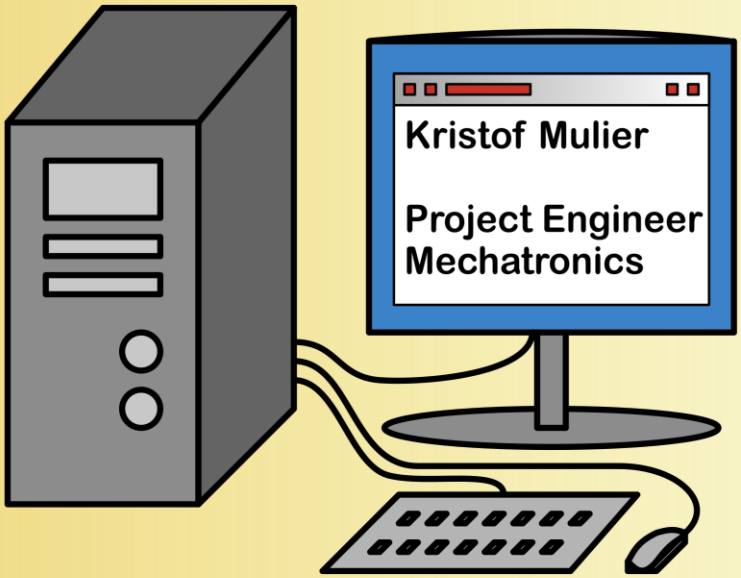
Introduction to:

- the STM32F7 microcontroller

- FreeRTOS



kristof.mulier@telenet.be

Contents

DISCLAIMER

I have written this book in my spare time, and present it free of charge “as is”. I do not warrant that the material in this book is error free. I will not be liable for any damages or injury, including but not limited to, direct or indirect, special or consequential damages that result from the use of, or the inability to use, the materials in this book. You warrant that you will use any information provided herein at your own risk and will not use the information in any way which will expose me to liability for patent infringement.

I have used company logos at certain spots in this book as a mere reference (for example to refer to ARM Holdings, the designer of the microcontroller core, and STMicroelectronics, the designer of the microcontroller). I am however not affiliated to any of these companies.

For any questions, please mail me:

kristof.mulier@telenet.be

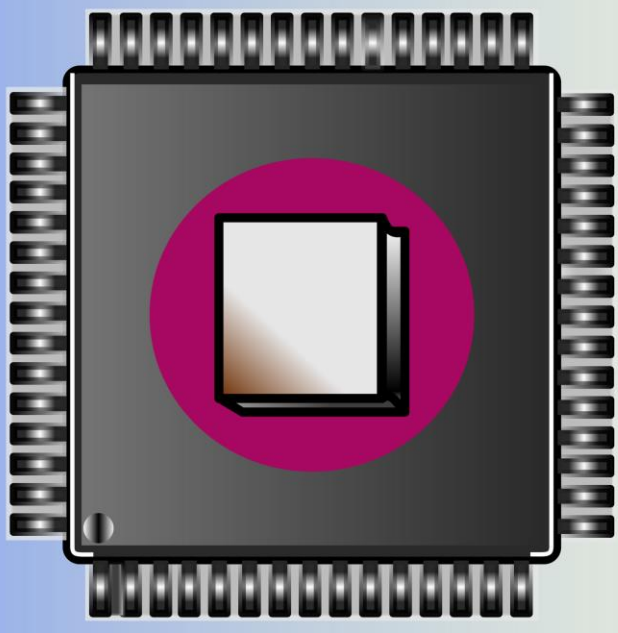
Kind regards,

The author

The STM32F7 microcontroller

Chapter 1

Documentation overview



The first and foremost challenge I faced when getting started with this microcontroller is finding out where to find all the documentation. There are so many companies and organizations involved that contribute to the final chip. The compiler, assembler and linker for your C-code is provided by GNU (the ARM engineers actively cooperate with GNU to maintain the toolchain). The processor core is designed by ARM. The silicon vendor STMicroelectronics designs the actual chip based on the ARM core and adds a set of peripherals. Both ARM and STMicroelectronics provide a library to interface with the chip hardware: CMSIS and ST HAL respectively. STMicroelectronics also delivers the programmer to flash your code onto the chip. And you might need a real-time operating system, provided by yet another company.
All these companies and organizations provide a set of documentation for their products. Therefore this chapter aims to give a complete overview to get you started. Click on the green boxes on the right to get redirected to the web URL.
PLEASE INFORM ME IF THE LINKS DO NOT WORK :-)

1.1 GNU Toolchain

GNU C lib
http://www.gnu.org/software/libc/manual/pdf/libc.pdf

GNU make
http://www.gnu.org/software/autoconf/manual/autoconf.pdf
http://www.gnu.org/software/automake/manual/automake.pdf
http://www.gnu.org/software/make/manual/make.pdf

GNU compiler
http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html
https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc.pdf

GNU assembler
https://sourceware.org/binutils/docs-2.25/as/index.html

GNU linker
https://sourceware.org/binutils/docs-2.25/ld/index.html

1.2 ARM

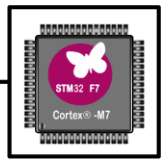
Cortex-M7
http://infocenter.arm.com/help/topic/com.arm.doc.dui0646b/DUI0646B_cortex_m7_dgug.pdf
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489d/DDI0489D_cortex_m7_trm.pdf
https://silver.arm.com/download/Unspecified/BX500-DA-10290-r0p0-00rel0/cortex_m7_software_developers_errata_notice_r0_v4.pdf

ARMv7-M arch
https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR580-DA-70000-r0p0-05rel0/DDI0403E_B_armv7m_arm.pdf
http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

ARM ABI
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

CMSIS specs
https://silver.arm.com/download/download.tm?pv=2615860

1.3 STMicroelectronics

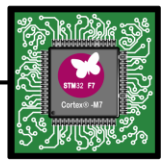


Progr. manual
PM0253
http://www.st.com/web/en/resource/technical/document/programming_manual/DM00237416.pdf

Datasheet
http://www.st.com/web/en/resource/technical/document/datasheet/DM00166116.pdf
http://www.st.com/web/en/resource/technical/document/errata_sheet/DM00145382.pdf

Ref. manual
RM0385
http://www.st.com/web/en/resource/technical/document/reference_manual/DM00124865.pdf

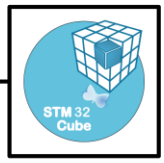
App. notes
http://www.st.com/stonline/stappl/resourceSelector/app?page=fullResourceSelector&doctype=application_note&SeriesID=1858
> AN2606: System mem boot mode
> AN2867: Oscillator design
> AN4661: Hardware design
> AN4676: Peripheral interconnections
> AN4731: STCube firmware examples



Board

Discovery board
http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/LN1848/PF261641#

Nucleo board
http://www.st.com/web/catalog/tools/FM116/CL1620/SC959/SS1532/LN1847/PF262631



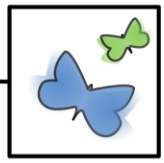
STM32CubeMX

STM32CubeMX
http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/SS1533/PF259242

STM32CubeF7
http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1897/PF261909
http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1897/PF263050

X-Cube 32F7PERF

STM32F7 performance software expansion for STM32Cube
http://www.st.com/web/en/catalog/tools/PF262415



Various manuals

User manuals
http://www.st.com/stonline/stappl/resourceSelector/app?page=fullResourceSelector&doctype=user_manual&SubClassID=961

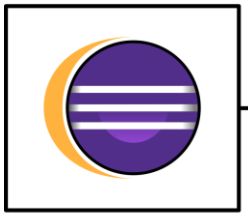
App notes
http://www.st.com/stonline/stappl/resourceSelector/app?page=fullResourceSelector&doctype=application_note&SubClassID=961

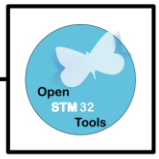
> AN4667: STM32F7 Series system architecture and performance

ST-Link
http://www.st.com/web/catalog/tools/FM146/CL1984/SC724/SS1677/PF251168?sc=internet/evalboard/product/251168.jsp

ST-Link Utility
http://www.st.com/web/en/catalog/tools/PF258168

1.4 IDE





OpenSTM32 Community
http://www.openstm32.org/HomePage

1.5 FreeRTOS

User Manual

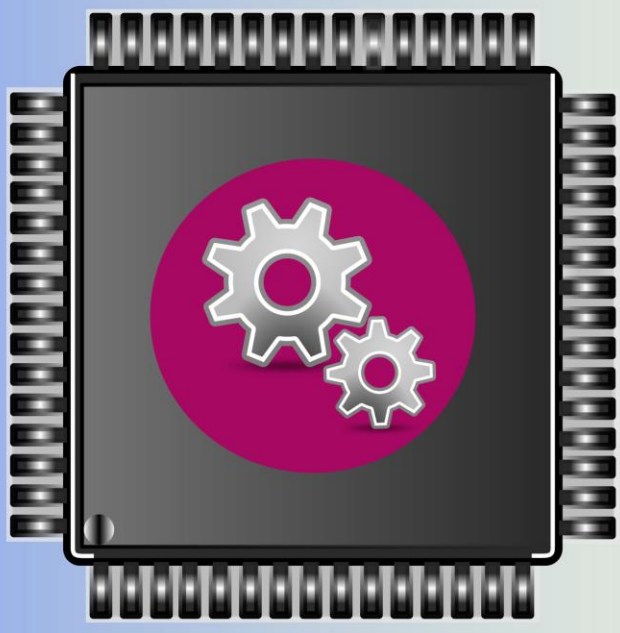
http://shop.freertos.org/FreeRTOS_Tutorial_Book_Generic_Cortex_M3_Edition_p/pdf_cortex-m3_tutorial_book.htm

Reference Manual

http://shop.freertos.org/FreeRTOS_Reference_Manual_PDF_p/pdf-reference-manual.htm

Chapter 2

Processor architecture and Memory map



2.1 Load store machine

ARM uses the load/store architecture. A load/store architecture only allows memory to be accessed by load and store operations. All operands for instructions (like ADD, SUB, ...) need to be present in CPU registers.

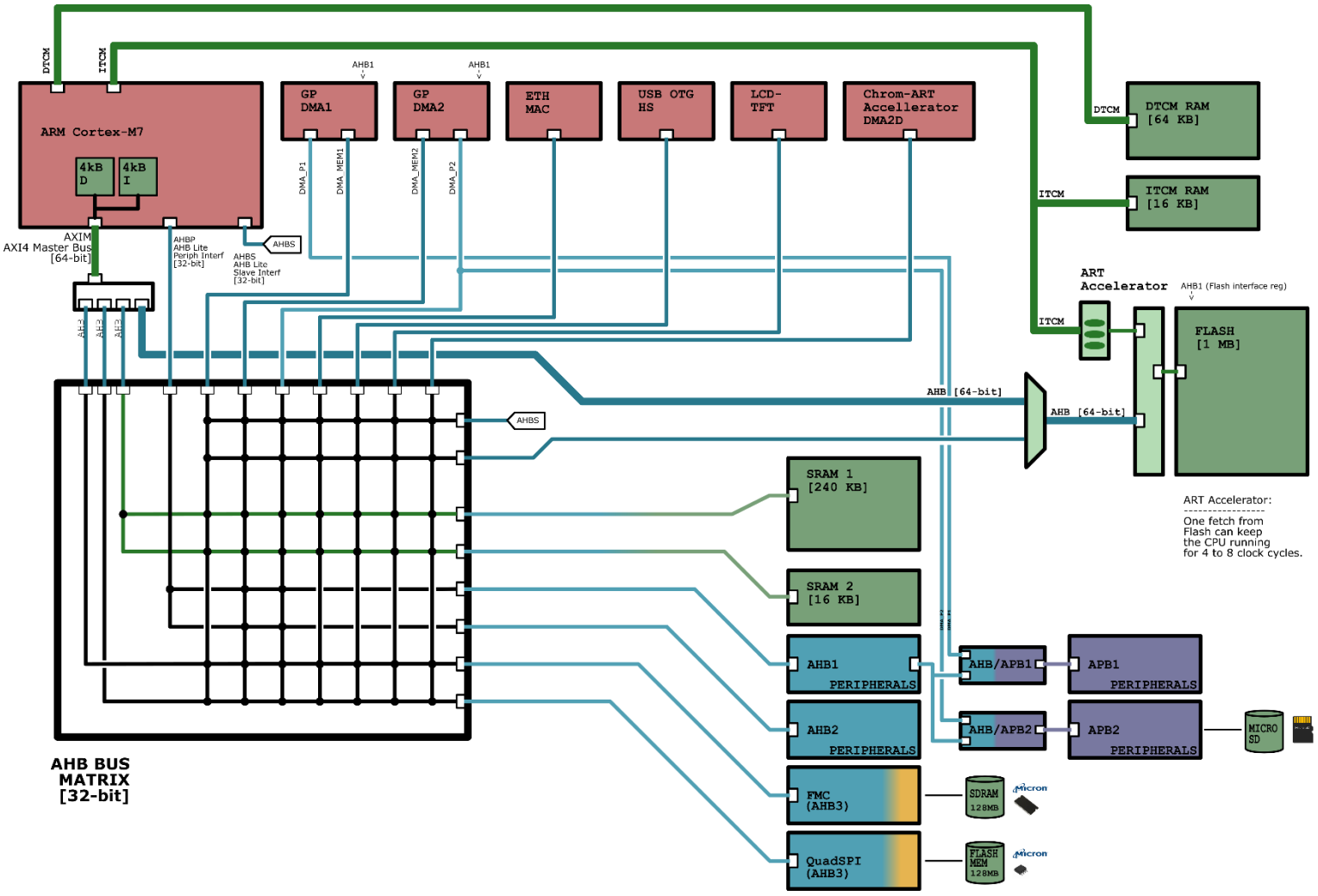
UNDER CONSTRUCTION

2.2 Calling convention

UNDER CONSTRUCTION

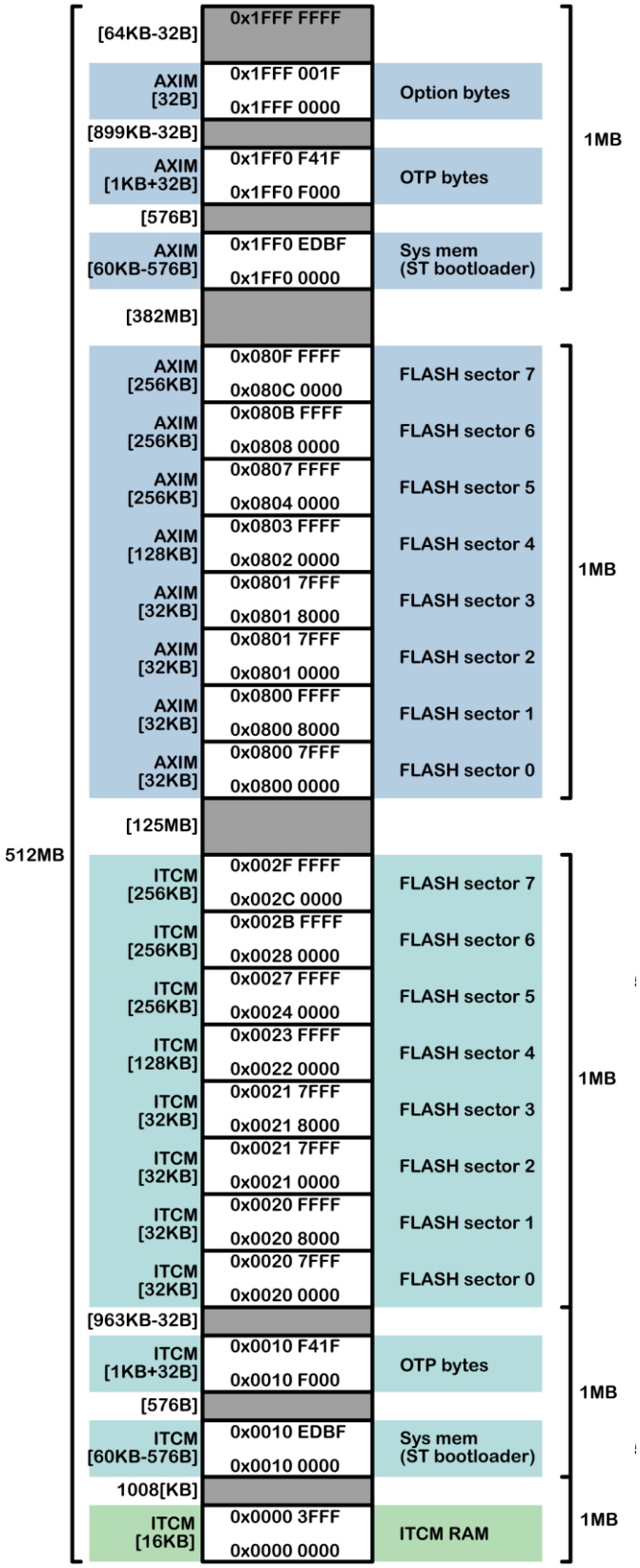
2.1 CPU architecture

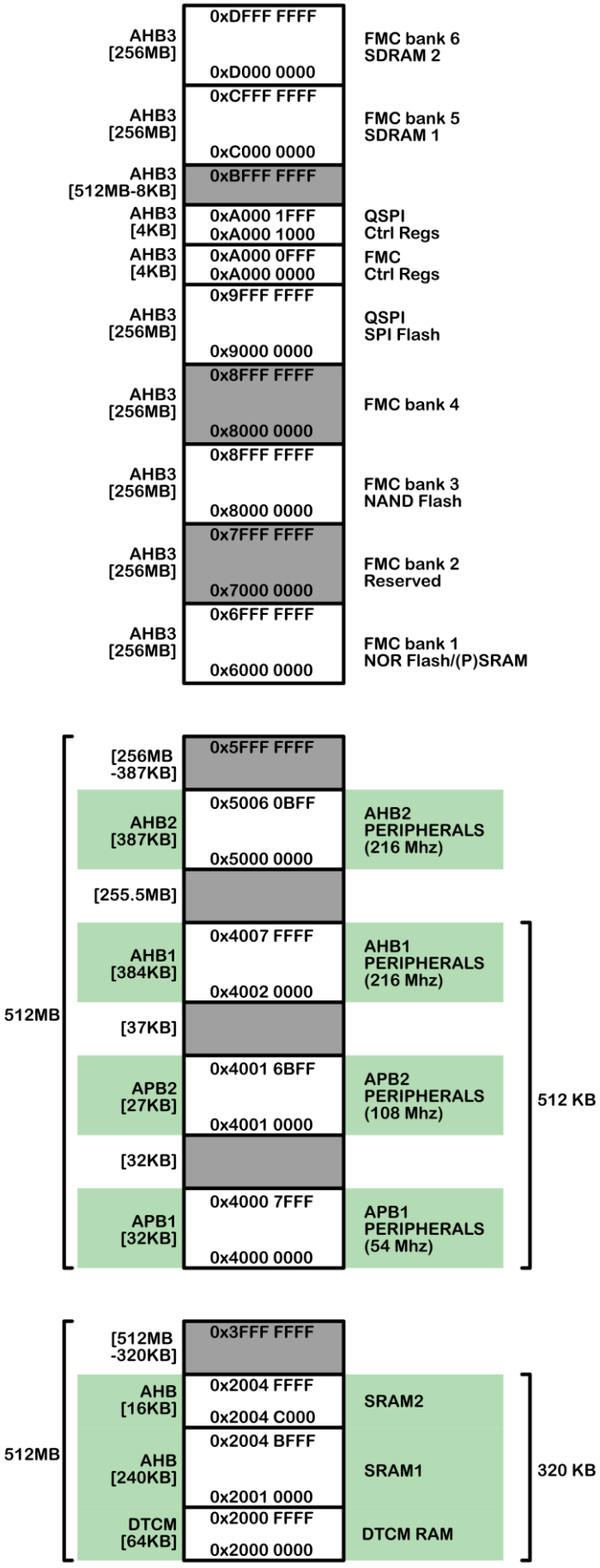
UNDER CONSTRUCTION
I still need to write some more documentation. But at least, you can find some nice figures already ;-)



2.2 Memory map

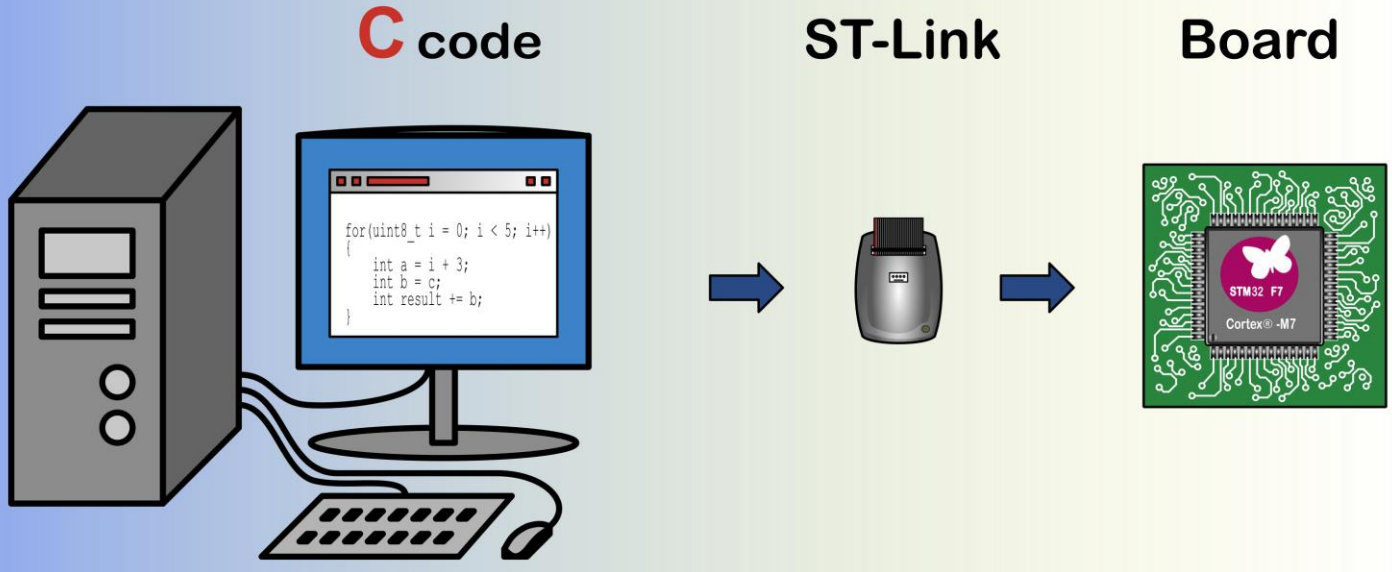
UNDER CONSTRUCTION




0xDFFF FFFF
AHB3 [256MB]
0xD000 0000
FMC bank 6 SDRAM 2
0xCFFF FFFF
AHB3 [256MB]
0xC000 0000
FMC bank 5 SDRAM 1
AHB3 [512MB-8KB]
0xBFFF FFFF
AHB3 [4KB]
0xA000 1FFF
0xA000 1000
QSPI Ctrl Regs
AHB3 [4KB]
0xA000 0FFF
0xA000 0000
FMC Ctrl Regs
0x9FFF FFFF
AHB3 [256MB]
0x9000 0000
QSPI SPI Flash
0x8FFF FFFF
AHB3 [256MB]
0x8000 0000
FMC bank 4
0x8FFF FFFF
AHB3 [256MB]
0x8000 0000
FMC bank 3 NAND Flash
0x7FFF FFFF
AHB3 [256MB]
0x7000 0000
FMC bank 2 Reserved
0x6FFF FFFF
AHB3 [256MB]
0x6000 0000
FMC bank 1 NOR Flash/(P)SRAM
512MB
[256MB -387KB]
0x5FFF FFFF
AHB2 [387KB]
0x5006 0BFF
0x5000 0000
AHB2 PERIPHERALS (216 Mhz)
[255.5MB]
AHB1 [384KB]
0x4007 FFFF
0x4002 0000
AHB1 PERIPHERALS (216 Mhz)
[37KB]
APB2 [27KB]
0x4001 6BFF
0x4001 0000
APB2 PERIPHERALS (108 Mhz)
[32KB]
APB1 [32KB]
0x4000 7FFF
0x4000 0000
APB1 PERIPHERALS (54 Mhz)
512 KB
512MB
[512MB -320KB]
0x3FFF FFFF
AHB [16KB]
0x2004 FFFF
0x2004 C000
SRAM2
AHB [240KB]
0x2004 BFFF
0x2001 0000
SRAM1
DTCM [64KB]
0x2000 FFFF
0x2000 0000
DTCM RAM
320 KB

Chapter 3

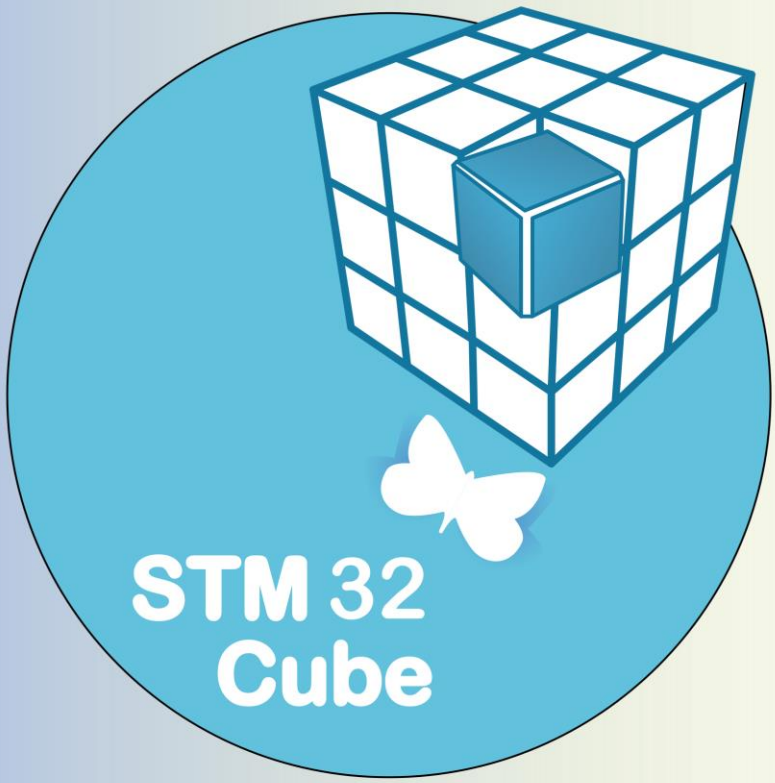
What you need to get started



UNDER CONSTRUCTION

Chapter 4

STM32CubeMX Code generation…



… and how to build it

4.1 STM32CubeMX Project – bare minimum code for STM32F7

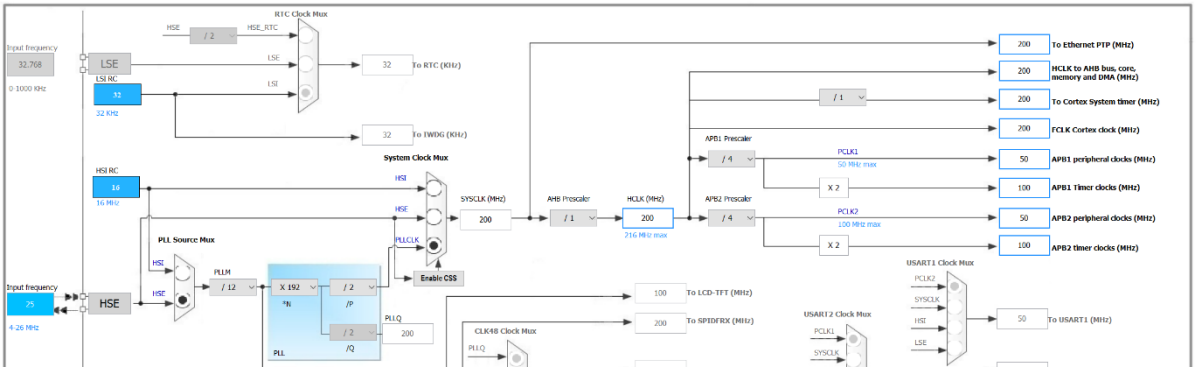
This chapter aims to investigate the generated code for an STM32F746NG microcontroller (the one used on the discovery board) in its most simple form. That is, all peripherals are disabled.

