Chapter 7

FreeRTOS-MPU

7.1 Chapter Introduction and Scope

The LPC17xx includes a Memory Protection Unit (MPU). This allows the entire memory map (including Flash, RAM, and peripherals) to be sub-divided into a number of regions, and access permissions to be assigned to each region, individually. A region is an address range consisting of a start address and a size.

FreeRTOS-MPU is a FreeRTOS Cortex-M3 port that includes integrated MPU support. It permits additional functionality and includes a slightly extended API, but is otherwise backward compatible with the standard Cortex-M3 port.

Using FreeRTOS-MPU will always:

- Protect the kernel from invalid execution by tasks.
- Protect the data used by the kernel from invalid access by tasks.
- Protect the configuration of Cortex-M3 core resources, such as the SysTick timer.
- Guarantee that all task stack overflows are detected as soon as they occur.

Also, at the application level, it is possible to ensure that tasks are isolated in their own memory space and that peripherals are protected from unintended modification.

FreeRTOS-MPU provides a simple interface to the MPU by hiding the register level MPU configuration from the user. However, writing an application for an environment that does not permit free access to all data can be challenging.

Scope

This chapter aims to give readers a good understanding of:

- The constraints the MPU hardware places on how memory regions can be defined.
- The access permissions that can be assigned to each memory region.
- The difference between User Mode tasks and Privileged Mode tasks.
- The FreeRTOS-MPU specific API.

7.2 Access Permissions

User Mode and Privileged Mode

The Cortex-M3 can execute code in either Privileged mode or User (unprivileged) mode. The standard FreeRTOS Cortex-M3 port executes all tasks in Privileged mode. FreeRTOS-MPU can execute tasks in either Privileged mode or User mode. The processor switches automatically to Privileged mode before executing an interrupt service routine. The kernel always switches to Privileged mode whenever a FreeRTOS-MPU API function is called, returning to its previous mode when the API function completes.

Tasks that execute in Privileged mode are not prevented from accessing any part of the Cortex-M3 core or from executing any of the Cortex-M3 instructions. MPU region access permissions can be used to prevent a Privileged mode task from making certain memory accesses—for example, writes to a region that is configured as read-only.

Tasks that execute in User mode are prevented from accessing certain Cortex-M3 resources and from executing certain Cortex-M3 instructions. For example, a User mode task cannot access the interrupt controller or execute CPS (Change Processor State) instructions[4]. MPU regions can be configured to prevent User mode access, while still permitting Privileged mode access.

Access Permission Attributes

Table 26 lists the access permission related definitions available in FreeRTOS-MPU. Examples of their use are provided later in this chapter.

[4] For complete details on User mode restrictions, refer to the ’ARM V7-M Architecture Application Level Reference Manual’, and the ‘Cortex-M3 Technical Reference Manual’, both of which are available directly from ARM.

Table 26. MPU region access permissions

FreeRTOS-MPU definition	Access for Privileged mode tasks	Access for User mode tasks
portMPU_REGION_READ_WRITE	Full Access	Full Access
portMPU_REGION_PRIVILEGED_READ_ONLY	Read Only	No Access
portMPU_REGION_READ_ONLY	Read Only	Read Only
portMPU_REGION_PRIVILEGED_READ_WRITE	Full Access	No Access
portMPU_REGION_EXECUTE_NEVER	Region cannot contain executable code.	

7.3 Defining an MPU Region

Overlapping Regions

A region is an address range to which access permissions can be applied. A maximum of eight regions can be defined at any one time. Regions are numbered from zero to seven.

If multiple regions define overlapping memory ranges, then the access permissions of the highest of the overlapping region numbers will be applied.[5] For example, if region two configures an address range for both read-and-write access at the same time as region three configures the same address range for read-only access, then the memory region will be configured for read-only access.

Predefined Regions and Task Definable Regions

Regions zero to four are used by the kernel to pre-configure a usable run time environment where:

- The Running state task has access to its own stack, but all other RAM is accessible only when the LPC17xx is running in Privileged mode.
- The area of Flash memory in which the kernel is located and the system peripherals are accessible only when the LPC17xx is running in Privileged mode.
- The Flash memory, other than that in which the kernel is located, and all non system peripherals (for example, UARTS and analog inputs) can be accessed by both Privileged and User mode tasks.

