How Developers Can Effectively Leverage MPUs with an RTOS

Jean J. Labrosse, Software Architect, Silicon Labs

Memory Protection Units (MPUs) have been available to developers for years but many don't fully embrace them. They aren't that complex and are fairly easy to configure. An MPU can improve both the safety and security of an embedded application by limiting access to memory and peripheral devices to only code that should access those resources.

This whitepaper describes Memory Protection Unit (MPU) concepts and how developers can understand their benefits and how to use them effectively.

Table of contents:

- MPU basic concepts and using them with an RTOS.
- Using an MPU in an ARM Cortex-M.
- How to organize code by processes, how they communicate, and handling I/Os outside assigned memory.
- Recommendations when using the Armv7-M MPU.
- Summary
- References

The Basics of MPUs and Using Them with an RTOS

What is an RTOS?

An RTOS (a.k.a. a Real-Time Kernel) is software that manages the time of a Central Processing Unit (CPU) as efficiently as possible. Most RTOSs are written in C and require a small portion of code written in assembly language to adapt the RTOS to different CPU architectures.

When you design an application (your code) with an RTOS, you simply split the work into tasks, each responsible for a portion of the job. A task (also called a thread) is a simple program that thinks it has the CPU all to itself. Only one task can execute at any given time on a single CPU. Your application code also needs to assign a priority to each task based on the task importance and a stack (RAM) for each task. In fact, adding low-priority tasks will generally not affect the responsiveness of a system to higher-priority tasks. A task is also typically implemented as an infinite loop. The RTOS is responsible for the management of tasks. This is called multitasking. Multitasking is the process of scheduling and switching the CPU between several sequential tasks. Multitasking provides the illusion of having multiple CPUs and maximizes the use of the CPU, as shown in **Figure 1**. Multitasking also helps in the creation of modular applications. With an RTOS, application programs are easier to design and maintain.

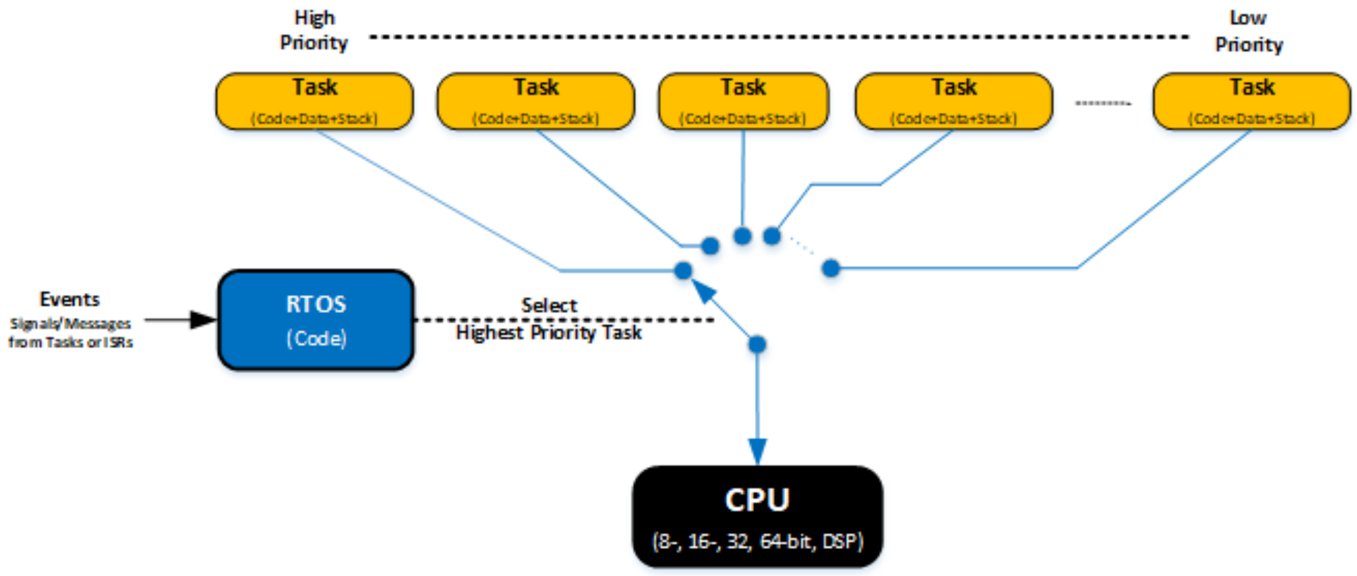


Figure 1. RTOS decides which task the CPU will execute based on events

Most commercial RTOSs are preemptive, which means that the RTOS always runs the most important task that is ready-to-run. Preemptive RTOSs are also event driven, which means that tasks are designed to wait for events to occur to execute. For example, a task can wait for a packet to be received on an Ethernet controller; another task can wait for a timer to expire, and yet another task can wait for a character to be received on a UART. When the event occurs, the task executes and performs its function, if it becomes the highest priority task. If the event that the task is waiting for does not occur, the RTOS runs other tasks. Waiting tasks consume zero CPU time. Signaling and waiting for events is accomplished through RTOS API calls. RTOSs allow you to avoid polling loops, which would be a poor use of the CPU's time. The example below shows how a typical task is implemented:

```
void  MyTask (void)
{
    while (1) {                         // Tasks are infinite loops.
        Wait for an event to occur;     // Task consumes no CPU time while waiting!
        Perform task operation;
    }
}                                       // A task doesn't return
```

The event that the task waits for can be triggered either from a task or a peripheral device interrupt handler through an RTOS API call. The API would typically run the RTOS scheduler, which would then decide to either switch to a new, more important task or simply resume the interrupted task (if the event was from an interrupt).

An RTOS provides many useful services to a programmer, such as multitasking, interrupt management, inter-task communication and signaling, resource management, time management, memory partition management and more.

An RTOS can be used in simple applications where there are only a handful of tasks, but it is a must-have tool in applications that require complex and time-consuming communication stacks, such as TCP/IP, USB (host and/or device), CAN, Bluetooth, Zigbee and more. An RTOS is also highly recommended whenever an application needs a file system to store and retrieve data as well as when a product is equipped with some sort of graphical display (black and white, grayscale or color). Finally, an RTOS provides an application with valuable services that make designing a system easier.

For performance reasons, most RTOSs are designed to run application code in privileged mode (a.k.a. Supervisor mode), thus allowing those applications full control of the CPU and its resources. This is illustrated in **Figure 2** where all tasks and ISRs have unrestricted access to memory and peripheral devices. Unfortunately, this implies that application code can corrupt the stacks or variables of other tasks whether accidently or purposely. In addition, allowing any task or ISR full access to all peripheral devices can have dire consequences.

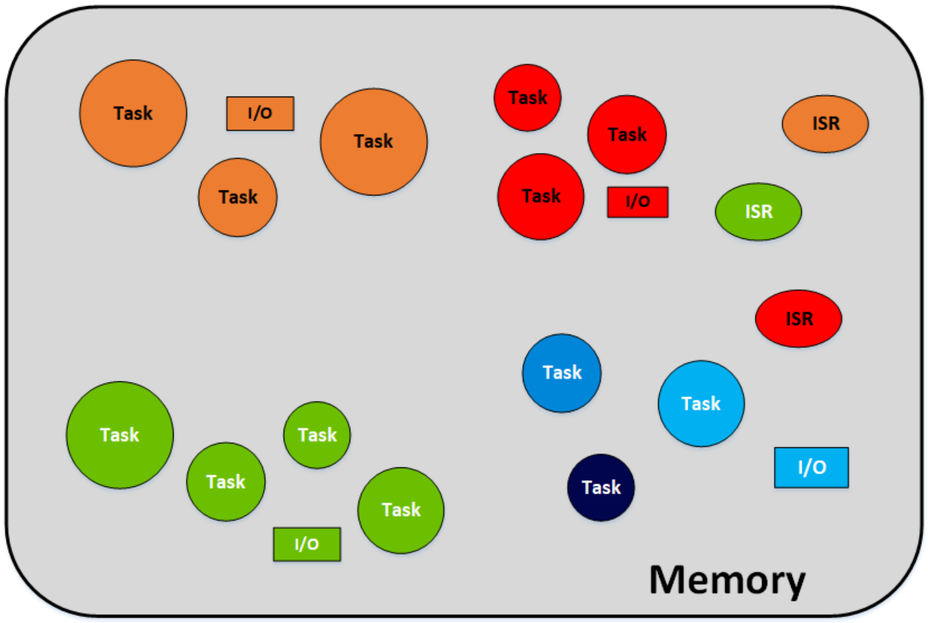


Figure 2. An RTOS and application code running with full privileges

What is an MPU?

A Memory Protection Unit (MPU) is hardware that limits access to memory and peripheral devices to only the code that need to access those resources. An MPU enhances both the stability and safety of embedded applications and is thus often used in safety critical applications such as medical devices, avionics, industrial control, nuclear power plants and more. MPUs are now finding their place in the Internet of Things (IoT) because limiting access to memory and peripherals can also improve product security. Specifically, crypto keys can be hidden from application code preventing access from an attacker. Isolating the flash memory controller with the MPU can also prevent an attacker from changing an application, thus only allowing trusted code to perform code updates.

With the help of an MPU, RTOS tasks are grouped into processes, as shown in **Figure 3**. Each process can consist of any number of tasks. Tasks within a process are allowed to access memory and peripherals that are allocated to that process. However, as far a task is concerned, it doesn't know that it's part of the same process except for the fact that it is given access to the same memory and I/Os as the other tasks within the process. When you add an MPU, very little has to change from a task's perspective since your tasks should be designed such that they don't interfere with each other unless they have to anyway.