Open **STM32CubeMX > New Project**

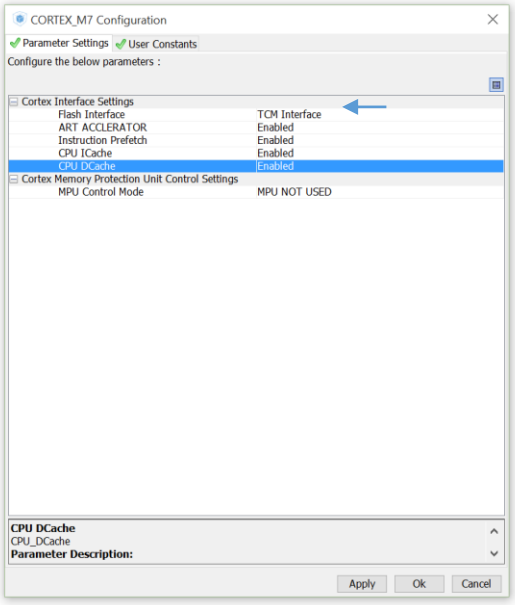
Next, select the tab "**MCU selector**". The other tab – Board selector – would generate much more files to accommodate for the other components on the board.

4.1.1 Settings for code generation

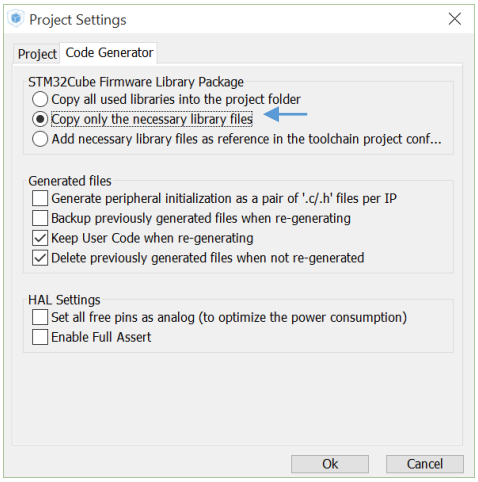
The clock is configured with an external oscillator:



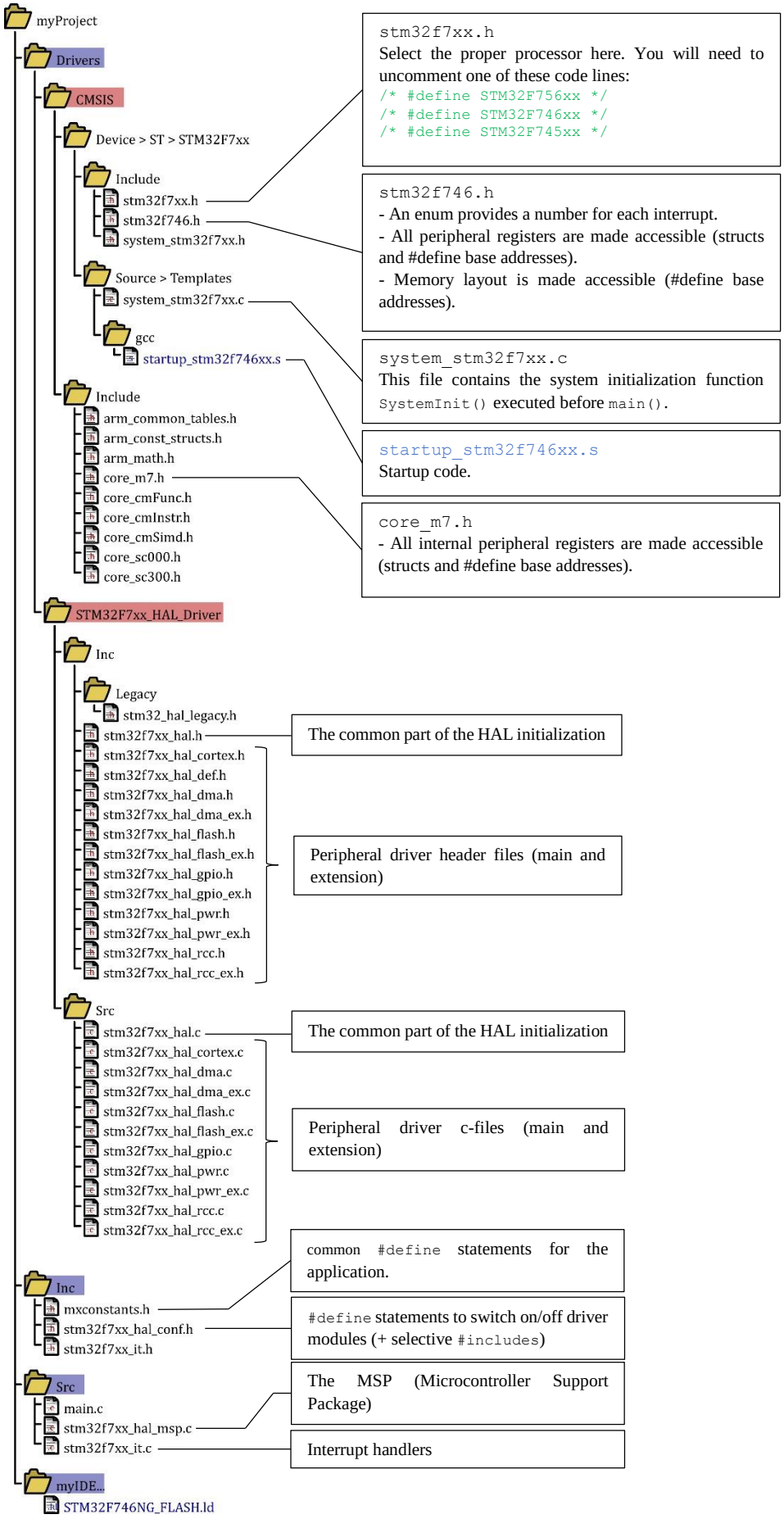
Another small adjustement to the default settings is shown in the following figure. The Flash interface is changed from AXI to TCM with ART ACCELERATOR enabled.



To limit the amount of generated source files to the bare minimum, check the box as shown below:



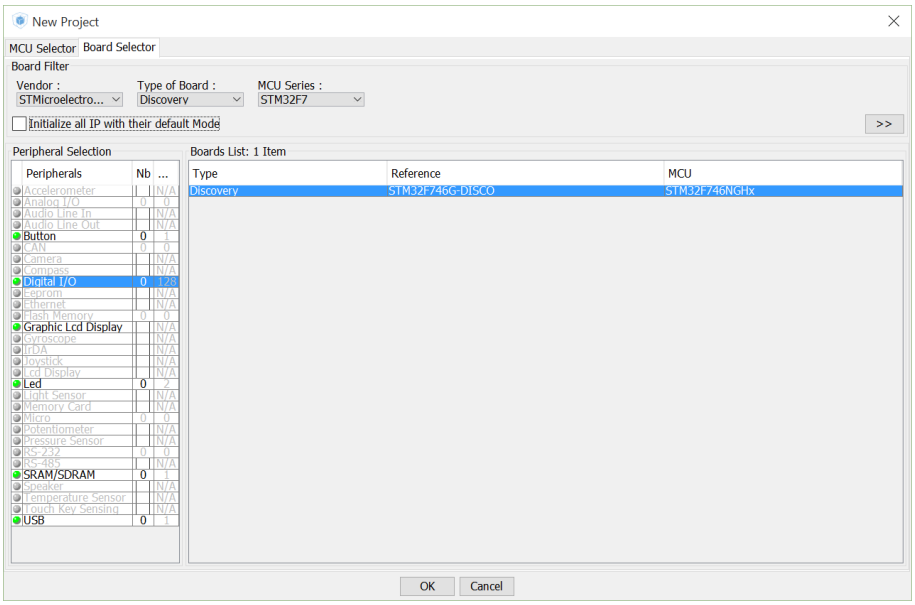
4.1.2 Overview of generated files



4.2 STMCubeMX Project – Code for Discovery board

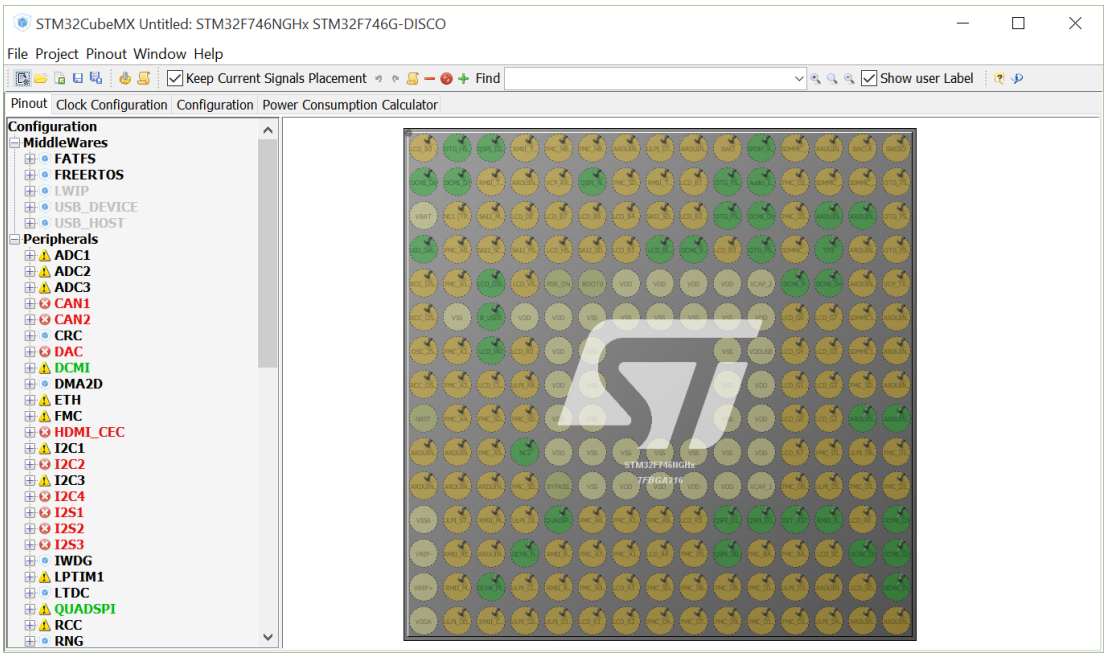
4.2.1 Settings for code generation

In this chapter we will generate a project for the Discovery Board. Open **STM32CubeMX > New Project**
Next, select the tab "**Board Selector**".

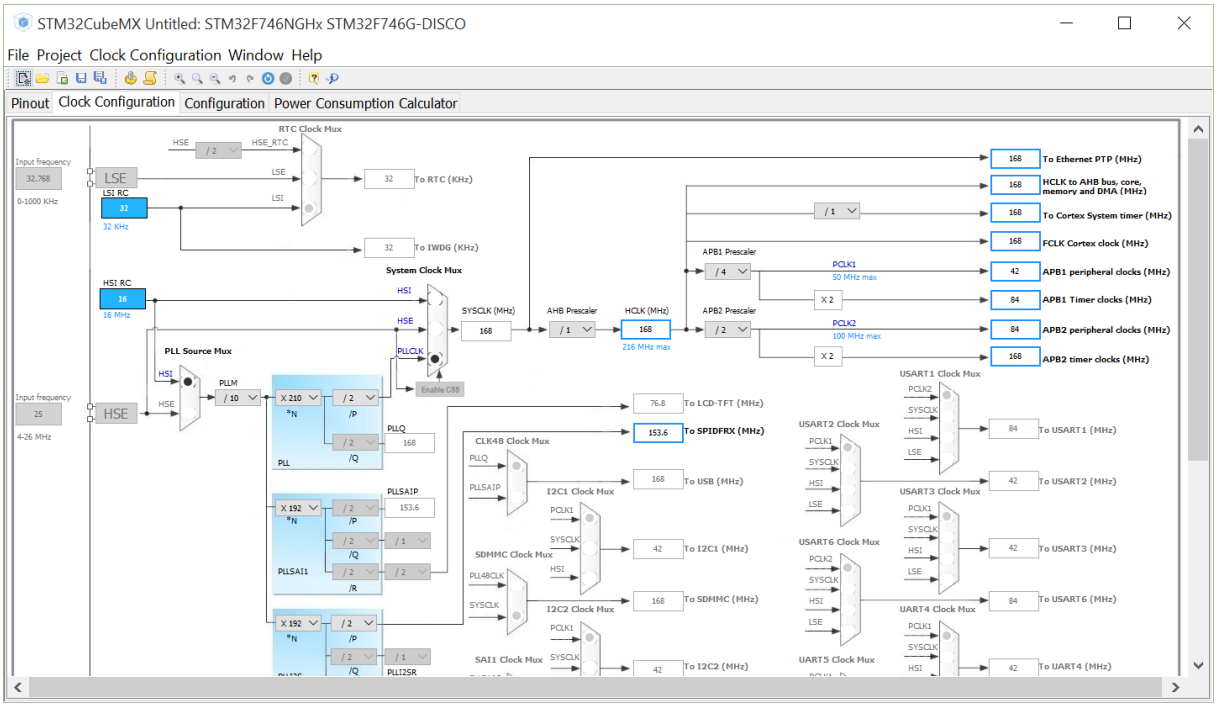


Normally all the configurations are correct. They should not be changed. The following figures show the configurations:

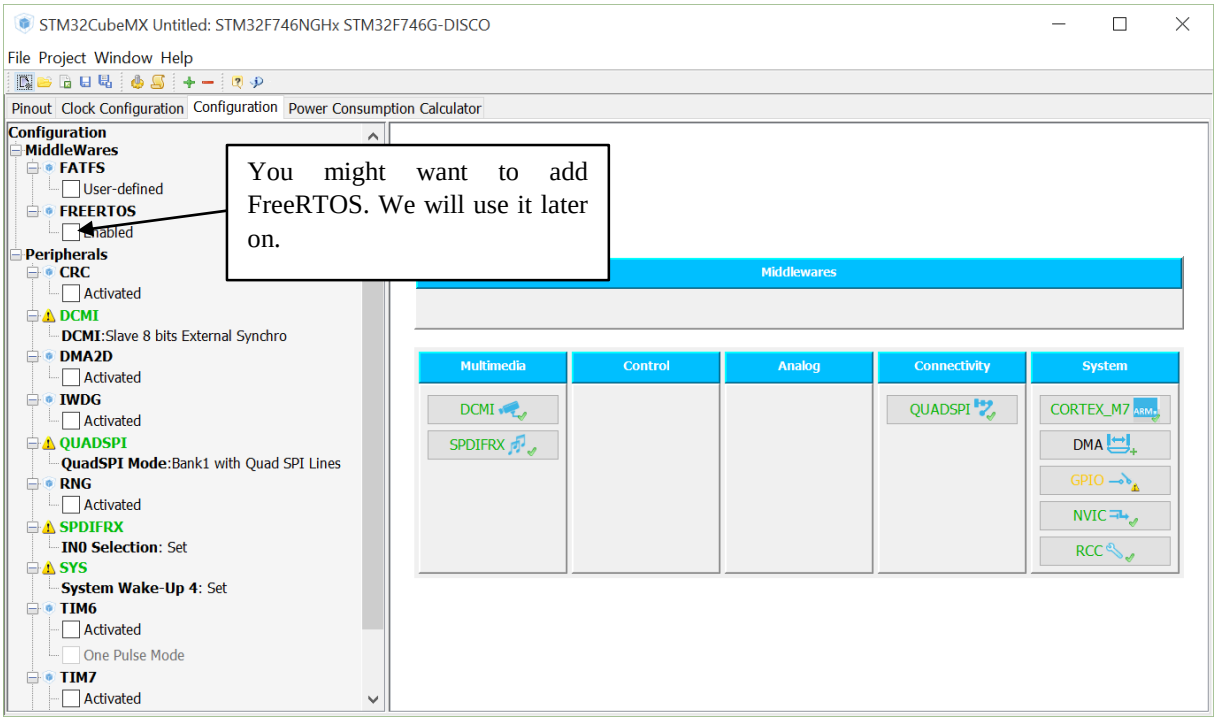
Pinout tab



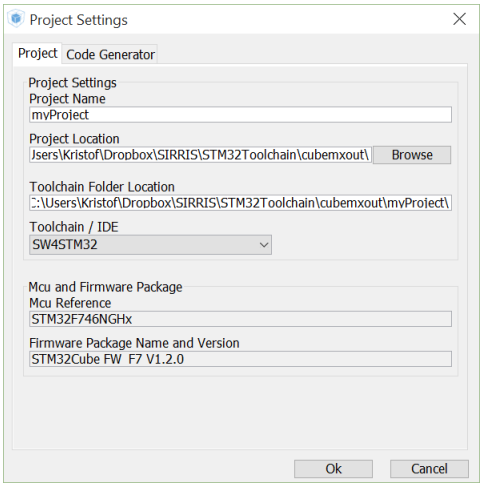
Clock Config tab

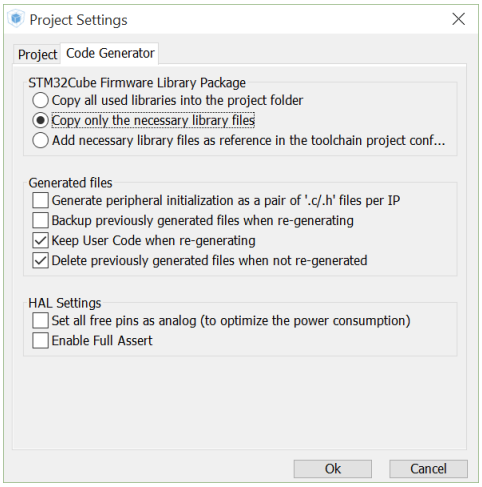


Configuration tab



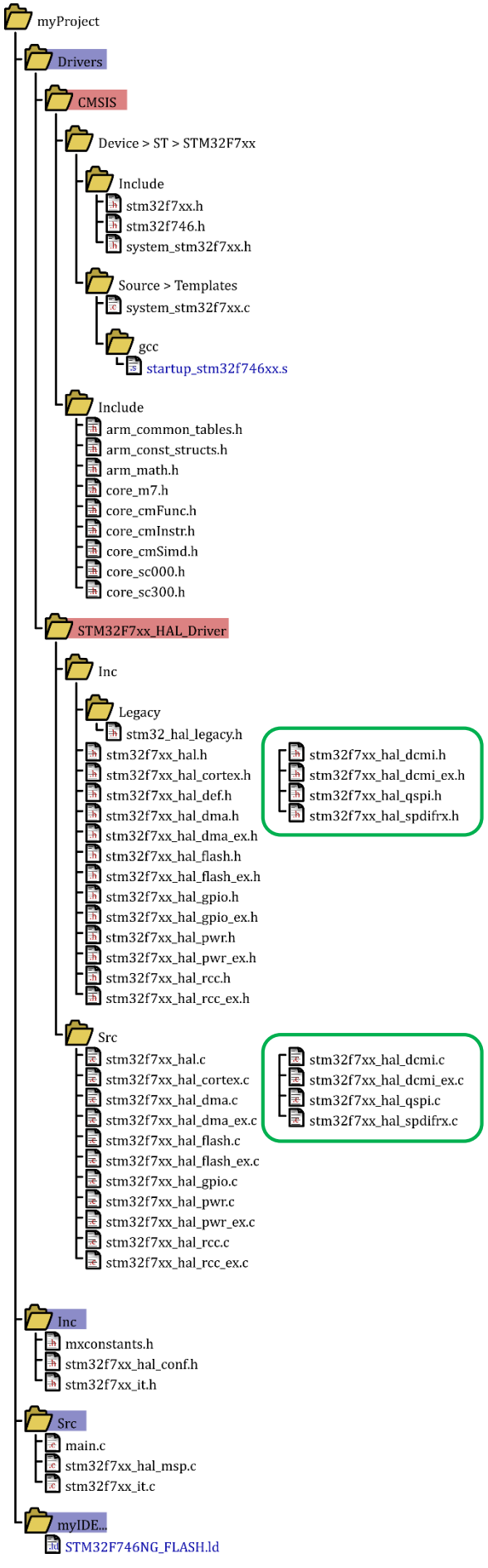
Click on **Project > Settings**, and the following pop-up menu appears. Choose a name for the project, and the desired Toolchain/IDE. We choose the SW4STM32 IDE – System Workbench for STM32.





Finally, click on **Project > Generate source code**.

4.2.2 Overview of generated files

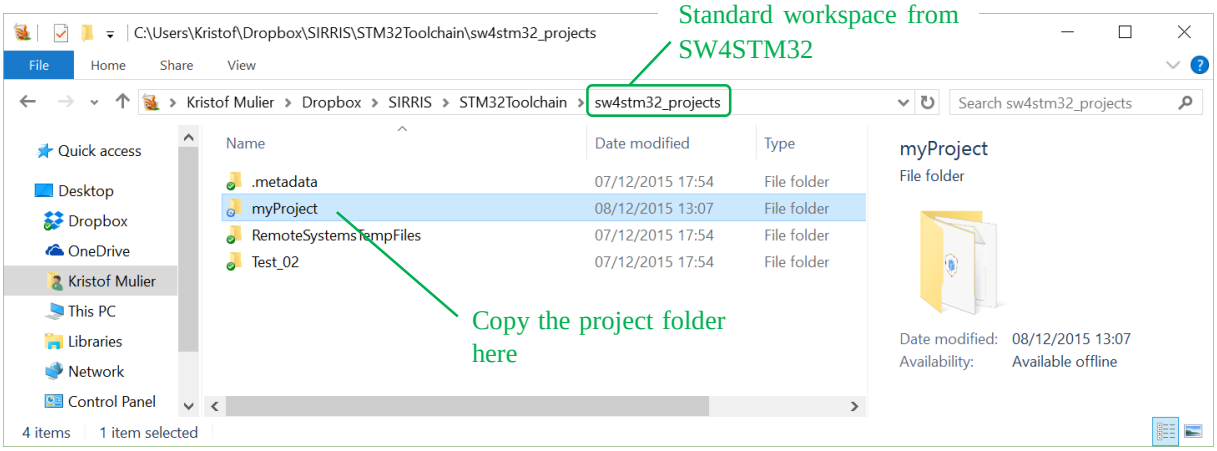


Note:
The files from FreeRTOS are not shown in this figure. We will dig into them later.