The kernel reconfigures the MPU during each context switch, so the remaining three regions can be defined differently by each task. The task-defined regions use the highest region numbers, so can be used to override the kernel-defined regions, although there are few circumstances in which that would be desirable.

[5] This applies to any range of memory that appears within more than one region definition—whether the two regions are completely coincident, or only partially overlapping.

Region Start Address and Size Constraints

The MPU hardware imposes two rules that region start address and size definitions must comply with:

1. The region size must be a binary power of two between 32 bytes and 64 gigabytes, inclusive. For example, 32 bytes, 64 bytes, 128 bytes, 256 bytes, and so on are all valid region sizes.

2. The start address must be a multiple of the region size. For example, a region that is configured to be 65536 bytes long must start on an address that is exactly divisible by 65536.

Most cross compilers include language extensions that can be used to force a variable to be placed on a specified address alignment. Listing 88 shows the syntax used for this purpose by the GCC, IAR, and Keil compilers.

```
/* Define and align an array using GCC syntax. */
char cAnArray[ 1024 ] __attribute__((aligned(1024)));

/* Define and align an array using IAR syntax. */
#pragma data_alignment=1024
char cAnArray[ 1024 ];

/* Define and align an array using Keil syntax.  Note this will only work for global
variables.  Keil also has a GCC compatibility mode where __attribute__ can be used.
*/
__align( 1024 ) char cAnArray[ 1024 ];
```

Listing 88. Syntax required by GCC, IAR, and Keil compilers to force a variable onto a particular byte alignment (1024-byte alignment in this example)

```
/* Define two arrays, access to each of which will be controlled by separate MPU
Regions (GCC syntax is shown). */
char cFirstArray[ 1024 ] __attribute__((aligned(1024)));
char cSecondArray[ 256 ] __attribute__((aligned(256)))
```

Listing 89. Defining two arrays that may be placed in adjacent memory

It is necessary to consider also how variables are placed in relation to each other. For example, consider the case shown in Listing 89. cFirstArray starts and ends on a 1024-byte boundary. cSecondArray starts and ends on a 256-byte boundary. As 1024 is divisible by 256, it is likely that the linker will place cSecondArray directly after and adjacent to cFirstArray. If a task has configured one MPU region to provide write access to cFirstArray, and another MPU region to provide write access to cSecond array, then the MPU will not prevent a write off

the end of cFirstArray, as might be the intent. A write outside the boundary of the first MPU region would not result in a memory protection fault but would result, instead, in a valid write into the second MPU. This situation can be avoided by making the size of cFirstArray 1025 bytes and the size of cSecondArray 257 bytes. The alignment requirements then prevent the linker from placing the arrays directly adjacent to each other. The actual alignment of the arrays, and the size of the MPU regions that control access to the arrays, do not change.

7.4 The FreeRTOS-MPU API

All the API functions available in the standard FreeRTOS Cortex-M3 port are also available in FreeRTOS-MPU. This section highlights some minor differences in the way xTaskCreate() is used, and introduces the API extensions that are specific to the MPU enabled kernel.

The xTaskCreateRestricted() API Function

xTaskCreateRestricted() is an extended version of xTaskCreate() that is used to create tasks with restricted execution privileges and restricted memory access rights.

xTaskCreateRestricted() requires all the parameters used by xTaskCreate(), plus four additional parameters that define the three task-specific MPU regions and a stack buffer. Attempting to use this number of parameters in a normal function parameter list would be cumbersome and could, potentially, make heavy use of stack space. Instead, FreeRTOS-MPU defines a structure called xTaskParameters that contains a member for each required parameter. xTaskParameters structures can be declared const and therefore remain in Flash. xTaskCreateRestricted() takes a pointer to an xTaskParameters structure as one of its two parameters. The second parameter is used to pass out a handle to the task being created—exactly as with the xTaskCreate() parameter of the same name. pxCreatedTask can be set to NULL if a handle to the task is not required.

```
portBASE_TYPE xTaskCreateRestricted( xTaskParameters *pxTaskDefinition,
                                     xTaskHandle *pxCreatedTask );
```