Figure 3 shows that processes can communicate with one another through shared memory. In this case, the same region(s) would appear in the MPU configuration table for both processes. An application can also contain system level tasks as well as ISRs that have full privileges, thus allowing them access any memory location, peripheral devices and the CPU itself.

If a task attempts to access a memory location or a peripheral device outside of its sandbox, then a CPU exception is triggered, and the exception handler can terminate the task or all tasks belonging to the process.

Exactly what happens when such a violation occurs greatly depends on the application and to a certain extent which task causes the violation. For example, if the violation is caused by a graphical user interface (GUI), then terminating and restarting the GUI might be acceptable and might not affect the rest of the system. However, if the offending task is controlling an actuator, the exception handler might need to immediately stop the actuator movement before restarting the task. Ideally, access violations are caught and corrected during product development because, otherwise, the system designer will need to assess all possible outcomes and make decisions on what to do when this happens in the field. Recovering from an MPU violation can get quite complicated.

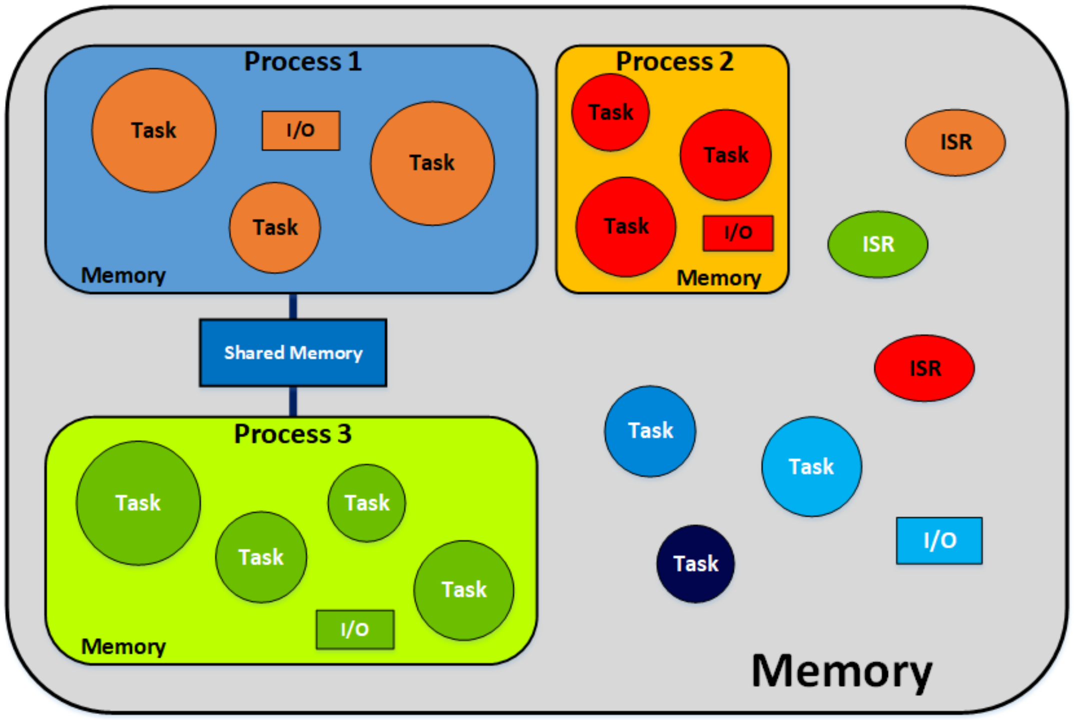


Figure 3. Separating an application into multiple processes

In RTOS-based applications, each task requires its own stack. Stack overflows are probably one of the most common issues facing developers of RTOS-based systems. Without hardware assistance, stack overflow detection is done by software and unfortunately rarely caught in time, which potentially makes the product unstable, at best. The MPU can help to protect against stack overflows, but unfortunately, it's not ideal.

The addressable addresses of a process are defined by a table (called the *process table*) that is loaded into the MPU when the RTOS switches-in a task. The table simply defines the memory (or I/O) ranges (called regions) that a task is allowed to access as well as attributes associated with those regions. Attributes for a region may specify if a task is allowed to read from or write to a region, or only be allowed to read, or execute code from the region (eXecute Never attribute, i.e. XN), etc. The eXecute Never attribute is highly useful as it can be used to prevent code from executing out of RAM, thus reducing the ability for hackers to perform code injection attacks. The number of entries in the process table depends on the MPU.

As shown in **Figure 4**, an MPU region can be used to detect stack overflows. In this case, a small region is used to overlay the bottom of each task stack. The MPU attributes are configured such that if any code attempts to write to that region, the MPU generates an exception. The size of the region determines how effective this technique would be at catching a stack overflow. The larger the region, the more chances you'd catch a stack overflow, but at the same time, the more RAM would be unavailable for your stack. In other words, the RedZone would be considered unusable memory because it's used to detect illegal writes. A good starting point for the RedZone size would be 32 bytes. If your task stack is 512 bytes then 32 bytes would only represent about 6 percent, leaving 480 bytes of usable stack space.

Because of the fairly limited number of regions available in an MPU, regions are generally setup to prevent access to data (in RAM) and not so much to prevent access to code (in flash). However, if your application doesn't make use of all the regions, security would also be improved by limiting access to code.

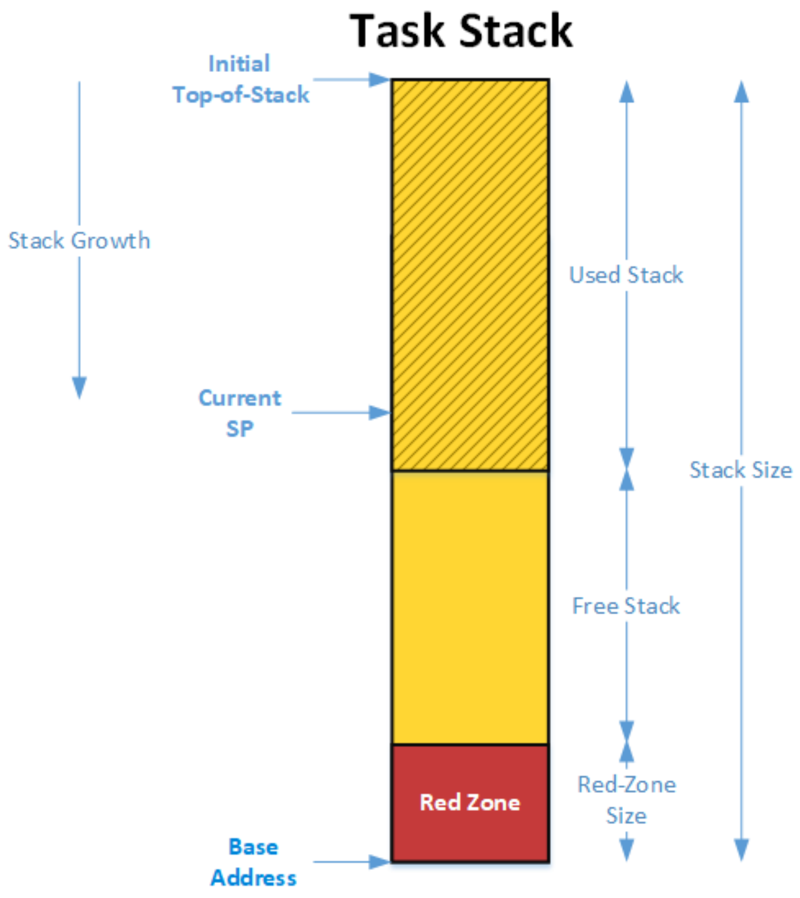


Figure 4. MPU region used to detect stack overflows

The process table is typically assigned to a task when the task is created. The RTOS simply keeps a pointer to the process table in the Task's Control Block (TCB). An RTOS context switch now includes additional code to update the MPU with the process table of the task being switched-in, as shown in **Figure 5**. You will notice that the MPU configuration doesn't need to be saved when a task is switched-out since the configuration for the task is always loaded from the process table which typically is placed in ROM (i.e. Flash).