These are some extra files (compared to the bare minimum STM32F7 configuration) generated for some peripherals on the discovery board.

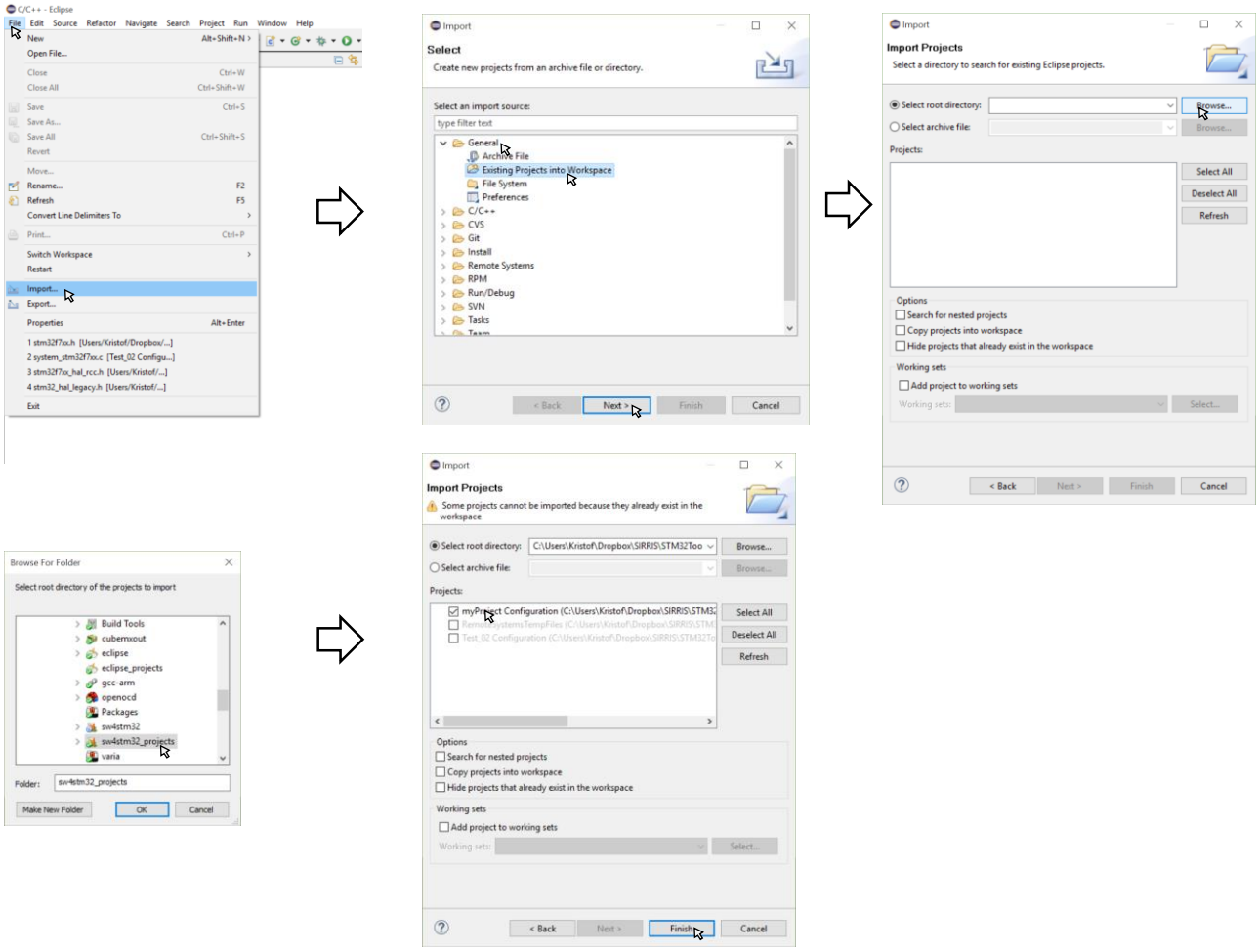
4.2.3 Import the generated code into SW4STM32

Suppose that the generated files are in the folder **C > ... > cubemxout > myProject**. Now copy the folder "**myProject**" to the standard workspace folder of your SW4STM32 IDE program (System Workbench for STM32).

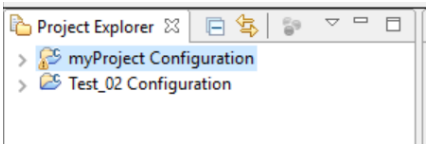


a. The Project Import

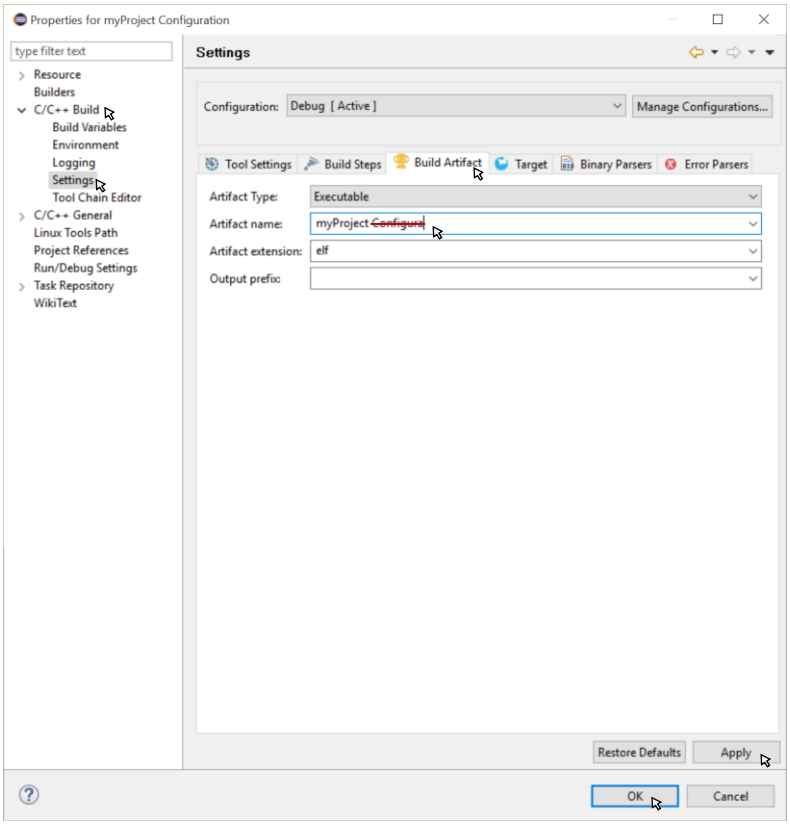
Now open the SW4STM32 IDE. Click on **File > Import**. A pop-up window appears in which you can define the import source. Click on **General > Existing Projects into Workspace > next**.



The **Project Explorer tab** from SW4STM32 shows all the projects you've got in the standard workspace folder. Notice that the new project "**myProject Configuration**" now appears in the list. For some strange reason, the expression "Configuration" is added to the name.



The official documentation from SW4STM32 advices not to change this name. The only thing you should do is open the properties window (right click on the project > Properties), click on **C/C++ Build > Settings > Build Artifact > Artifact name.** Delete the word "Configuration" there.



b. File overview in Project Explorer tab

Take a look at the **Project Explorer tab** on the left of the IDE. All the generated files can be viewed and edited here. But the hierarchical structure of the files has changed slightly. This change is only in the way that the IDE presents the files. The effective hierarchical structure – as one can look up in Windows Explorer Filesystem – has not changed at all.

Let us examine the files visible in the Project Explorer tab (see figure on next page):


C/C++ - myProject Configuration/Application/SW4STM32/startup_stm32f746xx.s - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
myProject Configuration
Includes
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32_projects/myProject/Drivers/CMSIS/Device/ST/STM32F7xx/Include
stm32f745xx.h
stm32f746xx.h
stm32f756xx.h
stm32f7xx.h
system_stm32f7xx.h
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32_projects/myProject/Drivers/CMSIS/Include
arm_common_tables.h
arm_const_structs.h
arm_math.h
core_cm0.h
core_cm0plus.h
core_cm3.h
core_cm4.h
core_cm7.h
core_cmFunc.h
core_cmInstr.h
core_cmSimd.h
core_sc000.h
core_sc300.h
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32_projects/myProject/Drivers/STM32F7xx_HAL_Driver/Inc
Legacy
stm32f7xx_hal_cortex.h
stm32f7xx_hal_dcmi_ex.h
stm32f7xx_hal_dcmi.h
stm32f7xx_hal_def.h
stm32f7xx_hal_dma_ex.h
stm32f7xx_hal_dma.h
stm32f7xx_hal_flash_ex.h
stm32f7xx_hal_flash.h
stm32f7xx_hal_gpio_ex.h
stm32f7xx_hal_gpio.h
stm32f7xx_hal_pwr_ex.h
stm32f7xx_hal_pwr.h
stm32f7xx_hal_qspi.h
stm32f7xx_hal_rcc_ex.h
stm32f7xx_hal_rcc.h
stm32f7xx_hal_spdifrx.h
stm32f7xx_hal.h
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32_projects/myProject/Drivers/STM32F7xx_HAL_Driver/Inc/Legacy
stm32_hal_legacy.h
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32_projects/myProject/Inc
mxconstants.h
stm32f7xx_hal_conf.h
stm32f7xx_it.h
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32/plugins/fr.ac6.mcu.externaltools.arm-none.win32_1.3.0.201507241045/tools/compiler/arm-none-eabi/include
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32/plugins/fr.ac6.mcu.externaltools.arm-none.win32_1.3.0.201507241045/tools/compiler/lib/gcc/arm-none-eabi/4.9.3/include
C:/Users/Kristof/Dropbox/SIRRIS/STM32Toolchain/sw4stm32/plugins/fr.ac6.mcu.externaltools.arm-none.win32_1.3.0.201507241045/tools/compiler/lib/gcc/arm-none-eabi/4.9.3/include-fixed
myProject Configuration/Debug
HEADER FILES
Application
SW4STM32
startup_stm32f746xx.s
User
main.c
stm32f7xx_hal_msp.c
stm32f7xx_it.c
Drivers
CMSIS
system_stm32f7xx.c
STM32F7xx_HAL_Driver
stm32f7xx_hal_cortex.c
stm32f7xx_hal_dcmi_ex.c
stm32f7xx_hal_dcmi.c
stm32f7xx_hal_dma_ex.c
stm32f7xx_hal_dma.c
stm32f7xx_hal_flash_ex.c
stm32f7xx_hal_flash.c
stm32f7xx_hal_gpio.c
stm32f7xx_hal_pwr_ex.c
stm32f7xx_hal_pwr.c
stm32f7xx_hal_qspi.c
stm32f7xx_hal_rcc_ex.c
stm32f7xx_hal_rcc.c
stm32f7xx_hal_spdifrx.c
stm32f7xx_hal.c
STM32F746NGHx_FLASH.ld
SOURCE FILES
/myProject Configuration/Application/User/main.c

4.2.4 Build the project: compile > assemble > link

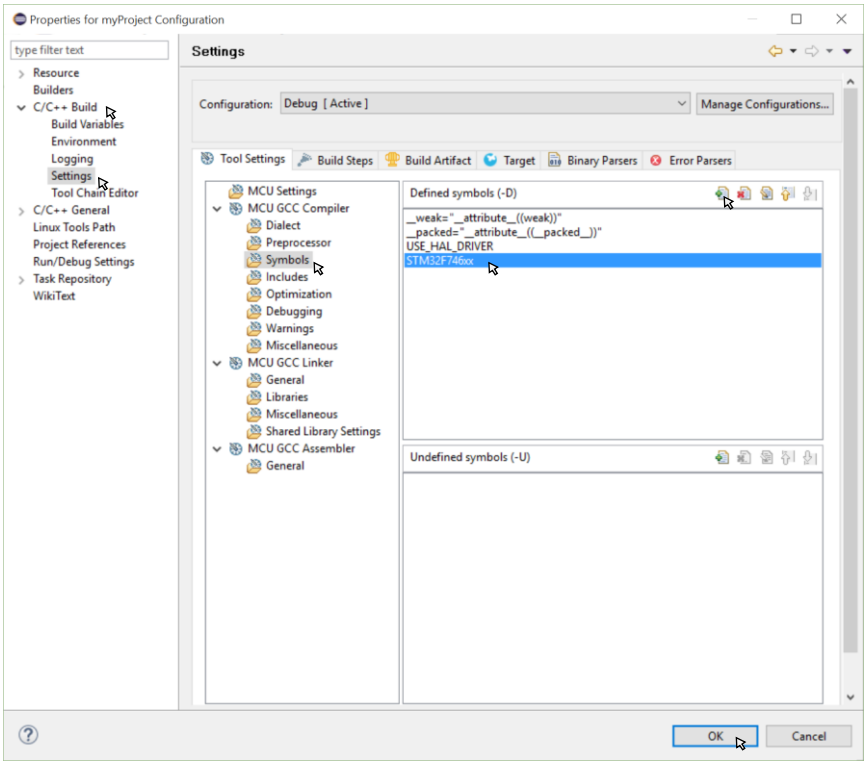
Check if the necessary global `#define` statements are done before starting the compilation. For example, you should incorporate the following statement to declare which specific STM32F7 microcontroller you're using:

```
#define STM32F746xx
```

One option is to open the `stm32f7xx.h` file and uncomment the line:

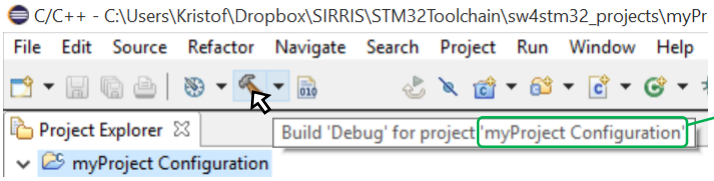
```
/* #define STM32F746xx */
```

Another option is to right click on the project (in the Project Explorer tab) and click **Properties > C/C++ Build > Settings > Tool Settings > Symbols > ...**



Note: on top of the Properties window, you can see the active **Configuration**. Some projects have a "debug" and a "release" configuration. The settings you are about to change are only valid for the active configuration. Switching to another configuration requires these steps to be redone.

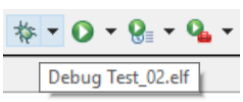
Now click on the  icon to build the project. This will invoke the **compiler > assembler > linker** toolchain on all your files. If you're unsure about what project gets actually built, hover with your mouse over the icon and wait a few seconds. A yellow box appears explaining which project (and its active configuration) gets built.



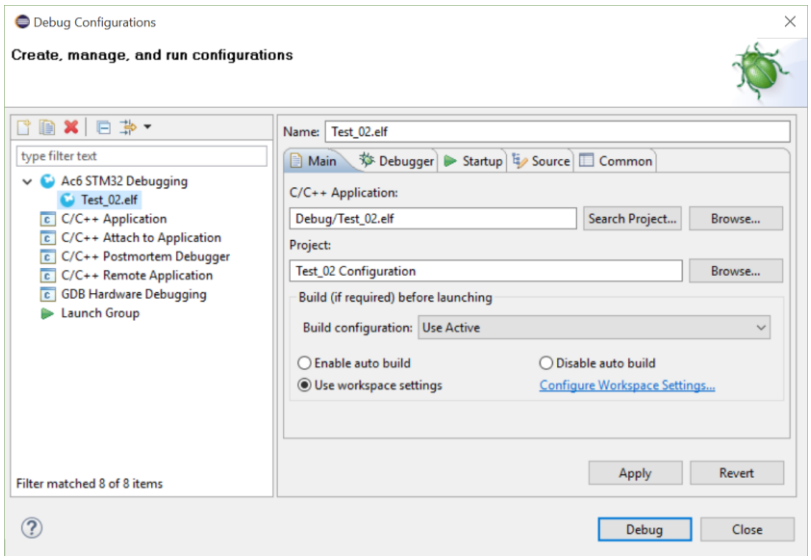
Yup, that's correct ☺

4.2.5 Debug the project

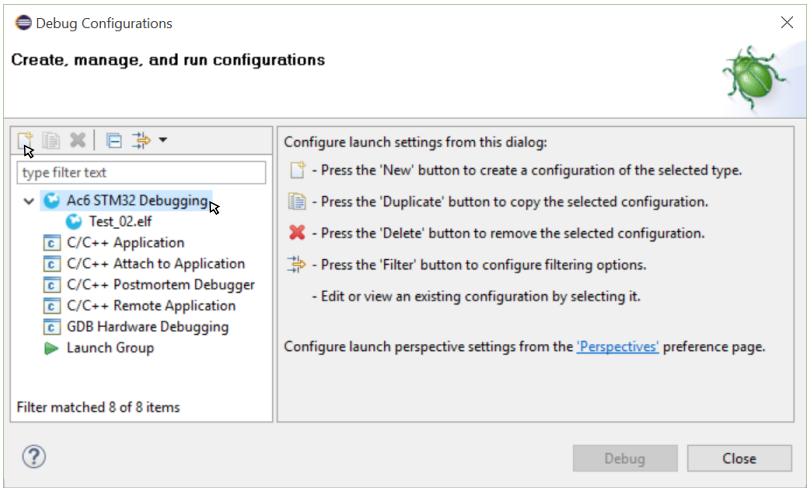
Hover with your mouse over the Debug icon. A yellow box appears to show you which `.elf` file will be used for the debug session. Click on the small black arrow next to the bug if the `.elf` file doesn't correspond to your project. A small menu appears, in which you should click **Debug Configurations**.

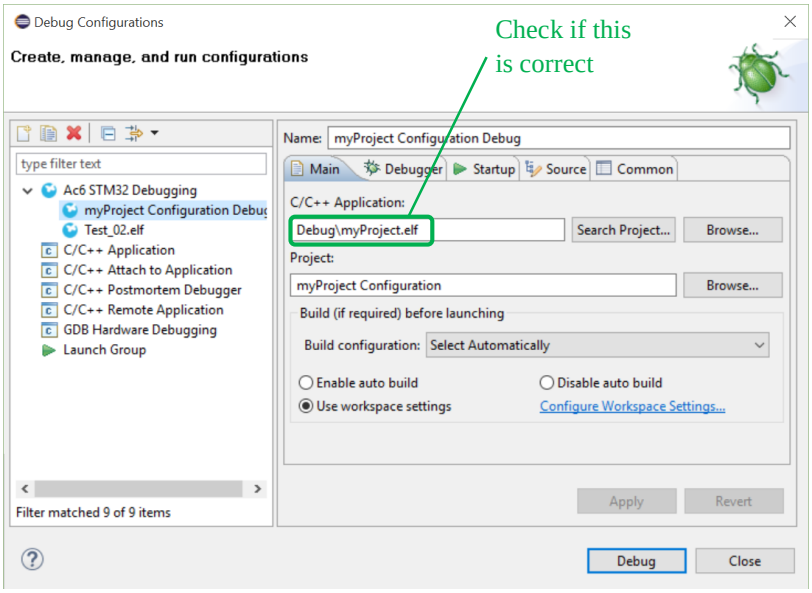


The **Debug Configurations** window appears. As you can see, the wrong `.elf` file `Test_02.elf` is selected.

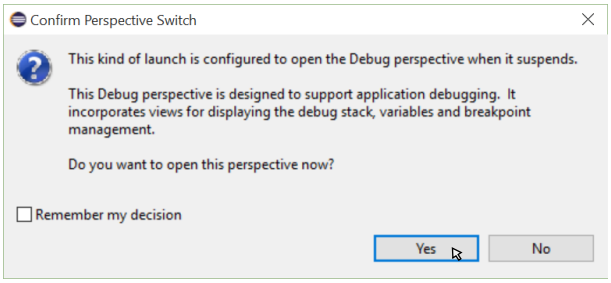


Click on `Ac6 STM32 Debugging` in the left pane. The right pane now displays some information about the possible options. Click on the 'New' button  in the top left corner. If everything goes well, a new Debug Configuration should appear, based on the active project.





Now that the **Debug Configuration** is correct, click on the  icon. The following pop-up window might appear:



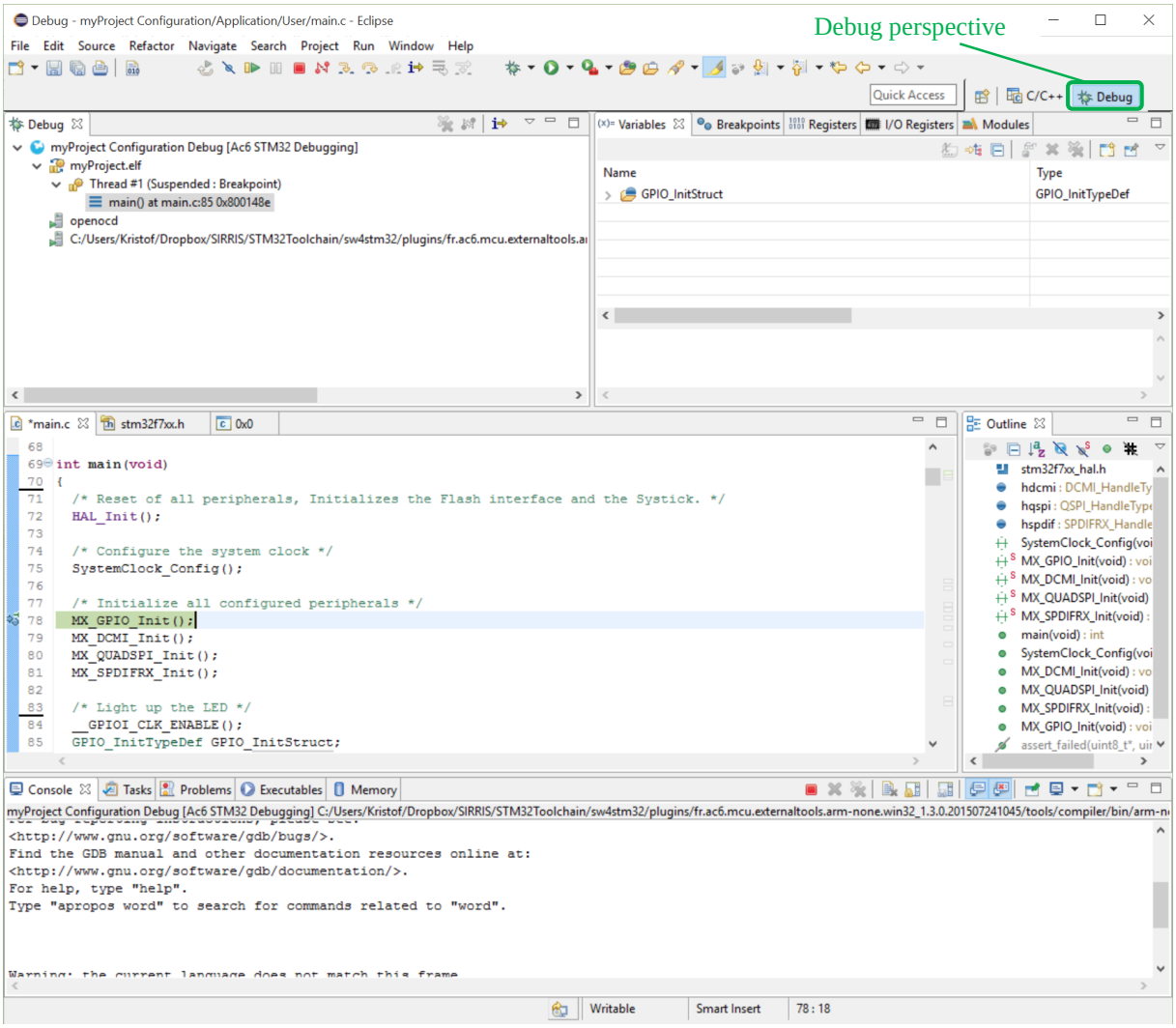
This dialog box basically asks to switch from the **C/C++ perspective** to the **Debug perspective** in your IDE. To go back to the coding perspective, click on C/C++ in the top right corner.
The debugger halts as soon as the code execution enters the `main()` function. Now you've got several buttons at your disposition:

 Resume execution

 Terminate Debug Session

 Step into function (F5)

 Step over function (F6)



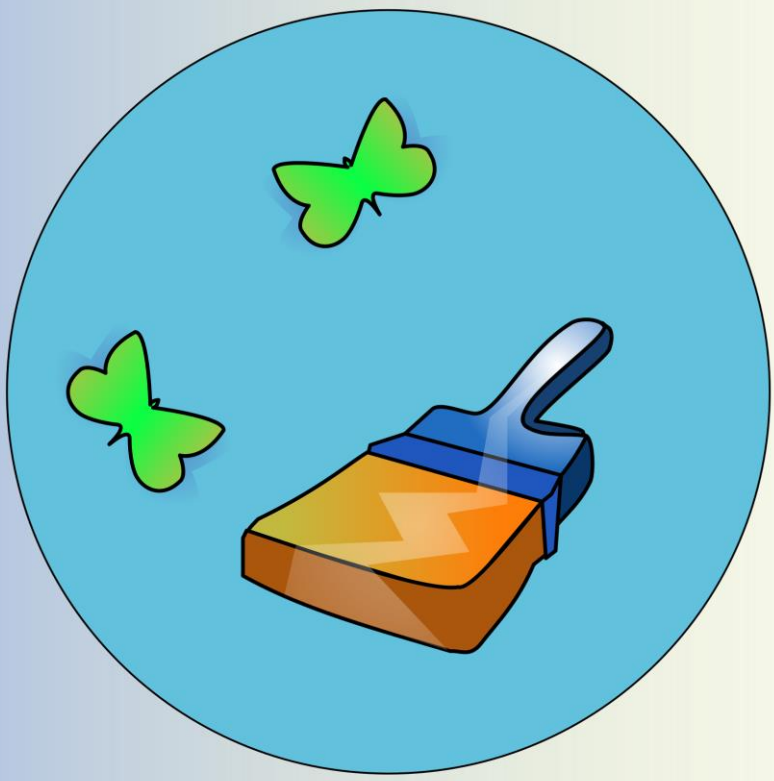
4.2.6 Flash the chip

UNDER CONSTRUCTION

You can flash your code to the chip by using the STLink Utility software. But that's a bit unpractical, since you need to leave your IDE and open another software.
But I'm working on a solution that enables you to do the flashing automatically, just by clicking a button in your IDE.
Stay tuned..

