 multiples) to make the region blocks fit as closely as possible to the required sizes. The results are:

ROM	0x136eb	<= 0x20000*5/8 = 0x14000
SRAM	0x6cac	<= 0x8000*7/8 = 0x7000
DRAM	0x8000	<= 0x8000

These values were used to create region blocks in the linker command file at pd1\APPM\IAR.AM\STM32\stm3264g_pd1.icf. **Now change the linker to use pd1.icf**. Right click on pd1 top node, Options, Linker, and change:

$PROJ_DIR$\stm32746g_pd0.icf to

$PROJ_DIR$\stm32746g_pd1.icf

[2] Be sure the configuration constants dependent upon SMX_CFG_MPU are off – see xcfg.h. We are not ready for them, yet.

Looking at pd1.icf, note the MPU region sizes. These must be powers of two, which is easy to do in hex. The rule for this is: There can be only one non-zero digit and it must be 1, 2, 4, or 8. Below this are the region block definitions. Note the size and alignment taken from the above calculations. Ignore (MPUPACKER) – it is a marker for our MpuPacker utility, discussed in section 7.11.1 Using MpuPacker. The rest of pd1.icf is the same as pd0.icf.

As development proceeds, region blocks will grow in size. The linker will inform you if the allocated size is exceeded. Then it is a simple matter to increase the allocated size by 1/8 or go to the next power of two and 5/8. This permits keeping region block sizes tight during development.

Default MPA

mpa_dflt is defined in pd1\BSP\ARM\STM32\mpa7.c. At the top, the region block names and sizes have been brought over from pd1.icf. Below this, mpa_dflt is defined. The macros making this possible, such as RGN and RA are defined in pd1\MPU\ARMM\mpatmplt.h. Each line in mpa_dflt defines RBAR, RASR, and a name for one region. The name is used only during debugging – it is very helpful, in smxAware, for example.

The first three regions of mpa_dflt are the memory regions defined in pd1.icf. The next three are IO regions. ffdemo requires all three of these. mcon requires just USART1. These IO memory mapped sections are 1K in size and 1K aligned, so there is no problem making them into v7M regions. However, the memory-mapped registers in USART1 and SDMMC1 are contained within the first 64 bytes and the registers in DMA2 within the first 128 bytes. So from Liu Table 11.7 we see 64 bytes corresponds to 5 and 128 to 7, both as used in mpa_dftl. It is not essential to tighten down regions, like this, but it does improve reliability vs. bugs and soft errors.

Idle Task

For idle, an MPA template, mpa_tmplt_init, has been defined in mpa7.c. It is similar to mpa_dflt, except that it has a large IO region. This is because idle first runs with ainit() as its main function and ainit() does many different IO accesses. In smx_Go(), following creation of idle, mp_MPACreate() is called to create a custom MPA for idle, using mpa_tmplt_init. After this, smx_Idle->mpap -> MPA for idle and smx_Idle->tp = mpa_tmplt_init.

MMFs

When the MPU is enabled, a task's MPA is loaded into the MPU whenever the task is dispatched. Hence the task is limited to the memory regions and their attributes in its MPA. For pd1 the regions are much smaller than the implemented memory sizes:

ROM	0x14000	vs. 0x100000
SRAM	0x7000	vs. 0x50000
DRAM	0x8000	vs. 0x2000000

This is useful because an access outside of used memory into implemented memory will not trigger a Bus Fault, but it will trigger an MMF. To see this, start pd1. You will get an MMF, which causes an immediate halt. To find the cause of the MMF, open the Call Stack window and click on the top entry. You will see that there is some code attempting to access location 0x20008100. Selecting MPU in the smxAware Text window you can see that this address is not in any MPU region, hence the MMF. Comment out the two lines of assembly code, and pd1 will run ok.

An MMF causes a system halt only when debugging. For a released system, it causes a branch to the smx Error Manager, which records it, then calls the smx_EMExitHook() callback function in smxmain.c. This is where code should be placed to delete the partition, report the MMF, and reboot the partition.