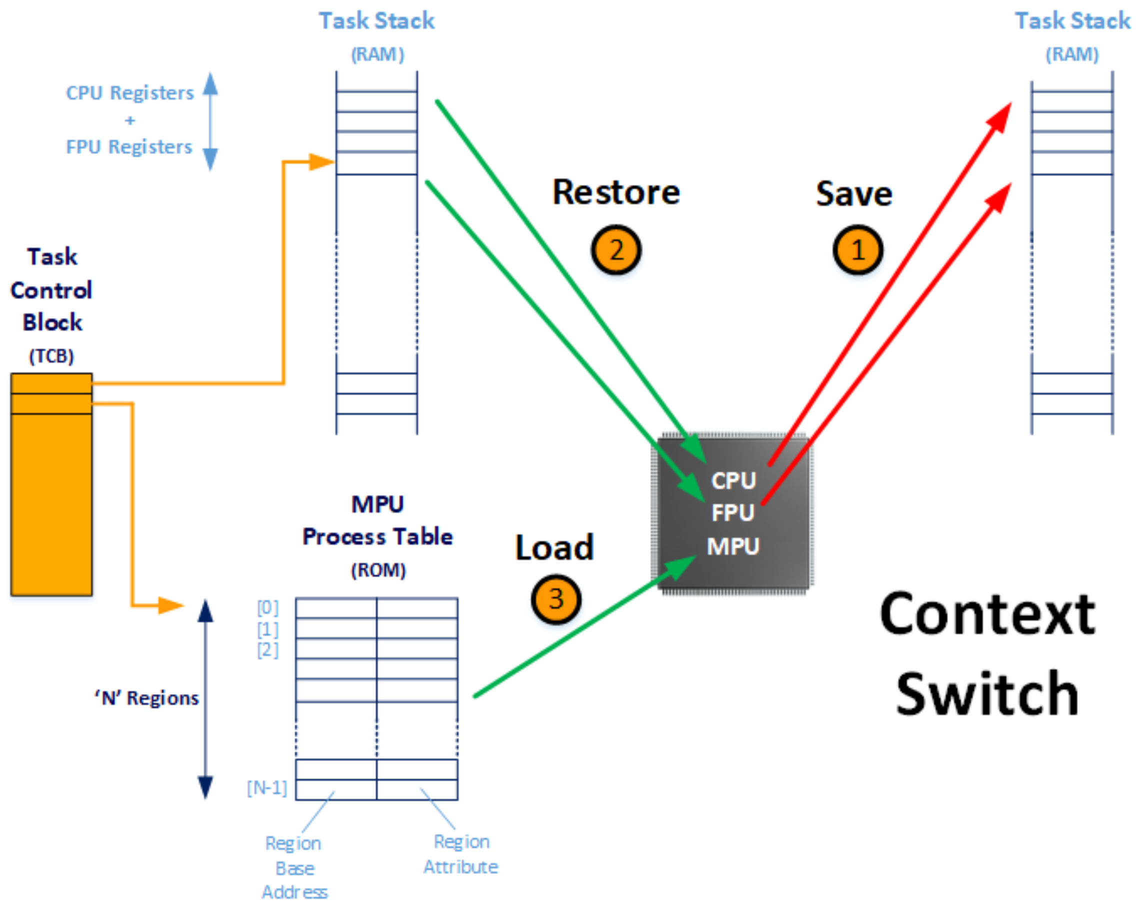


Figure 5. The MPU configuration is updated on a context switch

Section Summary

A Memory Protection Unit (MPU) is hardware that limits the access to memory and peripheral devices to only the code that needs to access those resources. Tasks are grouped into processes that are isolated from one another. If a task attempts to access a memory location or a peripheral device outside of its allotted space, then a CPU exception is triggered, and depending on the application, the offending task or the whole process can be terminated. The MPU can be used to detect stack overflows, but each task needs to give up a small portion of its stack to be used as a RedZone.

Each process is defined by a process table. A pointer to the process table is saved in the task's TCB when the task is created. This allows the RTOS to load the MPU with the task's process table when the task is switched-in. This operation obviously consumes extra CPU clock cycles, which adds to the context switch time.

Generically speaking, extending an RTOS to use an MPU seems to be quite straightforward; however, in practice, there are quite a few issues to consider which are addressed in the next section.

Using the MPU of an ARM Cortex-M

Introduced in 2004, the ARM Cortex-M architecture is currently the most popular 32-bit architecture on the market, adopted by most, if not all major MCU manufacturers. The Cortex-M was designed from the outset to be RTOS kernel friendly: dedicated RTOS tick timer, context switch handler, interrupt service routines written in C, tail-chaining, easy critical section management and more. Many Cortex-M MCU implementations are complemented with a floating-point unit (FPU), DSP extensions, highly versatile debug port and a memory protection unit (MPU). Let's look at how to use the Cortex-M MPU to improve the safety and security of embedded devices.

The ARM Cortex-M

In 2004, Arm introduced a new family of CPU cores called Cortex-M (M stands for Microcontroller) based on a Reduced Instruction Set Computer (RISC) architecture. The first Cortex-M was called the Cortex-M3, and the family has evolved to include a number of derivative cores: Cortex-M0/M0+, Cortex-M4, high-performance Cortex-M7, and the recently introduced Cortex-M23 and M33 with TrustZone security technology.

The programmer's model (see **Figure 6**) of the Cortex-M processor family is highly consistent. For example, R0 to R15, PSR, CONTROL and PRIMASK are available to all Cortex-M processors. Two special registers - FAULTMASK and BASEPRI - are available only on the Cortex-M3, Cortex-M4, Cortex-M7 and Cortex-M33, and the floating-point register bank and floating-point status and control register (FPSCR) is available on the Cortex-M4, Cortex-M7 and Cortex-M33 within the optional floating-point. Some Cortex-M implementations are also equipped with a Memory Protection Unit (MPU).

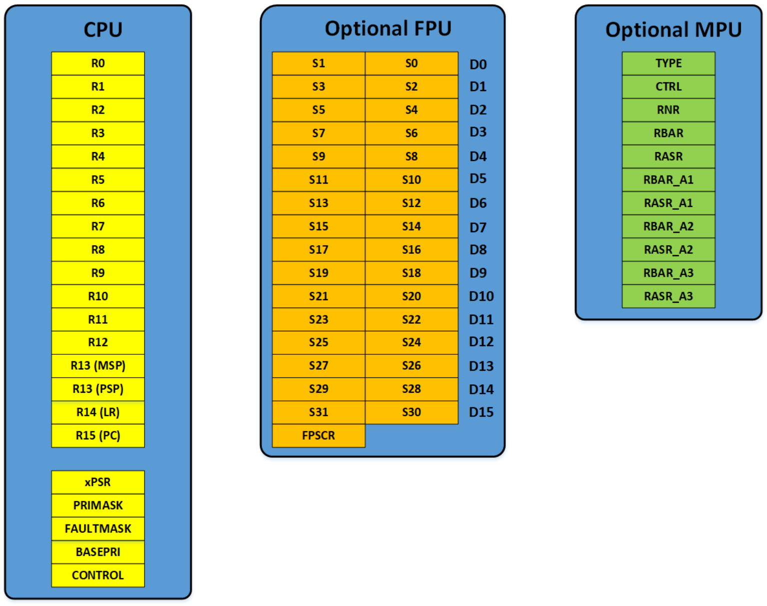


Figure 6. Armv7-M-based CPU register model

Both the CPU register and FPU registers (assuming the processor is equipped with one) are saved and restored by the RTOS during a context switch. Because the MPU configuration is obtained from a table, we only need to load the MPU registers when the task is switched-in. In other words, there is no need to save the MPU configuration for the task being switched-out. The details will be described in an upcoming section.

Cortex-M privilege levels

At power up, the Cortex-M starts in privileged mode, giving it access to all the features of the CPU. It can access any memory or I/O location, enable/disable interrupts, set up the nested vectored interrupt controller (NVIC), and configure the FPU and MPU, and so on.

To keep a system safe and secure, privileged mode code must be reserved for code that has been fully tested and is known to be trusted. Because of the thorough testing that most RTOSs undergo, RTOSs are generally considered trusted while most application code is not. There are few exceptions to this practice. ISRs, for example, are typically assumed to be trusted and thus also run in privileged mode, as long as those ISRs are not abused and kept as short as possible. This is a typical recommendation from most RTOS vendors.

Application code can be made to run on a Cortex-M in non-privileged mode, thus restricting what the code can do. Specifically, non-privileged mode prevents code from being able to disable interrupts, change the settings of the nested vectored interrupt controller (NVIC), change the mode back to privileged, and alter MPU settings as well as a few other things. This is a desirable feature because we don’t want untrusted code to give itself privileges and thus change the protection put in place by the system designer.

Since the CPU always starts up in privileged mode, tasks need to either be created from the get-go to run in non-privileged mode or switched to non-privileged (by calling an API) shortly after starting. Once in non-privileged mode, the CPU can only switch back to privileged mode when either servicing an interrupt or an exception.

SVC Handlers

Since non-privileged code cannot disable interrupts either through the CPU or the NVIC, application code is forced to use RTOS services to gain access to shared resource. Because RTOS services need to run in privileged mode (to disable interrupts during critical sections), non-privileged tasks must pass through a special mechanism on the Cortex-M called the SuperVisor *Call* (SVC) to switch back to privileged mode. The SVC behaves like an interrupt but is invoked by a CPU instruction. This is also known as a software interrupt.

On the Cortex-M, the `SVC` instruction uses an 8-bit argument to specify which of 256 possible RTOS functions (or services) the caller wants to execute. The system designer decides what RTOS services should be made available to non-privileged code. For example, you might not want to allow a non-privileged task to terminate another task (or itself). Also, none of these services would allow interrupts to be disabled as that would defeat one of the reasons to run code in non-privileged mode. Once invoked, the SVC instruction vectors to an exception handler called the SVC Handler.

This process is shown in **Figure 7**.

F7-1 Some non-privileged code executes SVC #5 to wait on a mutex.

F7-2 The SVC instruction forces the SVC exception handler to execute. The behavior is the same as if an interrupt was generated.

F7-3 The SVC handler extracts the argument (i.e., the value 5) and uses that to index into the SVC Jump Table.

F7-4 The desired RTOS service is executed (in privileged mode), and upon completion, the RTOS returns to the non-privileged code.

The SVC handler is part of the RTOS so you don’t have to worry about implementing that. In fact, your application code will invoke the same RTOS APIs regardless of whether your task runs in privileged or non-privileged mode.

Going through the SVC handler comes at a price: additional code and CPU cycles. On the Cortex-M3, the SVC handler adds about 1 Kbytes of code and executes between 75 and 125 CPU instructions to execute. So, any RTOS service invoked by non-privileged will require more processing time than if the same RTOS service was called from privileged mode.

