FreeRTOS – Part 1

Da-Wei Chang

Sources:

- *The freeRTOS website, https://www.freertos.org/*
- *Cortex-M3 Devices Generic User Guide*

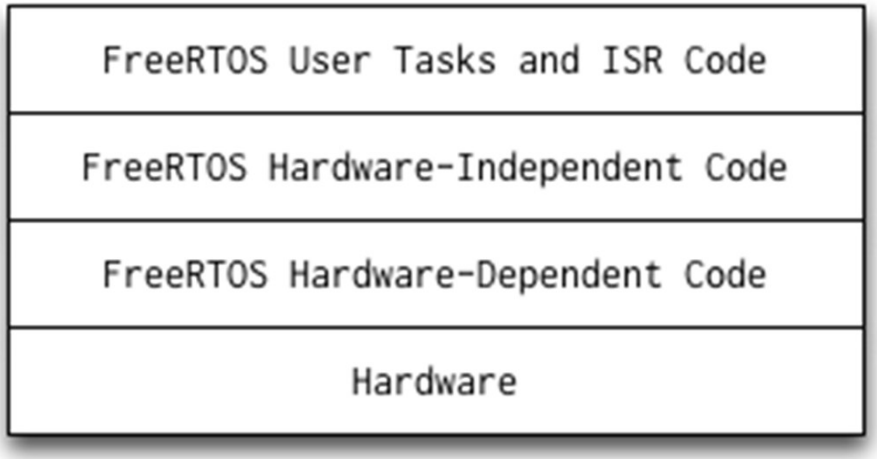
Embedded vs. Real-time

- Embedded
  - a computer system that is designed to do only a few things
  - e.g., TV remote control, in-car GPS, ABS, digital watch…
  - typically resource limited

- Real-time
  - do something within a certain amount of time
  - e.g., missile system…

Architecture Overview

- Minimal 3 .c files
  - about 9000 lines of code
- Typical binary size < 10 KB
- Breaks down into three main areas
  - Tasks
    - tasks.c, task.h
  - Communication
    - queue.[c/h]
  - Hardware dependent



Understanding the FreeRTOS Distribution

- FreeRTOS ports
  - Port: (compiler, processor/board)
  - ~ 20 compilers
  - >30 processors
- Library OS
  - Should be built with your application programs
- Configured by a header file: FreeRTOSConfig.h
  - e.g., configUSE_PREEMPTION
    - co-operative or pre-emptive scheduling

Top Level Directories

```
FreeRTOS
    ├─Source   Directory containing the FreeRTOS source files
    └─Demo     Directory containing pre-configured and port specific FreeRTOS demo projects
FreeRTOS-Plus
    ├─Source   Directory containing source code for some FreeRTOS+ ecosystem components
    └─Demo     Directory containing demo projects for FreeRTOS+ ecosystem components
```

FreeRTOS Libraries:
TCP/IP stack,
HTTP,
MQTT,
…

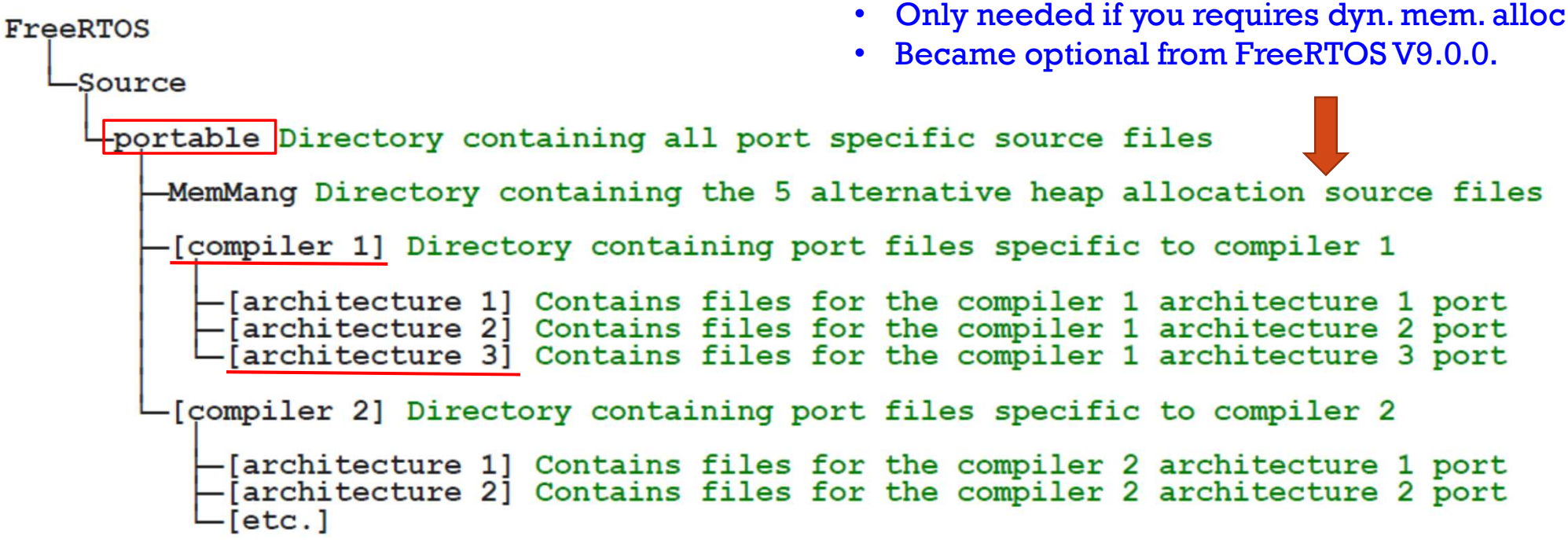
Core FreeRTOS Source Files

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

- tasks.c : task management, scheduling
- list.c : linked list
- queue.c : queue and semaphore services
- timers.c : software timer functionality
- event_groups.c : event group functionality
- croutine.c : implements the FreeRTOS co-routine functionality
  - intended for use on very small microcontrollers
  - rarely used now

Port-Specific Source Code

- Dynamic memory allocation methods
- Only needed if you requires dyn. mem. alloc.
- Became optional from FreeRTOS V9.0.0.



Data Types

- **TickType_t**
  - Used to store *tick count*
    - Tick count: number of timer/tick interrupts occurred since the system started
  - Can be unsigned 16-bit or unsigned 32-bit
    - uint16_t if the configUSE_16_BIT_TICKS [FreeRTOSConfig.h] is 1
    - uint32_t if the configUSE_16_BIT_TICKS [FreeRTOSConfig.h] is 0
    - Setting configUSE_16_BIT_TICKS as 1
      - more efficient on 8/16-bit processors
      - easier to overflow
      - typically, no need in 32-bit processors

Data Types

- **BaseType_t**
  - Set as the most efficient data type for the architecture
  - Typically, 32-bit type on a 32-bit architecture, 16-bit type on a 16-bit architecture…

Naming Convention - Variable Names

- Name of a variable is prefixed with its type
  - **c** for char
  - **s** for int16_t (short)
  - **l** for int32_t (long)
  - **x** for BaseType_t and any other non-standard types (structures, task handles, queue handles,…)
  - Prefixed with **u** if unsigned
  - Prefixed with **p** for pointer
- Examples
  - a variable of type uint8_t will be prefixed with **uc**
  - a variable of type pointer to char will be prefixed with **pc**

Naming Convention - Function Names

- Name of a function is prefixed with
  - the return type
  - the file they are defined within
  - Examples
    - **v**TaskPrioritySet() returns a **void** and is defined within task.c
    - **x**QueueReceive() returns a variable of type **BaseType_t** and is defined within queue.c
    - **pv**TimerGetTimerID() returns a **pointer to void** and is defined within timers.c
- Private functions do NOT follow the above naming rules, but are prefixed with **prv**
  - private function: static function, visible only within the file it is defined

Naming Convention - Macro Names

- written in UPPER CASE
- prefixed with lower case letters indicating where it is defined

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Naming Convention - Macro Names

- Naming Exception
  - Semaphore API, written almost entirely in macros, follows the function naming convention instead of the macro naming convention

For coding style consistency, you should follow the naming convention if you write code in FreeRTOS!

Heap Memory Management

Heap Management

- Heaps are dynamically allocated memory
- Traditionally, FreeRTOS kernel objects are NOT statically allocated at compile-time, but *dynamically allocated* at run-time
  - Including
    - Tasks (TCBs and Stacks)
    - Software Timers
    - Queues
    - Event Groups
    - Semaphores and Mutexes
  - Benefit
    - minimizes the RAM footprint
- From FreeRTOS V9.0.0, kernel objects can either be allocated statically or dynamically

Heap Management

- Why not allocate memory using the standard C lib. malloc()/free() functions
  - not always available on small embedded systems
  - the implementation can be relatively large
  - usually not thread-safe
  - usually not deterministic
    - amount of time taken to execute the functions will differ from call to call
  - may suffer from fragmentation
    - not good for memory-constrained embedded systems

Heap Management

- FreeRTOS puts memory allocation in the portable layer
  - Not a part of the core base
  - Reason
    - Different embedded systems have different memory allocation and timing requirements
      - a single dynamic memory allocation method will only be suitable for a subset of applications
    - Enable application writers to provide their own implementations
- Interfaces: pvPortMalloc(), vPortFree()
- FreeRTOS provides 5 example implementations
  - heap_*x*.c where *x* = 1-5
  - use one heap_*x*.c or provide your own (if you need dyma. mem. alloc.)