Task Stacks

Slot 7 is reserved for the task stack region. The main benefit of having a separate stack region is that stack overflow is caught immediately, causing an MMF. This protects whatever is "above" the stack such as a heap control block, another stack, or a global variable.

During debug it may be desirable for the system to continue running despite a stack overflow. This can be accomplished by adding a *pad* above the stack. smx will report a stack overflow when it scans the stack or when the current task stops running, but no MMF will occur unless the pad is exceeded. Below the stack is the *register save area*, and below that is optional *task local storage*. The latter may be helpful if you run out of MPU slots. See section 4.11.8 Task Local Storage.

smx supports two types of tasks: *normal* and *one-shot*. A normal task has a permanent stack, which may be pre-allocated or allocated from a heap by smx_TaskCreate(). In the first case, the stack block must be a region block. In the second case, eheap is able to find and allocate a region block from a heap. Either way smx_TaskCreate() creates the stack region and stores it in the task's TCB. Then mp_MPACreate(), which is called next (see smx_Idle in smx_Go), moves the region into MPA[7].

For a one-shot task the stack block is taken from the stack pool when the task is dispatched by the scheduler. The stack block must be a region block. The scheduler creates the stack region and loads it into MPA[7]. Then its MPA is loaded into the MPU, and the task is started. A one-shot task does not have an internal infinite loop like a normal task. When a one-shot task stops it releases its stack. Yet the one-shot task can be waiting at any smx object (e.g. semaphore, mutex, etc.) just like a normal task. As a consequence, many one shot tasks can share a single stack as long as they do not need to run concurrently. Since partitioning tends to increase the number of tasks in a system, one-shot tasks can help to limit memory growth.

Summary

At this point, all tasks are running under the MPU, and no application code changes have been made. Although not much has been done so far, there is already some benefit, namely: a latent bug or two might have been found and soft error protection has been improved.

7.2.2 pd2

In this step we put FatFs into a *pmode partition*. Since a partition must have at least one task, we will add ffdemo to the partition, for now.

Define Sections

The first step is to define regions for this new partition, which we will call *fs*. Regions are composed of *sections*. The C compiler puts everything into the well-known sections: *.text*, *.bss*,

.data, *.rodata*, and *.noinit*. There are two ways to create our own sections (which have pros and cons):

1. Compiler section switches.
2. Section pragmas.

Compiler section switches can be put into .xcc files. \CFG shows three .xcc files. (In the Open dialog box, change the filter to All Files (*.*) to see them, since .xcc is not a standard extension type.) These simply rename the sections created by the C compiler, such as:

```
--section .text=.sys_text
```

To apply these files to a group (folder), such as RTOS: in the project window, right click RTOS, Options, C/C++ Compiler, check “Override inherited settings”, Extra Options, check “Use command line options”, and enter:

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

which applies mpi_sys.xcc to all files under RTOS. The simplicity of the project window belies the complexity of the underlying project file. Unfortunately, when “Override inherited settings” is checked in a subgroup, all of the settings of groups that include the subgroup are copied into it. Then any changes made to a group must also be made to every overridden file or group under it, which is easy to forget. For this reason, the .xcc option should be used sparingly. Here it is used for RTOS and System, where it saves adding pragmas to a large number of modules, and for some STMicro HAL files, where it helps reduce the number of pragmas added to third party code, which is inconvenient when the code is revised by the third party. Note that overridden files are indicated by a checkmark in the gear column of the Workspace (project) window.

For other modules, it is preferrable to use section pragmas in the module. This avoids the above problem and it is necessary when not all functions or variables belong in the same regions. For example, in ffdemo.c, it is preferrable for ffdemo_init() and ffdemo_exit() to go into sys_code, since they are called during initialization and exit, respectively. Hence they do not belong in the fs partition. Another example is in sd_diskio_dma_rtos_bspv1.c, where the transfer completed callbacks and the trusted LSRs belong in sys_code. To do this,

```
#pragma default_function_attributes =
#pragma default_variable_attributes =
```

are placed ahead of them to end .fs_text and .fs_data from the start of the module.