Chapter 5

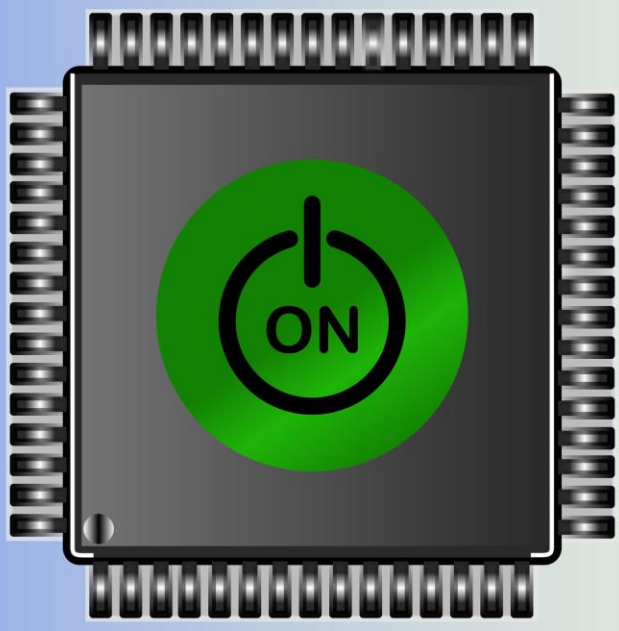
Code Cleanup



UNDER CONSTRUCTION

Chapter 6

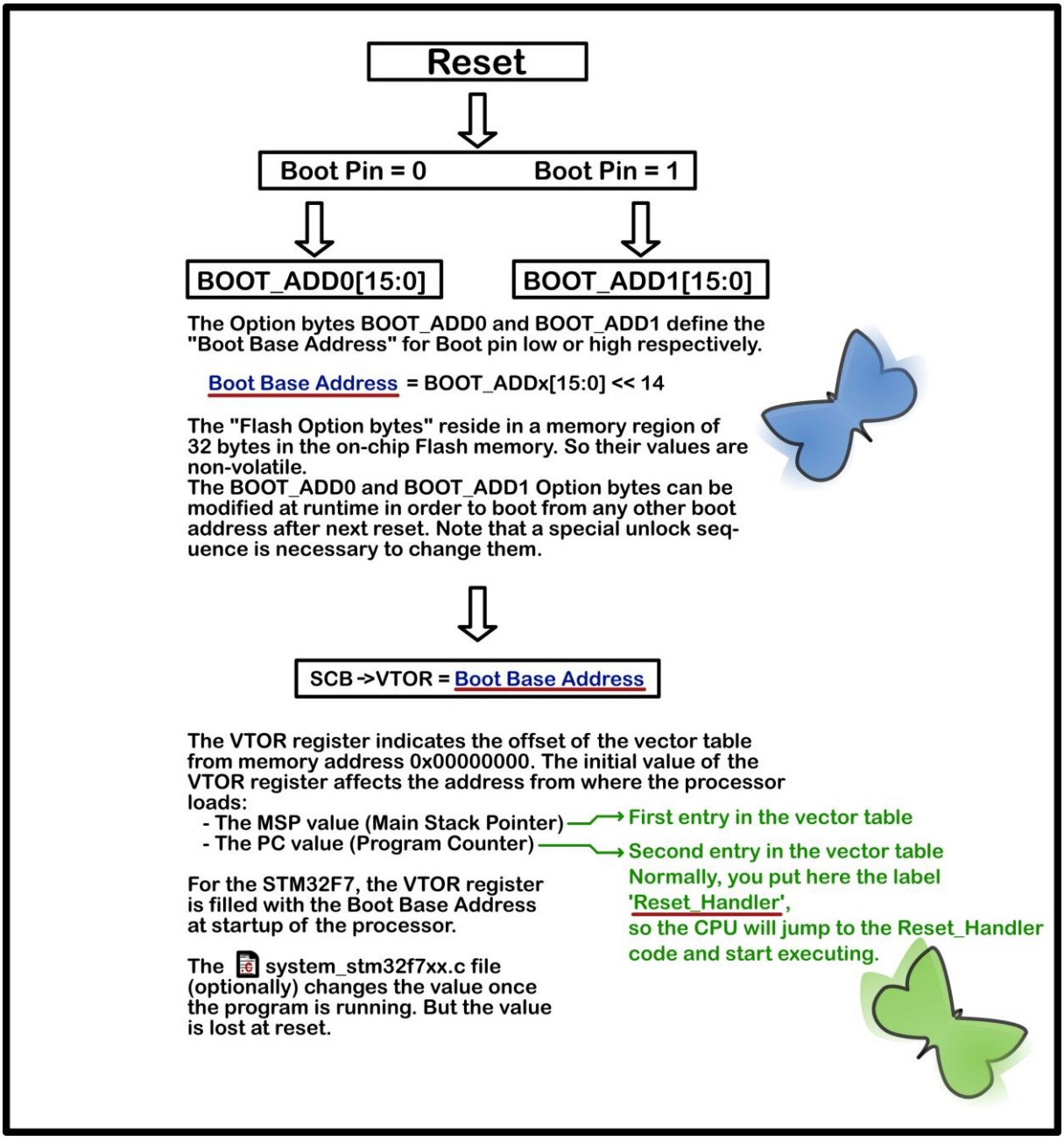
Startup sequence



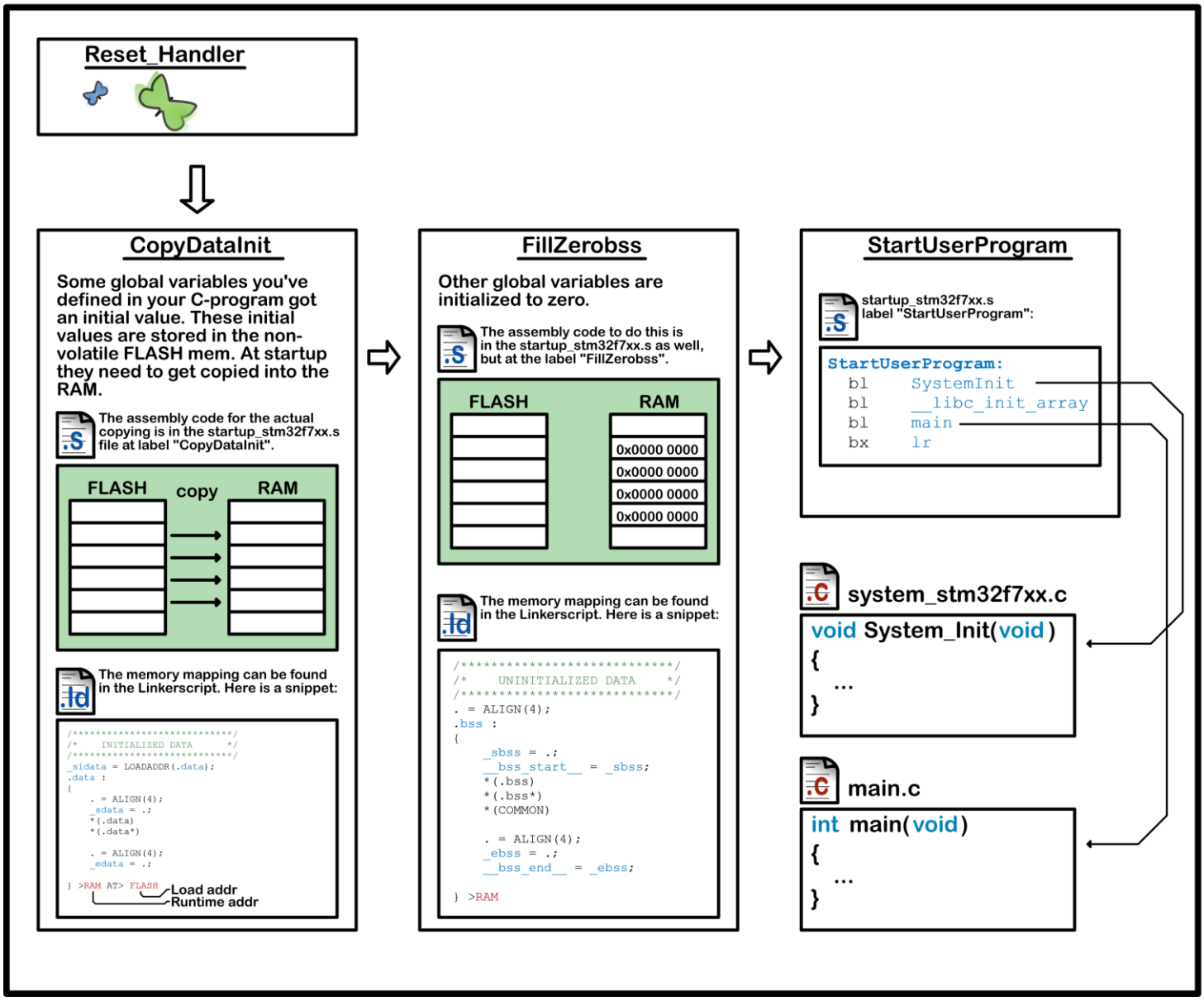
6.1 Startup sequence overview

The boot procedure of the microcontroller is not very clear. You need to combine the ARM core documentation and the STMicroelectronics Reference Manual to get an overview. I believe the startup sequence is as follows:

Upon reset the chip samples the Boot pin. If this pin is low, the chip will load its *Boot Base Address* from the `BOOT_ADD0[15:0]` register. If the pin is high it will be loaded from the `BOOT_ADD1[15:0]` register. Both registers reside in the "Flash Option bytes". This is a memory region of 32 bytes in the on-chip Flash memory. So their values are non-volatile. In other words, their values can be anything the user program has written to these registers before the reset occured.
Note: A left shift operation `<<14` is performed on the `BOOT_ADDx[15:0]` register when loading its value into the *Boot Base Address*.
Finally this *Boot Base Address* gets loaded into the `SCB->VTOR` register. This register informs the CPU where to find the *interrupt vector table*. The CPU loads the first entry from the vector table into its SP (Stack Pointer) and the second entry into its PC (Program Counter). The second entry normally holds the location of the `Reset_Handler` code. So that is where the CPU starts executing.



You can find the code from the `Reset_Handler` in the `startup_stm32f7xx.s` file. The `Reset_Handler` refers to a few other subroutines before entering the `main()` routine. The main routine should never exit. The figure below gives an overview:



6.2 startup_stm32f7xx.s file

```
/* ------------------------------------------------------------------------------------------*/
/*                            ARM specific assembler directives                              */
/* ------------------------------------------------------------------------------------------*/
/*                                                                                           */
  .syntax unified   /* option 'divided': the ARM and THUMB instructions have their          */
                    /* own separate syntaxes.                                               */
                    /* option 'unified': ...                                                */

  .cpu cortex-m7    /* Select the target processor. Same as -mcpu command line option       */

  .fpu softvfp      /* Select the floating-point unit to assemble for. Valid values         */
                    /* are the same as for the -mfpu commandline option.                    */
                    /* For this .S file, no hardware floating point is used.                */
                    /* Notice that for the rest of the files, the following command-        */
                    /* line options are used:                                               */
                    /*    -mfloat-abi=hard                                                  */
                    /*    -mfpu=fpv5-sp-d16                                                 */

  .thumb            /* This directive selects the instruction set being generated.          */
                    /* The value 16 selects Thumb, with the value 32 selecting ARM.         */

.global  g_pfnVectors         /* Make the label 'g_pfnVectors' visible to the Linker.       */
.global  Default_Handler      /* Make the label 'Default_Handler' visible to the Linker.    */
                              /* Note: the label 'Reset Handler' is automatically visible   */
                              /* to the linker, since it is defined as a section name.      */
                              /* (is this correct?).                                        */
/* ------------------------------------------------------------------------------------------*/
/*                               .text section                                               */
/* ------------------------------------------------------------------------------------------*/
.word  _sidata    /* Start address for the initialization values of the .data section,      */
                  /* defined in linker script.                                              */
.word  _sdata     /* Start address for the .data section, defined in linker script.         */
.word  _edata     /* End address for the .data section, defined in linker script.           */
.word  _sbss      /* Start address for the .bss section, defined in linker script.          */
.word  _ebss      /* End address for the .bss section, defined in linker script.            */
/* stack used for SystemInit_ExtMemCtl; always internal RAM used */

/* ------------------------------------------------------------------------------------------*/
/*                               .text.Reset_Handler section                                 */
/* ------------------------------------------------------------------------------------------*/
.section .text.Reset_Handler
.weak  Reset_Handler              /* Mark the symbol 'Reset_Handler' as being weak.         */
.type  Reset_Handler, %function   /* Mark the symbol 'Reset_Handler' as a function name.    */

    Reset_Handler: /*-------------------------<Reset_Handler>--------0x00000000------------*/
      ldr   sp, = estack
      movs  r1, #0
      b     LoopCopyDataInit

    CopyDataInit: /*-------------------------<CopyDataInit>---------0x00000008------------*/
      ldr   r3, = sidata
      ldr   r3, [r3, r1]
      str   r3, [r0, r1]
      adds  r1, r1, #4

    LoopCopyDataInit: /*---------------------<LoopCopyDataInit>-----0x00000010------------*/
      ldr   r0, = sdata
      ldr   r3, = _edata
      adds  r2, r0, r1
      cmp   r2, r3
      bcc   CopyDataInit
      ldr   r2, = sbss
      b     LoopFillZerobss

    FillZerobss: /*--------------------------<FillZerobss>----------0x0000001E------------*/
      movs  r3, #0
      str   r3, [r2], #4

    LoopFillZerobss: /*----------------------<LoopFillZerobss>------0x00000024------------*/
      ldr   r3, = _ebss
      cmp   r2, r3
      bcc   FillZerobss
```

```
    StartUserProgram: /*-----------------------<StartUserProgram>-----0x000000??-----------*/
      bl    SystemInit          /* Call SystemInit() function from file system stm32f7xx.c  */
      bl      libc init array   /* Call static constructors                                 */
      bl    main                /* Call the main() function                                 */
      bx    lr                  /* Jump to the address contained in R14 (the link register) */


.size  Reset_Handler, .-Reset_Handler /* The .size directive tells the assembler how much  */
                                      /* space the data that a symbol points to is using.  */
                                      /* The expression ".-ResetHandler" will calculate    */
                                      /* the total size in bytes of the function           */
                                      /* 'Reset_Handler', so that the linker can exclude   */
                                      /* the function if it's unused.                      */

/* -------------------------------------------------------------------------------------*/
/*                              .text.Default_Handler section                           */
/* -------------------------------------------------------------------------------------*/
.section .text.Default_Handler,"ax",%progbits   /* The ,"ax",@progbits tells the assembler */
                                                /* that the section is allocatable ("a"),  */
                                                /* executable ("x") and contains data      */
                                                /* ("@progbits").                          */
    Default_Handler:

    Infinite_Loop:
      b    Infinite Loop

.size  Default_Handler, .-Default_Handler



/* -------------------------------------------------------------------------------------*/
/*                              .isr_vector section                                     */
/* -------------------------------------------------------------------------------------*/
.section .isr_vector,"a",%progbits              /* The .isr_vector section is allocatable, */
                                                /* but not executable, and it contains     */
                                                /* data ("@progbits").                     */
.type  g_pfnVectors, %object
.size  g_pfnVectors, .-g_pfnVectors


g_pfnVectors:

  .word  _estack            /* The label '_estack' gets its value from the linkerscript. */
                            /* It holds the address of the end of RAM-memory. That is    */
                            /* 0x2005 0000 for the STM32F7.                              */
                            /* Note that this value is written as the first entry in the */
                            /* vector table.                                             */

  .word  Reset_Handler      /* 'Reset Handler' or 'Reset Vector'                         */
                            /* This symbol (or label) gets its value at link time, when  */
                            /* the linker decides where to put the .text.Reset_Handler   */
                            /* section.                                                  */
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler
```

```
  /* External Interrupts */
  .word     WWDG_IRQHandler                   /* Window WatchDog                          */
  .word     PVD_IRQHandler                    /* PVD through EXTI Line detection          */
  .word     TAMP_STAMP_IRQHandler             /* Tamper and TimeStamps through EXTI line */
  .word     RTC_WKUP_IRQHandler               /* RTC Wakeup through the EXTI line        */
  .word     FLASH_IRQHandler                  /* FLASH                            */
  .word     RCC_IRQHandler                    /* RCC                              */
  .word     EXTI0_IRQHandler                  /* EXTI Line0                       */
  .word     EXTI1_IRQHandler                  /* EXTI Line1                       */
  .word     EXTI2_IRQHandler                  /* EXTI Line2                       */
  .word     EXTI3_IRQHandler                  /* EXTI Line3                       */
  .word     EXTI4_IRQHandler                  /* EXTI Line4                       */
  .word     DMA1_Stream0_IRQHandler           /* DMA1 Stream 0                    */
  .word     DMA1_Stream1_IRQHandler           /* DMA1 Stream 1                    */
  .word     DMA1_Stream2_IRQHandler           /* DMA1 Stream 2                    */
  .word     DMA1_Stream3_IRQHandler           /* DMA1 Stream 3                    */
  .word     DMA1_Stream4_IRQHandler           /* DMA1 Stream 4                    */
  .word     DMA1_Stream5_IRQHandler           /* DMA1 Stream 5                    */
  .word     DMA1_Stream6_IRQHandler           /* DMA1 Stream 6                    */
  .word     ADC_IRQHandler                    /* ADC1, ADC2 and ADC3s             */

  ...

  .word     SPDIF_RX_IRQHandler               /* SPDIF_RX                         */

/*******************************************************************************
*
* Provide weak aliases for each Exception handler to the Default_Handler.
* As they are weak aliases, any function with the same name will override
* this definition.
*
*******************************************************************************/
   .weak      NMI_Handler
   .thumb_set NMI_Handler,Default_Handler

   .weak      HardFault Handler
   .thumb set HardFault Handler,Default Handler

   ...

   .weak      SPDIF_RX_IRQHandler
   .thumb set SPDIF RX IRQHandler,Default Handler

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

6.3 Linkerscript file

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x20050000;    /* end of RAM */

/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200;  /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* ------------------------------------------------------------------------------------------*/
/*                                MEMORY AREAS                                               */
/* ------------------------------------------------------------------------------------------*/
MEMORY
{
    FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
    /* FLASH (rx)      : ORIGIN = 0x00200000, LENGTH = 1024K */
    RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 320K
}

/* ------------------------------------------------------------------------------------------*/
/*                                OUTPUT SECTIONS                                            */
/* ------------------------------------------------------------------------------------------*/
SECTIONS
{
    /*****************************/
    /*      VECTOR TABLE         */
    /*****************************/
    .isr_vector :
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector)) /* Vector Table */
        . = ALIGN(4);
    } >FLASH

    /*****************************/
    /*      PROGRAM CODE         */
    /*****************************/
    .text :
    {
        . = ALIGN(4);
        *(.text)            /* .text sections (code) */
        *(.text*)           /* .text* sections (code) */
        *(.glue_7)          /* glue arm to thumb code */
        *(.glue_7t)         /* glue thumb to arm code */
        *(.eh_frame)

        /* Note: The section '.text.Reset_Handler' is one of the *(.text*) sections, such  */
        /* that it gets linked into the output .text section somewhere here.               */
        /* We can verify the exact spot where the Reset_Handler section is positioned, by  */
        /* examining the second entry of the vector table.                                 */
        /* A test has given the following results:                                         */
        /*    FLASH (rx) : ORIGIN = 0x0800 0000    ==>  Reset_Handler = 0x0800 1C91        */
        /*    FLASH (rx) : ORIGIN = 0x0020 0000    ==>  Reset_Handler = 0x0020 1CB9        */
        /*                                                                                 */
        /* In both cases, the Reset Handler section ends up a few hundred bytes after the  */
        /* vector table in Flash. But in the first case, the "Reset Handler" symbol points */
        /* to the Reset-code through AXIM-interface, whereas in the latter case it points  */
        /* to the Reset-code through the ITCM-interface.                                   */


        KEEP (*(.init))
        KEEP (*(.fini))

        . = ALIGN(4);
        _etext = .;         /* define a global symbol at end of code */

    } >FLASH
```

```
  /*****************************/
  /*      CONSTANT DATA        */
  /*****************************/
  .rodata :
  {
      . = ALIGN(4);
      *(.rodata)         /* .rodata sections (constants, strings, etc.) */
      *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
      . = ALIGN(4);
  } >FLASH

  .ARM.extab :
  {
      *(.ARM.extab* .gnu.linkonce.armextab.*)
  } >FLASH
  .ARM :
  {
      __exidx_start = .;
      *(.ARM.exidx*)
        exidx end = .;
  } >FLASH
  .preinit_array :
  {
      PROVIDE_HIDDEN (__preinit_array_start = .);
      KEEP (*(.preinit array*))
      PROVIDE HIDDEN (  preinit array end = .);
  } >FLASH
  .init_array :
  {
      PROVIDE HIDDEN (  init array start = .);
      KEEP (*(SORT(.init array.*)))
      KEEP (*(.init array*))
      PROVIDE_HIDDEN (__init_array_end = .);
  } >FLASH
  .fini_array :
  {
      PROVIDE HIDDEN (  fini array start = .);
      KEEP (*(SORT(.fini array.*)))
      KEEP (*(.fini_array*))
      PROVIDE_HIDDEN (__fini_array_end = .);
  } >FLASH

  /*****************************/
  /*    INITIALIZED DATA       */
  /*****************************/
  _sidata = LOADADDR(.data);
  .data :
  {
      . = ALIGN(4);
      _sdata = .;        /* create a global symbol at data start */
      *(.data)           /* .data sections */
      *(.data*)          /* .data* sections */

      . = ALIGN(4);
      _edata = .;        /* define a global symbol at data end */

  } >RAM AT> FLASH


  /*****************************/
  /*    UNINITIALIZED DATA     */
  /*****************************/
  . = ALIGN(4);
  .bss :
  {
       sbss = .;         /* define a global symbol at bss start */
      __bss_start__ = _sbss;
      *(.bss)
      *(.bss*)
      *(COMMON)

      . = ALIGN(4);
      _ebss = .;         /* define a global symbol at bss end */
      __bss_end__ = _ebss;

  } >RAM
```

```
    /*****************************/
    /* USER HEAP STACK SECTION  */
    /*****************************/
    /* User heap stack section, used to check that there is enough RAM left */
    ._user_heap_stack :
    {
        . = ALIGN(8);
        PROVIDE ( end = . );
        PROVIDE (  end = . );
        . = . +  Min Heap Size;
        . = . + _Min_Stack_Size;
        . = ALIGN(8);
    } >RAM


    /* Remove information from the standard libraries */
    /DISCARD/ :
    {
        libc.a ( * )
        libm.a ( * )
        libgcc.a ( * )
    }

    .ARM.attributes 0 : { *(.ARM.attributes) }
}
```

6.4 system_stm32f7xx.c file

The function **System_Init**(void) is called by the startup_stm32f7xx.s file before the **main**(void) routine. The **System_Init**(void) does the following:

-> Initialization of the FPU
-> Initialization of the clock, configured as internal HSI clock running at 16 Mhz
-> Initialization of external S(D)RAM
-> Relocation of the Vector Table

system_stm32f7xx.c

```
void SystemInit(void)
{
    /*-----------------------------------------------------------------------------*/
    /*|                            FPU                                            |*/
    /*-----------------------------------------------------------------------------*/
    /* The macros   FPU PRESENT and   FPU USED are set through the command line */
    /* options when invoking the gcc compiler. It is possible that the IDE      */
    /* does not represent their values correctly while coding.                  */
    #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
      /* set CP10 and CP11 Full Access */
      SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));
    #endif

    /*-----------------------------------------------------------------------------*/
    /*|                        RCC clock config                                   |*/
    /*-----------------------------------------------------------------------------*/
    /* Reset the RCC clock configuration to the default reset state             */

    RCC->CR |= (uint32_t)0x00000001;      /* Set HSION bit                      */
    RCC->CFGR = 0x00000000;               /* Reset CFGR register                */
    RCC->CR &= (uint32_t)0xFEF6FFFF;      /* Reset HSEON, CSSON and PLLON bits  */
    RCC->PLLCFGR = 0x24003010;            /* Reset PLLCFGR register             */
    RCC->CR &= (uint32_t)0xFFFBFFFF;      /* Reset HSEBYP bit                   */
    RCC->CIR = 0x00000000;                /* Disable all interrupts             */

    /*-----------------------------------------------------------------------------*/
    /*|                        External S(D)RAM                                   |*/
    /*-----------------------------------------------------------------------------*/
    /* This function configures the external memories (SRAM/SDRAM)              */
    /* This SRAM/SDRAM will be used as program data memory (including heap      */
    /* and stack).                                                              */
    #if defined (DATA_IN_ExtSRAM) || defined (DATA_IN_ExtSDRAM)
      SystemInit_ExtMemCtl();
    #endif

    /*-----------------------------------------------------------------------------*/
    /*|                        VECTOR TABLE location                              |*/
    /*-----------------------------------------------------------------------------*/
    /* The initial Vector Table address SCB->VTOR equals the Boot Base Address  */
    /* which is selected by the BOOT pin. The initial Vector Table location     */
    /* affects the boot behaviour (where the CPU makes its first jump...).      */
    /* Once the program is running, the Vector Table can be relocated. That is  */
    /* what the code below is doing.                                            */
    #ifdef VECT_TAB_SRAM
      SCB->VTOR = SRAM1 BASE | VECT TAB OFFSET; /* Relocation to Internal SRAM */
    #else
      SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Relocation in Internal FLASH */
    #endif
}
```

Note that the **SCB**->**VTOR** register gets a new value at the end of the function. This happens when the program is already running, so it has no impact on the boot process. I have noticed that the STM32Cube software will always relocate the **SCB**->**VTOR** register to 0x0800 0000, regardless of your choice for the Flash interface (AXIM vs ITCM).