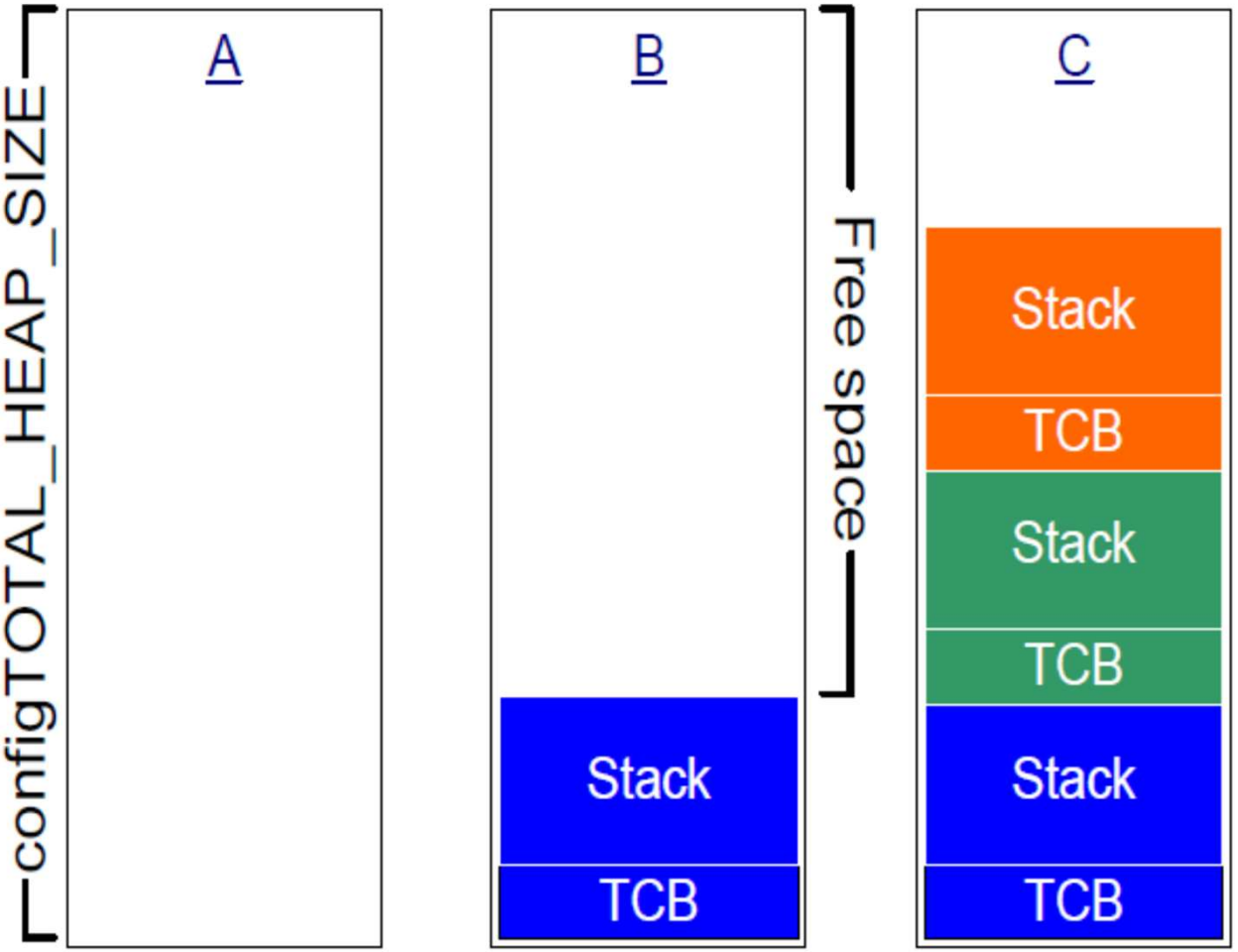
Heap_1

- Heap_1 provides
  - a simple pvPortMalloc() implementation
  - no vPortFree() implementation
- Suitable in
  - systems that create tasks and kernel objects before the scheduler has been started, and never free kernel objects
    - application never deletes a task, queue, semaphore, mutex, ...
  - systems not prefer dynamic allocation because of memory fragmentation and failed allocations
    - Heap_1 does not have memory fragmentation
      - since there is no *free* (vPortFree()) operations

Heap_1 is less useful since FreeRTOS added support for static allocation!

Heap_1

A: before any tasks have been created
B: after one task has been created
C: after three tasks have been created



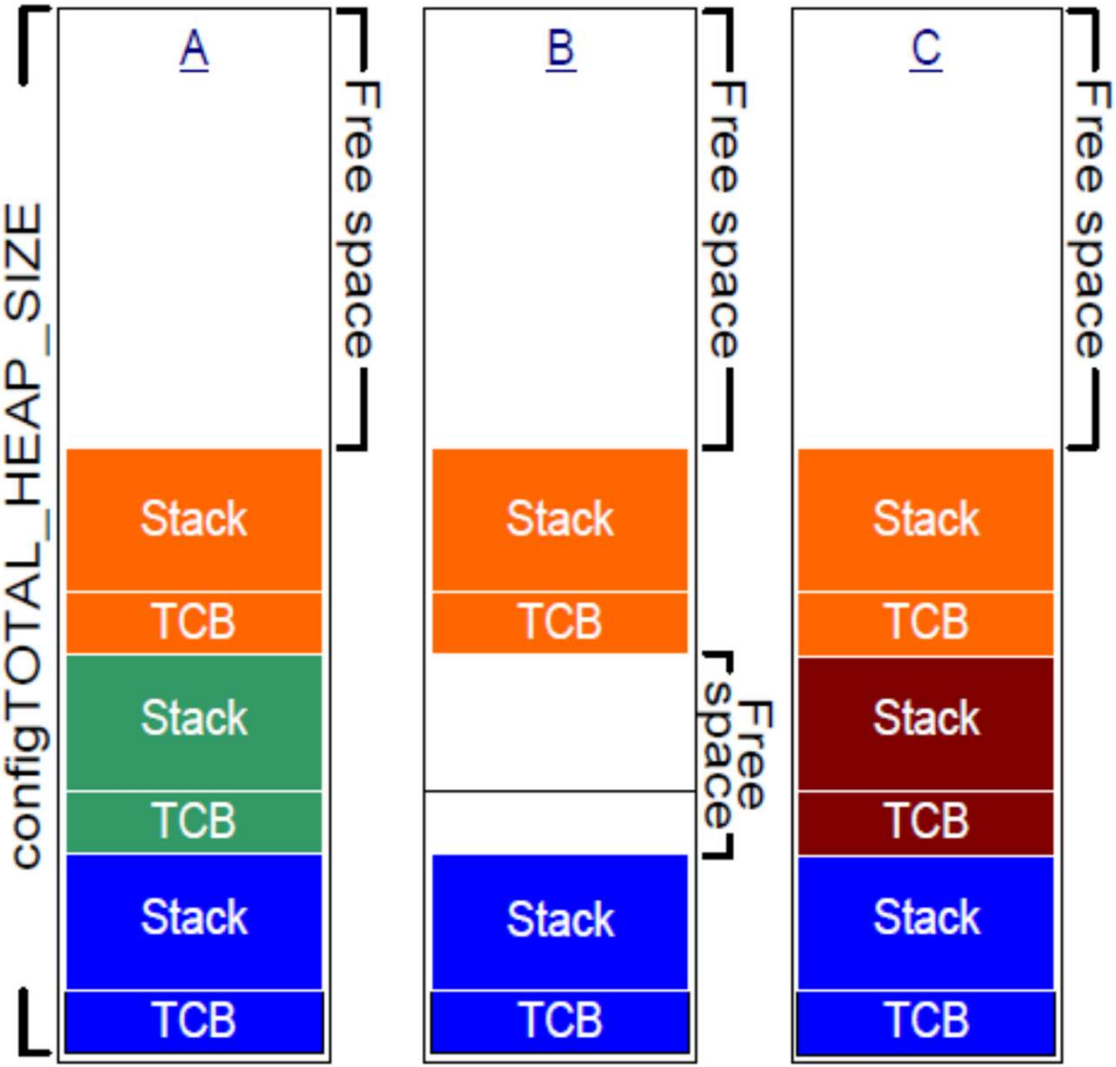
- TCB: task control block
- configTOTAL_HEAP_SIZE : the heap size, defined in FreeRTOSConfig.h

Heap_2

- Only retained for backward compatibility, **not recommended!**
  - Use heap_4 instead
- Use best-fit method to allocate memory, and allow memory to be freed
- Heap (with size **configTOTAL_HEAP_SIZE**) is statically allocated
- Method
  - Split (the best-fit) free memory when allocation, but
  - No merge when free
    - May cause fragmentation
- Suitable for
  - An application that creates and deletes tasks repeatedly, given that the tasks have the same sizes of stack
    - TCB and task stacks can be reused repeatedly
- Not deterministic, but
  - faster than most std. lib. malloc()/free() implementations

Heap_2

A: after 3 tasks have been created
B: after one task has been deleted
C: after another task (with the same stack size) has been created