However, section pragmas do not work for string literals, which is discussed in Eliminating MMFs below and section 4.5.4 String Literals.

Upgrade pd2.icf

The next step is to add new region blocks to the linker command file. Four new MPU region sizes have been added: fscsz, fsdsz, scsz, and sdsz.

Below them is a new block, *clib_code*. This is an *ordinary block* with alignment of 4. It is necessary in order to bring clib functions into sys_code. (clib_code is included in sys_code below.) Adding unknown code, such as FatFs, is likely to bring clib functions with it. The Module Summary in the map file is helpful to find the new modules holding these functions.

They are likely to appear in one of the lower groups. Some clib functions require variables, so *clib_data* is defined for these and it is included in sys_data.

Next come the new region blocks, *fs_code*, *fs_data*, *sys_code*, and *sys_data*. The first two are for the fs partition. Notice that each code block includes .xx_text and .xx_rodata sections, and each data block includes .xx_bss, .xx_data, and .xx_noinit sections. Although it's not necessary to specify ones that are not used, we strongly recommend always listing all of them to avoid wasting time debugging MMFs when the code changes and they become necessary later.

Next are sys_code and sys_data. These are included in all ptask templates. Their purpose is to allow direct access to system services and other services. For sys_code, note that .intvec is included first. This is necessary to enable the CPU to access the first two vectors in the vector table on startup – see \BSP\ARM\STM32\STM32F7xx\vectors.c. Next are two .sys code sections, then the clib_code block. For sys_data, CSTACK (the main stack) is included first so that overflowing it will trigger an MMF. Next are three .sys data sections and the clib_data block.

Now rom_block and ram_block are completely different than before: they include the region blocks defined above. ro adds all code not in the code region blocks, and rw adds all data not in the data region blocks. The sys blocks have been placed ahead of the fs blocks to minimize the gap between them. In the map, sys_code ends at about line 900. The *Block tail* is wasted space inside of the sys_code region block. It is 0x36ce (14,030) bytes. scsz = 0x20000 so a subregion (1/8) is 0x4000 bytes, which is larger than the tail. 0x8010392 + 0x36ce = 0x8014000, which is the starting address of fs_code, so it is not possible to use the space wasted between sys_code and fs_code region blocks, by disabling a subregion.

Memory Overflow

Memory sizes have grown as follows, due to organizing code and data into region blocks that have wastage at their ends and gaps between them due to the processor's alignment requirements:

ROM	0x136eb	to	0x18000	24%
SRAM	0x6cac	to	0xc000	143%
DRAM	0x8000	to	0x8000	0%

These are much larger increases than we will see in the end. There are many methods to improve memory efficiency, however it is too early to apply them now. If you are experiencing memory overflows at this stage, the best plan is to get a processor/board with more memory. If this is not feasible, the next best plan is leave out portions of code, as you work.

fs MPA template

mpa_tmplt_fs is shown in BSP\ARM\STM32\mpaf7.c. Note that it has been necessary to combine the USART1 and SDMMC1 IO regions in region 4, since there is no spare region. The first is located at 0x40011000 and the second is at 0x40012c000. The range to be covered is 0x2c00 - 0x1400 = 0x1800. The next larger power of 2 is 0x2000. The region must start at 0x40010000 and it will cover to 0x40012000, which is not enough, so region size 0x4000 must be chosen. It must start at 0x40010000 and will cover to 0x40014000, which is sufficient. This

covers from TIM1 to EXTI, which is twelve IO regions! In region 4, there are 6 region disables (N0, N1, N2, N3, N4, and N67) leaving windows at 0x1000 to 0x1800 and at 0x2800 to 0x3000. The first admits USART1 & 6; the second admits SDMMC1. This is acceptable, if UART6 is not used.