Listing 90. The xTaskCreateRestricted() API function prototype

Listing 91 contains the xTaskParameters structure definition, and the definition of the xMemoryRegion structure that xTaskParameters contains. The structure members are described in Table 27 and Table 28. Listing 90 shows how the structures are used.

```
/*
 * Defines a single MPU region.
 */
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} xMemoryRegion;

/*
 * Contains a member for each parameter required to create a restricted task.
 */
typedef struct xTASK_PARAMTERS
{
    pdTASK_CODE pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    unsigned portBASE_TYPE uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    xMemoryRegion xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} xTaskParameters;
```

Listing 91. Definition of the structures required by the xTaskCreateRestricted() API function

Table 27. xMemoryRegion structure members

Structure Member	Description
pvBaseAddress	The region start address. This must be a multiple of the region size as defined by the ulLengthInBytes value.
ulLengthInBytes	The region size in bytes. This must be a binary power of two having a value between 32 bytes and 4 gigabytes, inclusive.
ulParameters	The access permissions for the region, defined as the bitwise OR of the definitions contained in Table 26.

Table 28. xTaskParameters structure members

Structure Member	Description
pvTaskCode, pcName, usStackDepth, pvParameters	These parameters are the same as their xTaskCreate() equivalents. See Table 2.
uxPriority	In xTaskCreate(), uxPriority is used just to set the priority at which the task is initially created. In xTaskCreateRestricted(), it is also used to set the task to either Privileged mode or User mode. To create a User mode task, set uxPriority to the desired task priority. To create a Privileged mode task, bitwise OR the required task priority with portPRIVILEGE_BIT. For example, to create a User mode task at priority three, set uxPriority to 3. To create a Privileged mode task at priority three, set uxPriority to (3 | portPRIVILEGE_BIT). Source code examples are provided later in this chapter.

Table 28. xTaskParameters structure members

Structure Member	Description
puxStackBuffer	FreeRTOS-MPU uses an MPU region to ensure that the currently executing task can access its own stack, and that writes outside the valid stack space result in a memory protection fault. This means that the task stack start address and size must comply with the MPU region constraints already discussed—the size must be a binary power of two between 32 and 4 gigabytes, and the start address must be a multiple of the size. There are two ways to ensure compliance with the byte alignment requirements: 1. Provide an implementation of pvPortMallocAligned() that will allocate RAM from the heap with the specified byte alignment. The implementation is likely to be complex and potentially wasteful, so nothing further is mentioned in this book about this option. By default, pvPortMallocAligned() is not defined, and the standard pvPortMalloc() is used in its place. If pvPortMallocAligned() is implemented, then puxStackBuffer can be set to NULL. 2. Statically allocate a buffer (array) for use as a stack by the task being created, and use the compiler extensions to ensure that the buffer is correctly aligned. puxStackBuffer should then point to the start of the buffer. This is the method demonstrated later in this chapter.

Table 28. xTaskParameters structure members

Structure Member	Description
xRegions	An array of xMemoryRegion structures that define up to a maximum of three MPU regions (portNUM_CONFIGURABLE_REGIONS equals three). The kernel will automatically configure the MPU to use these regions each time the task being created enters the Running state. The regions can later be redefined using the vTaskAllocateMPURegions() API function. All three region definitions must be present in the xRegions array, even if only one or two are going to be used. To prevent a region definition being used, set all the members of its defining xMemoryRegion structure to zero.