6.5 main.c file

Finally the startup code calls the **main**(void) routine. The main routine itself is quite small, but it invokes a lot of other startup-routines that require some more study.

6.5.1 General Initialization of the HAL

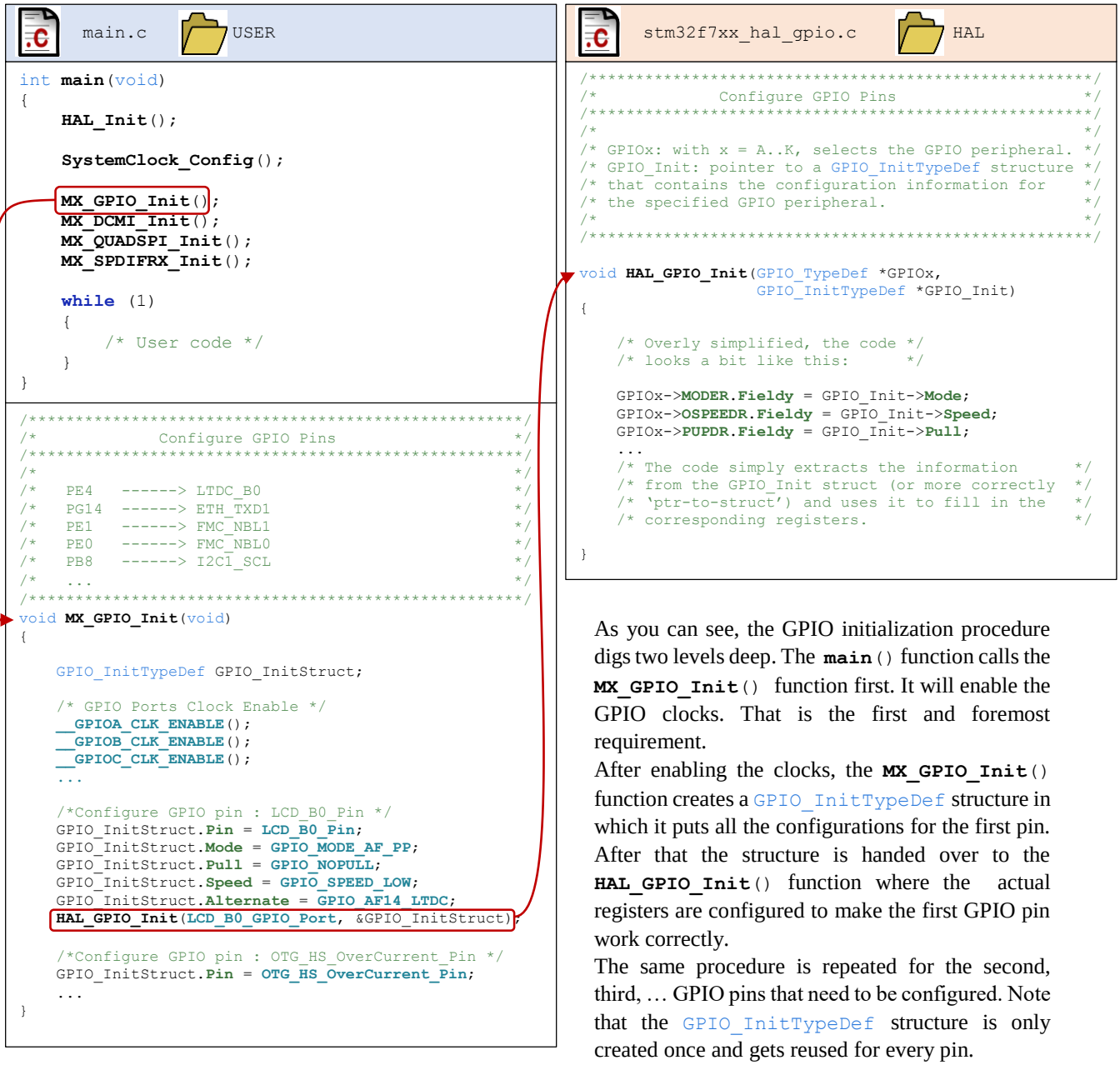
The first function call inside the main routine is **HAL_Init**(void). This function initializes the global part of the HAL (Hardware Abstraction Layer). The high level stuff is initialized in the **HAL_MspInit**(void) function.

6.5.2 Initialization of a special HAL driver

Just like the global HAL initialization, the individual HAL drivers go through different steps (or functions) to get initialized. The figure below shows these steps for the initialization of the GPIO driver. While looking at the figure, you should not forget that the GPIO peripheral is a special case. Indeed, for the *shared* and *system* peripherals, no handle or instance object is used:

> GPIO
> SYSTICK
> NVIC
> PWR
> RCC
> FLASH

So the case of the GPIO initialization is a bit different from other peripheral driver initializations. Anyway, here is the initialization procedure:



As you can see, the GPIO initialization procedure digs two levels deep. The **`main`**`()` function calls the **`MX_GPIO_Init`**`()` function first. It will enable the GPIO clocks. That is the first and foremost requirement.

After enabling the clocks, the **`MX_GPIO_Init`**`()` function creates a `GPIO_InitTypeDef` structure in which it puts all the configurations for the first pin. After that the structure is handed over to the **`HAL_GPIO_Init`**`()` function where the actual registers are configured to make the first GPIO pin work correctly.

The same procedure is repeated for the second, third, … GPIO pins that need to be configured. Note that the `GPIO_InitTypeDef` structure is only created once and gets reused for every pin.

6.5.3 Initialization of a normal HAL driver

Let us now consider the case of a "normal" peripheral driver initialization. A first observation is that the procedure goes three levels deep (remark the three red arrows below). We will study the initialization of the QUADSPI peripheral:

main.c — USER

```
int main(void)
{
    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_DCMI_Init();
    MX_QUADSPI_Init();
    MX_SPDIFRX_Init();

    while (1)
    {
        /* User code */
    }
}
```

```
/*********************************************************/
/*            Create peripheral handles                  */
/*********************************************************/

DCMI_HandleTypeDef hdcmi;
QSPI_HandleTypeDef hqspi;
SPDIFRX_HandleTypeDef hspdif;
...

/*********************************************************/
/*            QUADSPI Initialization                     */
/*********************************************************/
void MX_QUADSPI_Init(void)
{

  hqspi.Instance = QUADSPI;
  hqspi.Init.ClockPrescaler = 255;
  hqspi.Init.FifoThreshold = 1;
  hqspi.Init.SampleShifting = QSPI_SAMPLE_SHIFTING_NONE;
  hqspi.Init.FlashSize = 1;
  hqspi.Init.ChipSelectHighTime = QSPI_CS_HIGH_TIME...;
  hqspi.Init.ClockMode = QSPI_CLOCK_MODE_0;
  HAL_QSPI_Init(&hqspi);

}
```

stm32f7xx_hal_qspi.c — HAL

```
/*********************************************************/
/*            QUADSPI Initialization                     */
/*********************************************************/

HAL_StatusTypeDef HAL_QSPI_Init(QSPI_HandleTypeDef *hqspi)
{
    /* Overly simplified, the code */
    /* looks a bit like this:      */

    hqspi->Instance->CR = ...
    hqspi->Instance->DCR = ...
    hqspi->Instance->SR = ...
    ...
    /* The code simply extracts the information      */
    /* from the QSPI_Init struct (the peripheral     */
    /* handle gives access to this struct) and uses  */
    /* it to fill in the corresponding register      */
    /* (which are accessible through the handle as   */
    /* well).                                        */

    HAL_QSPI_MspInit(hqspi);

    return status;
}
```

stm32f7xx_hal_msp.c — USER(!)

```
void HAL_QSPI_MspInit(QSPI_HandleTypeDef* hqspi)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    __HAL_RCC_QSPI_CLK_ENABLE();

    /**QUADSPI GPIO Configuration
    PE2     ------> QUADSPI_BK1_IO2
    PB6     ------> QUADSPI_BK1_NCS
    PB2     ------> QUADSPI_CLK
    PD12     ------> QUADSPI_BK1_IO1
    PD13     ------> QUADSPI_BK1_IO3
    PD11     ------> QUADSPI_BK1_IO0
    */
    GPIO_InitStruct.Pin = QSPI_D2_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF9_QUADSPI;
    HAL_GPIO_Init(QSPI_D2_GPIO_Port, &GPIO_InitStruct);

    ...

}
```

In contrast to the GPIO initialization, the QUADSPI initialization starts with the creation of a **handle**. This object is passed on from one function to the next.

The **`MX_QUADSPI_Init()`** function fills in the proper configuration data into the `Init`-field from the handle (this `Init`-field itself is also a struct).

The handle is passed on to the **`HAL_QSPI_Init()`** function, which reads out the data from the `Init`-field and uses it to configure the actual registers. Once the registers are configured, the peripheral is almost ready. One more thing needs to be done. The handle is passed on to the **`HAL_QSPI_MspInit()`** function where the corresponding microcontroller pins are configured (intput, output, …).

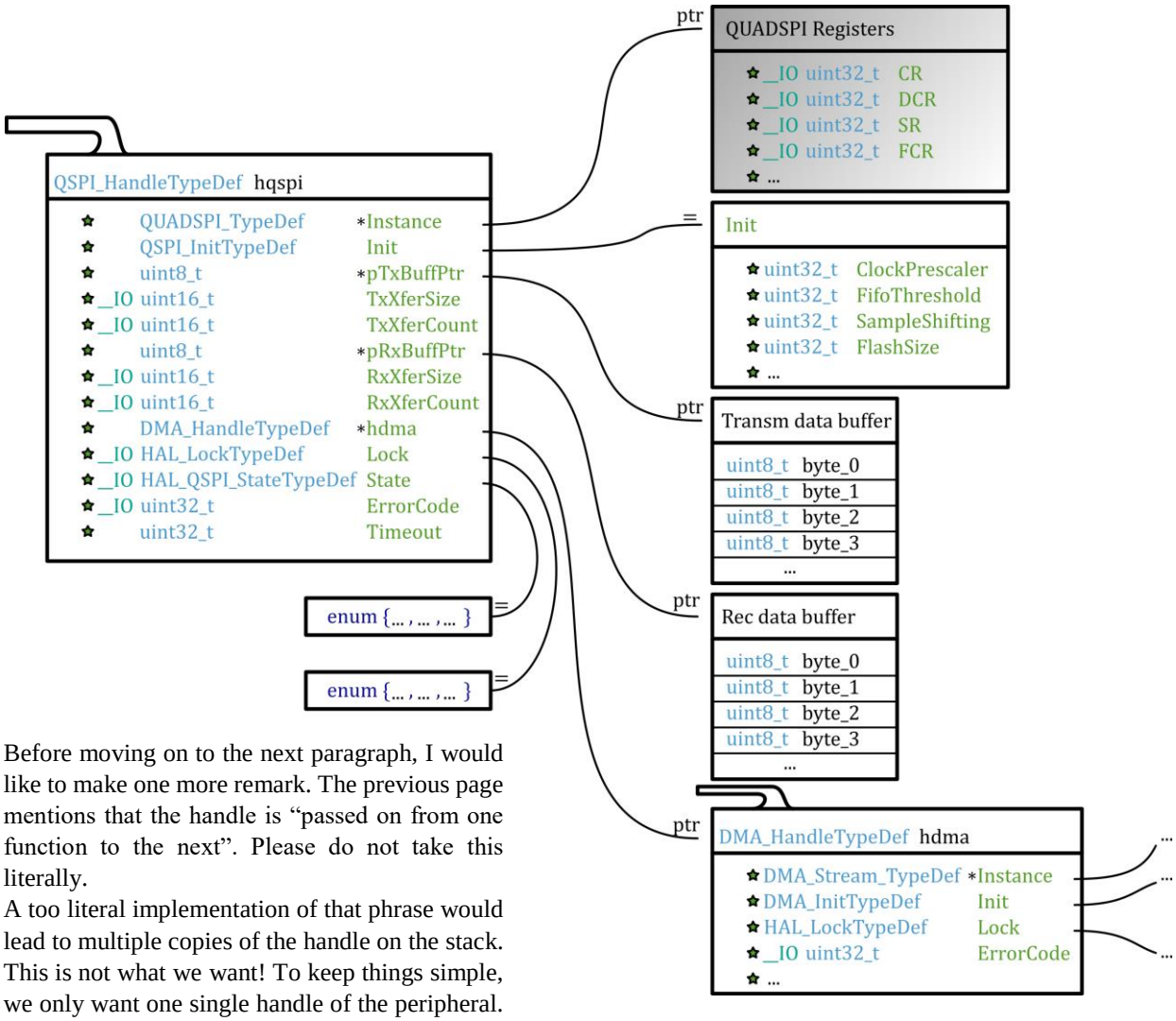
It is very interesting to take a closer look to the peripheral **`handle`**. We know that the **`handle`** is a struct. The figure below shows that the `hqspi` handle has 13 fields, each with different sizes. So this struct probably takes a few tens of bytes in the memory. It is one single block of consecutive bytes. They are not scattered around. Where does the struct live in the memory? To be honest, we don't know. The Linker will allocate some space for it. But there is no way to tell from the code where this struct is going to end up.
But one thing is for sure. The `*Instance` field holds the address of the QUADSPI registers. More precisely, it holds the address of the very first QUADSPI register (= QUADSPI_BASE). And that's enough. Because all the QUADSPI registers are located one after the other. The QUADSPI_BASE address is fixed in the silicon to be address `0xA0001000`. The following code line makes sure that the `*Instance` pointer holds the correct address:

```
hqspi.Instance = QUADSPI;
```

The right-side expression **`QUADSPI`** is a macro which expands into:

```
#define QUADSPI     ((QUADSPI_TypeDef *) QSPI_R_BASE)
```

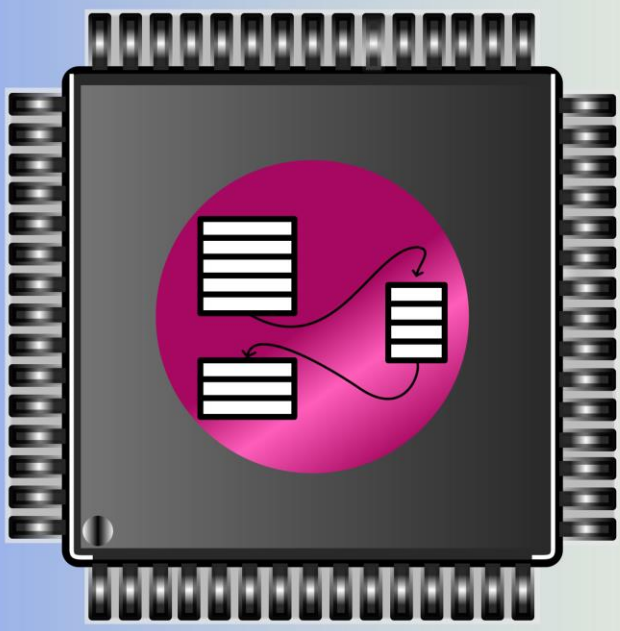
So it doesn't really matter where the `hqspi` handle itself resides in memory. As long as the `*Instance` field points to the set of QUADSPI registers, everything will work just fine.



Before moving on to the next paragraph, I would like to make one more remark. The previous page mentions that the handle is "passed on from one function to the next". Please do not take this literally.
A too literal implementation of that phrase would lead to multiple copies of the handle on the stack. This is not what we want! To keep things simple, we only want one single handle of the peripheral. So we need to pass on *a pointer* from one function to the next.

Chapter 7

Interrupts



7.1 The Interrupt Vector Table

The Interrupt Vector Table is an indirect Table containing the addresses in memory of the Interrupt Service Routines. Thanks to this table, the processor knows where to locate these routines. In a way, you could consider the entries in the Vector Table as “function pointers”.

7.1.1 The IVT in the startup file

The Interrupt Vector Table is established in the startup file. The following example is taken from the automatically generated CubeMX project for the STM32F746NG Device (Cortex-M7). Note that the Vector Table is not immediately filled with *addresses*, but instead with *labels*! If you define a function in one of your c-files with the exact same name, the linker notices it. Next, the linker will replace the corresponding *label* in the Vector Table by the *address* that points to the location where your function code resides.

```
/* ------------------------------------------------------------------------------------------*/
/*                                .isr_vector section                                        */
/* ------------------------------------------------------------------------------------------*/
.section .isr_vector,"a",%progbits              /* The .isr_vector section is allocatable, */
                                                /* but not executable, and it contains     */
                                                /* data (“@progbits”).                      */
.type  g_pfnVectors, %object
.size  g_pfnVectors, .-g_pfnVectors


g_pfnVectors:

  .word  _estack            /* The label '_estack' gets its value from the linkerscript. */
                            /* It holds the address of the end of RAM-memory. That is    */
                            /* 0x2005 0000 for the STM32F7.                              */
                            /* Note that this value is written as the first entry in the */
                            /* vector table.                                             */

  .word  Reset_Handler      /* 'Reset Handler' or 'Reset Vector'                         */
                            /* This symbol (or label) gets its value at link time, when  */
                            /* the linker decides where to put the .text.Reset Handler   */
                            /* section.                                                  */
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler


  /* External Interrupts */
  .word     WWDG_IRQHandler                 /* Window WatchDog                          */
  .word     PVD_IRQHandler                  /* PVD through EXTI Line detection          */
  .word     TAMP_STAMP_IRQHandler           /* Tamper and TimeStamps through EXTI line */
  .word     RTC_WKUP_IRQHandler             /* RTC Wakeup through the EXTI line         */
  .word     FLASH_IRQHandler                /* FLASH                                    */
  .word     RCC_IRQHandler                  /* RCC                                      */
  .word     EXTI0_IRQHandler                /* EXTI Line0                               */
  .word     EXTI1_IRQHandler                /* EXTI Line1                               */
  .word     EXTI2_IRQHandler                /* EXTI Line2                               */

  ...

  .word     SPDIF_RX_IRQHandler             /* SPDIF_RX                                 */
```

Even if the Vector Table contains the addresses of the handler routines, the Cortex-M core and the NVIC controller need a way to find the Vector Table itself. By convention, the Vector Table starts at hardware address zero. In

practice, it is located at the start address of Flash memory, which is not necessarily zero. It is the responsibility of the *Linker Script* to make sure that the Interrupt Vector Table ends up in the right spot:

```
/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x00200000, LENGTH = 1024K   /* FLASH mem on ITCM interface */
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 320K     /* DTCM + SRAM1 + SRAM2 */
}

/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  ...

}
```

What happens if some *labels* from the Vector Table are never used as function name? The startup code takes care of this problem. It provides so called “weak aliases” for every entry in the Vector Table. The weak aliases point to the `Default_Handler()` function.

```
/*******************************************************************************
* Provide weak aliases for each Exception handler to the Default_Handler.
* As they are weak aliases, any function with the same name will override
* this definition.
*******************************************************************************/
   .weak      NMI_Handler
   .thumb_set NMI_Handler,Default_Handler

   .weak      HardFault_Handler
   .thumb_set HardFault_Handler,Default_Handler

   .weak      MemManage_Handler
   .thumb_set MemManage_Handler,Default_Handler

   ...

/**
 * @brief  This is the code that gets called when the processor receives an
 *         unexpected interrupt (via the weak alias). This simply enters an infinite loop,
 *         preserving the system state for examination by a debugger.
 * @param  None
 * @retval None
*/
    .section  .text.Default_Handler,"ax",%progbits
Default_Handler:
  Infinite_Loop:
    b  Infinite_Loop
  .size  Default_Handler, .-Default_Handler
```

7.1.2 Interrupt numbers

Interrupts can be roughly devided into two groups: those that are 'internal' to the core and those that are 'external'. The first group of interrupts belong to the processor core. They are defined and implemented by the designers of the processor core. The very first interrupt in this group is the `Reset_Handler`. Other interrupts are mainly `Fault-` and `System Handlers`. The ARM Cortex-M7 core documentation provides more information about them.

The second group – the external ones – are implemented by the silicon vendor, which is STMicroelectronics in our case. Refer to the STM32F7 datasheet and Reference Manual to find more information.

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address or offset[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable[c]	0x00000010	Synchronous
5	-11	BusFault	Configurable[c]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable[c]	0x00000018	Synchronous
7-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[c]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[c]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[c]	0x0000003C	Asynchronous
16 and above[d]	0 and above[e]	Interrupt (IRQ)	Configurable[f]	0x00000040 and above[g]	Asynchronous

First group: Internal interrupts

Sec group: External interrupts

a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-6.
b. See *Vector table* for more information.
c. See *System Handler Priority Registers* on page 4-22.
d. The exception number range is implementation defined.
e. The IRQ number range is implementation defined.
f. See *Interrupt Priority Registers* on page 4-7.
g. Increasing in steps of 4. The vector address or offset range is implementation defined.

Every interrupt has an identification number. But confusion is likely to arise since the there are two different conventions. One should always carefully look at the context to distinguish which convention is getting used.

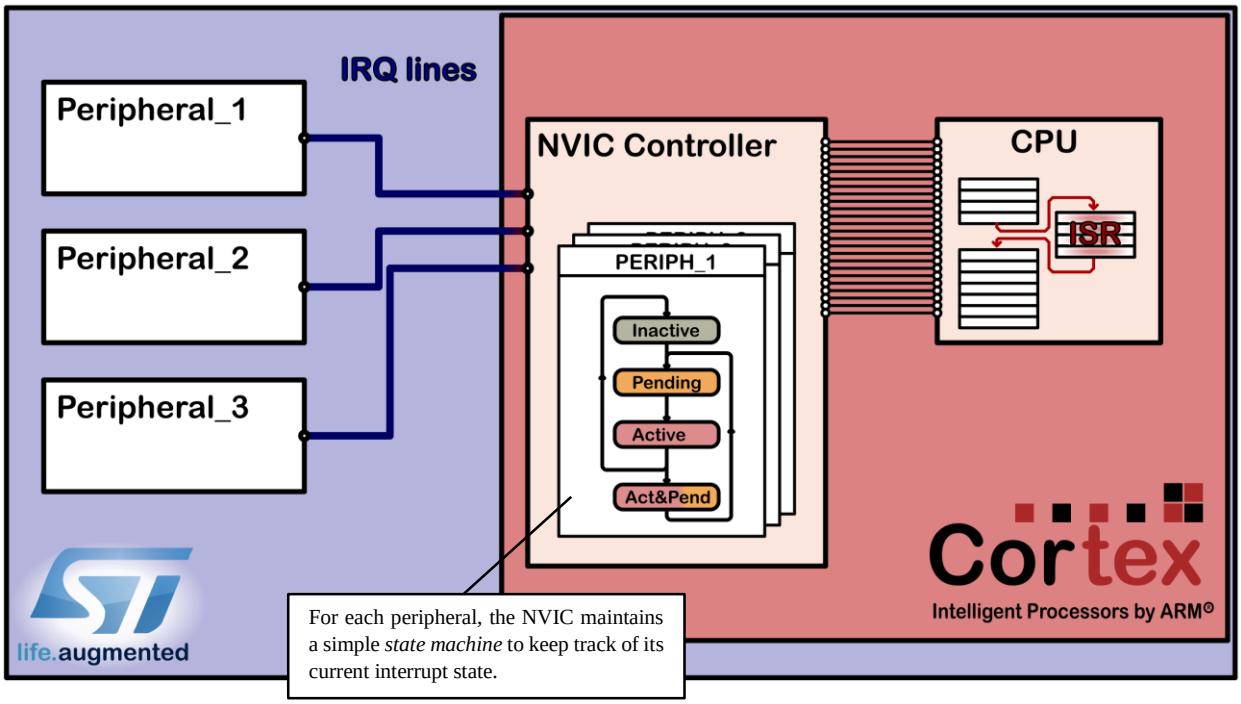
Two conventions in numbering interrupts exist:
Exception numbers: Start counting from 1 at the `Reset_Handler`. The external interrupts start from nr 16.
IRQ numbers: Start counting from -15 at the `Reset_Handler`. The external interrupts start from nr 0.