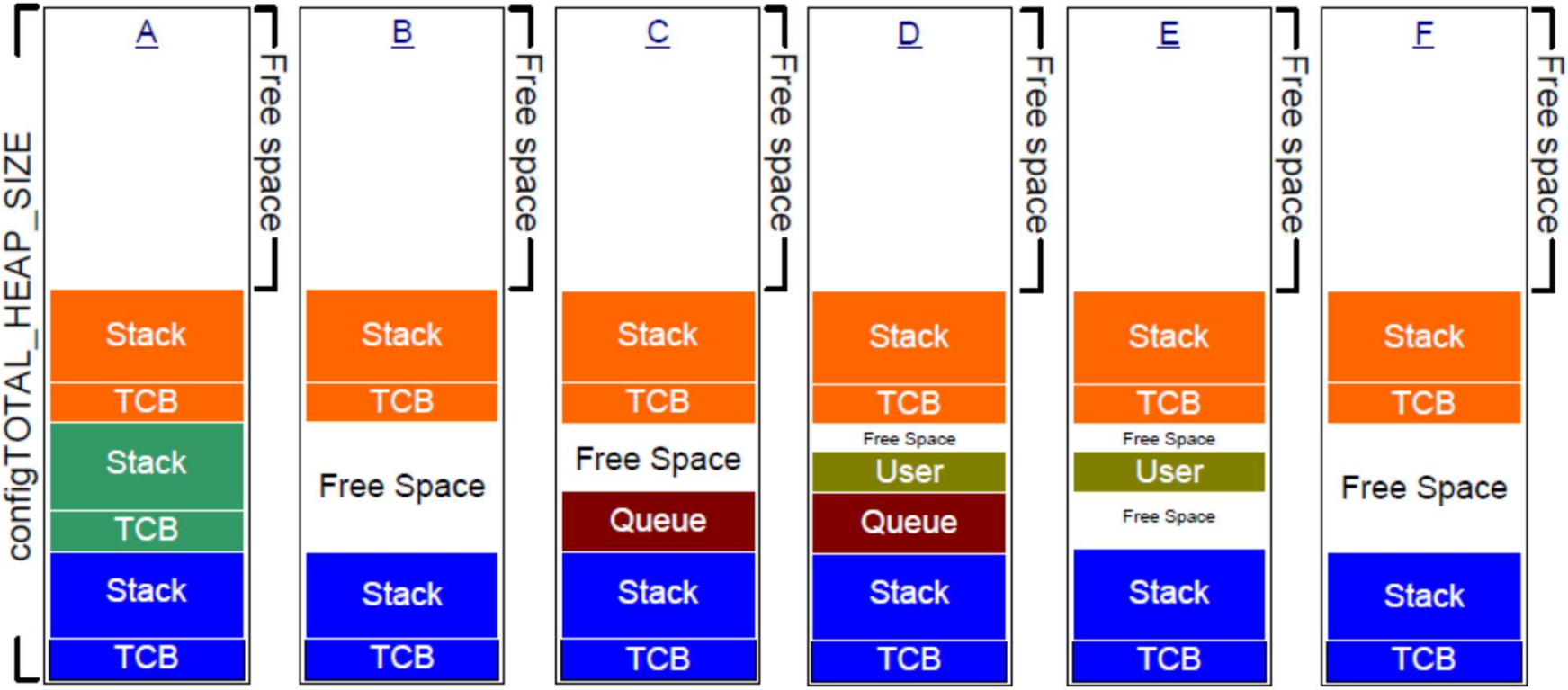
Heap_3

- Wrappers to the standard library malloc() and free() functions provided by the compiler
- heap size is defined by the linker configuration
  - Not the configTOTAL_HEAP_SIZE
- Suspend scheduler when calling malloc() and free()
  - to ensure thread-safe

Heap_4

- Heap with size configTOTAL_HEAP_SIZE is statically declared
  - Just like heap_1 and heap_2
- Use first-fit algorithm to allocate memory
  - A free list sorted with ascending address is maintained
- Allow memory free, and perform **merge** when free
  - adjacent free memory can be merged into a bigger one
    - Reduce memory fragmentation
- Suitable for
  - Applications that repeatedly allocate and free different sized RAM
- Not deterministic, but
  - faster than most std. lib. malloc()/free() implementations

Heap_4



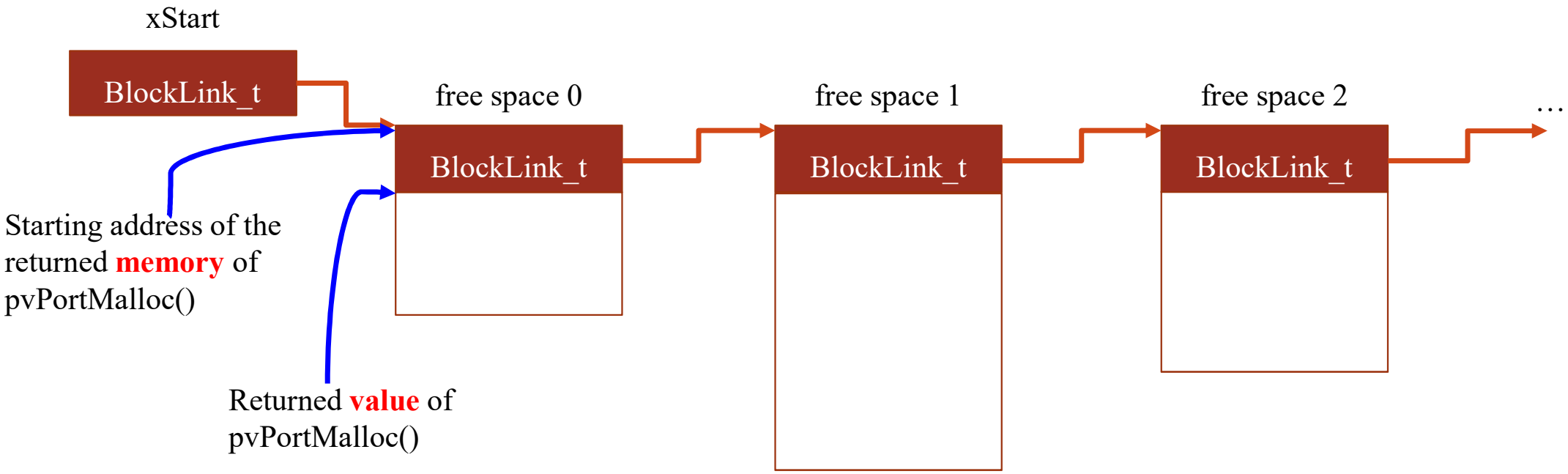
A: after 3 tasks have been created
B: after one task has been deleted (the freed TCB and Stack are **merged** into one free region)
C: after a FreeRTOS queue has been created (split the first free space)
D: after pvPortMalloc() has been called directly from the user (split the first free space)
E: after the queue has been deleted
F: after the user allocated memory has also been freed (3 free spaces **merged** into 1)

The Free List in Heap_4

```
/* a free block node */
typedef struct A_BLOCK_LINK
{
        struct A_BLOCK_LINK * pxNextFreeBlock;      /* The next free block in the list. */
        size_t xBlockSize;                          /* The size of the free block. */
} BlockLink_t;


/* List Traversal in pvPortMalloc() */
pxPreviousBlock = &xStart;
pxBlock = xStart.pxNextFreeBlock;

while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pxNextFreeBlock != NULL ) )
{
        pxPreviousBlock = pxBlock;
        pxBlock = pxBlock->pxNextFreeBlock;
}
……
```

The free list head

The Free List in Heap_4



- Each **free** space/block has a *BlockLink_t* located **at the beginning** of the free space
- Each **allocated** block also has a *BlockLink_t* located **just before** the returned address of the vPortMalloc()
  - ✓ Storing the block size so that the block can be correctly inserted back to the free list after vPortFree()

Setting the Starting Address of Heap

- By default, the heap (i.e., ucHeap) used by heap_4 is declared in heap_4.c
  - Heap start address is set automatically by the linker
- Why do users need to set the starting address of the heap?
  - e.g., ensure the heap is located in fast internal memory, not slow external memory
- Set the heap start address
  - Set configAPPLICATION_ALLOCATED_HEAP to 1
  - Declare the heap (byte-array) variable: ucHeap[configTOTAL_HEAP_SIZE], and
  - Configure the address of ucHeap
    - uint8_t **ucHeap [**configTOTAL_HEAP_SIZE**]** __attribute__ ((section(".my_heap")));
      - In GCC, the above code places the array in a memory section named .my_heap
    - uint8_t **ucHeap[** configTOTAL_HEAP_SIZE] **@ 0x20000000**;
      - In IAR, the above code places the array at the absolute address 0x20000000

Heap_5

- Can allocate memory from multiple separate memory spaces
- Suitable for
  - systems whose RAM does not appear as a single contiguous region in the system's memory map
- **Note:** must be explicitly initialized before pvPortMalloc() can be called
  - initialized using the vPortDefineHeapRegions()
    - specify the start address and size of each separate memory area in the heap

vPortDefineHeapRegions

Pointer to an array of heap regions

- void vPortDefineHeapRegions (const **HeapRegion_t** * const pxHeapRegions);

A heap region

```
typedef struct HeapRegion
{
    /* The start address of a block of memory that will be part of the heap.*/
    uint8_t *pucStartAddress;

    /* The size of the block of memory in bytes. */
    size_t xSizeInBytes;

} HeapRegion_t;
```

- HeapRegion_t structures in the array must be **ordered** by the start address
  - Smaller address first
- End of the array is marked by a HeapRegion_t structure with pucStartAddress member set to NULL