Listing 92 shows an example of an xTaskParameters structure configured to define a User mode task. Changing the uxPriority value from 1 to (1 | portPRIVILEGE_BIT) would cause the structure to define a Privileged mode task, instead.

```
/* A User task is to be created that requires read only access to an array.  First
define the array to comply with the size and alignment rules.  This example uses GCC
syntax. */
char cArray[ 128 ] __attribute__((aligned(128)));

/* Next define the xTaskParameters structure that includes an MPU definition giving
the task the required array access.  Only one of the possible three MPU regions are
being used, but all three have to be defined. */
static const xTaskParameters xCheckTaskParameters =
{
    vDemoTask,  /* pvTaskCode - the function that implements the task. */
    "Demo",     /* pcName             */
    400,        /* usStackDepth - defined in words, not bytes. */
    NULL,       /* pvParameters - not being used in this case. */
    1,          /* uxPriority - User mode priority 1. */
    cTaskStack, /* puxStackBuffer - the array to use as the task stack. */

    /* xRegions - In this case the xRegions array is used to create a single MPU
    region to provide read only access to just one array.  The parameters for
    the two unused regions are just set to 0 to prevent them having any effect. */
    {
        /* Base address     Length      Parameters */
        { cArray,           128,        portMPU_REGION_READ_ONLY },
        { 0,                0,          0 },
        { 0,                0,          0 }
    }
};
```

Listing 92. Using the xTaskParameters structure

Listing 92 shows the simple case where the MPU is being used to control access to a single variable (in this case an array), but the same technique can be used to control access to a set of variables by grouping the variables into a single structure. If this is not practical, then compiler extensions can be used to place the variables manually into a correctly sized and aligned memory area or section defined within the linker script.

Using xTaskCreate() with FreeRTOS-MPU

xTaskCreate() can be used to create both User mode and Privileged mode tasks, but cannot be used to allocate MPU regions to the tasks at the point of their creation. Instead, Privileged mode tasks will have access to the entire memory map, whereas User mode tasks will have access to any Flash and RAM memory that is not configured for Privileged-only access.

As with xTaskCreateRestricted(), set uxPriority to the desired task priority to create a User mode task, or bitwise OR the required task priority with portPRIVILEGE_BIT to create a Privileged mode task. This is demonstrated by Listing 93.

```
int main( void )
{
    /* Create a User mode task using xTaskCreate(). */
    xTaskCreate
       (
            vOldStyleUserModeTask,       /* The function that implements the task. */
            "Task1",                     /* Text name for the task. */
            100,                         /* Stack depth in words. */
            NULL,                        /* Task parameters. */
            3,                           /* Priority and mode (User in this case). */
            NULL                         /* Handle. */
       );

    /* Create a Privileged mode task using xTaskCreate().  Note the use of
    portPRIVILEGE_BIT where the task priority is specified. */
    xTaskCreate
       (
            vOldStylePrivilegedModeTask, /* The function that implements the task. */
            ( signed char * ) "Task2",   /* Text name for the task. */
            100,                         /* Stack depth in words. */
            NULL,                        /* Task parameters. */
            ( 3 | portPRIVILEGE_BIT ),   /* Priority and mode (Privileged in this
                                         case). */
            NULL                         /* Handle. */
       );

    /* Start the scheduler. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 93. Using xTaskCreate() to create both User mode and Privileged mode task with FreeRTOS-MPU

The vTaskAllocateMPURegions() API Function

Up to three MPU region definitions can be assigned to a task as the task is created. The regions can then be redefined using the vTaskAllocateMPURegions() API function.

```
void vTaskAllocateMPURegions( xTaskHandle xTask, const xMemoryRegion * const pxRegions );
```

Listing 94. The vTaskAllocateMPURegions() API function prototype

Table 29. vTaskAllocateMPURegions() parameters

Parameter Name/ Returned Value	Description
xTask	The handle of the task whose MPU region definitions are being modified (the subject task)—see the pxCreatedTask parameter of the xTaskCreate()/xTaskCreateRestricted() API function for information on obtaining handles to tasks. A task can modify the MPU regions assigned to it by passing NULL in place of a valid task handle.
pxRegions	An array of exactly three xMemoryRegion structures. To prevent a region definition from being used, set all members of its defining xMemoryRegion structure to zero. The kernel will automatically configure the MPU to use these definitions each time the task being modified enters the Running state.