Let us now look at the IRQ numbers. These are defined in an **enum** in the stm32f746xx.h file (from the CMSIS folder):

```
typedef enum
{
    /******  Cortex-M7 Processor Interrupt Numbers ***********************************/
    NonMaskableInt_IRQn         = -14,    /*!< 2 Non Maskable Interrupt                */
    MemoryManagement_IRQn       = -12,    /*!< 4 Cortex-M7 Memory Management Interrupt */
    BusFault_IRQn               = -11,    /*!< 5 Cortex-M7 Bus Fault Interrupt         */
    UsageFault_IRQn             = -10,    /*!< 6 Cortex-M7 Usage Fault Interrupt       */
    SVCall_IRQn                 = -5,     /*!< 11 Cortex-M7 SV Call Interrupt          */
    DebugMonitor_IRQn           = -4,     /*!< 12 Cortex-M7 Debug Monitor Interrupt    */
    PendSV_IRQn                 = -2,     /*!< 14 Cortex-M7 Pend SV Interrupt          */
    SysTick_IRQn                = -1,     /*!< 15 Cortex-M7 System Tick Interrupt      */
    /*****STM32 specific Interrupt Numbers ************************************************/
    WWDG_IRQn                   = 0,   /*!< Window WatchDog Interrupt                                */
    PVD_IRQn                    = 1,   /*!< PVD through EXTI Line detection Interrupt                */
    TAMP_STAMP_IRQn             = 2,   /*!< Tamper and TimeStamp interrupts through the EXTI line */
    RTC_WKUP_IRQn               = 3,   /*!< RTC Wakeup interrupt through the EXTI line               */
    FLASH_IRQn                  = 4,   /*!< FLASH global Interrupt                                   */
    RCC_IRQn                    = 5,   /*!< RCC global Interrupt                                     */
    EXTI0_IRQn                  = 6,   /*!< EXTI Line0 Interrupt                                     */
    EXTI1_IRQn                  = 7,   /*!< EXTI Line1 Interrupt                                     */
    EXTI2_IRQn                  = 8,   /*!< EXTI Line2 Interrupt                                     */
    EXTI3_IRQn                  = 9,   /*!< EXTI Line3 Interrupt                                     */

    ...

    SPDIF_RX_IRQn               = 97   /*!< SPDIF-RX global Interrupt                                */

} IRQn_Type;
```

7.2 NVIC Controller

The NVIC and the processor core interface are closely coupled. All interrupts – including the core exceptions – are managed by the NVIC.
To my understanding, the following model gives a correct view on how interrupts are processed. Each peripheral that can fire interrupts has a dedicated '**interrupt line**' or '**IRQ line**' routed to the NVIC Controller. There is no direct contact between the peripheral and the CPU. But this IRQ line provides the peripheral a way to request for an interrupt to take place.
The NVIC has the responsibility to schedule the incoming interrupt requests and make the CPU execute the most important ones first. One can assume that the NVIC maintains simple 'state machines' for this purpose. Every peripheral starts in the *inactive* state – there is no request for interrupt pending. Upon a request, the peripheral goes into the *pending* state. It waits there until the NVIC determines that no other more important interrupts are pending. When the CPU starts executing the ISR (Interrupt Service Routine) associated with that peripheral, it is said to be *active*. Usually the peripheral slides back into the *inactive* state upon completion of the ISR code.



Sources:

- ARM® Cortex®-M7 Devices Generic User Guide	ch 2.3	p33
- ARM® Cortex®-M7 Devices Generic User Guide	ch 4.2	p234
- ARM®v7-M Architecture Reference Manual	ch B3.4	p680
- STM32F7 Reference Manual	ch 10	p287

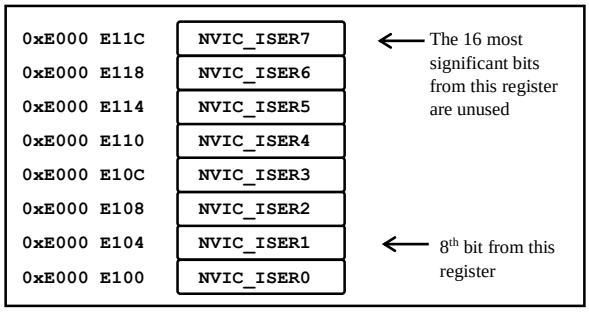
7.2.1 Exploring the NVIC state machine

The NVIC controller – just like any other peripheral – has a bunch of registers that control its actions. For most peripherals these registers would reside in the address space `0x4000 000 – 0x5FFF FFFF`. But the NVICs registers are located in the "Cortex-M7 internal peripherals" memory space. The NVIC has 4 kB allocated.

Most peripherals use the `stm32f7xx.h` file (from the CMSIS folder) to make their registers available to the user software. Not so for the NVIC. It uses the `core_cm7.h` file instead (also from CMSIS folder). Now let us consider the registers. You can find the following definitions in the `core_cm7.h` file:

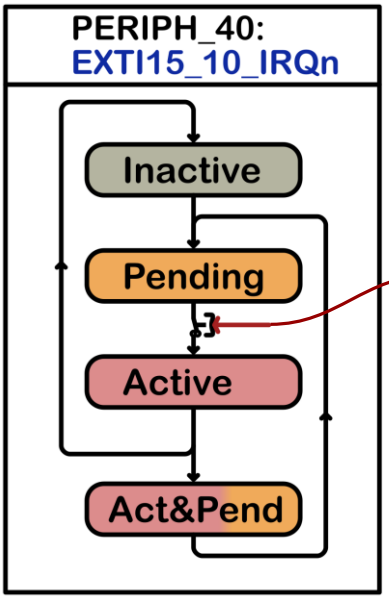
core_cm7.h

```
typedef struct
{
  __IO uint32_t  ISER[8];               /* 0xE000 E100 - 0xE000 E11F  | Int Set Enable Reg  */
       uint32_t  RESERVED0[24];

  __IO uint32_t  ICER[8];               /* 0xE000 E180 - 0xE000 E19F  | Int Clr Enable Reg  */
       uint32_t  RSERVED1[24];

  __IO uint32_t  ISPR[8];               /* 0xE000 E200 - 0xE000 E21F  | Int Set Pending Reg */
       uint32_t  RESERVED2[24];

  __IO uint32_t  ICPR[8];               /* 0xE000 E280 - 0xE000 E29F  | Int Clr Pending Reg */
       uint32_t  RESERVED3[24];

  __IO uint32_t  IABR[8];               /* 0xE000 E300 - 0xE000 E31F  | Int Active Bit Reg  */
       uint32_t  RESERVED4[56];

  __IO uint8_t   IP[240];               /* 0xE000 E400 - 0xE000 E4EF  | Int Priority Reg    */
       uint32_t  RESERVED5[644];

  __O  uint32_t  STIR;                  /* 0xE000 EF00                | Softw Trig Int Reg  */

}  NVIC_Type;


#define NVIC_BASE       0xE000 E100

#define NVIC            ((NVIC_Type *)NVIC_BASE)
```



The 32-bit register-arrays `ISER[8]`, `ICER[8]`, `ISPR[8]`, `ICPR[8]` and `IABR[8]` allocate one bit to each interrupt. A simple calculation learns us that they span 256 interrupts at the most (8 registers x 32 bits = 256 bits). However, they do not serve the 'internal' interrupts (System Handlers and Fault Handlers). So the number is capped down to 240 interrupts (the 16 most significant bits from the each 8th register are unused). In reality the silicon vendor does not implement all 240 interrupts, resulting in even more unused bits.

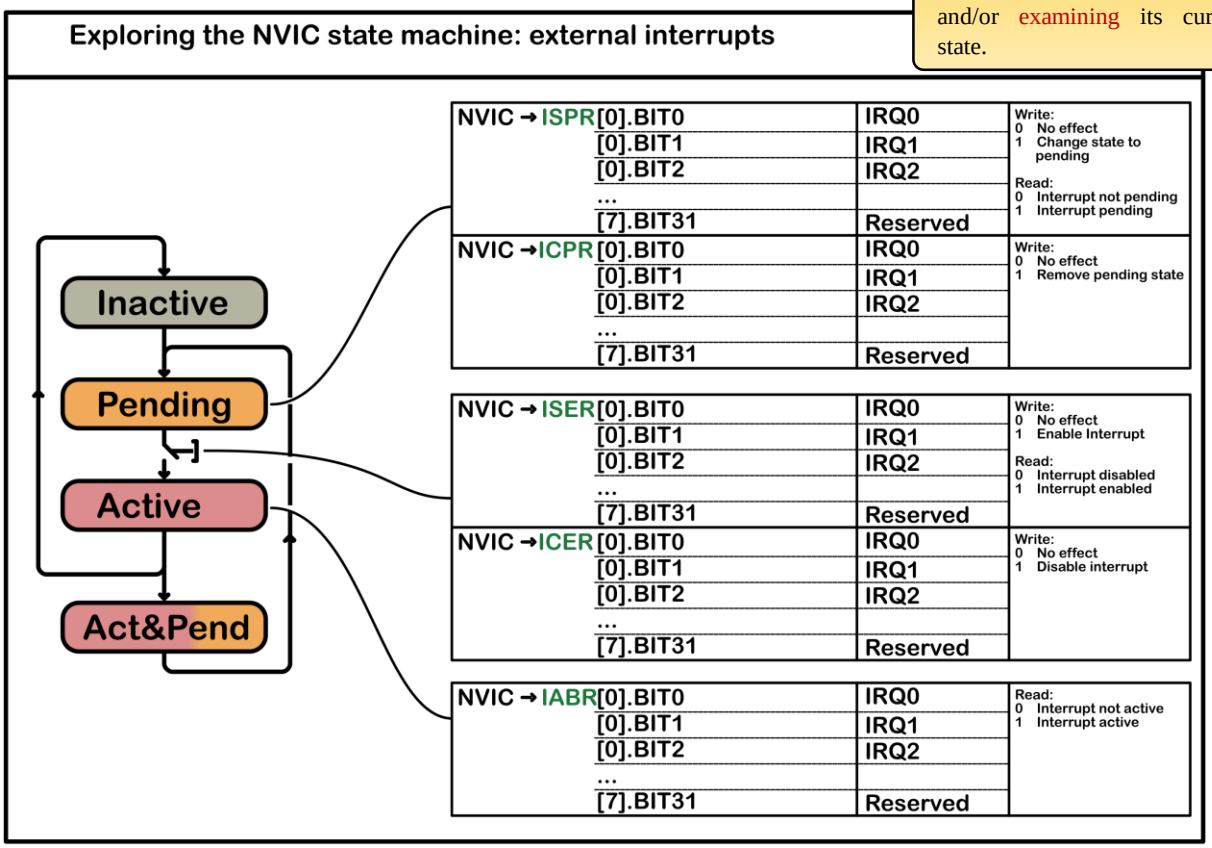
How do the bits map onto the actual interrupts? An example will clarify this. Suppose we want to 'enable' interrupt `EXTI15_10`. From the enumeration table in the file `stm32f746xx.h` we learn that this interrupt has number 40. From this we know that we need to set the 40th bit from the `NVIC->ISER[8]` array. That corresponds to setting the 8th bit from register `NVIC->ISER[1]`. Doing so will link the *Pending* state to the *Active* state in the NVIC state machine for the 40th peripheral: `EXTI15_10`. One could imagine some kind of switch between these two states getting pushed when the bit is set to 1.




```
NVIC -> ISER[1] = 1 << 8;   /* Set the 8th bit from          */
                            /* the second ISER register.    */
```

The NVIC controller provides more registers to shape or examine the state machine. The following figure demonstrates them all:

The **NVIC** registers can be used for both shaping the interrupt state machine to your needs and/or examining its current state.



Note:
The only NVIC registers not covered in the figure above are the *Priority fields*. Refer to the corresponding paragraph for a detailed study on assinging priorities to interrupts.

ARM provides a set of CMSIS functions to interact with the NVIC registers. The following table gives a complete overview:

CMSIS functions for NVIC control

CMSIS interrupt control function	Description
`void NVIC_SetPriorityGrouping(uint32_t priority_grouping)`	Set the priority grouping
`void NVIC_EnableIRQ(IRQn_t IRQn)`	Enable IRQn
`void NVIC_DisableIRQ(IRQn_t IRQn)`	Disable IRQn
`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`	Return true (IRQ-Number) if IRQn is pending
`void NVIC_SetPendingIRQ (IRQn_t IRQn)`	Set IRQn pending
`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`	Clear IRQn pending status
`uint32_t NVIC_GetActive (IRQn_t IRQn)`	~~Return the IRQ number of the active interrupt~~ Return 0 or 1 for the given IRQn
`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`	Set priority for IRQn
`uint32_t NVIC_GetPriority (IRQn_t IRQn)`	Read priority of IRQn

7.2.2 Find the active ISR

When looking for the current ISR being serviced by the CPU, you could call the function `NVIC_GetActive(IRQn_t IRQn)` multiple times until you get a hit. It's a viable strategy, but not very efficient. An alternative – and better – strategy is reading the `VECTACTIVE` byte from the **SCB**:

SCB -> ICSR.VECTACTIVE — ARM Cortex-M7 Devices, Generic User Guide : 4.3.3

[8:0]	VECTACTIVE[a]	RO	Contains the active exception number: **0** Thread mode. Nonzero The exception number[a] of the currently active exception. **Note** Subtract 16 from this value to obtain the CMSIS IRQ number required to index into the NVIC Interrupt Clear-Enable, Set-Enable, Clear-Pending, Set-Pending, or Priority Registers.

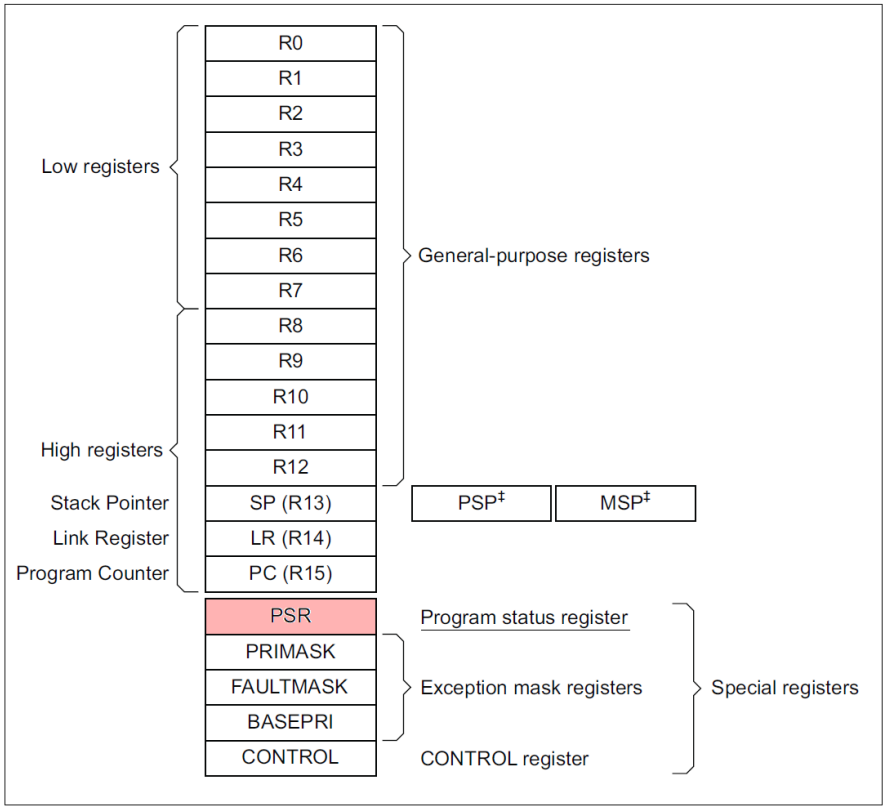
FreeRTOS provides the following two macros to reach the `VECTACTIVE` field (you can find them in the `portmacro.h` file):

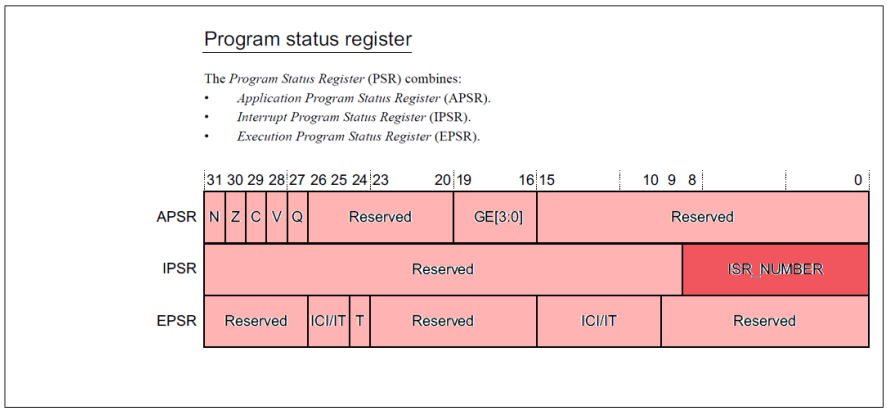
```
// Get the active exception number. Returns 0 if no exception is running (thread mode).
#define getActiveVector()       ((uint32_t)(( SCB->ICSR) & (0xFFUL)))

// Subtract 16 from this value to obtain the CMSIS IRQn number. Note that thread mode now
equals -16.
#define getActiveVectorIRQn()   (((int32_t)getActiveVector())-16UL)
```

Another option is to read the `IPSR` register. This is one of the special CPU registers. The `IPSR` register is not immediately visible on the CPU register map (see figure on the left). We have to look inside the `PSR` register. This register (colored red) is actually a combination of the following three registers:

- `APSR`
- `IPSR`
- `EPSR`

The next figure shows that the first bits [8:0] from the `IPSR` register contain the currently active ISR number.





Reading special CPU registers requires dedicated assembly instructions. CMSIS provides inline functions to do this. More in particular the **__get_IPSR**(void) function returns the content of the IPSR register. Confusion may arise when you find out that this function is implemented in the files cmsis_armccV6.h, cmsis_armcc.h and cmsis_gcc.h. But do now worry, only one of those files gets included in the final program. In our case the cmsis_gcc.h is included because we use the gcc compiler.
Here is the code from the **__get_IPSR**(void) function:

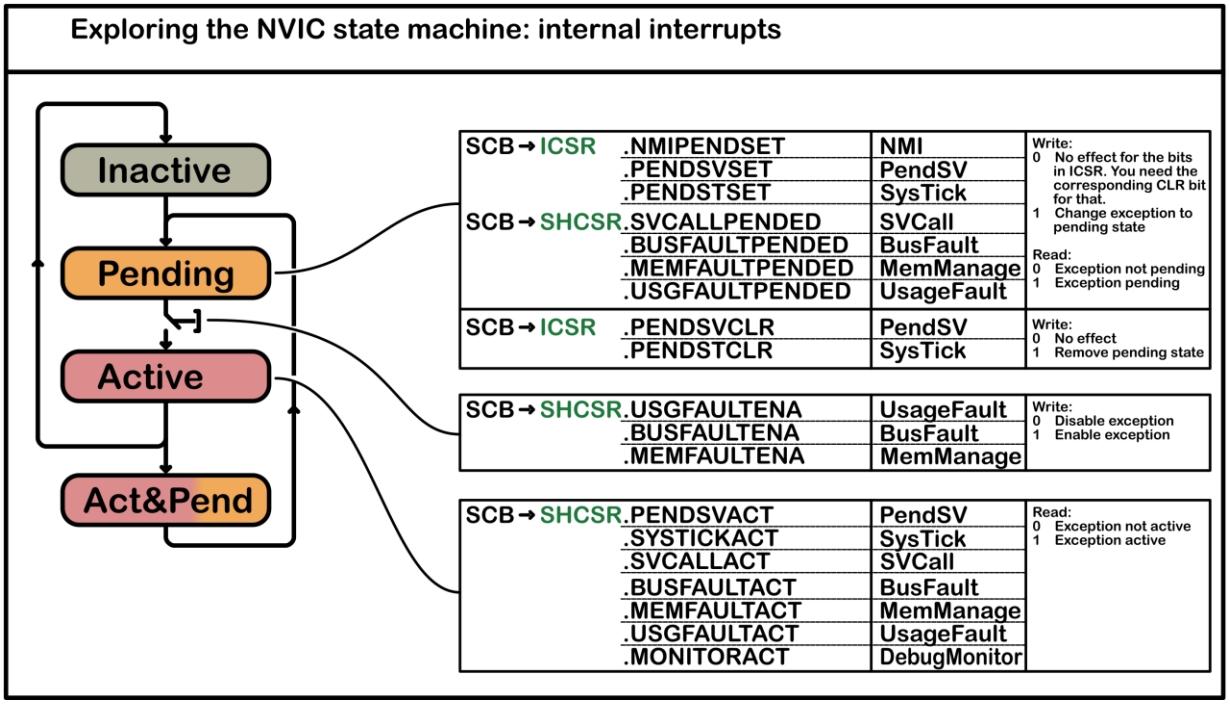
```
/**
  \brief       Get IPSR Register

  \details     Returns the content of the IPSR Register.

  \return      IPSR Register value. The value cannot be negative. Counting starts from zero.
               0 is returned for Thread mode. 16 is returned for IRQ0.
 */
__attribute__( ( always_inline ) ) __STATIC_INLINE uint32_t __get_IPSR(void)
{
    uint32 t result;

    __ASM volatile ("MRS %0, ipsr" : "=r" (result) );

    return(result);
}
```

Searching for the active ISR is not the only interesting feature that the ARM core offers. The SCB registers (System Control Block) offer a few more:

SCB->ICSR.ISRPENDING	Interrupt pending flag. This is a read-only bit. Excluding the NMI and faults, this bit indicates: **0** No interrupt is pending. **1** An interrupt is pending.
SCB->ICSR.VECTPENDING	Vector Pending. This field of 9 bits indicates the exception number of the highest priority pending enabled exception: **0** No pending exceptions. **Nonzero** The exception number of the highest priority & enabled exception. The value indicated by this field includes the effect of the BASEPRI and the FAULTMASK registers, but not any effect of the PRIMASK register.
SCB->ICSR.RETTOBASE	Return to Base. Indicates whether there are preempted active exceptions: **0** There are preempted *active* exceptions to execute. **1** There are none. Or the currently-executing exception is the only active one.