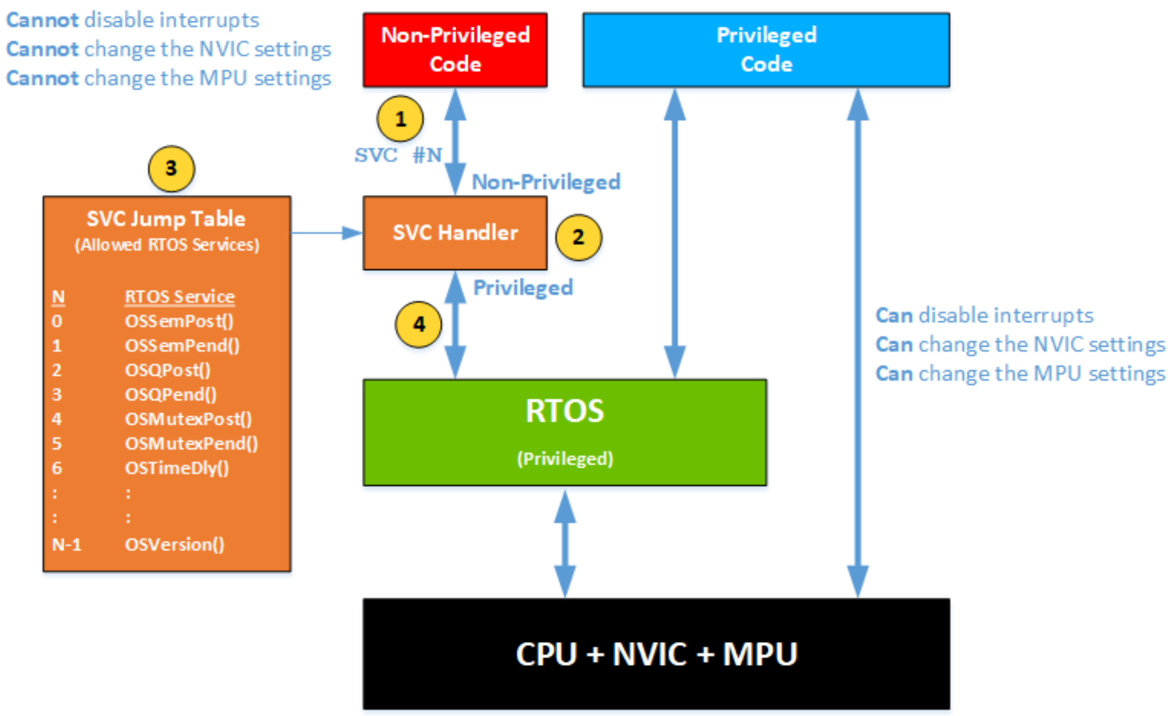


Figure 7. Limiting CPU, NVIC and MPU access from non-privileged code

Running code in non-privileged mode also prevents user code from disabling interrupts, thus reducing the chances of locking up the system. Of course, lockups are still possible if user code gets into an infinite loop, especially when that happens in a high-priority task or ISR. However, a lockup can be recovered in this case through the use of a watchdog.

Running in non-privileged mode still doesn't prevent application code from accessing any memory locations and peripheral devices or prevent code from executing out of RAM. This is where the MPU comes in.

The Cortex-M MPU in the Armv7-M architecture

The MPU on the Cortex-M (assuming Armv7-M) is a device that allows a process to have access to up to eight (8) or sixteen (16) memory or peripheral regions (depending on the MCU implementation). The location and size of each region is configurable. The size of each region must be a multiple of a power of two but cannot be smaller than 32 bytes. Also, the base address of a region must be aligned to an integer multiple value of the region size. So, if the region is 8K bytes, then the region must be aligned on an 8K boundary. Because of the relatively few regions available in the MPU, regions are typically used to limit access to RAM and peripherals and not so much code. However, at least one region must be used to provide access to code space.

A convenient way to organize the memory is to group the RAM needed for a process in one contiguous block as shown in **Figure 8**. Each of the processes would be set up in a similar fashion. An expanded view of Process A shows that it consists of four tasks, each with its own stack. Process A also manages a peripheral device. The white spaces represent unused memory or I/O space possibly due to alignment restrictions of the MPU.

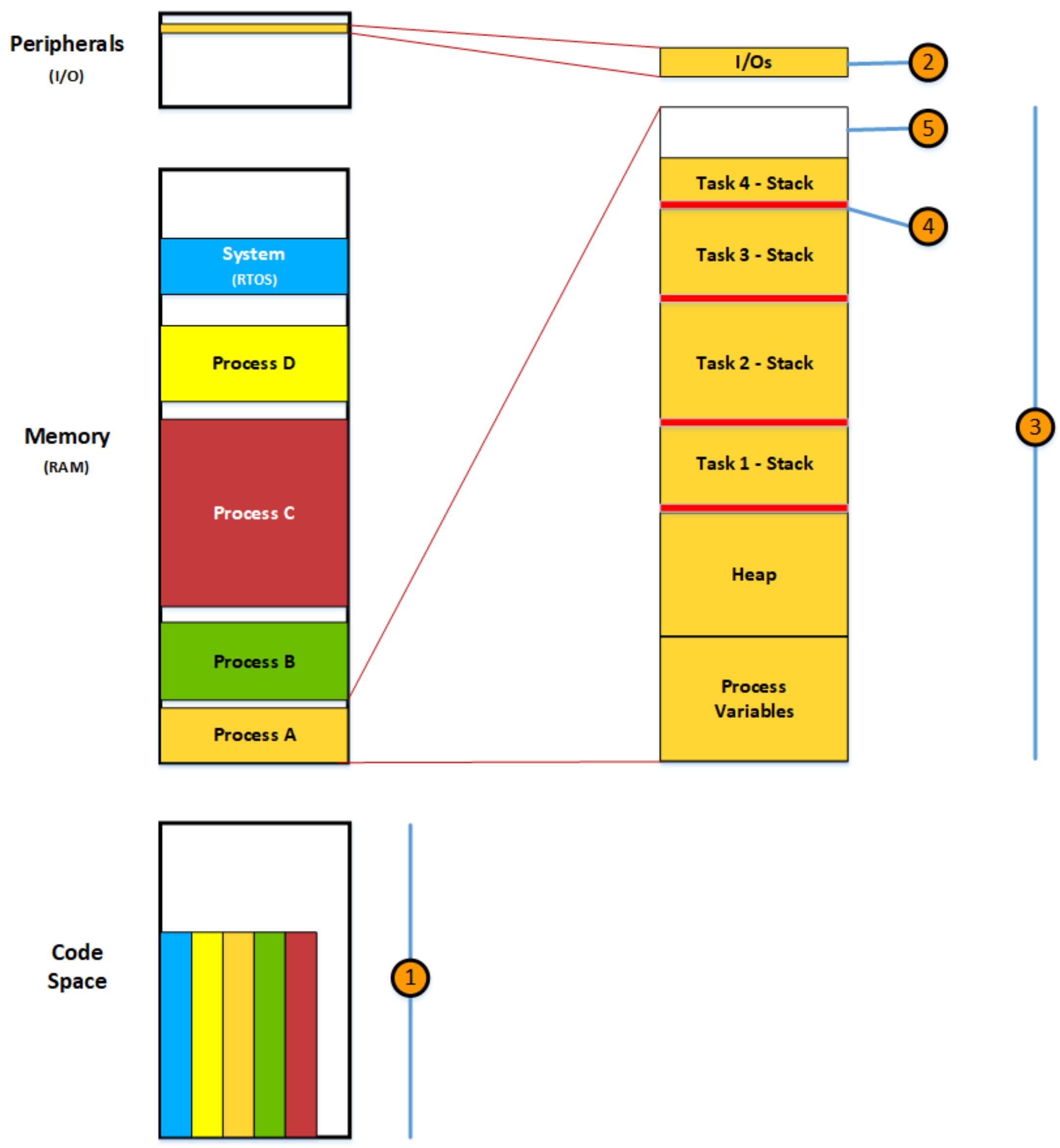


Figure 8. Grouping regions by process

F8-1 An MPU region is needed to provide access to code space. The region could be set up to only allow access to the code associated with the process but that can sometimes be problematic when a process shares code (i.e. libraries) with other processes so, each process is allowed to access the full code space.