Also note that the SDMMC1 and DMA2 regions were removed from the default template, mpa_dflt, since they are now in the fs template.

fs MPA

In ffdemo_init(), ffdemo create is followed by:

```
mp_MPACreate(ffdemo, &mpa_tmplt_fs, 0xFF, 8);
```

This gets a block for the MPA from the main heap that is large enough for 8 slots, transfers the first 7 slots from mpa_tmplt_fs, and loads the stack region from the ffdemo TCB into slot 7. In AppDbg.map search for fs_code in the Placement Summary to see its size. Then scan down to the end of the section where you find <Block tail>. This shows how much spare space is left. Do the same for fs_data. The sizes for the fs regions are (decimal):

```
fs_code   0x3000 (12288)    spare  0x4d4 (1236)    10%
fs_data   0x2800 (10240)    spare  0x294 (660)      6%
```

sys_code and sys_data allow the fs partition to directly access system services and data. These are temporary and will be replaced when the fs partition is moved to umode. The IO regions and the stack region have been previously discussed. The Event Buffer, EVB, requires a separate region since it is in DRAM. If it were moved into SRAM, it could be combined with sys_data, freeing up an MPU slot so USART1 and SCMMC1 could be separated.

Eliminating MMFs

Despite having tightened down the ffdemo task regions, pd2 runs smoothly. This is because we have already fixed all of the MMFs that normally occur when regions are tightened. If you were working with your own project, you would need to do this yourself. So, here is how to do it:

When an MMF occurs, open the Call Stack window and click on the top function (ignore <Exception frame>). This shows where the MMF occurred. Put a breakpoint there and run to it from the start. It generally works better to trace for an MMF in the disassembly window – tracing in the C source code window can be misleading. If you have left a variable out of your regions, you will generally find code like this:

```
ldr     rx, =variable
ldr     ry, [rx]
```

The first instruction will execute, but the second will refuse to execute. This is the sign of an MMF. Compare the address in rx to the MPU regions in the smxAware Text window of smxAware. You should find that it is not in any of them. The disassembly or C window will give you the variable name. Find it in your code and move it into one of your data regions. This is generally done with a section pragma, such as:

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

This is for a non-initialized variable in the fs_data region. For an initialized variable use ".fs_data".

If you have left out a function you generally find code like this:

```
bl      function
```

To the right of this is the address of the function. Comparing to the MPU regions in smxAware, you should find that it is not in any of them. Find the function in your code and move it into one of your code regions with:

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

This is for the fs_code region.

Handles, as parameters in system service calls, tend to cause difficulty. For example, a semaphore is created in hmode, then a utask attempts to signal it and gets an MMF. The problem is that the compiler attempts to pass the handle, not its address, as the parameter. This results in an attempt to access an address outside of the MPU. To avoid this problem, it is necessary to create an *alias handle* in a region of the utask and copy the actual handle into it after creating the object in hmode. Then specify the alias handle as the parameter in the system service call, instead of the actual handle.

A big problem is *string literals* (e.g. “abc”). The compiler puts all literals into section .rodata no matter where they occur in the code. This can be perplexing – everything else works, except the string literals. Often the string literal is staring you in the face, but you fail to recognize it. The best way to get them into one of your sections, for example .fs_rodata, is to use a .xcc file, as discussed previously. For example, this has to be done for “SDQueue” in smx_PipeCreate(), in SD_initialize(). You would think that literal would be put into .fs_rodata, but it’s not! In this case, -f $PROJ_DIR$\..\..\..\CFG\mpi_fsd.xcc has been put into Extra Options for FatFs.

An alternative is to define an array for a string, such as in ffdemo.c:

```
#pragma default_variable_attributes = @ ".fs_rodata"
const char hdr[] = "This is STM32 working with FatFs";
```

Then in ffdemo_main():

```
strcpy((char*)wtext, hdr);
```

puts the string literal into the beginning of wtext.