7.2.3 Exploring the NVIC state machine – for internal interrupts

All interrupts – including the *core exceptions* or *internal interrupts* – are handled by the `NVIC` controller. However, the NVIC register set does not cover them. They are covered by the `SCB` registers (System Control Block). The figure below gives an overview:



CMSIS and ST HAL apparently do not provide functions to use the register fields from this figure. But FreeRTOS does, even though the functions do not cover all the System and Fault Handlers. The following functions are extracted from the `portmacro.h` file:

portmacro.h

```
#define invokeSVCall()                                                          \
{                                                                               \
    __asm volatile( "SVC #0x00" );                                              \
    __asm volatile( "dsb" );                                                    \
    __asm volatile( "isb" );                                                    \
}

#define invokePendSV()                                                          \
{                                                                               \
    /* Set PendSV bit to request a context switch. */                           \
    SCB->ICSR = ( 1UL << 28UL );                                                \
    /* Barriers normally not required but just to be safe */                    \
    __asm volatile( "dsb" );                                                    \
    __asm volatile( "isb" );                                                    \
}
#define clrPendSV()                                                             \
{                                                                               \
    SCB->ICSR = ( 1UL << 27UL )                                                 \
    __asm volatile( "dsb" );                                                    \
    __asm volatile( "isb" );                                                    \
}
#define invokeSysTick()                                                         \
{                                                                               \
    SCB->ICSR = ( 1UL << 26UL )                                                 \
    __asm volatile( "dsb" );                                                    \
    __asm volatile( "isb" );                                                    \
}
```

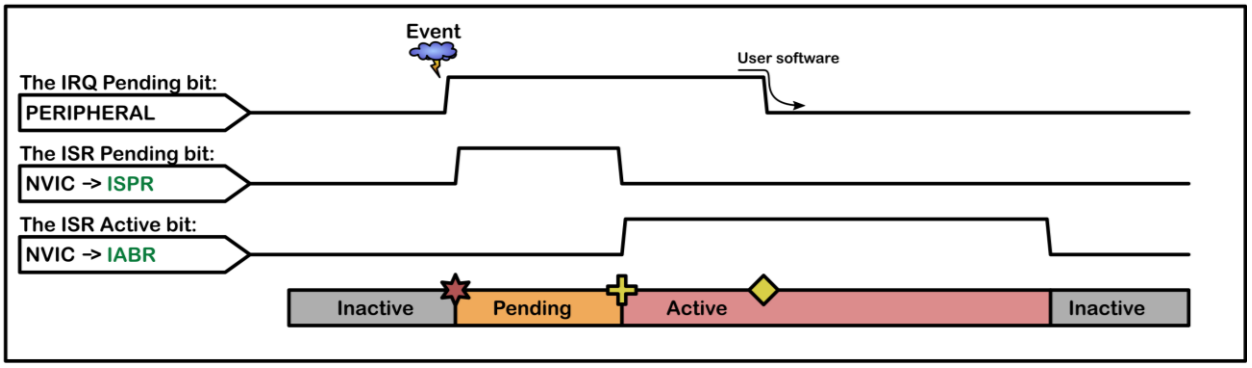
```
#define clrSysTick()                                                    \
{                                                                       \
    SCB->ICSR = ( 1UL << 25UL )                                         \
    __asm volatile( "dsb" );                                            \
    __asm volatile( "isb" );                                            \
}
```

7.2.4 When the interrupt fires...

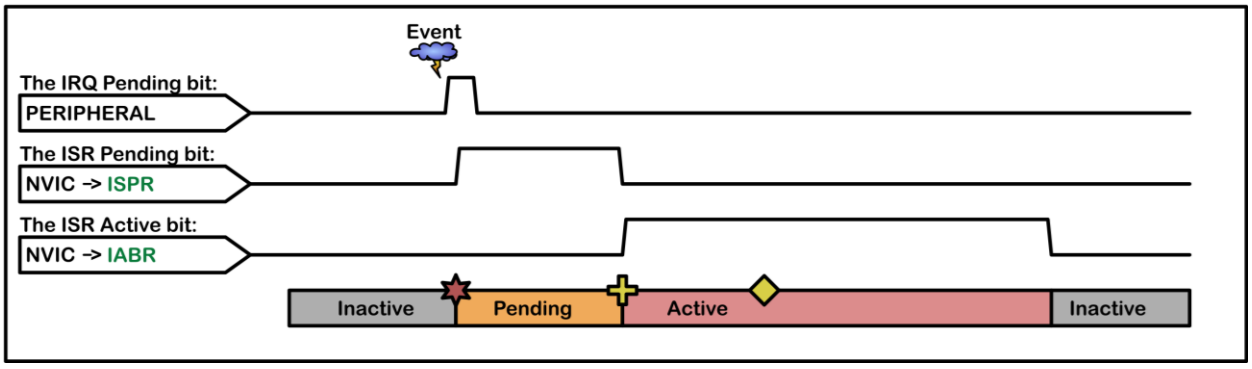
Most of the STM32 peripherals assert a specific signal as soon as an interrupt takes place: the ***IRQ Pending Bit*** (Interrupt Request Pending Bit). This signal is routed to the NVIC controller, to inform it about the interrupt. The IRQ Pending Bit should not be confused with the ISR Pending Bit. The IRQ Pending Bit informs the NVIC controller that a certain interrupt should take place whereas the ***ISR Pending Bit*** reflects the state in which the interrupt is.

interrupt line = IRQ Pending Bit

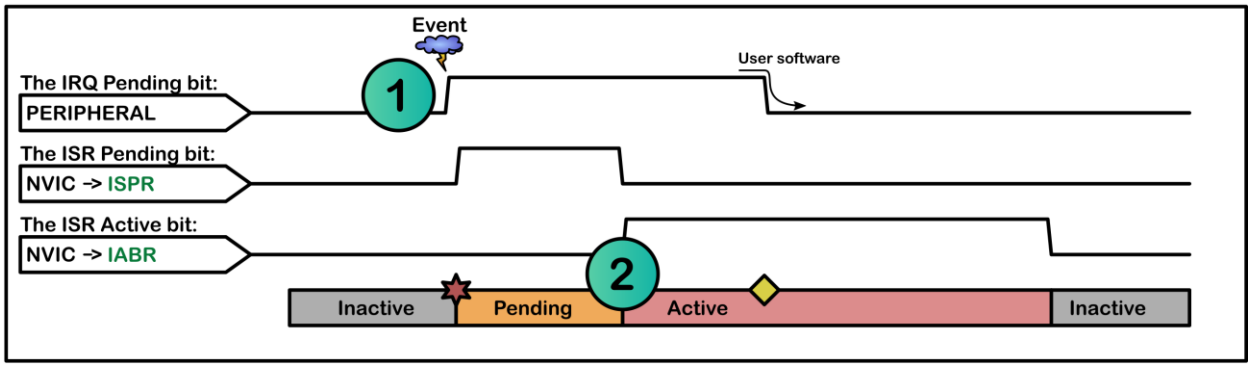
The processor supports both **level-sensitive** and pulse interrupts. A level-sensitive interrupt line is held asserted for a relatively long time. Typically the bit stays high until the user software in the Interrupt Routine clears it. If you forget to do that, the NVIC controller will notice that the interrupt line is still high when finishing the ISR, such that the interrupt becomes pending again.



For a **pulse-interrupt** the peripheral will hold the interrupt line high for a relatively short time. To ensure the NVIC detects the interrupt, the peripheral must assert the interrupt signal for at least one clock cycle, druing which the NVIC detects the pulse and latches the interrupt. But that's a worry for the silicon designers. The embedded software engineer will just notice that he doesn't have to manually clear the IRQ Pending Bit in his ISR routine.



This is a good moment to say a word about enabling interrupts. We know from other paragraphs that the enabling procedure involves two important phases:
(1) Enable the interrupt line(s) in the peripheral. Without doing this, the interrupt line will just stay dead.
(2) Enable the interrupt in the NVIC. Without doing this, the interrupt will never get to the *Active* state, but keeps hanging in the *Pending* state.



7.2.5 Handling interrupts

To handle interrupts, you basically need to do two things:
(1) Write a function (somewhere in your c-files) with the exact same name as an entry from the IVT in the `startup.S` file. The linker will replace that entry with the start address of your function code.
(2) Clear the IRQ Pending bit as soon as possible when the CPU enters the ISR function.

Here is an example for the `EXTI15_10` interrupt:

```
void EXTI15_10_IRQHandler(void)
{
    EXTI -> PR.PR15 = 1;       /* This bit is cleared by programming it to '1'    */

    /* User code here */

}
```

7.2.6 Handling interrupts using the HAL

STMicroelectronics provides a few HAL-functions to do these things. The following function is a general EXTI interrupt handler. It takes the `GPIO_Pin` as input, and clears the corresponding IRQ Pending bit. Next it jumps to the callback function that the user should implement.

File: stm32f7xx_hal_gpio.c

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    /* EXTI line interrupt detected */
    if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}
```

The previous function used a macro to clear the IRQ Pending bit:

```
File: stm32f7xx_hal_gpio.h

/**
  * @brief  Clears the EXTI's line pending bits.
  * @param  __EXTI_LINE__: specifies the EXTI lines to clear.
  *          This parameter can be any combination of GPIO_PIN_x where x can be (0..15)
  * @retval None
  */

#define __HAL_GPIO_EXTI_CLEAR_IT(__EXTI_LINE__) (EXTI->PR = (__EXTI_LINE__))
```

Now the user code should implement the following functions:

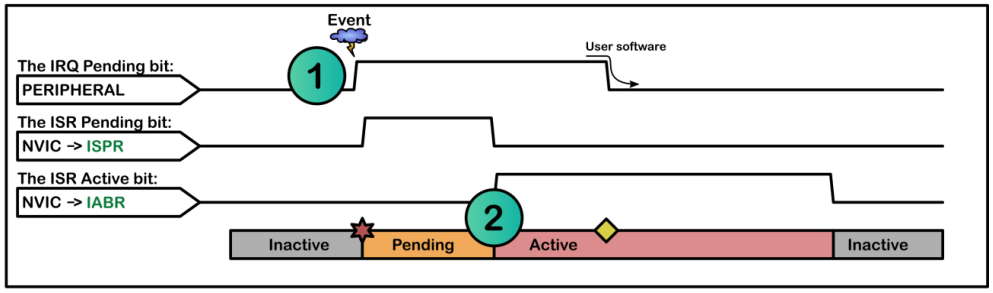
```
/**
 * This is the ISR routine where the CPU starts its execution at the moment the interrupt
 * appears.
 */
void EXTI15_10_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);   // Go to the general EXTI interrupt handler
                                             // from the ST HAL.
}

/**
 * This is the callback function, called by HAL_GPIO_EXTI_IRQHandler().
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_15)
    {
        /* User code here */
    }
}
```

7.3 Enabling Interrupts

Before an interrupt can be used, the programmer should first *enable* it. The enable procedure differs between the two different kinds of interrupts. In case you forgot: 'internal' vs 'external'. Internal interrupts belong to the processor core. They are designed and implemented by the ARM engineers. By 'external' we refer to those interrupts that do not belong to the processor core, but are implemented by the silicon vendor STMicroelectronics. These interrupts are mostly attached to peripherals.

7.3.1 Enable 'external' interrupts

Remember this figure. As you know, the enabling procedure involves two phases.



So the user software must do the following two things:

- Inform the involved peripheral(s) (eg. Timer, GPIO, ...)
- Inform the NVIC controller

Let us consider setting GPIO pin PC15 as "interrupt pin" as an example.

(a) Inform the involved peripheral(s)

Quite some peripherals are involved:

- **SYSCFG** (System Configuration Controller) – some might argue this is not a standalone peripheral.
- **EXTI** (Extended Interrupts and Events Controller) – some might argue this is not a standalone peripheral.
- **GPIOC**

The following code will inform all of these peripherals that pin PC15 will generate an interrupt on a rising edge. Note: This code will not work straight away. The bitfields from the registers can be made accessible if you define the proper structs.

```
/*------------------------------GPIOC-------------------------------------------------------*/
RCC -> AHB1ENR.GPIOCEN = 0b1;          /* Enable GPIOC clock.                                 */
delay(..)                              /* Wait until clock is stable.                         */
GPIOC -> MODER.MODER15 = 0b00;         /* Input mode for pin 15 from port C.                  */
GPIOC -> PUPDR.PUPDR15 = 0b10;         /* Pull-down mode.                                     */

/*------------------------------SYSCFG------------------------------------------------------*/
RCC -> APB2ENR.SYSCFGEN = 0b1;         /* Enable SYSCFG clock.                                */
delay(..)                              /* Wait until clock is stable.                         */
SYSCFG -> EXTICR4.EXTI15 = 0b0010;     /* Connect EXTI line 15 to pin PC15.                   */

/*------------------------------EXTI--------------------------------------------------------*/
EXTI -> IMR.MR15 = 0b1;                /* Interrupt request from line 15 is not masked.       */
EXTI -> RTSR.TR15 = 0b1;               /* Rising trigger enabled for line 15.                 */
EXTI -> FTSR.TR15 = 0b0;               /* Falling trigger disabled for line 15.               */
```

The same can be achieved through using the HAL-functions. The single function `HAL_GPIO_Init(..)` will not only configure the `GPIOC` registers, but also those from `SYSCFG` and `EXTI`.

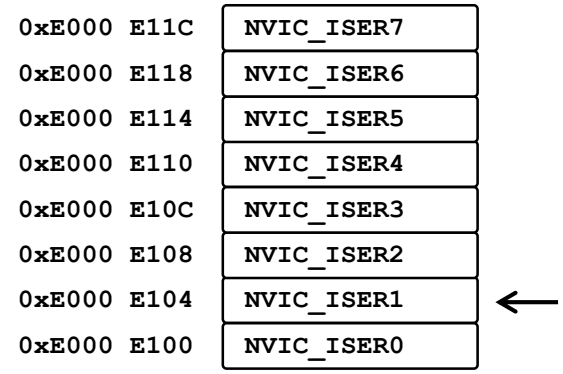
```
GPIO_InitTypeDef GPIO_InitStruct;

__HAL_RCC_GPIOC_CLK_ENABLE();
GPIO_InitStruct.Pin = GPIO_PIN_15;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_PULLDOWN;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

(b) Inform the NVIC

We know from the Interrupt Vector Table that the interrupt `EXTI15_10_IRQn` has number 40. At least, that's the number when you start counting from zero in the list of external interrupts. To inform the NVIC that interrupt number 40 should be enabled, we have to set the 40th bit from the array `NVIC->ISER[0..7]`. Indeed, the NVIC has an array of 8 registers from which every bit corresponds to enabling/disabling one interrupt.

8 registers x 32 bits = 256 bits

So these registers could reach 256 interrupts in total. We know that these registers only serve the so called 'external' interrupts, and the silicon vendor can at most implement 240 such interrupts. So register NVIC_ISER7 will always have its upper 16 bits unused.



We have to set the 40th bit from the array, so that corresponds to the 8th bit from the second `ISER` register. So we get:

```
NVIC -> ISER[1] = 1 << 8;     /* Set the 8th bit from the second ISER register */
```

The same can be achieved through the HAL-function:

```
HAL_NVIC_EnableIRQ(40);
```

Which is basically a simple wrapper of the following CMSIS function. It does not matter which one you use. The HAL-function is just a wrapper of the CMSIS function. In other words, the HAL-function does nothing more than calling the CMSIS function. The CMSIS function will set the right bit in the **`NVIC`**`->ISER` registers.

```
NVIC_EnableIRQ(40);
```

Below is the internal code of the **`NVIC_EnableIRQ`**`(IRQn_Type IRQn)` function. Notice that the comment states the value cannot be negative. This proves that the 'internal' interrupts – System Handlers and Fault Handlers – cannot be reached through these NVIC registers. They need special attention. We will get to them shortly.

```
/**
  \brief   Enable External Interrupt
  \details Enables a device-specific interrupt in the NVIC interrupt controller.
  \param [in]      IRQn  External interrupt number. Value cannot be negative.
 */
__STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
{

  NVIC->ISER[(((uint32_t)(int32_t)IRQn) >> 5UL)] = (uint32_t)(1UL << (((uint32_t)(int32_t)IRQn) & 0x1FUL));

}
```

To make things more simple, you can replace the number '40' in the code above by the enum value `EXTI15_10_IRQn`. As mentioned before, these interrupt numbers are defined in an enum in the `stm32f746xx.h` file (from the CMSIS folder).

7.3.2 Enable Internal interrupts

The internal interrupts are mostly `Fault Handlers` and `System Handlers`. The `NVIC_ISER` registers do not cover them. This is how you enable them:

Reset	Permanently enabled	
NMI	Permanently enabled	
HardFault	Permanently enabled(?)	
MemManage	MemManage enable bit:	`SCB->SHCSR.MEMFAULTENA`
BusFault	BusFault enable bit:	`SCB->SHCSR.BUSFAULTENA`
UsageFault	UsageFault enable bit:	`SCB->SHCSR.USGFAULTENA`
SVCall	Permanently enabled(?)	
PendSV	Permanently enabled(?)	
SysTick	Enable counter: Enable exception:	`SysTick->CTRL.ENABLE` `SysTick->CTRL.TICKINT`

Since these 'internal' interrupts do not belong to any external peripheral, the enabling procedure is just one phase.

7.4 Interrupt priorities

Before going into more details, I want to emphasize the following very important convention:

Cortex M Convention:
HIGH PRIORITY
= LOW NUMERICAL VALUE

FreeRTOS Convention:
HIGH PRIORITY
= HIGH NUMERICAL VALUE

(lowest priority is 0)

7.4.1 Meaning of the Priority bits

The priority of every interrupt is defined by a one-byte field. The effect from each of those 8 bits on the final priority depends on a number of factors. This paragraph digs into those factors, and how you can adjust them to fit your needs.

(a) Implemented vs non-implemented bits

Every interrupt priority is defined by 1 byte. In theory this would allow a choice of 256 priority levels. In practice the silicon vendors only support a subset. The STM32F7 chips from STMicroelectronics support 16 priority levels (source: STM32F7 Reference Manual):

```
*
*  +----+----+----+----+----+----+----+----+
*  | b7 | b6 | b5 | b4 |////|////|////|////|
*  +----+----+----+----+----+----+----+----+
*  16 priority levels        not implemented
*  implemented
*       |
*       `->  In the file "stm32f746xx.h" you will find
*            nr. implemented prio bits   =   NVIC PRIO BITS         -> for STM32F7 this is 4
*            nr. unimplemented prio bits = (8 - __NVIC_PRIO_BITS)   -> for STM32F7 this is 4
*
```

This is fixed in silicon. You cannot change it!

stm32f746xx.h

```
#define __NVIC_PRIO_BITS          4       /*!< CM7 uses 4 Bits for the Priority Levels       */
```

Note that the numerical value of the priority is shifted to the left by the number of unimplemented priority bits. If you ever write directly to the priority registers in the NVIC, you must remember to follow this convention!

(b) Group priority vs subpriority bits

To increase priority control in systems with interrupts, the NVIC supports priority grouping. This divides each interrupt priority register entry into two fields:

* An upper field that defines the group priority.
* A lower field that defines a subpriority within the group.

"group priority" and "pre-empt priority" are synonyms

Only the group priority determines preemption of interrupt exceptions. When the processor is executing an interrupt exception handler, another interrupt with the same group priority as the interrupt being handled does not preempt the handler.
If multiple pending interrupts have the same group priority, the subpriority field determines the order in which they are processed. If multiple pending interrupts have the same group priority and subpriority, the interrupt with the lowest IRQ number is processed first.

The interrupt priority grouping is determined by the PRIGROUP field:

SCB->AIRCR.**PRIGROUP**

The **PRIGROUP** field indicates the position of the binary point that splits the PRI_n fields from the Interrupt Priority Registers into separate "group priority" and "subpriority" bits. For example, if **PRIGROUP** equals 0b101 (=5) the following split applies:

```
*
*                                    (PRIGROUP + 1)
*                                  |<----------------
*             PRI_n:          x  x  y  y  y  y  y  y
*                            \_ __/\_______ _______/
*                              '           '
*                       Group pr bits    Subpr bits
*                          [7:6]            [5:0]
*                                            |
*                                            `-> PRIGROUP = 5
*
```

This split seems to be independent from the number of supported priority levels. In other words, the split is executed on the complete byte, regardless of how many bits are unimplemented. Continuing the example above with **PRIGROUP** equal to 5:

```
*
*                                    (PRIGROUP + 1)
*                                |<-----------------------------
*                   +----+----+  +----+----+----+----+----+----+
*                   | b7 | b6 |  | b5 | b4 |////|////|////|////|
*                   +----+----+  +----+----+----+----+----+----+
*                   `----,----´  `----,----´
*                Group pr bits    Subpr bits
*                     [7:6]          [5:4]
*
```

7.4.2 Making the split (set the group priority)

The stm32f7xx_hal_cortex.h file from the HAL folder provides a few **#define** statements to facilitate setting the **PRIGROUP** value. The comments on the right describe what happens when you assign a certain value to the **PRIGROUP** field.

stm32f7xx_hal_cortex.h

```
/* ---------------------------------------------------------,-----------------------------------,   */
/*                                                          |                                   |   */
/*    The value to fill in into SCB->AIRCR.PRIGROUP         |   The resulting split             |   */
/*   ____________________________________________           |   _____________________           |   */
/*                                                          |                                   |   */
#define NVIC_PRIORITYGROUP_0     ((uint32_t)0x00000007) /*   0 bits for pre-emption priority  |   */
                                                        /*   4 bits for subpriority           |   */
#define NVIC_PRIORITYGROUP_1     ((uint32_t)0x00000006) /*   1 bits for pre-emption priority  |   */
                                                        /*   3 bits for subpriority           |   */
#define NVIC_PRIORITYGROUP_2     ((uint32_t)0x00000005) /*   2 bits for pre-emption priority  |   */
                                                        /*   2 bits for subpriority           |   */
#define NVIC_PRIORITYGROUP_3     ((uint32_t)0x00000004) /*   3 bits for pre-emption priority  |   */
                                                        /*   1 bits for subpriority           |   */
#define NVIC_PRIORITYGROUP_4     ((uint32_t)0x00000003) /*   4 bits for pre-emption priority  |   */
                                                        /*   0 bits for subpriority           |   */
/*---------------------------------------------------------`-----------------------------------´   */
```

The ideal moment to make a decision on the split between “group” and “subpriority” is at the very beginning of your program. This is exactly what happens in the HAL initialization phase:

stm32f7xx_hal.c

```
HAL_StatusTypeDef HAL_Init(void)
{
   __HAL_FLASH_ART_ENABLE();

  HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

  HAL_InitTick(TICK_INT_PRIORITY);

  HAL_MspInit();  /* Init the low level hardware */


  return HAL_OK;
}
```

Apparently the default priority grouping is:
- 4 bits for **pre-emption** priority
- 0 bits for **subpriority**

The function **HAL_NVIC_SetPriorityGrouping**(...) is a wrapper for the CMSIS function **NVIC_SetPriorityGrouping**(...). This function simply writes its parameter to the PRIGROUP field:

SCB->AIRCR.**PRIGROUP**

But before doing so, the function will make sure that the given parameter is not out of range, and that the proper key is written to the VECTKEY field. Otherwise, the processor ignores the write.

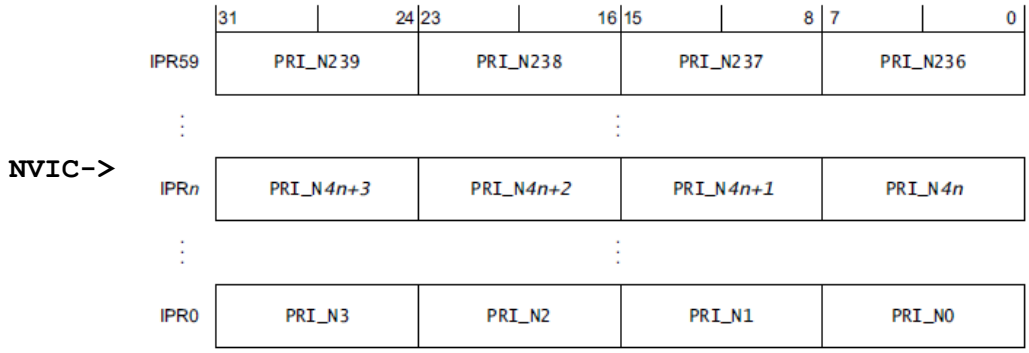
core_cm7.h

```
__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
{
    uint32_t reg_value;

    /* only values 0..7 are used */
    uint32_t PriorityGroupTmp = (PriorityGroup & (uint32_t)0x07UL);

    /* read old register configuration    */
    reg_value  =  SCB->AIRCR;

    /* clear bits to change  */
    reg_value &= ~((uint32_t)(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk));

    /* Insert write key and priorty group */
    reg_value  =  (reg_value                                   |
                  ((uint32_t)0x5FAUL << SCB_AIRCR_VECTKEY_Pos) |
                  (PriorityGroupTmp << 8U)                      );

    SCB->AIRCR =  reg_value;
}
```

7.4.3 Priority fields

(a) Priority fields for 'external' interrupts

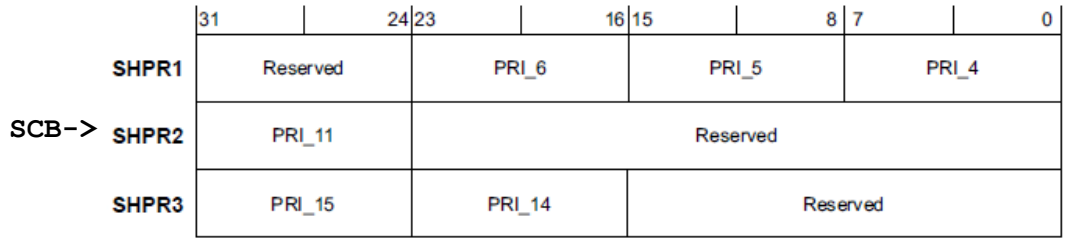
The `NVIC_IPR0 - NVIC_IPR59` registers provide a priority field for each interrupt. These registers are byte-accessible. Each register holds four priority fields as shown:



Note that the NVIC provides 240 fields in total. This is the maximum amount of 'external' interrupts supported (remember that 'external' in this context means: an interrupt not belonging to the core but to a peripheral implemented by the silicon vendor).