```
void vAFunction( xTaskHandle xTask )
{
/* Define an xMemoryRegion array that defines an 8K block from address 0 to
be read only, and a 2K block from address 0x10004000 to be accessible only from
privileged mode. The array defines only two of the possible three MPU regions,
but must contain all three entries.  The members of the unused entry are just set
to zero so it has no effect. */
static const xMemoryRegion xRegions[ 3 ] =
{
    /* Base address Length  Access parameters */
    { 0x00,         8096,   portMPU_REGION_READ_ONLY },
    { 0x10004000,   2048,   portMPU_REGION_PRIVILEGED_READ_WRITE },
    { 0,            0,      0 } /* The third entry is not used so is just set to
                                   zero. */
}

    /* Change the MPU regions of the task referenced by xTask to those defined by
    xRegions. */
    vTaskAllocateMPURegions( xTask, xRegions );

    /* Also change the MPU regions used by this task to those defined by xRegions. */
    vTaskAllocateMPURegions( NULL, xRegions );
}
```

Listing 95. Using vTaskAllocateMPURegions() to redefine the MPU regions associated with a task

The portSWITCH_TO_USER_MODE() API Macro

A Privileged mode task can call portSWITCH_TO_USER_MODE() to lower its own privilege to User mode. There is no way for a User mode task to raise its privilege to Privileged mode.

portSWITCH_TO_USER_MODE() does not take any parameters.

7.5 Linker Configuration

FreeRTOS-MPU requires the linker script to define two named sections as described by Table 30, and eight linker variables as described by Table 31.

The syntax used to define the required sections and variable depends on the tool chain being used. Listing 96 and Listing 97 provide an example that uses GNU LD syntax. LD is the linker that is distributed with GCC. The easiest way to generate a suitable linker script is to start with a pre-configured example from a FreeRTOS-MPU demo application.

Table 30. Named linker sections required by FreeRTOS-MPU

Section name	Description
privileged_functions	The section into which the kernel executable image is to be placed. privileged_functions should incorporate the vector table, starting at address zero, with the kernel image starting immediately after the vector table. An MPU region is used to protect access to the privileged_functions section, so its size must be a binary power of two to comply with the MPU region definition rules.
privileged_data	The section into which the kernel data is to be placed. As the section is protected by an MPU region, its start address and size must comply with the MPU region definition rules.

Table 31. Linker variables required by FreeRTOS-MPU

Variable name	Variable value
__FLASH_segment_start__	The start address of the LPC17xx Flash memory.
__FLASH_segment_end__	The end address of the LPC17xx Flash memory.
__privileged_functions_end__	The end address of the privileged_functions named section.
__SRAM_segment_start__	The start address of the LPC17xx SRAM memory.
__SRAM_segment_end__	The end address of the LPC17xx SRAM memory.

Table 31. Linker variables required by FreeRTOS-MPU

Variable name	Variable value
__privileged_data_start__	The start address of the privileged_data named section.
__privileged_data_end__	The end address of the privileged_data named section.

```
/* Given the memory map…. */
MEMORY
{
    FLASH (rx) : ORIGIN = 0x0,        LENGTH = 0x80000
    SRAM (rwx) : ORIGIN = 0x10000000, LENGTH = 0x8000
    AHBRAM0    : ORIGIN = 0x2007c000, LENGTH = 0x4000
    AHBRAM1    : ORIGIN = 0x20080000, LENGTH = 0x4000
}

/* ….define the variables required by FreeRTOS-MPU.  First ensure the section sizes
are a binary power of two to comply with the MPU region size rules. */
_Privileged_Functions_Region_Size = 16K;
_Privileged_Data_Region_Size = 256;

/* Then define the variables themselves. */
__FLASH_segment_start__         = ORIGIN( FLASH );
__FLASH_segment_end__           = __FLASH_segment_start__ + LENGTH( FLASH );
__privileged_functions_start__ = ORIGIN( FLASH );
__privileged_functions_end__    = __privileged_functions_start__ +
                                  _Privileged_Functions_Region_Size;
__SRAM_segment_start__          = ORIGIN( SRAM );
__SRAM_segment_end__            = __SRAM_segment_start__ + LENGTH( SRAM );
__privileged_data_start__       = ORIGIN( SRAM );
__privileged_data_end__         = ORIGIN( SRAM ) + _Privileged_Data_Region_Size;
```

Listing 96. Defining the memory map and linker variables using GNU LD syntax

