Strong Security Using the Cortex-M MPU

by Ralph Moore

March 2017

Introduction

Embedded systems are being drawn more into the IoT and thus, security in the form of protection of critical system resources is becoming increasingly important. Most security mechanisms, for example, depend upon secret keys, and most embedded systems have mission-critical software, both of which must not be compromised. Effective protection can only be achieved via hardware means.

The Cortex-M Memory Protection Unit (MPU) is difficult to use, but it is the main means of hardware memory protection available for Cortex-M processors. These processors are in widespread use in small- to medium-size embedded systems. Hence, it behooves us to learn to use the Cortex-M MPU effectively in order to achieve the reliability, security, and safety that modern embedded systems require.

MPU Basics

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.
- **Privileged Thread Mode**: privileged tasks (*ptasks*) run in this mode. It can be entered only from handler mode.
- **Unprivileged Thread Mode**: unprivileged tasks (*utasks*) run in this mode. It can be entered from either of the above two modes.

In the discussions that follow, the first two modes are collectively called *pmode* and the third mode is called *umode*. Similarly, I refer to *pcode*, *ucode*, *pSSRs*, *uSSRs*, etc. These are not industry-standard terms, but rather are introduced here to simplify discussions.

Cortex-M0/1/3/4/7 MPUs have 8 slots. Each *active* slot defines a *memory region* with its own 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 are usually filled with 0's to disable them.

Two unfortunate aspects of the Cortex-M MPU are that memory region sizes must be powers of 2, ranging from 32 bytes to 4 GB, and memory regions must start on multiples of their sizes. These requirements undermine the utility of the MPU by making it difficult to use without wasting substantial memory. This and uncertainty about how to define MPU regions have, I think, been major impediments to better usage of MPUs in embedded systems.

How to define MPU regions is discussed in this paper. Where necessary for specificity, examples assume the SMX® RTOS and the IAR EWARM tool suite. However, the techniques presented are applicable to all RTOSs and tool suites with similar capabilities.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

Defining Sections

The process starts with defining *sections* in the code. In the following discussion, *.taskA_code* and *.taskA_data* are the code and data sections reserved for taskA. .taskA_code contains the main task function and any subroutines that are specific to it. .taskA_data contains static variables used by taskA, if any.

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

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

/* Place taskA functions here. */

#pragma default_function_attributes =
```

where .taskA_code is a name to identify the section. Several functions can be enclosed above. Also, the above structure can be repeated in other modules, and all taskA functions will be combined into a single .taskA_code section.

For data:

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

/* Place taskA 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 .taskA_data section containing all of the static variables specific to taskA.

Creating and Locating Linker Blocks

The linker plays a prominent role in assigning the sections defined in the C code to actual memory locations. For example, in the linker command file (.icf extension for ILINK) for the .taskA_code section:

```
define region ROM  = mem:[from 0x00200000 to 0x002FFFFF];
…
define block taskA_code with size = 1024, alignment = 1024 {ro section .taskA_code};
…
place in ROM {block taskA_code};
```

Note that alignment equals the size as required by the MPU. Also note that the section and block names differ by only a ".". This is not a requirement, but it is convenient.

Memory Protection Array (MPA) Template

Now, in a C file or header file, define:

```
#pragma section="taskA_code"
```

Redefining the linker block as a compiler section is an EWARM idiosyncrasy that I don't understand, but it works.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

Finally, we define the template for the MPA of taskA:

```
const MPA mpa_tmplt_taskA =
{
  /*0*/   {RA("ucom_data")  | V | S0, RW_DATA | RSI("ucom_data") | EN},
  /*1*/   {RA("ucom_code") | V | S1, UCODE      | RSI("ucom_code") | EN},
  /*2*/   {RA("taskA_data")  | V | S2, RW_DATA | RSI("taskA_data")  | EN},
  /*3*/   {RA("taskA_code") | V | S3, UCODE      | RSI("taskA_code")  | EN},
  /*4*/   {/* taskA_stack */     V | S4, RW_DATA}
};
```

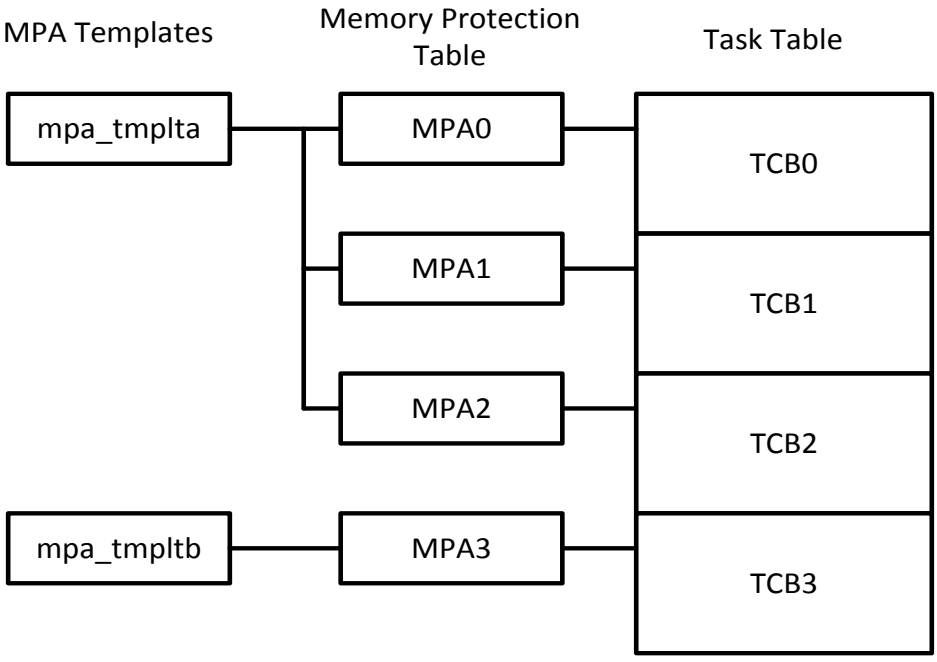
The macros used above are defined as follows:

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

In the above template two common regions have been introduced for subroutines and static data common to taskA and other tasks. Also, a region is reserved for the taskA stack.

Task Control

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



To the left of TT is the Memory Protection Table (MPT). This table has a Memory Protection Array (MPA) for each task. MPAs are in the same order as TCBs, and each TCB contains an index into MPT to access its MPA. A task's MPA is loaded into the MPU when the task is dispatched. Thus, each task has its own set of regions when it is running. The overhead on task switching time is about 25% for an MPA with 5 regions.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

The switching overhead applies to all task switches, whether the MPA changes or not and whether the task is a utask or a ptask. Normally each MPA has at least one dynamic region, the task stack. Other dynamic regions are planned for the future. Thus, loading the MPU on every task switch is necessary. Also, it simplifies MPU usage.

MPA Templates

To the left of MPT, in the above figure, two MPA templates are shown. Note that tmplta is shared between three MPAs and hence it is shared between three tasks. These tasks comprise a group of tasks that share code and data and are probably part of a subsystem, such as networking or file I/O. Such a task group is comparable to a process in a GPOS system. In a GPOS system processes typically use a Memory Management Unit (MMU) to provide isolation and protection. Here tasks use an MPU for the same purposes. Also shown, tmpltb is used by one MPA and hence by one task. This solitary task can be isolated and protected from all other tasks in the system.

A template is defined as follows:

```
const MPA mpa_tmplt_taskA =
{
  /*0*/   {RA("ucom_data")  | V | S0, RW_DATA | RSI("ucom_data") | EN},
  /*1*/   {RA("ucom_code") | V | S1, UCODE     | RSI("ucom_code") | EN},
  /*2*/   {RA("taskA_data")  | V | S2, RW_DATA | RSI("taskA_data")  | EN},
  /*3*/   {RA("taskA_code") | V | S3, UCODE     | RSI("taskA_code")  | EN},
  /*4*/   {/* taskA_stack */     V | S4, RW_DATA}
};
```

After taskA has been created, its MPA is loaded with the MPA template as follows:

```
taskA = smx_TaskCreate(taskA_main, 2, 0, 0, “taskA”);
smx_TaskSet(taskA, SMX_ST_MPA, mpa_tmpltb_taskA);
```

As previously noted, before taskA starts running, its MPA is loaded into the MPU. As a consequence, taskA can access only the regions shown above and only as permitted by the region attributes. If it tries to access an address outside of the above five regions, a Memory Manage Fault (MMF) will occur.

Region attributes are defined as follows:

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

where XN, RW, and RO are MPU attributes that mean *execute never*, *read/write*, and *read-only*, respectively. Hence taskA_data cannot be executed, and taskA_code cannot be written. Nor can taskA access system code or data, nor the code or data of other tasks, unless it shares a region with them, such as ucom_code. Any attempt to do so will cause a MMF.

An MMF provides the opportunity to take corrective action, such as stopping the task, rebooting the system, notifying the operator, notifying a remote system, etc. Thus, system takeover by malware can be averted. As a result, system code and mission-critical code are protected from taskA. This level of protection is as good as that achievable with an MMU.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

Task Stacks

MPA[4] is reserved for a protected task stack region, which must come from a stack pool so that it can be correctly sized and aligned, as required by the MPU. Stack pool stacks are allocated to tasks when they are dispatched by the scheduler. Following stack allocation, the MPA[4] region is automatically created and loaded by the scheduler. Task stacks are not only protected, but also, task stack overflows are detected and reported as MMFs as soon as they occur. This is helpful for debugging and also blocks a common malware attack method.

MPU Background Region

The MPU background region is like a flood – it breaks down all the barriers in pmode. Consequently, any ptask can access any other ptask's code and data, and even handler and ISR code and data. Also, there is no overflow detection. This is no good. For this reason, I have implemented background region switching. This consists of defining two system regions: sys_code and sys_data. These privileged regions are permanently present in MPU[7] and MPU[6], respectively. Thus, they are present for all tasks and have higher priority than task regions so that overlapping task regions cannot override them.

sys_code contains all handler and ISR code, and sys_data contains the Main Stack (MS), which is used by handlers and ISRs. The MPU_BR_ON() and MPU_BR_OFF() macros enclose every handler and ISR as follows:

```
SECTION `.sys_code`:CODE:NOROOT
THUMB
smx_SVC_Handler:
smx_MPU_BR_ON

; … SVC_Handler code

smx_MPU_BR_OFF
pop    {pc}
```

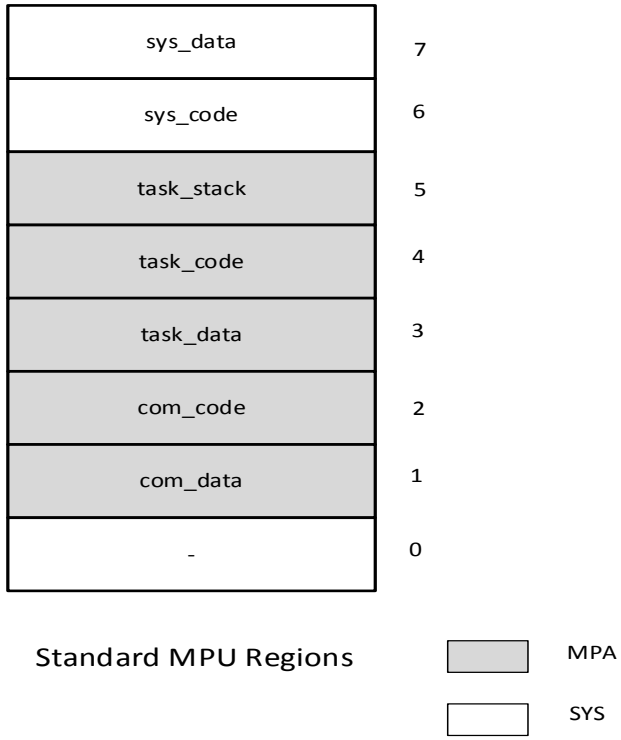
MPU_BR_ON() turns the background region on for handlers and ISRs. MPU_BR_OFF() turns it off if the RETTOBASE processor flag is 1 and mpu_br_off global flag is true. The former means that the handler or ISR is not nested, and the latter means that the task about to run does not use background mode. mpu_br_off is set when a task is dispatched, if its mpav flag is true. A task's mpav flag is set when its MPA is loaded. Each of the above macros is just a few lines of code and adds minimal overhead to a handler or ISR. If a handler or an ISR can be entirely contained within sys_code and sys_data, these macros can be omitted – i.e. background region is not turned on for them.

Now ptasks can be isolated from other ptasks, and handlers and ISRs can be isolated from ptasks. This improves system reliability and is an important stepping stone on the path to utasks.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

MPU Format

The following MPU format results from previous discussions:

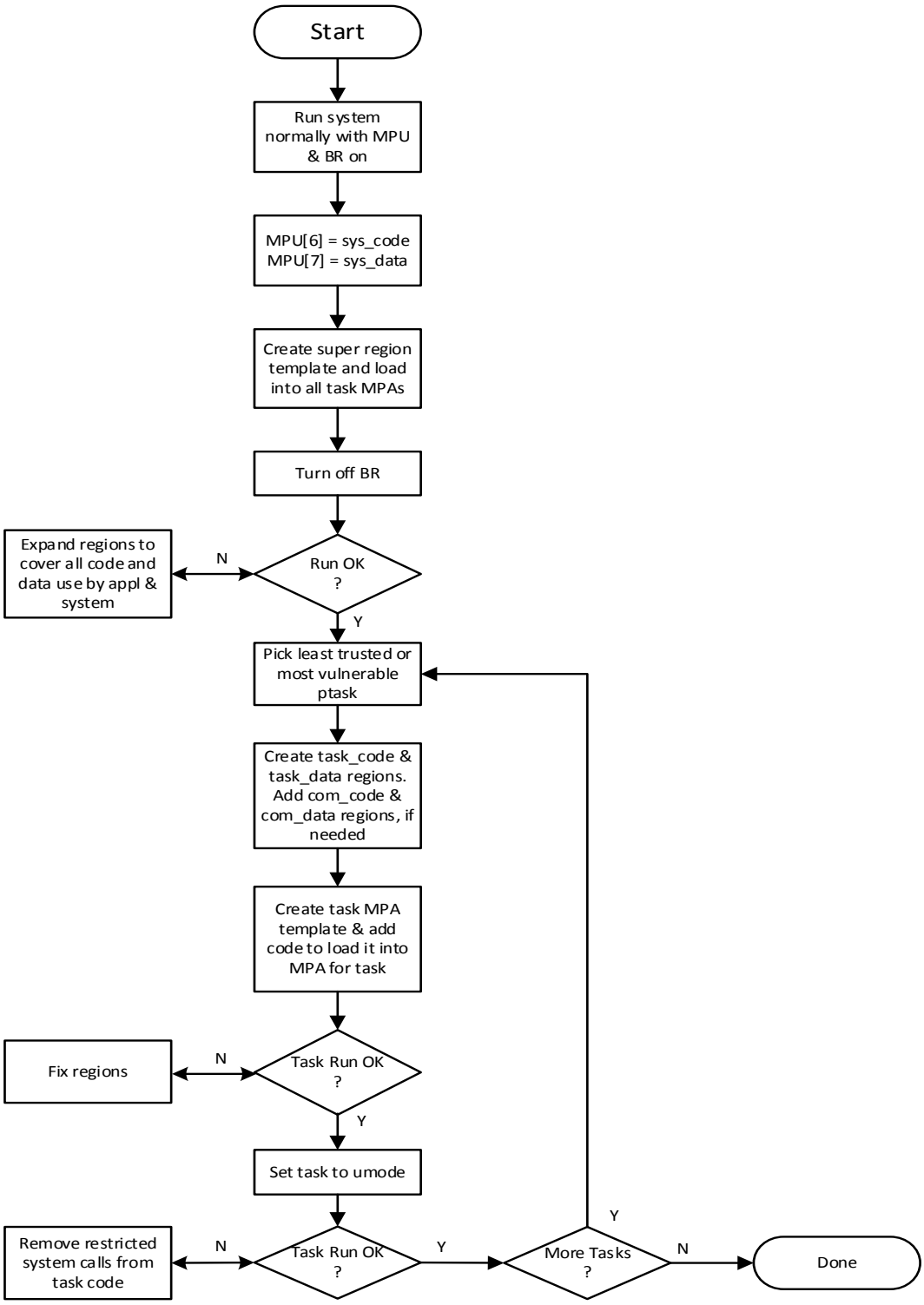


This seems to be a good format for the MPU. The top two regions contain the Main Stack and code for handlers and ISRs, respectively. These are privileged regions that are present for all tasks. The shaded regions correspond to Memory Protection Arrays (MPAs) that are loaded when tasks are dispatched. An MPA may be unique to a task or may be shared between a group of tasks. MPAs apply to both privileged tasks (ptasks) and unprivileged tasks (utasks). The bottom region (0) is available for a system region, such as an RO region for C libraries, tables, and text strings, or it could be added to the MPA if more regions are needed for tasks. In the event of a region overlap, the higher number region's attributes prevail.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

Step by Step Conversion

Here we present a step-by-step procedure to provide MPU security to late- and post-project systems. It, of course, can also be applied to new projects. The goal is to achieve the reliability, security, and safety that modern embedded systems require. The following flow chart provides an overview of the porting process:



Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

1. Start

To start, it is assumed that the RTOS library includes the MPU-Plus™ files. Add a call to sb_MPUInit() near the beginning of the startup code, and temporarily disable loading MPU[6] & [7] in it. This turns on the MPU and enables its Background Region (BR). The application should run normally with these changes.

2. System Regions

Next, define .sys_code and .sys_data sections. .sys_code should contain all handler and ISR code. This is done as in the following example for assembly code:

```
SECTION `.sys_code`:CODE:NOROOT
THUMB

smx_PendSV_Handler:
MPU_BR_ON       ; turn on MPU background region
… ; Handler code
MPU_BR_OFF      ; turn off MPU background region
cpsid   f
sb_INT_ENABLE
pop     {pc}
```

and for C code:

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

void sb_OS_ISR0(void)
{
   MPU_BR_ON();        /* turn on MPU background region */
   … /* ISR body or call ISR function here
   MPU_BR_OFF();       /* turn off MPU background region */
   sb_OS_ISR_EXIT();
}
#pragma default_function_attributes =
```

Then in the linker command file:

```
define block sys_code   with size = 4096, alignment = 4096 {ro section .sys_code};
define block sys_data   with size    = 512, alignment = 512  {block CSTACK};
```

Of course, the actual sizes depend upon the application. They must be the next power of two that is large enough. (If not large enough, the linker will complain.) The alignments must equal the sizes. Now enable loading sys_code into MPU[6] and sys_data into MPU[7] in sb_MPUInit().

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

3. Super Regions

The next step is to define super regions for the SRAM, ROM, and DRAM in the system. These regions serve as temporary replacements for BR. Consult the linker map to determine the starting address and how much memory is being used in each memory area. Then pick the next larger power of two for the region size. The following template is an example:

```
MPA const mpa_tmplt_app =
{
  /*0*/ {0x20000000 | V | S0, PRW_DATA | N7 | (0x10 << 1) | EN},    /* SRAM in use */
  /*1*/ {0x00200000 | V | S1, PCODE      | N7 | (0x11 << 1) | EN},    /* ROM in use */
  /*2*/ {0xC0000000 | V | S2, PRW_DATA      | (0x14 << 1) | EN},    /* RAM in use */
  /*3*/ {0x40011000 | V | S3, PIO            | (0x9  << 1) | EN},    /* USART1 */
  /*4*/ {       0        | V | S4, 0}                                  /* empty */
};
```

This template is loaded into the Memory Protection Array (MPA) for every task, so the tasks will run without BR. If a task gets a Memory Manage Fault (MMF), then it needs access to something outside of all regions. This can be fixed by enlarging regions, adding a new region in slot 4, or in the worst case, not loading a template into its MPA, thus leaving the task operating in BR. Such a task can be fixed later, possibly by dividing it into smaller tasks.