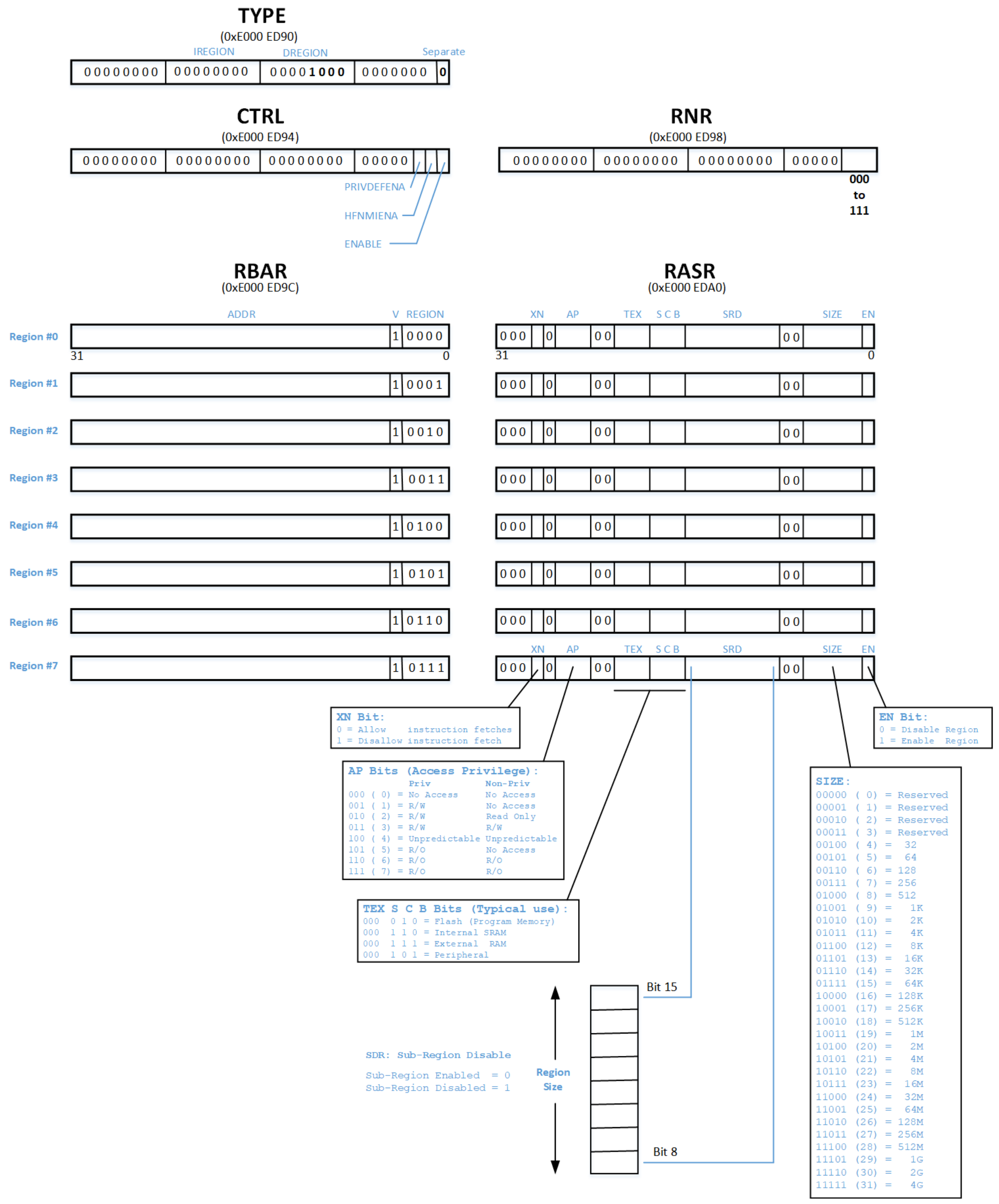
F8-1 An MPU region is needed to allow all tasks within the process to access peripheral devices assigned to the process. For example, if Process A manages an Ethernet controller, then the region must allow access to all the registers associated with this device.

F8-3 An MPU region is used to access all the RAM allocated to the process. It is assumed here that process global variables and a process heap are shared by all tasks within the process. As a side note, it’s not possible to use a global heap that can be used by all processes because you would not be able to set up an MPU table to separate the dynamically allocated memory of one process with that of another.

F8-4 An MPU region is used for RedZone stack checking. In fact, we only need a single region to cover all the task stacks in a process because we simply need to move the RedZone during a context switch. This, however, implies that each task will require a slightly different MPU process table. That being said, this greatly depends on how the RTOS manages the MPU during a context switch. For example, the RTOS might decide to only load the first seven regions from the MPU process table and load the last region with the base address of the stack to set the RedZone. Most of the time, the RTOS stores the base address of the task stack in the task’s control block (TCB). Using this scheme, all tasks within a process can share the exact same process table yet properly set the RedZone for the task’s stack.

F8-5 This represents unused RAM caused by the Cortex-M’s MPU requirement that the size of all regions must be a binary power of two. So, if Process A requires 7 Kbytes or RAM, then 1K would be lost due to the fact that Process A would need to be 8 K. Instead of letting that space go to waste, you might simply want to increase the size of certain stacks within the process to reduce the chance of getting stack overflows. However, the drawback to this is that if you ever need to add functionality to a process, then you might not remember how much memory you can reclaim. In fact, from a safety-critical point of view, if you qualify your system with a memory configuration, then you might not be able to reclaim it back. Thus, it’s probably best to allocate the stacks needed for the process and live with the wasted space.

From a programmer’s point-of-view, the Cortex-M MPU is a fairly simple device that consists of 19 32-bit registers as shown in **Figure 9**. You will note that this model differs from the one presented in **Figure 6** because some of the registers are actually banked and thus indirectly addressable, but internally, this is how they appear:



```
XN Bit:
0 = Allow    instruction fetches
1 = Disallow instruction fetch
```

```
EN Bit:
0 = Disable Region
1 = Enable  Region
```

```
AP Bits (Access Privilege):
               Priv           Non-Priv
000 ( 0) = No Access      No Access
001 ( 1) = R/W            No Access
010 ( 2) = R/W            Read Only
011 ( 3) = R/W            R/W
100 ( 4) = Unpredictable Unpredictable
101 ( 5) = R/O            No Access
110 ( 6) = R/O            R/O
111 ( 7) = R/O            R/O
```

```
TEX S C B Bits (Typical use):
000  0 1 0 = Flash (Program Memory)
000  1 1 0 = Internal SRAM
000  1 1 1 = External  RAM
000  1 0 1 = Peripheral
```

```
SDR: Sub-Region Disable

Sub-Region Enabled  = 0
Sub-Region Disabled = 1
```

```
SIZE:
00000 ( 0) = Reserved
00001 ( 1) = Reserved
00010 ( 2) = Reserved
00011 ( 3) = Reserved
00100 ( 4) =  32
00101 ( 5) =  64
00110 ( 6) = 128
00111 ( 7) = 256
01000 ( 8) = 512
01001 ( 9) =   1K
01010 (10) =   2K
01011 (11) =   4K
01100 (12) =   8K
01101 (13) =  16K
01110 (14) =  32K
01111 (15) =  64K
10000 (16) = 128K
10001 (17) = 256K
10010 (18) = 512K
10011 (19) =   1M
10100 (20) =   2M
10101 (21) =   4M
10110 (22) =   8M
10111 (23) =  16M
11000 (24) =  32M
11001 (25) =  64M
11010 (26) = 128M
11011 (27) = 256M
11100 (28) = 512M
11101 (29) =   1G
11110 (30) =   2G
11111 (31) =   4G
```

Figure 9. The Cortex-M MPU registers

The ***TYPE*** register is used to determine the number of MPU regions supported by the MPU, and the DREGION field of this register will always read as 0, 8 or 16. The ***CTRL*** register is used to configure some aspects of the MPU, but practically speaking, this register is used to enable or disable the MPU. In fact, the MPU should be disabled prior to changing the configuration of any or all of the regions. The ***RNR*** number allows you to address a specific MPU region.

Referring to **Figure 9**, you will notice that the lower five bits of the RBAR have a fixed value. When set to 1, the 'V bit' indicates that the lower 4-bits are used to specify the region number. The upper bits of RBAR are used to specify the base address of the region. The base address must be aligned on a boundary that matches the size of the region; e.g. a 1 Kbytes region must align on a 1 Kbytes boundary.

For the most part, setting up the attributes for a given region is fairly straightforward:

RASR.XN	It's highly recommended that you set this bit to 1 when the region covers RAM and you don't expect to execute code out of that region. This would catch code injection attacks from a hacker.
RASR.AP	If the region covers a RAM region, then you'd set the bits to '011', and if the region covers ROM, you'd set this field to '110'.
RASR.TEX S C B	**Figure 9** shows the typical value of these bits based on where the memory region resides.
RASR.SRD	This field allows you to subdivide a region into eight equal parts. This feature can greatly reduce wasted memory. For example, a 16 Kbytes region has eight 2 Kbytes sub-regions, so if a process only needs 5 Kbytes (3 sub-regions), then you can disable five of those sub-regions and assign them to a different process(es).
RASR.SIZE	This field is a bit more complicated to set because it requires some manual intervention and specifically looking at the linker map file to determine the encoded binary power of two size attribute.
RASR.EN	This bit enables (1) or disables (0) the region. If you don't need all eight regions, you must disable the region so that you don't inadvertently enable regions from a different process.