```
/* Defining privileged_functions at the start of the Flash memory, but after the
vector table. */
SECTIONS
{
    /* Privileged section at the start of the flash - vectors must be first
    whatever. */
    privileged_functions :
    {
        KEEP(*(.isr_vector))
        *(privileged_functions)
    } > FLASH

    .text :
    {
        /* Non privileged code kept out of the first 16K of flash. */
         = __privileged_functions_start__ + _Privileged_Functions_Region_Size;

        *(.text*)
        *(.rodata*)

    } > FLASH

       /* Rest of section definitions go here - including the privileged_data
       definition. */
}
```

Listing 97. Defining the privileged_functions named section using GNU LD syntax

7.6 Practical Usage Tips

Accessing Data from a User Mode Task

A User mode task cannot access RAM that is outside its own stack space, unless the address falls within the range of one of the task's MPU region definitions. If, for example, a User mode task needs the value of a globally declared queue handle, then, to be accessible, the value must first be copied into a variable that is on the task stack. There are several ways to achieve this, including:

- Initially, create the task in Privileged mode, and then copy the global variable value into a stack variable, before switching the task into the required User mode. This method is demonstrated in Listing 98.

- Pass the value of the global variable into the task using the task parameter. This method is demonstrated in Listing 99.

```
/* The handle to a queue is stored in a global (or file scope) variable. */
xQueueHandle xGlobalQueue;

void vATask( void *pvParameters )
{
xQueueHandle xStackQueue;

    /* This task was created in Privileged mode so can access the global variable.
    Copy the value of the global variable into a stack variable while the task is
    still in Privileged mode. */
    xStackQueue = xGlobalQueue;

    /* Now set the task into User mode.  From this point on the task can no longer
    access the value of the global variable, but can access its local stack copy. */
    portSWITCH_TO_USER_MODE();

    for( ;; )
    {
        /* The main task functionality is performed in User mode. Data can be sent
        to or from the queue using xStackQueue as the handle. */
    }
}
```

Listing 98. Copying data into a stack variable before setting the task into User mode

```
/* The handle to a queue is stored in a global (or file scope) variable. */
xQueueHandle xGlobalQueue;

void vATask( void *pvParameters )
{
xQueueHandle xStackQueue;

    /* This task was created in User mode so cannot access the global variable.  It
    can access variables stored on its own stack and the task parameter.  The value
    of xGlobalQueue is passed into this task using the task parameter and then copied
    into the local stack variable, casting to the appropriate type. */
    xStackQueue = ( xQueueHandle ) pvParameters;

    for( ;; )
    {
        /* The main task functionality is done here.  Data can be sent to or from the
        queue using xStackQueue as the handle. */
    }
}
```

Listing 99. Copying the value of a global variable into a stack variable using the task parameter

Intertask Communication from User Mode

Code executing in User mode cannot access RAM outside its own stack and the MPU regions that are configured for it. This does not prevent User mode tasks from using queues or semaphores to communicate with other tasks or interrupts.

The RAM used by queues and semaphores is owned and controlled by the kernel and can be accessed only when the processor is executing in Privileged mode. Calling an API function such as xQueueSend() causes the processor to switch temporarily into Privileged mode, from where the data being queued can be copied from the User mode task into the kernel controlled queue storage area. Similarly, calling an API function such as xQueueReceive() causes the processor to switch temporarily into Privileged mode, from where the data being received can be copied from the kernel controlled queue storage area into the User mode task.

FreeRTOS-MPU Demo Projects

FreeRTOS-MPU is included in the main FreeRTOS download. Some heavily commented FreeRTOS-MPU demo applications are located in sub-directories with names that start ‘Cortex-MPU’ within the FreeRTOS\Demo directory.