For v8M, a region overlap causes an MMF when the overlapping area is accessed. This can be particularly puzzling during debug because you see that the object causing the MMF is in a region, so what’s the problem? The MPU window in smxAware, flags overlapping regions, so watch for this. Otherwise, you need to carefully compare MPU regions. Stacks and pmsgs are the primary cause of overlapping regions. If a stack comes from the main heap, do not create a separate stack region. This leaves MPU[7] available for another region. In this case, PSPLIM is used to detect stack overflows. For pmsgs, it is best to use an auxiliary slot in the current task’s MPA.

Sometimes, to find the cause of an MMF, it is necessary to trace in assembly. Tracing in C often gives misleading results. For example, in C, it may look like a function is out of range, whereas actually a parameter is the cause of the problem.

Summary

We now have FatFs and ffdemo running in an isolated partition with fairly tight regions. It is left as an exercise to the reader to do the same for mcon. However, since mcon is highly-trusted code, there is no reason to do this other than to improve reliability or possibly catch latent bugs. Idle is left as is because it must perform functions such as heap management and profiling, which require wide memory access. Also, in case of a system shutdown, aexit() runs under idle.

7.2.3 pd3

Before moving the fs partition to umode, we must get it to make system calls via the SVC Exception, because it will not be able to access system calls directly.

SVC Functions

MPU\ARMM\svctmplt.c contains shell functions for all services considered safe for use from umode. It does not contain shell functions for services that could disrupt system operation, such as smx_SysPowerDown(). In some cases a service may be allowed from umode, but is limited in what it can do. For example, smx_TaskCreate() can be used to create a umode child task, but not a pmode task. svctmpl.c cannot be used in the project file because the jump table in it brings in all smx and other services whether they are used or not. So svc.c is derived from svctmplt.c to include only services actually used, in this case, by fs partition. As can be seen, it is also considerably smaller. This is useful for IO services that may or may not be needed.

The ssndx enum in svc.c has 23 entries. The first, LIM, is the limit, the last, END, is the number of entries, excluding itself, and in between are symbols for the 21 services provided. Each symbol defines the n in the SVC N instruction. At the top of svc.c is the jump table, smx_sst[], used by the SVC Handler (SVCH) with n as its index. Here the service function names are listed. These must be in the same order as the ssndx enum. Note that the first entry is the limit = END = 22. This is used by SVCH() to determine if n is valid. If not SMXE_PRIV_VIOL is reported to the Error Manager, which takes control.

It is pretty easy to get ssndx and smx_sst[] out of step. When this happens the actual function activated will not be what you expected. It is fairly simple to find and fix this problem.

Following the jump table are the shell functions, which call SVC N via one of the sb_SVC macros, using the symbols defined in the ssndx enum. (The sb_SVC macros are defined in svc.h.) Each shell function has the same name as the service it represents, with a u added to the prefix, e.g. smxu_. The header file, xapiu.h, defines the shell functions, and then maps each service to a shell function using mapping macros.

All that is required to cause the fs partition to make service calls via SVCH() is to add

```
#include “xapiu.h”
```

after other includes in each module that calls a service. Since, xapiu.h uses mapping macros, all parameters in each service call must be specified. This means that default parameters must be added. For example:

```
smx_TaskStart(ffdemo);
```

must be changed to:

```
smx_TaskStart(ffdemo, 0);
```

Default parameter values are specified in xapi.h.

Never in hmode

SVC functions can be called from pmode or umode, but must not be called from hmode. Since ISRs and trusted LSRs run in hmode, this is an easy mistake to make. The problem is that the n parameter should be stored in the task stack. But in hmode, there is no task stack, only the main stack, so n is stored in the main stack. However, here it is not protected and the results can be pretty wild. Usually a SMXE_PRIV_VIOL will be reported because the n delivered to SVCH() is too large. At other times, the wrong service will be called, which might report some other error such as SMXE_INV_TCB. This can be very confusing until you realize what is wrong.