Listing 1 shows the assembly language code of an optimized function that loads all eight MPU regions. I show this as an example of how efficiently we can change the MPU configuration, but this is not something you have to worry about. It's really the responsibility of the RTOS to determine the best way to manage the MPU. However, you will need to follow the RTOS guidelines on how to set up the MPU process table for each task. For this particular implementation, you need to create an MPU process table that assigns all eight regions even if fewer are used. The prototype for the function is:

```
void OS_MPU_ProcessSet (ARM_MPU_Region_t *p_process);
```

p_process is a pointer to an MPU process table that contains eight pairs of RBAR and RASR values. `ARM_MPU_Region_t` is a data type defined by ARM's Cortex Microcontroller Software Interface Standard (CMSIS) (see reference [2]) and is declared as follows:

```
typedef  struct
{
    uint32_t  RBAR;   // Region base address
    uint32_t  RASR;   // Region attributes (type, region size, enable, etc.)
} ARM_MPU_Region_t;
```

So, for **each** task, you would need to declare an array of `ARM_MPU_Region_t` containing eight entries as follows:

```
const  ARM_MPU_Region_t  MyTask_MPU_Tbl[8] =
{
  //------------------- RBAR value -----------------   ---------------- RASR value -------------
   {.RBAR = ARM_MPU_RBAR(0UL, Region 0 base address),  .RASR = ARM_MPU_RASR(Region 0 attributes)},

   {.RBAR = ARM_MPU_RBAR(1UL, Region 1 base address),  .RASR = ARM_MPU_RASR(Region 1 attributes)},

   {.RBAR = ARM_MPU_RBAR(2UL, Region 2 base address),  .RASR = ARM_MPU_RASR(Region 2 attributes)},

   {.RBAR = ARM_MPU_RBAR(3UL, Region 3 base address),  .RASR = ARM_MPU_RASR(Region 3 attributes)},

   {.RBAR = ARM_MPU_RBAR(4UL, Region 4 base address),  .RASR = ARM_MPU_RASR(Region 4 attributes)},

   {.RBAR = ARM_MPU_RBAR(5UL, Region 5 base address),  .RASR = ARM_MPU_RASR(Region 5 attributes)},

   {.RBAR = ARM_MPU_RBAR(6UL, Region 6 base address),  .RASR = ARM_MPU_RASR(Region 6 attributes)},

   {.RBAR = ARM_MPU_RBAR(7UL, &MyTask_Stk[0]),         .RASR = ARM_MPU_RASR(1,                 // XN
                                                                 ARM_MPU_AP_RO,     // AP
                                                                 0,                 // TEX
                                                                 1,                 // Share
                                                                 1,                 // Cache
                                                                 0,                 // Buffer
                                                                 0,                 // SRD
                                                                 ARM_MPU_REGION_SIZE_32B)}
};
```

Specifies the region number to be

The base address of the stack to add

Prevent execution from RAM,

Read Only,

Internal SRAM,

Note that the last entry contains the base address of the task's stack and also assumes that the RedZone size is 32 bytes.

Listing 1. Configuring all 8 MPU regions.

```
OS_MPU_ProcessSet:           ; R0 contains 'p_process'
    PUSH   {R4-R9}
;
    LDR    R1,=0xE000ED9C    ; MPU->RBAR

    DBM    0x0F              ; Make sure outstanding transfers are done
    MOVS   R2, #0            ; Disable the MPU
    STR    R2, [R1, #-8]
;
;                              Update the first 8 MPU regions
    LDMIA R0!, {R2-R9}       ; Read  8 words from the process table
    STMIA R1,  {R2-R9}       ; Write 8 words to   the MPU
    LDMIA R0!, {R2-R9}       ; Read  the next 8 words from the process table
    STMIA R1,  {R2-R9}       ; Write those    8 words to the MPU
;
;                              Write the next 8 MPU regions
    LDMIA R0!, {R2-R9}       ; Read  8 words from the process table
    STMIA R1,  {R2-R9}       ; Write 8 words to   the MPU
    LDMIA R0!, {R2-R9}       ; Read  the next 8 words from the process table
    STMIA R1,  {R2-R9}      ; Write those 8 words to the MPU
;
    DSB   0x0F              ; Memory barriers to ensure subsequent data + instruction
    ISB   0x0F              ; Transfer using updated MPU settings
;
    MOVS  R2, #5            ; Enable the MPU (assumes PRIVDEFENA is 1)
    STR   R2, [R1, #-8]

    POP   {R4-R9}
    BX    LR
;
    ALIGN 4
```

Section Summary

The MPU in the Cortex-M is a fairly simple device. The RTOS is responsible for configuring the MPU on every context switch. However, it's the application developer's responsibility to set up the MPU process table for the application. Tasks within a process can share the same MPU process table if the RTOS sets up the RedZone for each task.

There are still a few things to take care of to get an application running with an MPU. Specifically, how do you group the RAM by process? How does a process communicate with another process? What happens if a task accesses memory or a peripheral device outside its allocated memory space? Apart from task stacks, should kernel objects be allocated within the process memory space? We will address these questions in the next section.

How to organize code by processes, how processes communicate, and handling I/Os outside assigned memory.

The complexity of using an MPU has more to do with organizing the memory of an application than the mechanics of updating this highly useful device. In this section we'll look at how a process can communicate with another process, what happens when a process attempts to access a memory location or a peripheral device outside its allocated space, how you group RAM by process, and conclude with a list of recommendations when using the Cortex-M MPU.

Inter-Process Communication

Figure 10 shows different ways that processes can communicate with each other. These are just some of the possible scenarios, and in fact, an application can have a combination of the techniques described below. The Cortex-M also has special instructions to allow lock-free data structures, making shared access simple and efficient but assumes non-blocking.

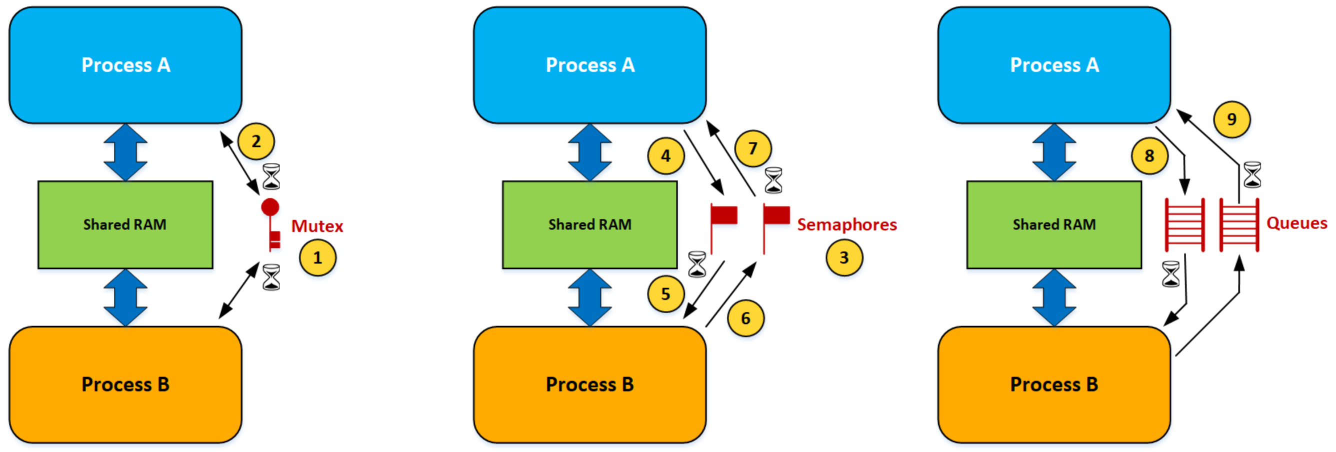


Figure 10. Inter-process communication

F10-1 A mutex is used to ensure that two processes do not access the same data at the same time. You should note that the mutex actually resides in RTOS memory space, and through RTOS APIs, the mutex is accessible to either process. Of course, there could be multiple mutexes, each providing access to different resources shared by two (or more) processes.

F10-2 Tasks needing access to a shared resource guarded by the mutex must first acquire the mutex. Once the task is done accessing the shared resource, the mutex is released. The hourglass represents an optional timeout in case a task is not willing to wait forever for the mutex to be released by its current owner.

F10-3 Semaphores can also be used by processes to signal each other about data availability.

F10-4 A task within Process A deposits data into an agreed-upon area in the shared RAM and then signals the semaphore on the left.

F10-5 A task within Process B waits for the signal from Process A through the semaphore. A signal indicates that data is available. Again, the hourglass represents an optional timeout to avoid waiting forever for a signal. If the signal doesn't occur within the timeout period, the task would be resumed by the RTOS. In this case, however, the task will know there hasn't been anything deposited in the shared area.

F10-6 Process B can acknowledge the fact that it processed the data (if a timeout didn't occur).

F10-7 After signaling the semaphore, Process A waits for an acknowledgement with an optional timeout.

F10-8 Alternatively, communication can use an RTOS's message queue mechanism. In this case, a buffer from dynamically allocated memory is obtained from the shared RAM area; the buffer needs to be accessible by both processes. The sender task in Process A fills the buffer and sends a pointer to a task in Process B.

F10-9 Similar to the semaphore case, the task in Process B can wait for a reply and specify an optional timeout.

Handling faults

As previously mentioned, the job of the MPU is to ensure that tasks within processes only access memory or peripheral devices that are assigned to them. But what if these tasks attempt to access data outside of those regions? The answer is that the MPU triggers a CPU exception called the Memory Manage (a.k.a. MemManage) Fault.

What happens when a fault is detected greatly depends on the application and is probably one of the more difficult things to determine. Needless to say, these types of faults should be detected and corrected during development. However, one of the reasons to use the MPU is to protect against those cases where an invalid memory or peripheral access occurs in the field, either because of some corner case that was not caught during system verification or through some unauthorized access.

The MemManage fault is generally handled by the RTOS. Ideally, your embedded system has some mechanism to record and report back faults to developers so corrections (if needed) can be included in the product's next release. A file system is a good place to record these faults, assuming of course that it can still be relied upon by the fault handler.

When a fault occurs, the fault handler could perform the following sequence of operations (shown in Listing 2 as pseudo-code):

Listing 2. Pseudo-Code of Possible Fault Handler

```
void  OS_MPU_FaultHandler (void)
{
    // Terminate the offending task/process                                 (1)
    // Release resources owned by the task/process                          (2)
    // Run a user provided 'callback' (based on the offending task)         (3)
    // If we have a file system:
                                                                            (4)
    //    Store information about the cause
    // Do we restart the task/process?                                      (5)
    //    Yes, Restart the task/process
    //       Alert a user
                                                                            (6)
    //    No,  Reset the system
                                                                            (7)
}
```

L2-1 The system designer needs to determine what to do when a fault occurs. At a minimum, the offending task must be terminated, but do we also need to terminate the other tasks in the process? There might not be a single answer, and in fact, it could depend on which task caused the fault. As a result, the MPU fault handler should be designed to perform different operations based on the task or process that triggered it.