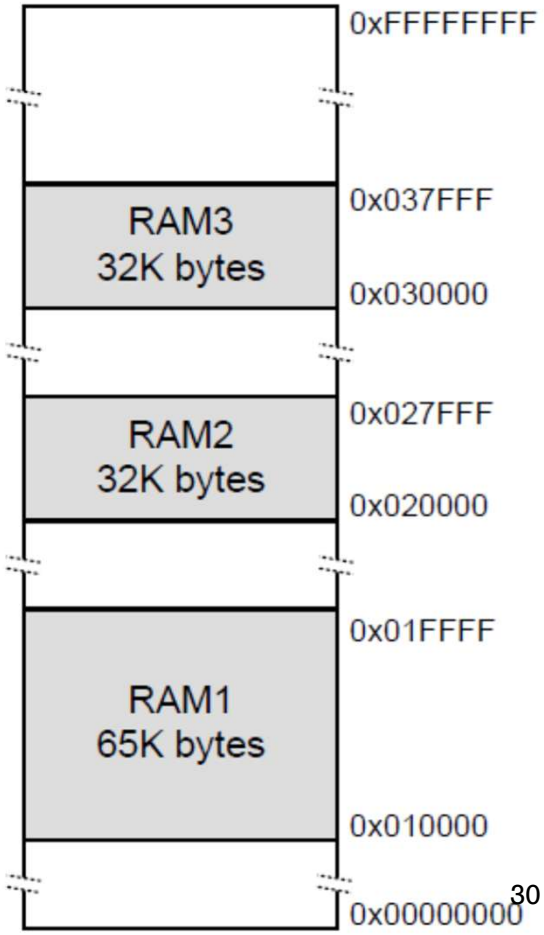
Heap_5 Example

```
/* Define the start address and size of the three RAM regions. */
#define RAM1_START_ADDRESS     ( ( uint8_t * ) 0x00010000 )
#define RAM1_SIZE              ( 65 * 1024 )

#define RAM2_START_ADDRESS     ( ( uint8_t * ) 0x00020000 )
#define RAM2_SIZE              ( 32 * 1024 )

#define RAM3_START_ADDRESS     ( ( uint8_t * ) 0x00030000 )
#define RAM3_SIZE              ( 32 * 1024 )

const HeapRegion_t xHeapRegions[] =
{
    { RAM1_START_ADDRESS, RAM1_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL,               0         }
};

int main( void )
{
    /* Initialize heap_5. */
    vPortDefineHeapRegions( xHeapRegions );

    /* Add application code here. */
}
```

Smaller address first

End of the array

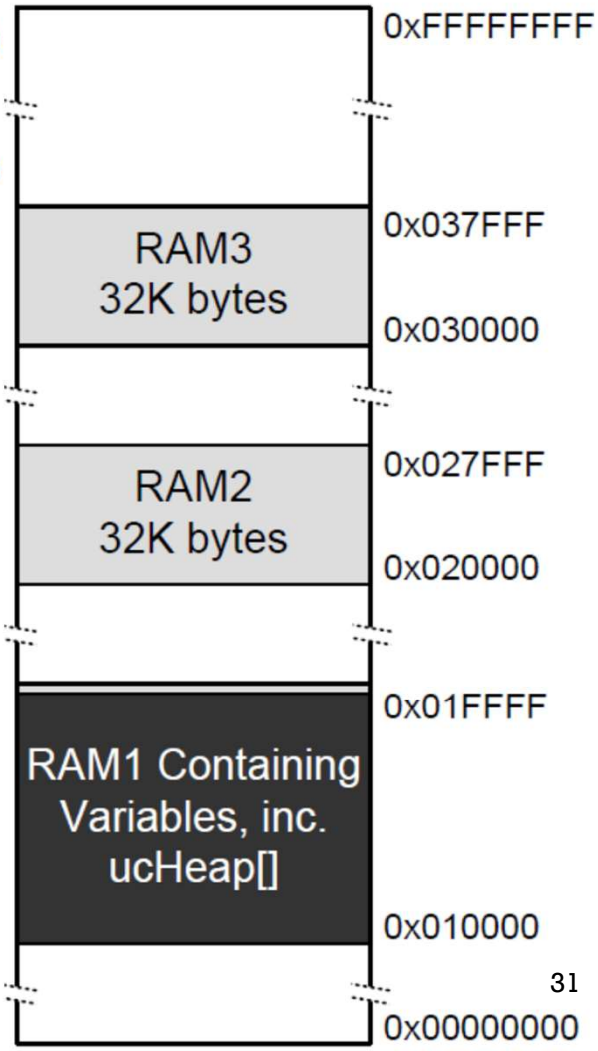
All the RAMs are used as the heap, not a typical case!



A Typical Heap_5 Example

```
#define RAM2_START_ADDRESS     ( ( uint8_t * ) 0x00020000 )
#define RAM2_SIZE              ( 32 * 1024 )

#define RAM3_START_ADDRESS     ( ( uint8_t * ) 0x00030000 )
#define RAM3_SIZE              ( 32 * 1024 )

#define RAM1_HEAP_SIZE ( 30 * 1024 )
static uint8_t ucHeap[ RAM1_HEAP_SIZE ];

const HeapRegion_t xHeapRegions[] =
{
    { ucHeap,              RAM1_HEAP_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL,                0           }
};
```

- **RAM2 and RAM3 not included in the linker script**
- **RAM2 and RAM3 are entirely used for heap**
- **Only part of the RAM1 (i.e., uCHeap[]) is used for heap**



Heap Related Utility Functions

- **xPortGetFreeHeapSize()**
  - returns the number of free bytes in the heap
  - not available when heap_3 is used
- **xPortGetMinimumEverFreeHeapSize()**
  - returns the minimum number of unallocated bytes that have ever existed in the heap since the FreeRTOS application started executing
  - an indication of how close the application has ever come to running out of heap space
  - only available when heap_4 or heap_5

Heap Related Utility Functions

- **vApplicationMallocFailedHook()**
  - the *error* hook function
  - called when pvPortMalloc() returns NULL (i.e., mem. allocation fails)
  - **enabled** when configUSE_MALLOC_FAILED_HOOK is set to 1

Task Management

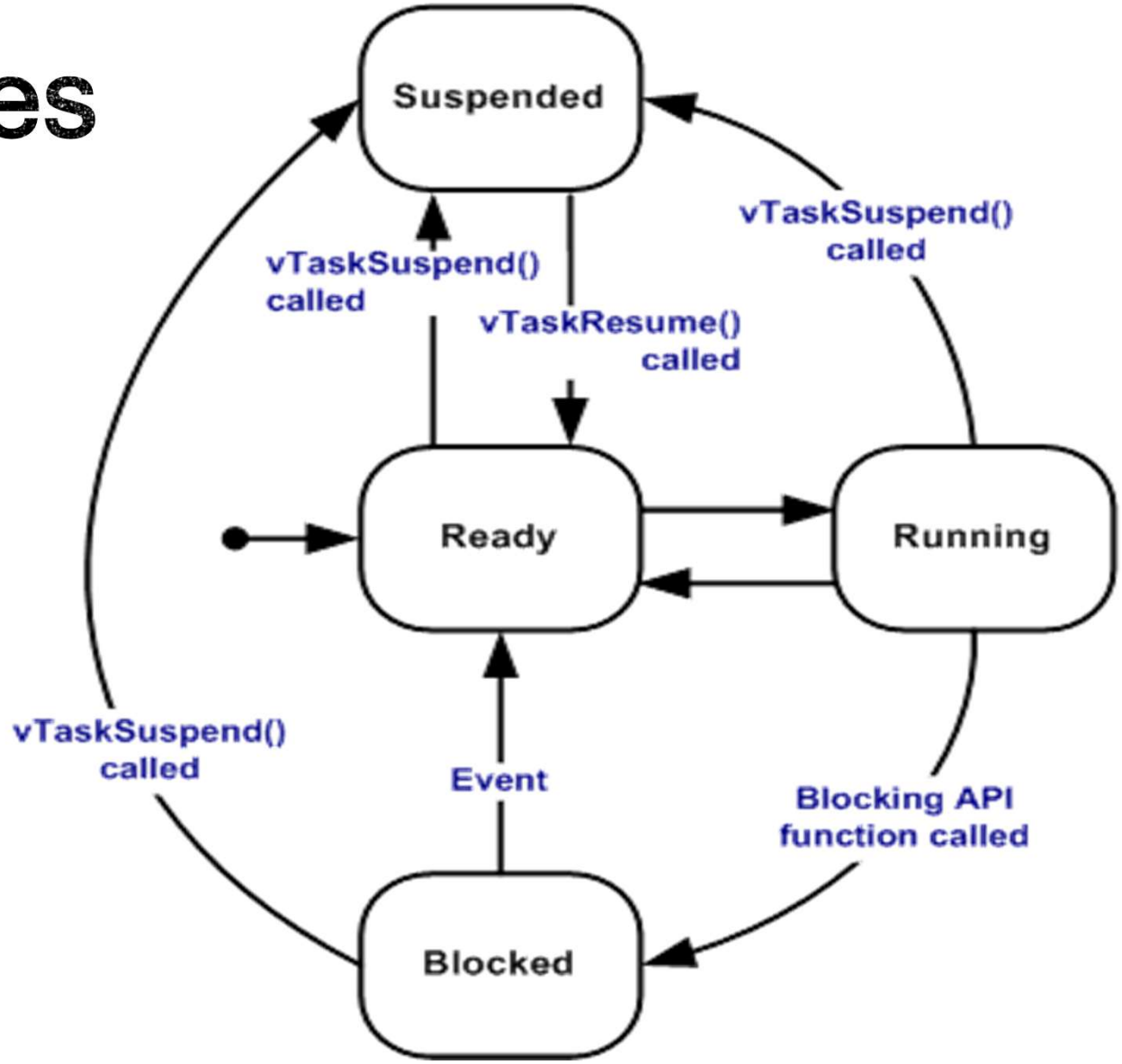
Task

- Supports full preemption
- Prioritised
- Each task has its own stack

Task States

- **Running**
  - currently utilizing the processor
- **Ready**
  - Ready to run, waiting to be scheduled
- **Blocked**
  - waiting for either a temporal or external event
    - vTaskDelay()
    - wait for queue, semaphore, event group, notification …
  - typically have a '***timeout***' period
- **Suspended**
  - Enter/exit this state via vTaskSuspend()/xTaskResume()
  - No timeout