At the top of sd_diskio_dma_rtos_bspv1.c we have:

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

But starting at line 645 is ISR and LSR code. So, ahead of this put:

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

This reverses the effect of xapiu.h.

Summary

We now have the fs partition making system calls via the SVC exception.

7.2.4 pd4

Finally we are ready to move the fs partition uptown to umode!

ucom Regions

The first thing is to define the *ucom_code* and *ucom_data* regions. These regions are common to utasks and replace the sys_code and sys_data regions used by ptasks. In RTOS\MPU\svc.c, note that smx_sst[] is left in sys_code because it is used by SVCH(), which runs in hmode. Below this, the shell functions are put into .ucom_text, and xapiu.h is included for prototypes.

For the STM32F746 group containing the HAL files, we removed the project override on the C/C++ Extra Options tab that used mpi_sys.xcc to locate all of the files in sys_code and sys_data regions, and instead we added this override to some files and pragmas to others to locate them elsewhere. The SD driver files are put into the fs sections since they are used only by the file system, and other files and routines are put into ucom sections since they may be needed by any code. (Keep in mind that putting things in ucom goes counter to good security, because it is shared by multiple partitions. For higher security systems, an alternative is to duplicate the HAL routines and data needed by multiple partitions, giving them slightly different names, so each can

be located in only one partition.) The remaining HAL files do not have project overrides nor pragmas, so their code and data fall into the default sections (.text, .data, etc) which is fine because those routines are called only during startup, which runs in pmode.

Next, we have modified pd2.icf to produce pd4.icf (pd3 uses pd2.icf). In particular, uccsz and ucdsz have been defined, and below them ucom_code and ucom_data are defined. ucom_code includes ucom sections and clib_code. The *.ucom_reset* section is a special section defined in BSP\ARM\STM32\STM32F7xx\reset.c. It has the first two elements of the *intvec* (interrupt vector) table, which are needed for system startup. Note that ucom_code is included in sys_code, which is expected to be at the start of rom_block, and that is where the processor expects to find pointers to CSTACK and to __iar_program_start, when it first starts running. After initialization, the VTOR register points to the real intvec table (see vectors.c). Including the ucom sections in the sys sections allows ptasks to access the clib functions, SVC shells, and other common functions and data.

New MPA

Next, a new MPA is required for fs in umode. In \BSP\ARM\STM32\mpaf7.c under UMODE TEMPLATES is a new template *mpa_tmplt_ufs*. Note that ucom_code and ucom_data have replaced sys_code and sys_data, the EVB region has been removed, and nothing else has changed from mpa_tmplt_fs. EVB is accessed only by system services and thus its region is not needed here. Note: the smx_EVBLogUser() functions, used to log user functions, can be called in umode, but they are accessed via SVC shell functions.

In mp_MPACreate() in ffdemo_init(), mpa_tmplt_ufs has replaced mpa_tmplt_fs.

fs_heap

One more change is necessary because FatFs requires a heap. Since the main heap cannot be used from umode, we must create a new heap, *fs_heap*. At the top of smxmain.c, bins and variables are defined for the main heap, and space for the main heap, itself, is allocated. (It is necessary to allocate it here because its size is determined by SMX_CFG_HEAP_SPACE, defined in acfg.h.) Below this, fs_heap bins and variables are defined. Since this a small, low-activity heap, it is given only one bin. Its size is defined near the top of pd4.icf as 0x1000 bytes.

fs_heap is initialized after the main heap (heap0) in smx_HeapsInit(). This consists primarily of putting fs_heap in the fs_heap section, initializing three fields in the fs_hv structure, and then calling smx_HeapInit()[3] which is in xheap.c. This calls eh_Init() in \XBASE\eheap.c. eheap is an RTOS-agnostic heap, which is the basis for smx_Heap. eh_Init() creates the heap, loads the remaining fields in fs_hv. The fs_hv structure, *EHV*, is defined in eheap.h. Then eh_Init() assigns a heap number to fs_heap, which is fs_hn. Finally in XFMW\FatFs\option\syscall.c, ff_memalloc() calls:

```
smx_HeapMalloc(msize, 0, fs_hn);
```