L2-2 The offending task (or process) being terminated might own resources (kernel objects, buffers, I/Os, etc.) that would need to be released to avoid affecting other tasks/processes. The RTOS is aware of some of these resources and could automatically release them.

L2-3 The task that caused the fault might be controlling actuators or other types of outputs that should be placed in a safe state to avoid harm to people or assets. A user-defined callback should be provided by the embedded system designers to take care of system specific actions. The callback is stored in the task's control block (TCB) during task creation. To improve system safety and security, tasks should be created only during startup while the CPU is in privileged mode, and tasks should never be deleted at run-time unless because of a fault.

Since the TCB resides in RTOS space, the callback would not be accessible from user code, thus preventing potentially unsafe and unsecure code from invoking the callback, either unintentionally or maliciously.

L2-4 If the embedded system has some form of data storage capability, you might want to log information about the fault: what was the offending task, the value of CPU registers, what action was taken, etc.

L2-5 Depending on the task that caused the fault, the task could simply be restarted, and the system can thus recover from this situation.

L2-6 If the system was able to recover and if the system contains a display, it might be useful to alert an operation. Also, if the system has network connectivity, then notifying the service department and preferable the development team could help to avoid the issue in future releases.

L2-7 If the system cannot recover then there might be no other choice than to reset the system.

The MPU process table can be altered to include a per-task callback that would be called from `OS_MPU_FaultHandler()`. Of course, if all the tasks need to perform the same operation upon a fault, then you can either not use this feature or have the callback for all the MPU process tables point to the same callback. I believe the latter option is the most flexible and would be my preferred choice as a system designer as it offers greater flexibility for future releases. That being said, you will probably need to consult your RTOS provider to determine if this feature is available.

```
typedef  struct
{
    ARM_MPU_Region_t    MPU_Tbl[8][2];       // RBAR and RASR entries
    void               (*FaultCallback)(..);  // NULL pointer if no callback
} OS_MPU_PROCESS_TBL;
```

Creating the MPU Process Tables

Probably the biggest difficulty when using an MPU is grouping memory by process and creating the MPU process table. This is partly because you need a more intimate understanding of your toolchain: compiler, assembler and linker/locator as well as the application.

Let's assume I'm using the IAR toolchain (i.e. EWARM), but the concepts are similar enough that you'll be able to adapt these for the tools you use. Unless otherwise directed, the linker will place data (i.e. RAM) in one of three sections illustrated in **Figure 11**.

- Uninitialized data
- Zero initialized data
- Initialized data

As the name implies, uninitialized data corresponds to variables that have not been given an initial value at compile time or are not declared static.

Zero initialized data corresponds to data that was declared static and gets initialized to zero at startup. The linker groups this as one contiguous block so that startup code can perform a block set (to 0).

Initialized data corresponds to data that has an initial value (e.g. int x = 10;). Again, the linker groups this data into a contiguous block but creates a parallel block in ROM that contains the initial value of each of the corresponding variables in RAM. At startup, the whole block is copied from ROM to RAM.

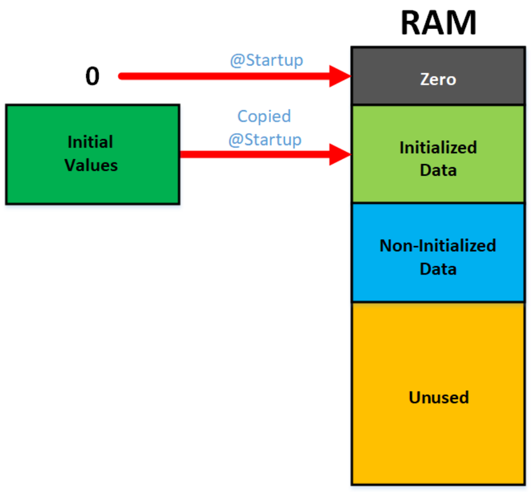


Figure 11. RAM sections

As previously discussed, RAM for a process must be grouped continuously as shown in **Figure 12**. To accomplish this, we need to bypass the compiler/linker standard sections and create new sections that we will be grouped by process. The toolchains are typically capable of creating multiple blocks of zero and initialized sections as shown in **Figure 12**.

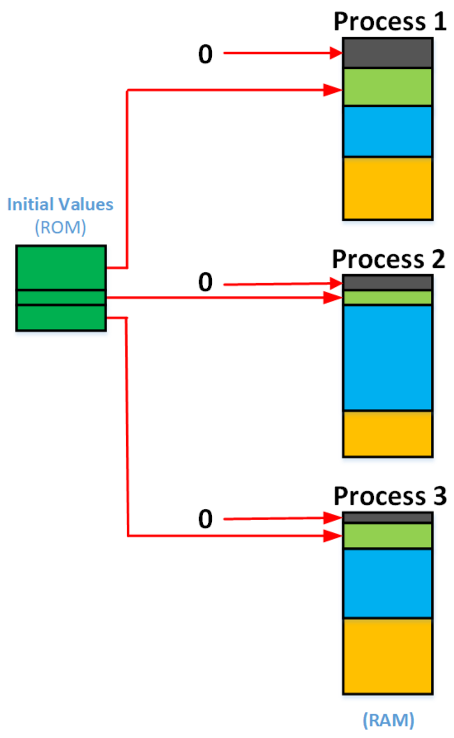


Fig 12. RAM sections for MPU-based application

Creating named RAM sections

To group data by process, we need to use the EWARM `#pragma directive default_variable_attributes` and wrap all the variables that are to be grouped together in a process.

```
#pragma default_variable_attributes = @”.Process1”

// All variables that we want to be part of the section named “.Process1”.

#pragma default_variable_attributes =
```

If your application contains variables declared in assembly language files, then you'll also need to make sure the assembly language file contains the appropriate assembler directives.

Grouping RAM by blocks

Your application will surely contain code that is not necessarily associated with any specific process. In this case, it would be best to create named sections for those modules and then combine the sections into a common block of code. You would then use the `#pragma` directive described above to create different named sections, one for each module, and use the linker's `block` directive as shown below to group these sections.

```
define  block  COMMON_RAM_BLOCK       with alignment =  4K, size =  4K
{
    section  .DRIVER_RAM,
    section  .COMMOM_RAM,
    section  .MATH_RAM,
    section  .STRING_RAM,
}


define  block  PROCESS_AI_RAM_BLOCK   with alignment = 16K, size = 16K
{
    section  .AI_DRIVER_RAM,          // Analog input driver
    section  .RTD_LIN_RAM,            // RTD          linearization
    section  .THERMOCOUPLE_LIN_RAM,   // Thermocouple linearization
    section  .UNIT_CONVERSION_RAM,    // Shared RAM with AO module
}


define  block  PROCESS_AO_RAM_BLOCK   with alignment =  8K, size =  8K
{
    section  .AO_DRIVER_RAM,          // Analog output driver
    section  .4_20MA_LIN_RAM,         // 4-20 mA  linearization
    section  .ACTUATOR_LIN_RAM,       // Actuator linearization
    section  .UNIT_CONVERSION_RAM,    // Shared RAM with AI module
}


define  block  SHARED_RAM_BLOCK       with alignment =  2K, size =  2K
{
}
```

You will note that the block directive allows you to specify the size and alignment of a memory block. It's important that both values are the same in order to place the start address of the block in the MPU process table. Also, the amount of RAM needed for each block depends on the application. I decided to use 16K, 8K, 4K and 2K bytes for the sake of illustration.

Locating RAM Blocks

We can now place all the blocks in the MCU's addressable space using two linker directives: `region` and `place`:

```
define  region  RAM = Mem:[from 0x20000000 size 64K];

place  in  RAM
{
    block  RAM_ALL with fixed order
    {
        block  PROCESS_AI_RAM_BLOCK,
        block  PROCESS_AO_RAM_BLOCK,
        block  COMMON_RAM_BLOCK,
        block  SHARED_RAM_BLOCK
    }
}
```

The `region` directive specifies the addressable memory of the MCU. There could be different region directives if your RAM is not all contiguous.

The `place in RAM` directive specifies to locate the blocks in the RAM area. You will notice that we needed to put blocks within a block to specify the order of block placement. In fact, to reduce the amount of wasted space, larger blocks should go first.