Task States


Suspended
vTaskSuspend()
called
vTaskResume()
called
vTaskSuspend()
called
Ready
Running
vTaskSuspend()
called
Event
Blocking API
function called
Blocked

Task Priorities

- Task priority from 0 to (configMAX_PRIORITIES – 1)
  - **configMAX_PRIORITIES** is defined in FreeRTOSConfig.h
  - 0 is the lowest priority
    - = tskIDLE_PRIORITY, the priority of the idle task
- You can configure the configMAX_PRIORITIES, but be careful
  1. configMAX_PRIORITIES cannot be larger than 32 when
     - configUSE_PORT_OPTIMISED_TASK_SELECTION = 1
     - The port uses a '**count leading zeros**' type of instruction for task selection in a single instruction
  2. Consider the RAM usage…
     - Number of priorities ↑ ➔ RAM usage ↑

Task Priorities

- Any number of tasks can share the same priority

- Ready tasks of equal priority will share the CPU time using the round robin (RR) scheduling scheme, if
  - configUSE_TIME_SLICING is **not defined**, or if
  - configUSE_TIME_SLICING is **set to 1**

Task Function

- a C function
- The entry point of a task
- Prototype
  - void *v*FuntionName (void **pv*Parameters);
- Typically, a task will be implemented as an infinite loop
- **Must NOT return** from the task function
- If a task is no longer required, it should be explicitly deleted
  - via **vTaskDelete()**
- Multiple tasks can have the same task function
  - Each task still has its own stack and automatic (stack) variables

Task Function

Type TaskFunction_t
- All functions that implement a task should be of this type

A task should have the following structure…

```
void vATaskFunction( void *pvParameters )
{
   for( ;; )
   {
      -- Task application code here. --
   }

   /* Tasks must not attempt to return from their implementing
   function or otherwise exit.  In newer FreeRTOS port
   attempting to do so will result in an configASSERT() being
   called if it is defined.  If it is necessary for a task to
   exit then have the task call vTaskDelete( NULL ) to ensure
   its exit is clean. */
   vTaskDelete( NULL );
}
```

Task Control Block (TCB)

```
typedef struct tskTaskControlBlock
{
    volatile portSTACK_TYPE    *pxTopOfStack;    /* stack top */
    xListItem                  xGenericListItem;  /*ready list */
    xListItem                  xEventListItem;
    unsigned portBASE_TYPE     uxPriority;
    portSTACK_TYPE             *pxStack;         /* stack start */
    signed char                pcTaskName[configMAX_TASK_NAME_LEN];

    #if ( portSTACK_GROWTH > 0 )
        portSTACK_TYPE         *pxEndOfStack;   /* overflow check for grow-up stack*/
    #endif
    #if ( configUSE_MUTEXES == 1 )
        unsigned portBASE_TYPE uxBasePriority;  /* for priority inheritance */
    #endif
    ..... Skipped...
} tskTCB;
```

Task Control Block (TCB)

- Stack related fields
  - pxStack: stack start (*the minimum address*)
  - pxEndOfStack: stack end
    - Used only for grow-up stacks
  - pxTopOfStack: stack top
  - **Stack overflow checking** ← What is stack overflow?
    - For grow-up stack: pxTopOfStack > pxEndOfStack
    - For grow-down stack: pxTopOfStack < pxStack
- Priority related fields
  - uxPriority: current priority
  - uxBasePriority: original priority while the task is temporarily elevated to the "inherited" priority
    - For implementing priority inheritance

Task Control Block (TCB)

- Lists related fields
  - xGenericListItem: ready list
  - xEventListItem: event list

- **Where is the '*task state*'?**
  - TCB does NOT explicitly maintain the task state!
  - FreeRTOS tracks task state *implicitly* by putting tasks in the appropriate list (e.g., ready list, suspended list, etc.)

Creating Tasks

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

- configSUPPORT_DYNAMIC_ALLOCATION (FreeRTOSConfig.h) **must be 1**, or left undefined (in which case it will default to 1), for xTaskCreate() to be available
- RAM for the task (including task stack) is allocated from the **FreeRTOS heap**

Parameters of xTaskCreate()

- pvTaskCode
  - pointer to the task function (task entry point)
- pcName
  - a descriptive name for the task
  - included purely as a debugging aid
  - configMAX_TASK_NAME_LEN defines the maximum length a task name can take, including the NULL terminator
    - longer task names will be silently truncated
- usStackDepth
  - the stack size (in **words**)

Parameters of xTaskCreate()

- pvParameters
  - Used to pass data to the created task
- uxPriority
  - The priority of the task
  - lowest priority: **0**, highest priority: **configMAX_PRIORITIES – 1**
  - Systems that include **MPU support** can create a **privileged-mode task** by setting bit **portPRIVILEGE_BIT** in uxPriority
    - E.g., create a privileged task at priority 2 ➔ set uxPriority to (2 | portPRIVILEGE_BIT)
- pxCreatedTask
  - a handle to the created task
  - can be used to reference the task in the later API calls
    - E.g., change the task priority, delete the task…
  - Set to NULL if your application has no use for the task handle

Return Value of xTaskCreate()

- pdPASS
  - indicates that the task has been created successfully
- pdFAIL
  - indicates that the task has not been created
    - E.g., free memory in the heap is not enough to hold the TCB and stack of the to-be-created task

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

An Example

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    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 2 provides more information on heap memory management. */
    for( ;; );
}
```

Multiple tasks can have the same priority!!!

should not reach here…

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Creating a Task from Another Task

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before entering the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

The Idle Task

- Created automatically by the RTOS
  - Exists to ensure there is always at least one task that is able to run
- Run at the lowest possible priority
  - tskIDLE_PRIORITY
- Defined in prvIdleTask() [tasks.c]

Idle Task Hook

- a **function** that is called **during each cycle** of the idle task
- It is common to use idle hook to place the CPU into sleep mode
- Two options to run your code at the idle priority
  - Write your code in an idle task hook
    - Set configUSE_IDLE_HOOK to 1 (FreeRTOSConfig.h)
    - Define a function that has the following name and prototype
      - void vApplicationIdleHook(void);
    - **You cannot call any API functions that might cause the idle task to block!!!**
  - Create an idle-priority task
    - more flexible, but
    - has a higher RAM usage overhead

Stack Initialization

- When a task is created, its stack is initialized
- Storing task context in task stacks is common
- The initialized stack is look as if the new task is about to run and interrupted by a context switch
  - So, the scheduler can treat newly created tasks exactly the same way as it treats tasks that have been running for a while
- The stack init. code is CPU dependent
  - The ARM Cortex-M3 processor port is shown in the next slide

pxPortInitialiseStack

For ARM Cortex-M3

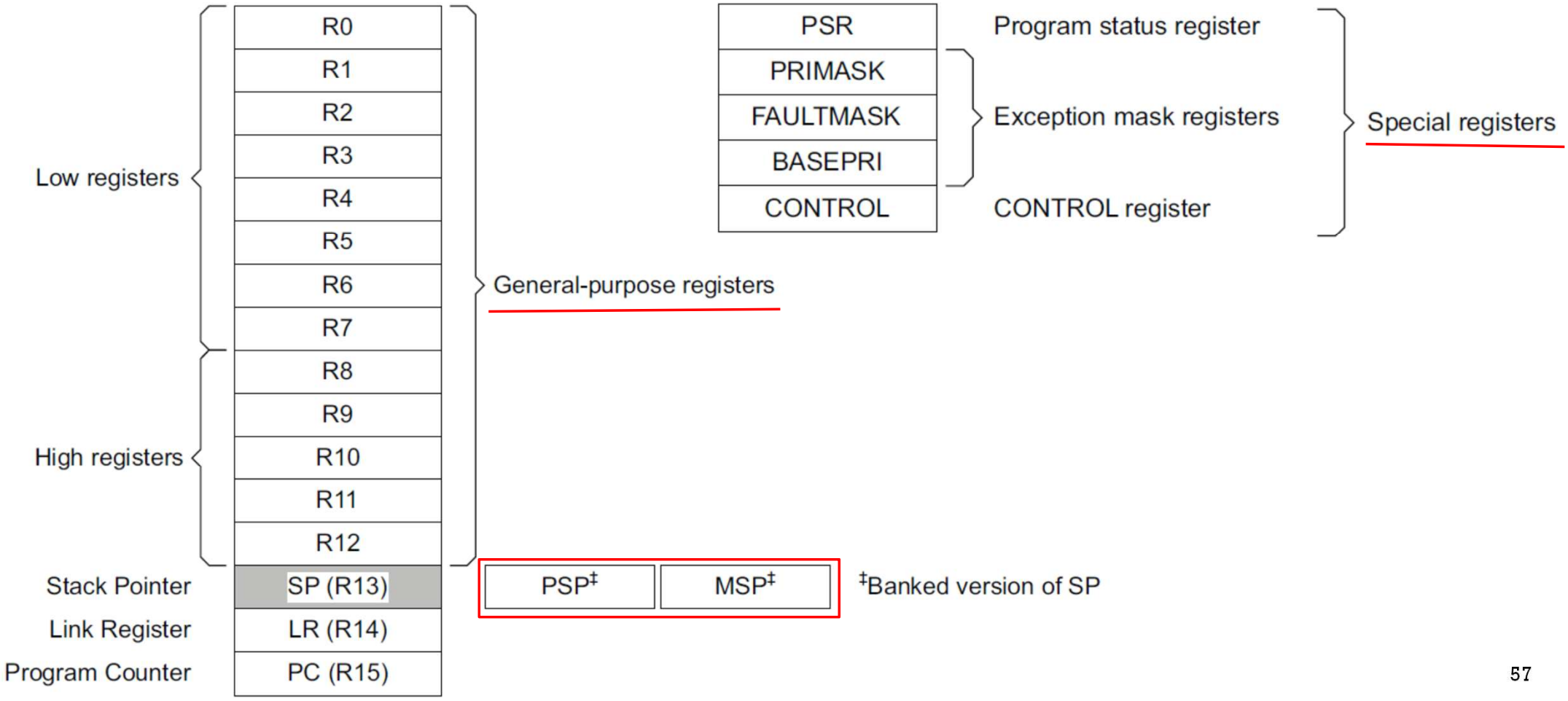
```
unsigned int *pxPortInitialiseStack( unsigned int *pxTopOfStack, pdTASK_CODE pxCode,
                                     void *pvParameters )   {
```

Simulate stack push by CPU HW