[3] smx_HeapInit() is called by $Sub$$__call_ctors() in smxmain.c, which is called by __cmain() in the EWARM startup code, prior to its calling C++ initializers, which require a heap.

and ff_memfree() calls:

```
smx_HeapFree(mblock, fs_hn);
```

umode

The final step is in smx_TaskCreate() in ffdemo_init() to replace 0 with SMX_FL_UMODE as the flags parameter. The fs partition is now running in umode. (You can verify this by checking *umode* in ffdemo_main vs. *umode* in mcon_main().) As a consequence, mission critical code and system code are protected from the fs partition by the *pmode barrier*. What that means is that any code, including malware, running in the fs partition can access hmode and pmode only via the SVC exception, and that access is limited by the SVC shell functions that have been provided in svc.c.

Background Region (BR)

When BR is on, except in umode, the processor can access all implemented memory. BR on in umode has no effect. Put a breakpoint in ffdemo_main() and bring up the MPU Register window (it is near the bottom of the Group menu). You will see that MPU_CTRL = 5. This means that both BR and the MPU are on. This is true for all utasks. If an interrupt occurs, while in umode, the processor switches to hmode, and BR allows the ISR to access all implemented memory. The same is true for exceptions.

ptasks are different. BR is off in ptasks. For example, in mcon_main(), MPU_CTRL = 1, meaning that BR is off and MPU is on. ptasks rely on sys_code and sys_data to directly access the services they need. If an interrupt occurs, sys_code allows the ISR shell in vectors.c to run. (If not, it must be moved into sys_code.) Then smx_ISR_ENTER() saves the state of BR and turns it on; smx_ISR_EXIT() restores BR to its previous state if control returns to the point of interrupt. Otherwise, smx_PendSV_Handler() runs next, and BR remains on for LSRs that might be dispatched by it.

Reversion to MPU Off

You may be having difficulty finding a problem and you feel that the MPU or partitioning is interfering with your effort, or may be the cause. Or you may be making a major change and do not want to be interrupted with MMFs. Whatever the reason, reversion to MPU off is easy. First, set SMX_CFG_MPU to 0 in xcfg.h and xarmm_iar.inc. This automatically disables several other SecureSMX features. Next: Top node Options, Linker, and change pd4a.icf to pd0.icf. It is not necessary to change section pragmas in the code because the Linker will now ignore them, and all #include “xapiu.h” statements are disabled. Then, since SMX is provided in library form in these demos, exclude the library from the project and add the nompu version, which was built with SMX_CFG_MPU 0. We recommend that you give this a try to verify that pd4 runs normally with the MPU off. Remember to reverse these changes before continuing.

Where inherited settings have been overridden there will be a check mark in the gear column of the project window. It is recommended that you enable inherited settings in case the problem is due to changes not being made to all duplicated project sections. Doing this will eliminate duplicated sections in the project file. First save a copy of the .ewp file and restore it when done, to avoid having to redo the overrides.

Sizes

ROM and SRAM usage have continued to increase, as shown in the map file:

```
ROM     0x136eb  to  0x28000  106%
SRAM    0x6cac   to  0xc000   77%
DRAM    0x8000   to  0x8000   0%
```

This is using pd4a.icf, and the percentages are vs. pd0 sizes. Now is a good time to run MpuPacker to see if any improvement is possible. It is in the BIN directory, and it is documented in section 8.11.1 Using MpuPacker, but some information is presented here. Make sure it set for pd4. It generates two files: MpuPacker.txt and MpuPackerDiag.txt in APPM\IAR.AM\STM32. In the EWARM Open dialog, click "All files (*.*)" in the lower right corner to see these. Comparing the first file to **pd4a.icf**, we see that no improvement in ordering can be made.