A significant gain has been made at this point: handlers and ISRs are running, as they were before, but all or most tasks are running in reduced memory regions with strictly controlled attributes (e.g. RO, XN, etc.) This is likely to reveal latent errors. In addition, significant spare memory, if present, has been protected from access by wild pointers and malware.

4. Task-Specific Regions

The next step is to identify the most untrusted or vulnerable task or group of tasks to isolate from the rest of the system. This might be a networking subsystem or third-party code. For simplicity, we will deal with a single task, taskA, here.

The first step is to group code and data into task-specific regions and to define blocks in the linker command file to hold these regions. The linker can pull together parts of regions from different modules so that code and data reorganization is not necessary, though perhaps desirable. It is convenient to name a task's regions after the task, e.g.: taskA_code and taskA_data.

Next, define common code and data regions to hold RTOS and other system services and to hold common data needed by them. These might be named pcom_code and pcom_data, respectively. At this point, taskA is a ptask, so pcom_code needs to include the actual code for the RTOS and other system services that is needed by taskA, and pcom_data needs to include data needed for these services.

Then, create mpu_tmplt_taskA and modify the code to load it into the MPA for taskA, instead of mpa_tmplt_app. taskA is now partially isolated from all other tasks. Will it run? This is where the tire meets the road. Memory Manage Faults (MMFs) from taskA are likely to occur due to references outside of its regions or due to attribute violations (e.g. writing to ROM.). These require good debugger or trace tools to find easily.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission

5. umode Operation

The final step, is to make taskA a utask. This is done by setting its umode flag. Now when it is dispatched, PendSV_Handler()[1] will set CONTROL = 0x3, which causes the processor to run in unprivileged thread mode using the task's stack. In addition, add

```
#include "xapiu.h"
```

ahead of the task's code. This forces the SWI API to be used for RTOS service calls and other system service calls rather than direct calls, as in pmode. At this point, it is probably desirable to pull all of taskA's code together into a single module since xapiu.h applies to all code that follows the point where it is included. (I draw a barbed wire barrier ahead of this point to remind me that the code above runs in pmode and the code after runs in umode.) Before actually running taskA, replace its pcom regions with ucom_code and ucom_data. The first contains the system service shells that implement SWI system services thus protecting them and their data from taskA.

When taskA first starts running as a utask, PRIVILEGE VIOLATION errors are likely, indicating that restricted service calls are being made. These are calls that should not be made from utasks, such as TaskStop(), PowerDown(), etc. This necessitates recoding to not use those services. One approach is to split taskA into a ptask, which directly calls these services (e.g. TaskCreate()) and a utask, which does not. Alternatively, taskA could start as a ptask, make all of the restricted service calls that it, then restart itself as a utask. (It must restart itself so that the PendSV_Handler() will change CONTROL to 0x3.)

Once you get taskA running as a utask, you have a task which cannot harm critical system resources. It can only access its own code, data, and stack, plus common code and common data shared with other utasks in its subsystem.

Conclusion

If all has gone well, untrusted code is running in utasks, trusted code is running in ptasks, and you and your boss can sleep well again. Critical parts of the system are strongly isolated from utasks. Though ptasks provide less security than utasks, they are convenient stepping stones to utasks, and they provide increased protection for software that must run in privileged mode.

It will probably take a fair bit of work to achieve the necessary changes. The important aspect of the above procedure is that it lays out a logical process for this work and after each step, the system can be tested and if it is not running properly, problems can be traced and fixed. You will not be confronted with an unmanageable number of problems all at once. Small steps will lead to wonderful outcomes. This procedure naturally leads to a succession of security releases, each making your system less vulnerable to hacking and dealing with vulnerabilities in order of importance.

An additional benefit of MPU conversion is a more reliable system due to finding latent bugs as the conversion proceeds and providing greater protection against environmental events such as energetic particles and voltage spikes.

For more information, see www.smxrtos.com/mpu.

Ralph Moore
ralph@smxrtos.com

[1] For Cortex-M processors the RTOS scheduler runs inside of the PendSV handler.

Copyright © 2016-2017 by Micro Digital, Inc. All rights reserved. The information on this page is proprietary to Micro Digital, Inc. and may not be copied, stored, or transmitted, without written permission