Creating the MPU Process Table for Each Task

Now that RAM is grouped by process, you can go back and edit the MPU table for each task/process. However, to do this, the compiler must know the names of the blocks so, you will need to use the `#pragma section` directive as follows:

```
#pragma  section = "COMMON_RAM_BLOCK"
#pragma  section = "PROCESS_AI_RAM_BLOCK"
#pragma  section = "PROCESS_AO_RAM_BLOCK"
#pragma  section = "SHARED_RAM_BLOCK"
```

The two process tables can now look as follows (assuming you are not using the version that contains the per-task callback as described in the previous section):

```
const  ARM_MPU_Region_t  MyProcess_AI_MPU_Tbl[8] =
{
   //------------------- RBAR value -----------------   ---------------- RASR value -------------
    {.RBAR = ARM_MPU_RBAR(0UL, __segment_begin("COMMON_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                  // XN
                          ARM_MPU_AP_RW,      // AP
                          0,                  // TEX
                          1,                  // Share
                          1,                  // Cache
                          0,                  // Buffer
                          0,                  // SRD
                          ARM_MPU_REGION_SIZE_4K)},

    {.RBAR = ARM_MPU_RBAR(1UL, __segment_begin("SHARED_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                  // XN
                          ARM_MPU_AP_RW,      // AP
                          0,                  // TEX
                          1,                  // Share
                          1,                  // Cache
                          0,                  // Buffer
                          0,                  // SRD
                          ARM_MPU_REGION_SIZE_2K)},

    {.RBAR = ARM_MPU_RBAR(2UL, __segment_begin("PROCESS_AI_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                  // XN
                          ARM_MPU_AP_RW,      // AP
                          0,                  // TEX
                          1,                  // Share
                          1,                  // Cache
                          0,                  // Buffer
                          0,                  // SRD
                          ARM_MPU_REGION_SIZE_16K)},

    {.RBAR = ARM_MPU_RBAR(3UL, 0),            // Unused regions, address = 0 and RASR = 0
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(4UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(5UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(6UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(7UL, &Process_AI_Task_Stk[0]),  // RedZone for the task
     .RASR = ARM_MPU_RASR(1,                              // XN
                          ARM_MPU_AP_RO,                  // AP
                          0,                              // TEX
                          1,                              // Share
                          1,                              // Cache
                          0,                              // Buffer
                          0,                              // SRD
                          ARM_MPU_REGION_SIZE_32B)}
};
```

```
const  ARM_MPU_Region_t  MyProcess_AO_MPU_Tbl[8] =
{
   //------------------- RBAR value -----------------   ---------------- RASR value -------------
    {.RBAR = ARM_MPU_RBAR(0UL, __segment_begin("COMMON_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                // XN
                          ARM_MPU_AP_RW,    // AP
                          0,                // TEX
                          1,                // Share
                          1,                // Cache
                          0,                // Buffer
                          0,                // SRD
                          ARM_MPU_REGION_SIZE_4K)},

    {.RBAR = ARM_MPU_RBAR(1UL, __segment_begin("SHARED_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                // XN
                          ARM_MPU_AP_RW,    // AP
                          0,                // TEX
                          1,                // Share
                          1,                // Cache
                          0,                // Buffer
                          0,                // SRD
                          ARM_MPU_REGION_SIZE_2K)},

    {.RBAR = ARM_MPU_RBAR(2UL, __segment_begin("PROCESS_AO_RAM_BLOCK"),
     .RASR = ARM_MPU_RASR(1,                // XN
                          ARM_MPU_AP_RW,    // AP
                          0,                // TEX
                          1,                // Share
                          1,                // Cache
                          0,                // Buffer
                          0,                // SRD
                          ARM_MPU_REGION_SIZE_8K)},

    {.RBAR = ARM_MPU_RBAR(3UL, 0),          // Unused regions, address = 0 and RASR = 0
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(4UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(5UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(6UL, 0),
     .RASR = 0},

    {.RBAR = ARM_MPU_RBAR(7UL, &Process_AO_Task_Stk[0]),  // Redzone for the task
     .RASR = ARM_MPU_RASR(1,                              // XN
                          ARM_MPU_AP_RO,                  // AP
                          0,                              // TEX
                          1,                              // Share
                          1,                              // Cache
                          0,                              // Buffer
                          0,                              // SRD
                          ARM_MPU_REGION_SIZE_32B)}
};
```

Recommendations when using the Armv7-M MPU

Run user code in non-privileged mode:
It's possible to use the MPU and yet still run all the application code in privileged mode. Of course, this means that application code would be able to change the MPU settings and would thus defeat one of the purposes of having the MPU. Initially running the application in privileged mode might allow easier migration of your application code. At some point, though, most of your application code will need to run in non-privileged mode, and you will thus need to add the SVC handler.

Set PRIVDEFENA to 1:
This allows privileged code to have access to the full memory map. Ideally, most of your application will run in non-privileged mode, and only ISRs and the RTOS will run in privileged mode. This recommendation avoids consuming three MPU regions for every task to give privileged code access to any RAM location, any code and any peripheral device. The decision of setting PRIVDEFENA to 1 might already have been made by the RTOS supplier and not something you can change.

ISRs have full access:
The processor switches to privileged mode whenever an interrupt is recognized and the ISR starts. Since PRIVDEFENA would be set to 1, ISRs have access to any memory of I/O location anyway. You simply don't want to reconfigure the MPU upon entering an ISR and reconfigure it back upon exit. So, ISRs should be considered system-level code and thus should indeed be allowed to have full access.

Also, ISRs should always be as short as possible and simply signal a task to perform most of the work needed by the interrupting device. Of course, this assumes that the ISR is kernel aware and the task has a fair amount of work dealing with the interrupting device. For example, processing an Ethernet packet should not be done at the ISR level. However, toggling an LED or updating the duty cycle of a pulse width modulation (PWM) timer might be done directly in the ISR.

Set the XN bit to 1:
The eXecute Never bit of the RASR register should be set for all RAM or peripheral regions if your application code is not expecting to execute code out of RAM. Setting the XN bit for peripheral devices may seem strange, but it doesn't hurt and protects against hackers who would look at ways to get into your system.

Limit peripheral device access to its process:
You should set aside one or more MPU regions to limit access of a process to only its own peripherals. In other words, if a process manages USB ports, then it should only have access to USB peripherals or peripherals related to the needs of the USB controllers such as DMA.

Limit RTOS APIs:
The system designer needs to determine which RTOS API should be available to application code. Specifically, do you want to prevent application code from creating and deleting tasks or other RTOS objects like semaphores, queues, etc. after system initialization? In other words, should RTOS objects only be created at system startup but not during run-time? If so, then the SVC handler lookup table should only contain the APIs you want to expose to the application. However, even if ISRs run in privileged mode and thus have access to any of the RTOS APIs, a good RTOS would prevent creating and deleting RTOS objects from ISRs anyway.

Allocate RTOS objects in RTOS space:
Task stacks are located within a process's memory space. However, RTOS objects (semaphores, queues, task control blocks, etc.) should preferably be allocated in kernel space and be accessed by reference. In other words, you don't want to allocate RTOS objects in a process's memory space because that would mean application code can, whether purposely or accidentally, modify these objects without passing through RTOS APIs.

No global heap:
It’s virtually impossible to set up an MPU to use a global heap (i.e., a heap used by all processes), so you should avoid those if at all possible. Instead, as previously suggested, you should allow process-specific heaps if a process requires dynamically allocated memory such as Ethernet frame buffers.

Don’t disable interrupts:
If your application runs in non-privileged mode, any attempt to disable interrupts will be ignored. The problem with this is that you will have no indication from the CPU that interrupts have *not* been disabled.

A bus fault will be triggered if your application runs in non-privileged mode and you attempt to disable interrupts through the NVIC.

Protect access to code:
Although MPU regions are generally used to provide or restrict access to RAM and peripheral devices, if you have spare regions and you are able to organize code (via linker commands) by processes then, it might be useful to limit code access to code. This prevents certain types of security attacks like *Return-to-libc* [2].

Reduce inter-process communications:
Just like tasks should be designed to be as independent as possible, processes should also follow the same rule. So, either processes don’t communicate with one another or you keep inter-process communication to a minimum.

If you have to communicate with other processes, simply set aside a shared region containing an out and an in buffer. The sender places its data in the out buffer and then triggers an interrupt to wake-up the receiving process. Once the data is processed, the response (if needed) can be placed in the in-buffer of the sender, and an interrupt can be used to notify the sender.

Determine what to do when you get an MPU fault:
Ideally, all MPU faults are detected and corrected during development. You should plan for faults to occur in the field either because of an unexpected failure or bug, or because your system was subjected to a security attack. In most cases, it’s recommended to have a controlled shutdown sequence for either each task or each process. Whether you restart the offending task, all tasks within a process or the whole system depends on the severity of the fault.

Have a way to log and report faults:
Ideally, you’d have a way to record (possibly to a file system) and display the cause of the fault to allow the developer(s) to fix the issue(s).

Summary

A Memory Protection Unit (MPU) is hardware that limits the access to memory and peripheral devices to only the code that needs to access those resources. If a task attempts to access a memory location or a peripheral device outside of its allotted space, then a CPU exception is triggered, and depending on the application, corrective actions must be taken.

The MPU found in a Cortex-M MCU is a fairly simple device and relatively easy to configure. However, the complexity in using the MPU is more oriented toward the allocation of storage (mostly RAM) by process and the creation of MPU process tables that will be loaded into the MPU during a context switch. I provided a list of recommendations that would make better use of an MPU in your application.

Software alone cannot prevent access to memory or peripheral devices not assigned to tasks in an RTOS environment. You need hardware to accomplish this, and the MPU is currently the only mechanism available on the Cortex-M (Armv7-M) that can do that.

Migrating an application to use the MPU is a fairly easy but tedious process. Adding an MPU will also impose overhead on your application: you have additional registers to load during a context switch, and user code should run in non-privileged mode to avoid having such code alter the MPU settings.

References

[1] Jean J. Labrosse, “Detecting Stack Overflows (Part 1 of 2)
https://www.micrium.com/detecting-stack-overflows-part-1-of-2/
March 8, 2016
Jean J. Labrosse, “Detecting Stack Overflows (Part 2 of 2)
https://www.micrium.com/detecting-stack-overflows-part-2-of-2/
March 14, 2016

[2] Wikipedia, “Return-to-libc attacks”
https://en.wikipedia.org/wiki/Return-to-libc_attack

[3] ARM, “MPU functions for the Armv7-M”
http://www.keil.com/pack/doc/CMSIS/Core/html/group__mpu__functions.html