The second file provides diagnostic information. For rom_block, there are no gaps, but there are 0x873b bytes free at the end. rom_block = 0x28000, so actual size used is 0x28000 – 0x873b = 0x1f8c5. The next smaller region block size (from 0x40000) is 0x20000, so reducing to that (and using 8/8 subregion multiple) would save 0x8000 bytes.

Looking at "Block Tails" in the Diag file, we see that the ucom_code tail can be reduced by changing the region size (i.e. opt = "R"). ucom_code is in sys_code, which is in rom_code. Somehow, ucom_code was way too big. It can be reduced to 0x2000*5/8. sys_code is a little too big and can be reduced to 0x10000*7/8. These can be determined by looking at their sizes in the map file and calculating the correct region size and multiple, but it is easier to let MpuPacker guide you. The R means to divide the region size by 2 at least once, and the S means to reduce the subregion size by 1/8 or more. This can be done iteratively. For example, if it says R, divide the region size by 2 and restore the multiplier to 8/8. Relink and run MpuPacker again, and if there is still an R there, do it again until the R is gone. If an S is there now, reduce the multiple by 1/8 and try again. Repeat if S remains. When no letter is indicated in the opt column, you are done.

By reducing ucom_code and sys_code, there is now more space in rom_block, so with region size 0x20000 and 8/8 multiple the map file shows 0x873b byte block tail. Subregion size is 0x4000, so this is more than 2 subregions, and we change 8/8 to 6/8, leaving 0x73b bytes in the block tail. Now:

```
ROM     0x136eb  to  0x18000  24%
```

This is a dramatic reduction. Now change the linker to use **pd4b.icf**. Looking at Block Tails in MpuPackerDiag.txt there are no tails larger than subregions. However, looking above in this file, we see rom_block now has a gap of 0x2000 and there is 0x73b free. The latter is less than rom_block subregion size = 0x20000/8 = 0x4000.

To work on the gap, the map file shows that fs_code ends at 0x8015000 and the last code ends at 0x80178c5, so there are 0x28c5 bytes not in a region block. Up to 0x2000 of this space can be formed into a *plug block* with 4-byte alignment and put into the gap, thus reducing rom block size by up to 0x2000. This plug block is called *pb1_code*. Now change the linker to use **pd4c.icf**. Notice it selects individual object modules to put into the plug block. (These must be files that do not contain pragmas to control section location or you will get link errors (see 8.11.4 Using Plug Blocks).) Looking at MpuPacker.txt, we see that pb1_code is located between sys_code and

fs_code, as expected. Looking at MpuPackerDiag, we see that the gap is gone. This is because, pb1sz = 0x2000 in pd4c.icf. Also, End Free space went from 0x73b to 0x2525, adding 0x1dea more bytes at the end of rom_block for it to grow. If End Free had been greater than the subregion size of 0x4000 for rom_block, it would have allowed reducing the multiplier by 1/8 reducing rom_block by 0x4000.

This significant reduction to 24% plus more space for code in rom_block illustrates what can be done. Also there are other techniques that can reduce memory waste even further, as discussed in Section 8.11 Reducing Memory Waste for v7M. However, it is clear that the above work is best left until the end of the project, unless packing is so poor that things won't fit in memory.

Looking at MpuPackerDiag.txt, we see that ucom_code, sys_code, and fs_code have "tails". Tails are unused memory at the ends of blocks. Having unused memory at the ends of regions is better than having all of it after all code and all data because it allows code and data to grow within regions, thus enabling partition-only updates.

Performance

The difference in average 4096-byte file write and read performances from pd0 to pd4 is less than the jitter from one measurement to the next. Therefore the performance of the SD card is the limiting factor. Measured performances are: 1.17 mbps write and 3.72 mbps read.

Summary

The fs partition is now running in umode. Hence, mission critical code and system code are safe from malware that may have infected the fs partition. The price for this enhanced security in memory is small and in performance is none.