```
    pxTopOfStack--;                                          /* full descending stack */
    *pxTopOfStack = portINITIAL_XPSR;                        /* xPSR */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) pxCode;               /* PC init. as the entry point*/
    pxTopOfStack--;
    *pxTopOfStack = portTASK_RETURN_ADDRESS;    /* LR */
    pxTopOfStack -= 5;                                       /* R12, R3, R2, R1, and R0. */
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters;  /* R0 used for passing parameters*/
    pxTopOfStack -= 8;                                       /* R11, R10, R9, R8, R7, R6, R5 and R4. */
    return pxTopOfStack;
}
```

no specific data stored in the stack for R1-R12:
In the ARM architecture, those registers are undefined at reset

Core Registers of ARM Cortex-M3



Core Registers

- General-purpose registers (R0-R12)
  - 32-bit general-purpose registers for data operations
- Stack Pointer (R13)
  - Either MSP (Main Stack Pointer) or PSP (Process Stack Pointer)
  - In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use
    - 0 = *Main Stack Pointer* (MSP). This is the reset value.
    - 1 = *Process Stack Pointer* (PSP)
- Link Register (R14)
  - stores the return information for function calls and exceptions
- Program Counter (R15)
  - On reset, the processor loads the PC with the value of the reset vector (at address 0x00000004)
  - Bit 0 of PC is reserved, loaded into the **EPSR T-bit** at reset, and must be **1**

Processor Modes and Privilege Levels

- 2 **processor modes**
  - *Thread* mode
    - execute application software
    - processor enters the Thread mode when it comes out of reset
  - *Handler* mode
    - handle exceptions, returns to Thread mode when exception processing finished
    - Always in *privileged* level
- 2 **privilege levels**
  - *Unprivileged*
    - Cannot execute certain instructions (e.g., CPS) and cannot use certain resources (e.g., system timer, NVIC, or system control block…)
    - Can change to privilege level via **SVC** instruction

Change Processor State

SuperVisor Call

  - *Privileged*
    - can use all the instructions and has access to all resources
    - can write to the **CONTROL** register to change the privilege level for Thread mode software

Stacks

	Full	Empty
Ascending	Full-ascending	Empty-ascending
Descending	Full-descending	Empty-descending

- The processor uses a **full** *descending* stack
  - **Full**: SP (stack pointer) holds the address of the last pushed item
  - *Descending*: Decrements the SP when push
- **2 Stacks**, 2 stack pointer registers
  - *Main stack* & *Process stack*

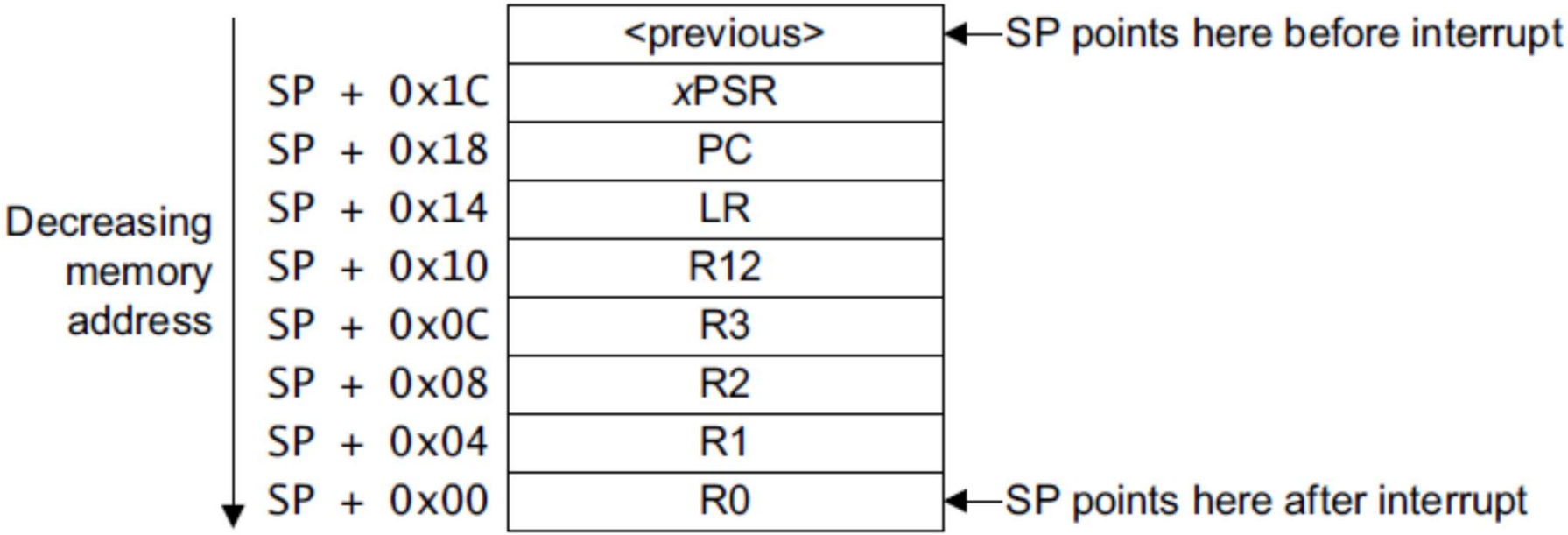
Determined by the **CONTROL** register

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged[a]	Main stack or process stack
Handler	Exception handlers	Always privileged	Main stack

In Handler mode, the processor always uses the main stack

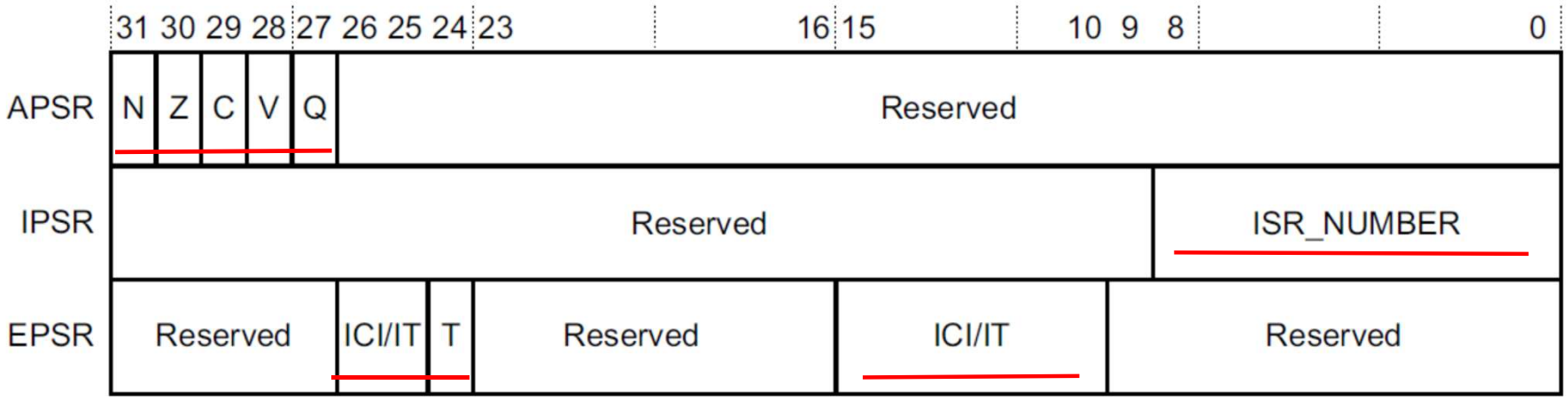
Stack Before/After an Exception Occurs

For CM3, **xPSR, PC (R15), R14, R12, R3-R0** are **saved by CPU** *on exception entry*!



Special-purpose Program Status Registers (xPSR)

- Application PSR (APSR)
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)



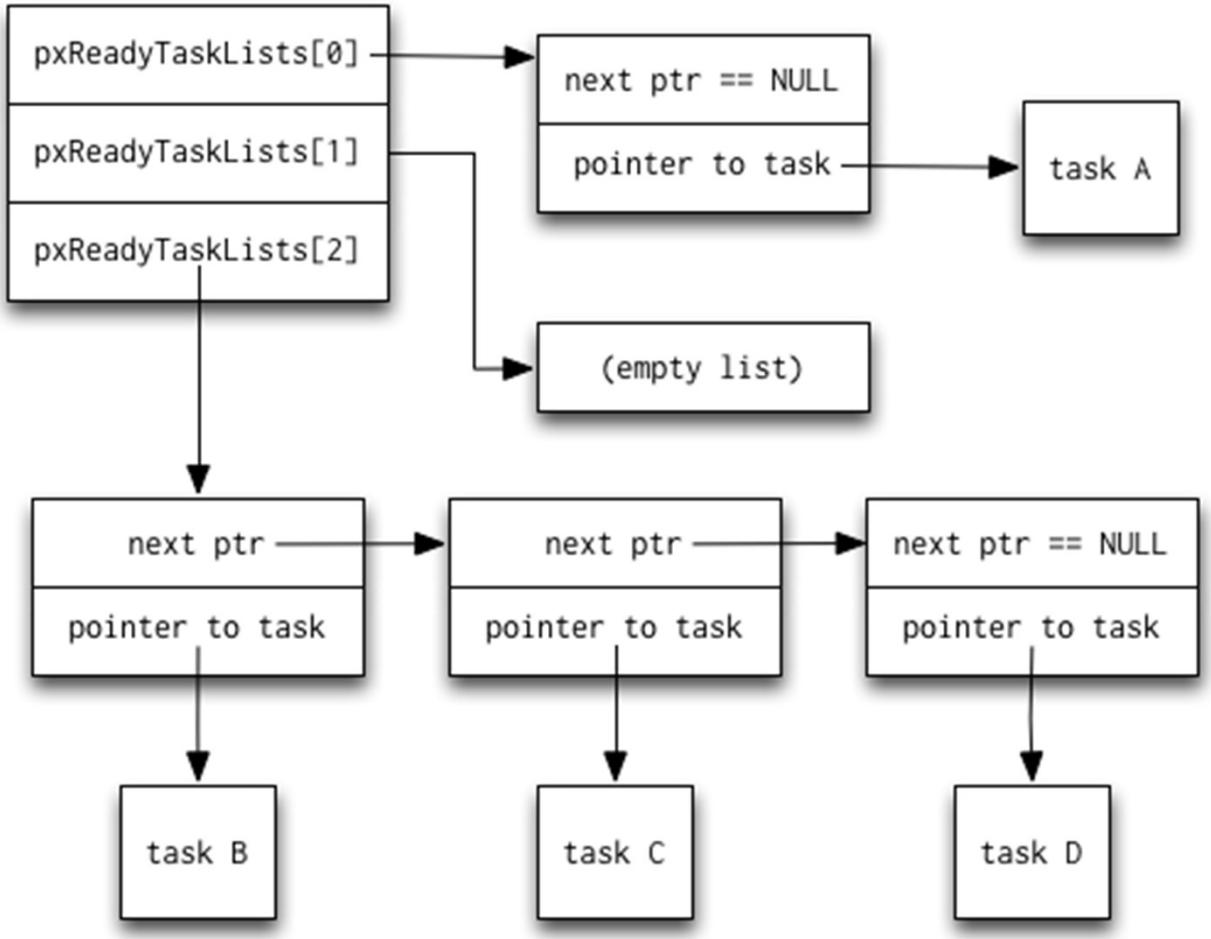
Scheduling and Ready List

- Ready list
  - keeps track of all tasks that are currently ready to run

static xList **pxReadyTasksLists**[configMAX_PRIORITIES];

- *pxReadyTasksLists[i] chains all the ready tasks with priority i*

Basic View of the Ready List



Select the Highest Priority Task

The version ***without*** port-optimized task selection

A *high watermark* of the top ready priority; raised when the priority of the task inserted into the ready list is higher than the watermark

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()                                  \
    {                                                                       \
        UBaseType_t uxTopPriority = uxTopReadyPriority;                     \
                                                                            \
        /* Find the highest priority queue that contains ready tasks. */   \
        while (listLIST_IS_EMPTY(&(pxReadyTasksLists[uxTopPriority])))      \
        {                                                                   \
            configASSERT(uxTopPriority);                                    \
            --uxTopPriority;                                                \
        }                                                                   \
                                                                            \
        /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of  \
        the same priority get an equal share of the processor time. */      \
        listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB, &(pxReadyTasksLists[uxTopPriority]));  \
        uxTopReadyPriority = uxTopPriority;                                 \
    } /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

selects the highest-priority ready task and puts it in the pxCurrentTCB

Select the Highest Priority Task

The version *with* port-optimized task selection

The bitmap

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()                                  \
    {                                                                       \
        UBaseType_t uxTopPriority;                                          \
                                                                            \
        /* Find the highest priority list that contains ready tasks. */    \
        portGET_HIGHEST_PRIORITY (uxTopPriority, uxTopReadyPriority);       \
        configASSERT( listCURRENT_LIST_LENGTH( &(pxReadyTasksLists[uxTopPriority])) > 0);  \
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[uxTopPriority]));  \
    } /* taskSELECT_HIGHEST_PRIORITY_TASK() */


#define portGET_HIGHEST_PRIORITY(uxTopPriority, uxReadyPriorities)          \
        uxTopPriority = (31UL - (uint32_t)ucPortCountLeadingZeros((uxReadyPriorities)))


__attribute__((always_inline)) static inline uint8_t ucPortCountLeadingZeros(uint32_t ulBitmap) {
    uint8_t ucReturn;

    __asm volatile ("clz %0, %1" : "=r" ( ucReturn ) : "r" ( ulBitmap ) : "memory" );
    return ucReturn;
}
```

Context Switch (for ARM CM3)

- taskSELECT_HIGHEST_PRIORITY_TASK() is called by vTaskSwitchContext() [tasks.c]
  - Both functions do not perform the real context switch job
  - Real context switch code is typically in ***portable*** directory
    - Since context switch is processor dependent
  - In ARM Cortex M3 (CM3)
    - vTaskSwitchContext() is called by the **xPortPendSVHandler**(), the handler for handling **pendSV exception**
    - In CM3, pendSV is typically used for implementing context switch

- How to issue/pend a pendSV exception?
  - Set a bit in the interrupt control register
    - portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
    - The register is mapped to a predefined memory address (memory mapped register)

Pend SV Handler

```
void xPortPendSVHandler( void )
{
    __asm volatile
    (
    "   mrs r0, psp                         \n"  /* Get the process stack pointer, described later*/
    "   isb                                 \n"  /* Instruction Synchronization Barrier, flush pipeline */
    "                                       \n"
    "   ldr r3, pxCurrentTCBConst           \n"  /* Get the location of the current TCB. */
    "   ldr r2, [r3]                        \n"
    "                                       \n"
    "   stmdb r0!, {r4-r11}                 \n"  /* Push the remaining registers into the process stack. */
    "   str r0, [r2]                        \n"  /* Save new stack top to the 1st TCB member, see TCB structure*/
    "                                       \n"
    "   stmdb sp!, {r3, r14}                \n"  /* Push r3 (cur. TCB addr) and r14 (return addr) into the main stack*/

    "   mov r0, %0                          \n"  /* Raise basepri to configMAX_SYSCALL_INTERRUPT_PRIORITY */
    "   msr basepri, r0                     \n"  /*  as above,  described later */
    "   bl vTaskSwitchContext               \n"
```

We need to use R3 and R14 after the call of vTaskSwitchContext().
So, we save them on the main stack.

xPSR, PC (R15), R14, R12, R3-R0 are saved by CPU on exception entry!

Call vTaskSwitchContext()

Pend SV Handler (cont.)

```
"    mov r0, #0                                  \n“  /* restore basepri to 0 */
"    msr basepri, r0                             \n"
"    ldmia sp!, {r3, r14}                        \n“  /* restore r3 and r14 */
“                                                \n”
"                                               \n“  /* Restore the context, including the critical nesting count */
"    ldr r1, [r3]                                \n“  /* Get the new value of pxCurrentTCB (for new highest priority task) */
"    ldr r0, [r1]                                \n"  /* 1st item in pxCurrentTCB is the process stack top of the task */
"    ldmia r0!, {r4-r11}                         \n"  /* Pop the registers to restore the context */
"    msr psp, r0                                 \n“  /* Restore the process stack */
"    isb                                         \n“
“    bx r14                                      \n”  /* return from interrupt (r14 = EXC_RETURN)*/
"                                                \n"
"    .align 4                                    \n"
"pxCurrentTCBConst: .word pxCurrentTCB           \n"
::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
);
}
```

CM3 Stacks

	Full	Empty
Ascending	Full-ascending	Empty-ascending
Descending	Full-descending	Empty-descending

- The processor uses a **full** *descending* stack
  - SP (stack pointer) holds the address of the last pushed item
  - Decrements the SP when push
- **2 Stacks**, 2 stack pointer registers
  - *Main stack* & *Process stack*

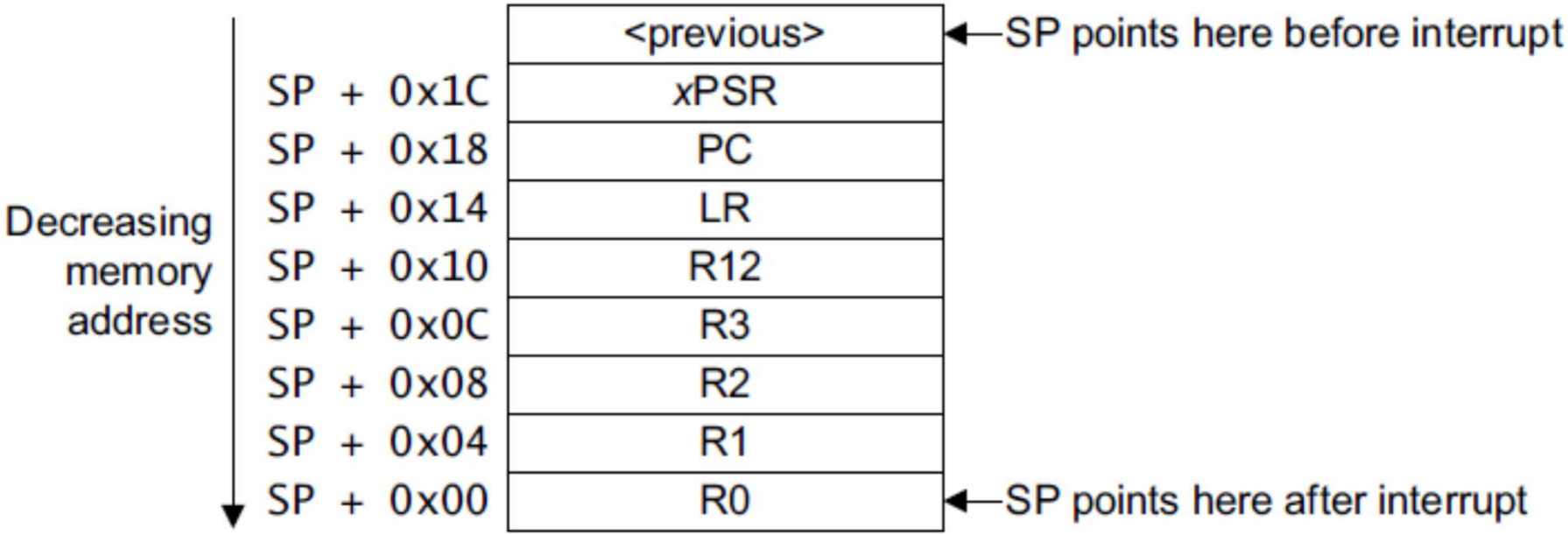
Determined by the **CONTROL** register

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged[a]	Main stack or process stack
Handler	Exception handlers	Always privileged	Main stack

In Handler mode, the processor always uses the main stack

Stack Before/After an Exception Occurs

For CM3, **xPSR, PC (R15), R14, R12, R3-R0** are saved by CPU ***on exception entry***!

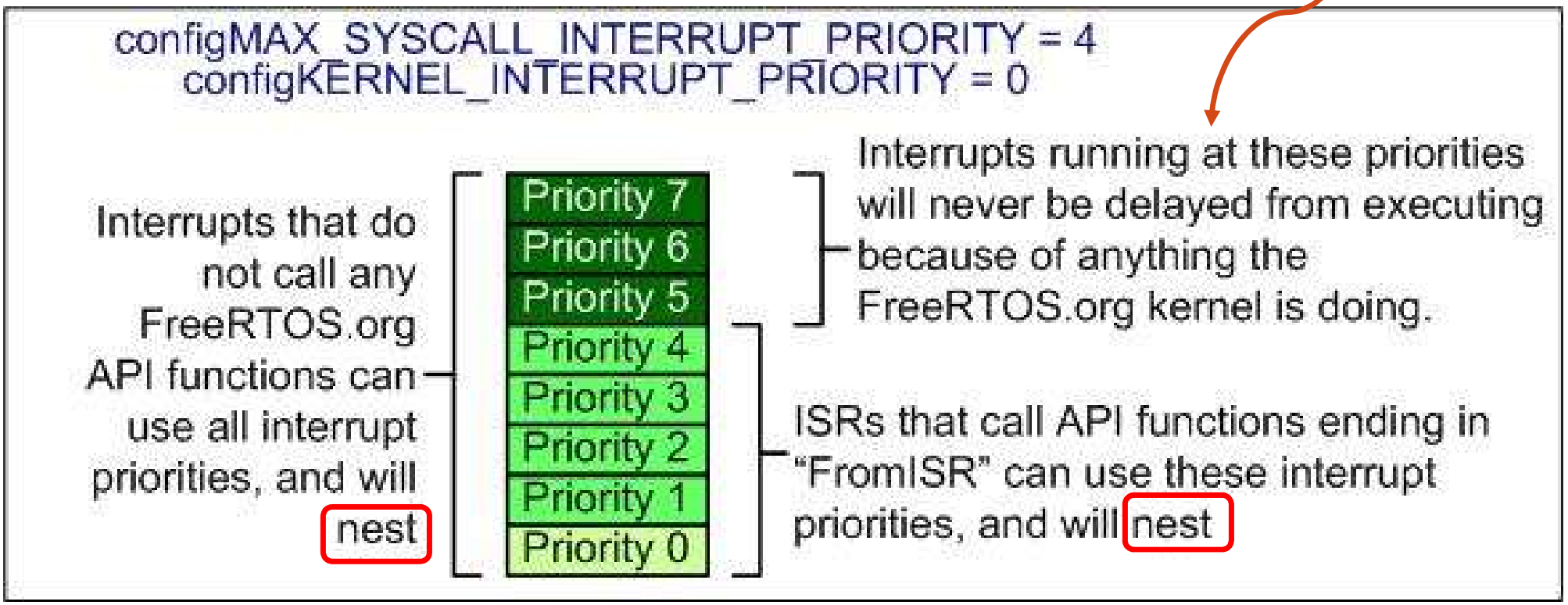


Interrupt Priority Configurations

- An ISR may call *xxx*FromISR() FreeRTOS APIs
  - Typically, an ISR executes on an (pre-configured) interrupt priority
  - Q: Will an ISR/interrupt be pended when FreeRTOS enters its critical section?
  - Q: How to set the interrupt priority for a given ISR/interrupt?
- 2 priorities settings provided by FreeRTOS
  - **configKERNEL_INTERRUPT_PRIORITY**
    - the interrupt priority used by the RTOS kernel itself
    - should be set to the ***lowest*** priority
  - **configMAX_SYSCALL_INTERRUPT_PRIORITY**
    - the ***highest*** interrupt priority from which *xxx*FromISR() functions can be called
    - Renamed as **configMAX_API_CALL_INTERRUPT_PRIORITY** in newer ports
    - Some ports do not have this configuration!!!
      - In those ports, ISRs that call *xxx*FromISR() functions should be in the priority of **configKERNEL_INTERRUPT_PRIORITY**

An Example

FreeRTOS kernel does not completely disable interrupts, even inside critical sections



Source: *https://www.freertos.org/a00110.html#kernel_priority*

CM3 Exception Types

- **Reset**
  - invoked on power up or a warm reset
  - reset handler runs as privileged execution in the Thread mode
    - Different from the other exception handlers
- **NMI**
  - has the **highest** priority other than **reset**
  - cannot be masked or prevented from activation by any other exception
  - can be signaled by a peripheral or triggered by software
- **HardFault**
  - occurs because of an error during exception processing
  - has **higher** priority than any ***configurable-priority*** exception

CM3 Exception Types

- **MemManage**
  - occurs because of a memory protection related fault
    - e.g., execute an instruction in a **Execute Never (XN)** memory region
- **BusFault**
  - occurs because of a memory related fault for an instruction/data memory transaction
  - might be from an error detected on a memory bus
- **UsageFault**
  - a fault related to instruction execution
    - e.g., undefined instruction, unaligned access, error on exception return, division by zero…

CM3 Exception Types

- **SVCall**
  - triggered by the **SVC (Supervisor Call)** instruction
  - Used to implement the system call mechanism in an OS environment
- **PendSV**
  - pending system-level service
  - typically used to for **context switches** when no other exception is active
- **SysTick**
  - generated when the system timer reaches zero
  - can be used for system ticks in an OS
- **Interrupt (IRQ)**
  - an exception signaled by a peripheral, or generated by a software

Properties of CM3 Exceptions

Exception number[a]	Exception type	Priority	Vector address or offset[b]	Activation
1	Reset	-3, the highest	0x00000004	Asynchronous
2	NMI	-2	0x00000008	Asynchronous
3	HardFault	-1	0x0000000C	-
4	MemManage	Configurable[c]	0x00000010	Synchronous
5	BusFault	Configurable[c]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	UsageFault	Configurable[c]	0x00000018	Synchronous
7-10	Reserved	-	-	-

Properties of CM3 Exceptions

Exception number[a]	Exception type	Priority	Vector address or offset[b]	Activation
11	SVCall	Configurable[c]	0x0000002C	Synchronous
12-13	Reserved	-	-	-
14	PendSV	Configurable[c]	0x00000038	Asynchronous
15	SysTick	Configurable[c]	0x0000003C	Asynchronous
16	Interrupt (IRQ)	Configurable[d]	0x00000040[e]	Asynchronous