(b) Priority fields for 'internal' interrupts

For the 'internal' interrupts, one should look to the `SCB` (System Control Block) instead:



Handler	Field
MemManage	PRI_4
BusFault	PRI_5
UsageFault	PRI_6
SVCall	PRI_11
PendSV	PRI_14
SysTick	PRI_15

(c) Adjust priority fields

One could directly write to the priority fields for adjusting the interrupt priority. But both ARM And STMicro-electronics provide some library functions to facilitate the priority adjustments. STMicroelectronics provides the following function in the HAL folder:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority)
```

This function expects at first the `IRQ` number to identify the interrupt. Next it asks the `PreemptPriority` and `SubPriority` from which it will calculate the proper sequence of bits that need to be written to the `NVIC PRI_N` field.
When this calculation is finished, it will call the CMSIS function to do the actual register adjustement:

```
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
```

The function expects to get the completely calculated `priority` value that should be written to the registers. The only operation this CMSIS function will do is shifting the value 4 steps to the left. This corresponds to the number of unimplemented priority bits in the STM32F7 chip.

Both functions **`HAL_NVIC_SetPriority(..)`** and **`NVIC_SetPriority(..)`** can handle negative `IRQn` numbers. So they can be used to set the priorities of 'internal' interrupts (System Handlers and Fault Handlers). The functions detect that the number is negative, and refrain from interacting with the NVIC registers. Instead, they will interact with the proper SCB registers.

We shall now explore the whole procedure, starting with the HAL function.

stm32f7xx_hal_cortex.c

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority)
{
  uint32 t prioritygroup = 0x00;

  /* Check the parameters */
  assert_param(IS_NVIC_SUB_PRIORITY(SubPriority));
  assert_param(IS_NVIC_PREEMPTION_PRIORITY(PreemptPriority));

  prioritygroup = NVIC_GetPriorityGrouping();

  NVIC_SetPriority(IRQn, NVIC_EncodePriority(prioritygroup, PreemptPriority, SubPriority));
}
```

core_cm7.h

```
/**
 * This function simply returns the SCB->AIRCR.PRIGROUP field
 */
__STATIC_INLINE uint32_t NVIC_GetPriorityGrouping(void)
{
  return ((uint32_t)((SCB->AIRCR & SCB_AIRCR_PRIGROUP_Msk) >> SCB_AIRCR_PRIGROUP_Pos));
}
```

core_cm7.h

```
/**
  \brief   Encode Priority
  \details Encodes the priority for an interrupt with the given priority group,
           preemptive priority value, and subpriority value.
           In case of a conflict between priority grouping and available
           priority bits (__NVIC_PRIO_BITS), the smallest possible priority group is set.

  \param [in]     PriorityGroup  Used priority group. This parameter normally corresponds to the SCB->AIRCR.PRIGROUP
                                 register.

  \param [in]   PreemptPriority  Preemptive priority value. The value starts from 0 and can go up to (2^n - 1) with n
                                 the number of PreemptPriorityBits.

  \param [in]       SubPriority  Subpriority value. The value starts from 0 and can go up to (2^n - 1)  with n the
                                 number of SubPriorityBits.

  \return                        Encoded priority. This value can be used in the function \ref NVIC_SetPriority().
                                 Note that the encoding is actually not complete. In the function NVIC SetPriority()
                                 the final left shift by (8U - __NVIC_PRIO_BITS) takes place.
 */
__STATIC_INLINE uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)
{
    uint32 t PriorityGroupTmp = (PriorityGroup & (uint32 t)0x07UL);   /* only values 0..7 are used          */
    uint32_t PreemptPriorityBits;
    uint32_t SubPriorityBits;

    /*
     * I have tested the following two code lines, and found this truth table:
     *    +--------------------+---------------------+------------------+
     *    | PriorityGroup(Tmp) | PreemptPriorityBits | SubPriorityBits  |
     *    +--------------------+---------------------+------------------+
     *    | 0                  | 4                   | 0                |
     *    | 1                  | 4                   | 0                |
     *    | 2                  | 4                   | 0                |
     *    | 3                  | 4                   | 0                | <-- This is the default
     *    | 4                  | 3                   | 1                |
     *    | 5                  | 2                   | 2                |
     *    | 6                  | 1                   | 3                |
     *    | 7                  | 0                   | 4                |
     *    +--------------------+---------------------+------------------+
     */
    PreemptPriorityBits = ((7UL - PriorityGroupTmp) > (uint32_t)(__NVIC_PRIO_BITS)) ? (uint32_t)(__NVIC_PRIO_BITS) :
                                                                                       (uint32 t)(7UL - PriorityGroupTmp);

    SubPriorityBits     = ((PriorityGroupTmp + (uint32_t)(__NVIC_PRIO_BITS)) < (uint32_t)7UL) ? (uint32_t)0UL :
                                                 (uint32_t)((PriorityGroupTmp - 7UL) + (uint32_t)(__NVIC_PRIO_BITS));
```

```
    /*
     * After the number of PreemptPriorityBits and SubPriorityBits is calculated, the following three operations get
     * executed:
     *
     * (1) PreemptPriority &= 0b0000 0111; //In case PreemptPriorityBits = 3
     * (2) SubPriority &= 0b0000 0001;     //In case SubPriorityBits = 1
     * (3) PreemptPriority <<= 1;          //In case SubPriorityBits = 1
     *
     * Finally the following result is returned:
     *
     *      return (PreemptPriority | SubPriority);
     *
     * Note that all these operations are executed in the following return statement:
     */
    return (
              ((PreemptPriority & (uint32_t)((1UL << (PreemptPriorityBits)) - 1UL)) << SubPriorityBits) |
              ((SubPriority     & (uint32_t)((1UL << (SubPriorityBits    )) - 1UL)))
            );
    /*
     * The returned value can be used in the NVIC SetPriority(..) function. The first thing this NVIC SetPriority(..)
     * function will do is to left-shift the value by 4 (that is for the STM32F7 in which the first four bits are not
     * used).
     */
}
```

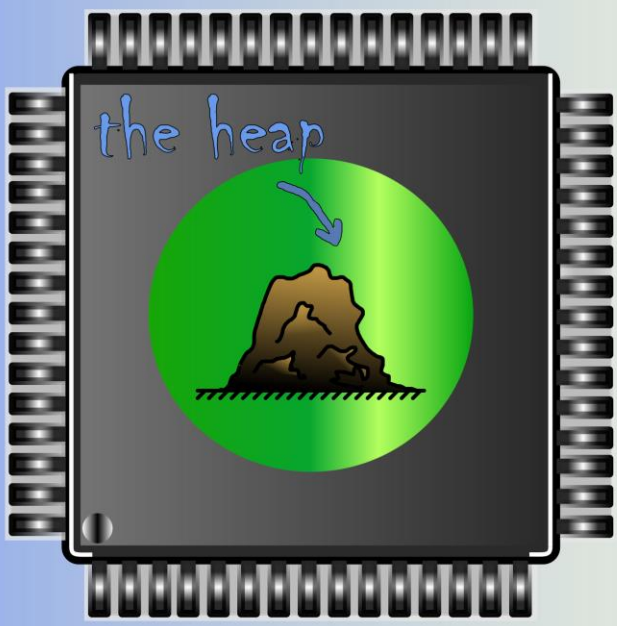
core_cm7.h

```
/**
  \brief   Set Interrupt Priority
  \details Sets the priority of an interrupt.
  \note    The priority cannot be set for every core interrupt.
  \param [in]      IRQn       Interrupt number.
  \param [in]      priority   Priority to set.
 */
__STATIC_INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
  if ((int32_t)(IRQn) < 0)
  {
    SCB->SHPR[(((uint32_t)(int32_t)IRQn) & 0xFUL)-4UL]
                        = (uint8_t)((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
  }
  else
  {
    NVIC->IP[((uint32_t)(int32_t)IRQn)]
                        = (uint8_t)((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
  }
}
```

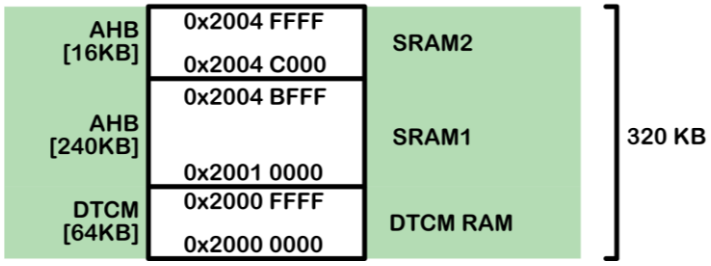
The FreeRTOS operating system

Chapter 1

FreeRTOS memory layout



Understanding the memory layout is a prerequisite for gaining deeper insight in FreeRTOS. We know that a big part of the memory layout is "burned in the silicon". Think about the peripheral registers. Each of those registers has a fixed memory address. But this paragraph is about the GP RAM – General Purpose RAM. The STM32F7 microcontroller provides 320 KB of this memory type. The way this RAM gets allocated is purely defined by software.



When you write software in C, your variables get a location in RAM (in contrast with constants that are usually placed in FLASH).

A variable can be:

Static variables: Static variables get a fixed address in RAM. They 'live' for the entire duration of your program. They can have a global scope, file scope or even function scope. But the scope has nothing to do with the fundamental property of a static variable: the fact that its address in RAM is fixed – throughout the entire program.

Automatic variables: Automatic variables get an address on the stack. These are the local variables that get created inside a function, although 99% of the C programmers do not use the 'automatic' keyword. As soon as the function returns (= has finished), all the automatic variables are removed from the stack.

Dynamic variables: Dynamic variables get an address in the heap. Your program has to allocate memory for their creation through the `malloc()` function. They live on the heap until you kill them. Call the `free()` function to do that. Note that many simple embedded systems do not even have a heap (and thus no dynamic variables).

1.1 Linkerscript memory layout

The linkerscript determines the general layout of the RAM. Let us therefore analyse those parts of the linkerscript that apply to the RAM:

We know that ARM uses a "full descending stack". Therefore it is quite logic that the stack starts at the end of RAM.
We also know that the STM32F7 microcontroller uses two stack pointers: MSP (Main SP) and PSP (Process SP). This `_estack` definition applies only to the MSP. FreeRTOS assigns a "process stack" for each task. We will consider that later on.

LinkerScript.ld

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the stack */
_estack = 0x20050000;    /* end of RAM */

/* Generate a link error if heap and stack don't fit into RAM */
 Min Heap Size = 0x200;  /* 512 bytes: required amount of heap  */
 Min Stack Size = 0x400; /* 1 Kbytes:  required amount of stack */

/*----------------------------------------------------------------------------------------*/
/*                                MEMORY AREAS                                            */
/*----------------------------------------------------------------------------------------*/
MEMORY
{
    FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
    RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 320K
}

/*----------------------------------------------------------------------------------------*/
/*                                OUTPUT SECTIONS                                         */
/*----------------------------------------------------------------------------------------*/
SECTIONS
{
    ...
```

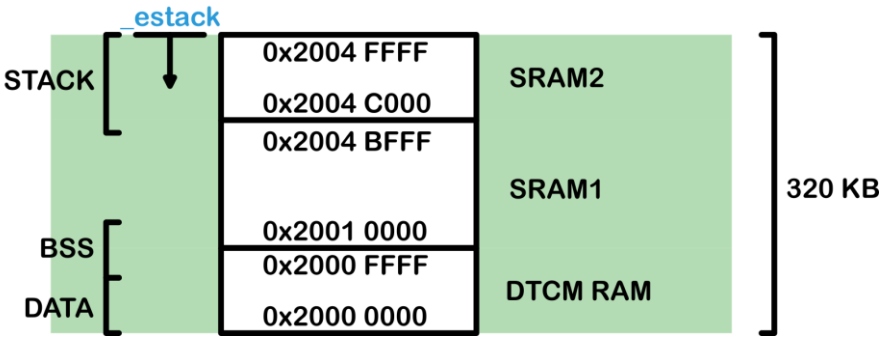
```
    /****************************/
    /*     INITIALIZED DATA     */
    /****************************/
    _sidata = LOADADDR(.data);
    .data :
    {
        . = ALIGN(4);
         sdata = .;        /* create a global symbol at data start */
        *(.data)           /* .data sections */
        *(.data*)          /* .data* sections */

        . = ALIGN(4);
         edata = .;        /* define a global symbol at data end */

    } >RAM AT> FLASH

    /****************************/
    /*    UNINITIALIZED DATA    */
    /****************************/
    . = ALIGN(4);
    .bss :
    {
        _sbss = .;         /* define a global symbol at bss start */
          bss start    =  sbss;
        *(.bss)
        *(.bss*)
        *(COMMON)

        . = ALIGN(4);
         ebss = .;         /* define a global symbol at bss end */
          bss end    =  ebss;

    } >RAM



    /****************************/
    /* USER_HEAP_STACK SECTION  */
    /****************************/
    /* User_heap_stack section, used to check that there is enough RAM left */
    . user heap stack :
    {
        . = ALIGN(8);
        PROVIDE ( end = . );
        PROVIDE ( _end = . );
        . = . + _Min_Heap_Size;
        . = . +  Min Stack Size;
        . = ALIGN(8);
    } >RAM
}
```

The initialized `.data` section goes both into FLASH and RAM. Its address in FLASH is called the "load address", and its address in RAM is the "runtime address". The startup code needs to make the copy from FLASH to RAM.
This `.data` section is typically used for global or static variables in your C program that have an initial value.

The uninitialized `.bss` section goes only in RAM. The startup code needs to fill this section with zeros.
This `.bss` section is typically used for global or static variables in your C program that have an initial value equal to zero.

The `._user_heap_stack` output section is only used to check that there is enough RAM left. There are no 'input sections' defined here. This section is thus empty.

The following figure shows how the linkerscript has arranged the general purpose RAM. Notice that the `.data` and `.bss` sections are placed at the beginning of RAM. They contain the global and static variables from your C program. The stack is positioned at the end of RAM. That's a logic decision considering the fact that ARM uses a "full descending stack".



Later on we will analyse the several "process stacks" that belong to created tasks in the FreeRTOS operating system. But for now it's sufficient to know that the "main stack" starts at the end of RAM.

1.2 FreeRTOS heap initialization

The FreeRTOS operating system builds further on the memory layout defined in the linkerscript. The operating system will do the following:

The stack: The linkerscript positioned the stack at the end of RAM. And that's okay for the "main stack" which is used for interrupts and other code that runs outside the operating system. FreeRTOS provides a "process stack" for every created task.

The heap: The linkerscript did not really create a heap. The script just checked if there is enough space for one. FreeRTOS will go one step further and really creates one.

This paragraph digs deeper into the FreeRTOS heap. FreeRTOS provides many implementations. For more information refer to the FreeRTOS website. Every implementation comes with its own `heap_n.c` file. I have chosen for the `heap_4.c` version as a compromise between simplicity and functionality.

The very first – and most important – code you find in that file is:

heap_4.c

```
/*-----------------------------------------------------------*/
/*                  ALLOCATE THE MEMORY FOR THE HEAP         */
/*                                                           */
            static uint8_t heap[ configTOTAL_HEAP_SIZE ];
/*                                                           */
/*-----------------------------------------------------------*/
```

15 360

So the heap is nothing more than an array of 15.360 consecutive bytes in RAM. Where will this array be located? When you compile the file `heap_4.c` and you look at the "disassembly" (the raw assembly code), you will see:

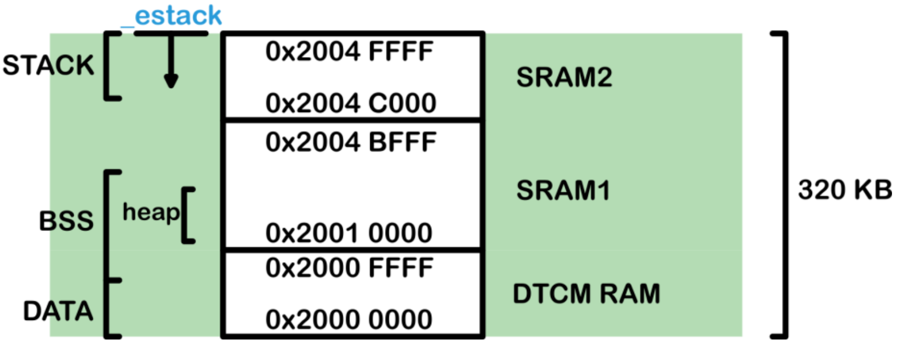
```
Disassembly of section .bss:

    00000000 <heap>:                                    /* Static variable 'heap'          */
            ...

    00003c00 <xStart>:                                  /* Static variable 'xStart'        */
            ...

    00003c08 <p_End>:                                   /* Static variable 'p_End'         */
        3c08:    00000000         andeq    r0, r0, r0

    00003c0c <freeBytesRemaining>:                      /* Static variable 'freeBytesRemaining'       */
        3c0c:    00000000         andeq    r0, r0, r0

    00003c10 <minEverFreeBytesRemaining>:               /* Static variable 'minEverFreeBytesRemaining */
        3c10:    00000000         andeq    r0, r0, r0

    00003c14 <xBlockAllocatedBit>:                      /* Static variable 'xBlockAllocatedBit'       */
        3c14:    00000000         andeq    r0, r0, r0
```

From this snippet in the assembly code you can conclude that:

- The variable `'heap'` is one of the many static variables in the `heap_4.o` file.
- The variable `'heap'` is located at the beginning of the `.bss` section of the `heap_4.o` file.
- The variable `'heap'` consumes `0x3C00` memory locations in that `.bss` section.

Note that the last conclusion corresponds to 15 360 bytes. That is the total heap size we've defined at the start. You are of course free to increase or decrease that value according to your needs.

At the final compilation stage – when the linker takes over – the heap is put in the RAM memory somewhere in the `.bss` section. The heap has a constant size of 15 360 bytes. The following figure gives an impression of the RAM memory layout:



The fact that the heap is a fixed memory block inside the `.bss` section makes it technically speaking equivalent to any other statically defined array. It is not really a heap in the purest sense. Nevertheless, this implementation works fine, it's simple and predictable. In other words perfect for lightweight embedded applications.

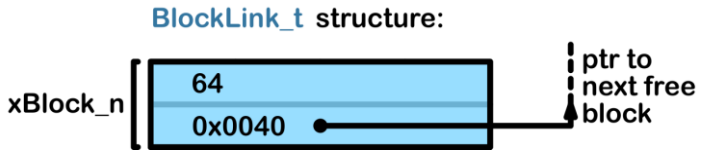
Now let us consider the initialization of the heap.

heap_4.c

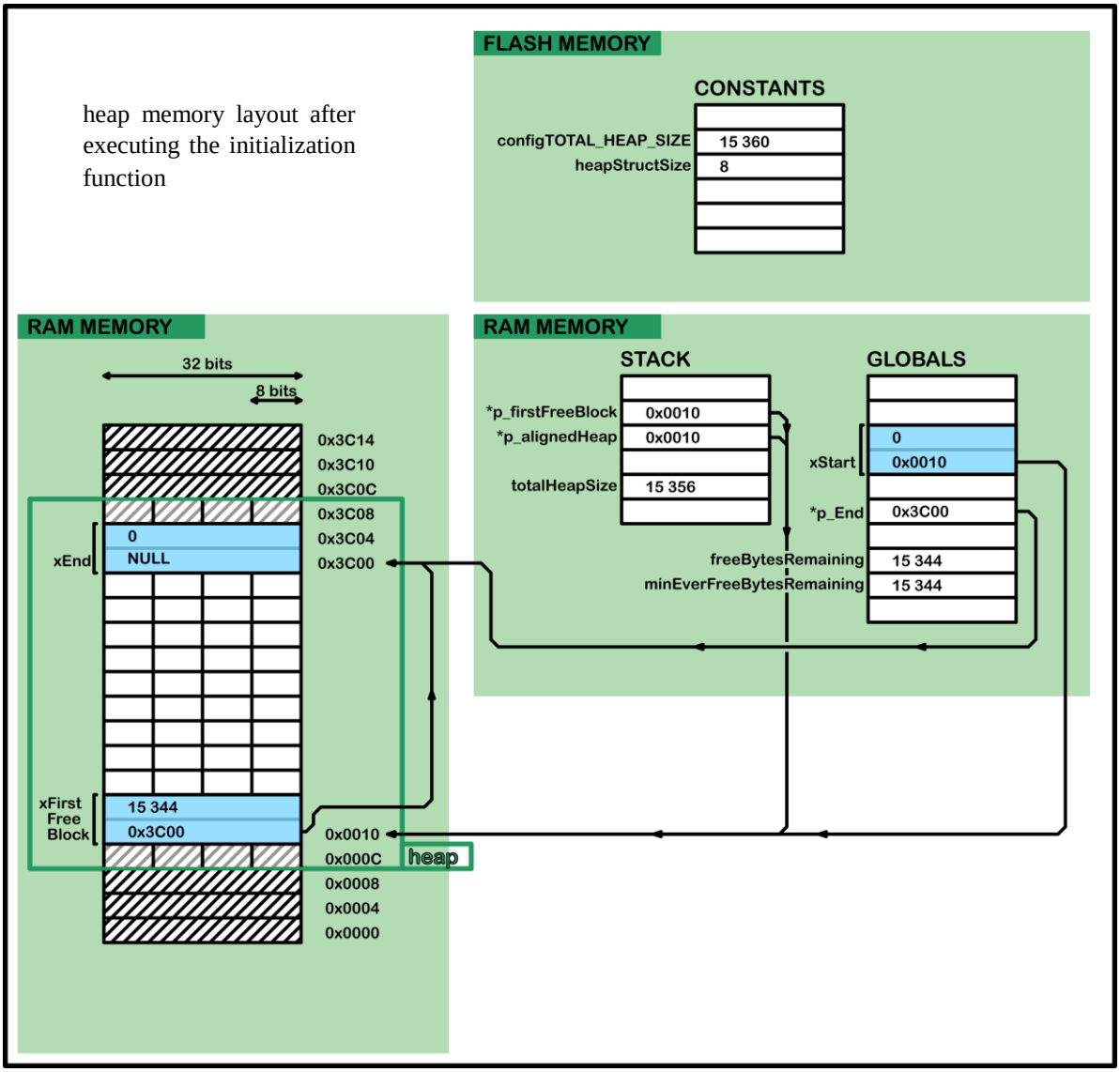
```
/*********************************************************************   15 360
/*                  ALLOCATE THE MEMORY FOR THE HEAP                  */
/*                                                                    */
            static uint8_t heap[ configTOTAL_HEAP_SIZE ];
/*                                                                    */
/**********************************************************************/


/* Define the linked list structure.  This is used to link free blocks in order     */
/* of their memory address.                                                         */
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *p_nextFreeBlock;   /*<< The next free block in the list. */
    uint32_t size;                          /*<< The size of the free block.      */
} BlockLink_t;

static BlockLink_t xStart, *p_End = NULL;           //
static uint32_t freeBytesRemaining = 0U;            //
static uint32_t minEverFreeBytesRemaining = 0U;     //
static const uint32_t heapStructSize = 8;           // Must be a multiple of 8.

/* ------------------------------------------------------------------------------------*/
/*                                                                                     */
/*      |_ _|  _ _    (_) | |_   (_)  __ _   | | (_)  ___  ___                          */
/*       | |  | ' \   | | |  _|  | | / _` |  | | | | |_ / / -_)                         */
/*      |___| |_||_|  |_|  \__|  |_| \__,_|  |_| |_| /__| \___|                         */
/* ------------------------------------------------------------------------------------*/

static void prvHeapInit( void )
{
    BlockLink_t *p_firstFreeBlock;
    uint8_t *p_alignedHeap;
    uint32_t addr;
    uint32_t totalHeapSize = configTOTAL_HEAP_SIZE;     //15 K

    /* Ensure the heap starts on a correctly aligned boundary. */
    addr = ( uint32_t ) heap;
    if( ( addr & (0x0007) ) != 0 )
    {
        /*********************************/
        /*    TRUTH TABLE:               */
        /* 0    ->  0       17  ->  24   */
        /* 1    ->  8       18  ->  24   */
        /* ..               ..           */
        /* 8    ->  8       24  ->  24   */
        /*                               */
        /*                               */
        /* 9    ->  16                   */
        /* 10   ->  16                   */
        /* ..                            */
        /* 16   ->  16                   */
        /*                               */
        /*********************************/
```

```
        addr += ( 8 - 1 );
        addr &= ~( ( uint32_t ) (0x0007) );
        totalHeapSize -= (addr - ( uint32_t ) heap);
    }

    p_alignedHeap = ( uint8_t * ) addr;

    /* xStart is used to hold a pointer to the first item in the list of free
    blocks.  The void cast is used to prevent compiler warnings. */
    xStart.p_nextFreeBlock = ( void * ) p_alignedHeap;
    xStart.size = ( uint32_t ) 0;

    /* pxEnd is used to mark the end of the list of free blocks and is inserted
    at the end of the heap space. */
    addr = ( ( uint32_t ) p_alignedHeap ) + totalHeapSize;
    addr -= heapStructSize;
    addr &= ~( ( uint32 t ) (0x0007) );
    p End = ( void * ) addr;
    p_End->size = 0;
    p_End->p_nextFreeBlock = NULL;

    /* To start with there is a single free block that is sized to take up the
    entire heap space, minus the space taken by xEnd. */
    p firstFreeBlock = ( void * ) p alignedHeap;
    p_firstFreeBlock->size = addr - ( uint32_t ) p_firstFreeBlock;
    p_firstFreeBlock->p_nextFreeBlock = p_End;

    /* Only one block exists - and it covers the entire usable heap space. */
    minEverFreeBytesRemaining = p_firstFreeBlock->size;
    freeBytesRemaining = p_firstFreeBlock->size;

    /* Work out the position of the top bit in a uint32_t variable. */
    xBlockAllocatedBit = ( (uint32_t) 1 ) << ( (uint32_t)31 );
}
```

The easiest way to get insight in this code is using an example. Suppose that the linker has assigned the heap array to memory space `0x000C` to `0x3C08`. That is exactly 15 360 bytes. I know that this allocation is impossible because the RAM memory starts from address `0x2000 0000`. But this example is just an illustration.
The first thing that the **`prvHeapInit`**`()` function does is properly aligning the heap on an 8-byte boundary. In our example this implies that the start and end addresses cannot be used. Therefore the heap will only use the space from address `0x0010` to `0x3C04`.
The FreeRTOS heap implementation uses the concept of a 'linked list' to divide the heap into memory blocks. For example, the following structure in the heap defines that the next 64 bytes belong to block n, and the next free block is located at memory location `0x0040`:



After executing the **`prvHeapInit()`** function, the heap is initialized and looks like the figure below.



Notice that the heap is now completely defined with the following three blocks:

`xStart`: This `xStart` block is empty. It is not intended to contain any data, but it holds a pointer to the first real memory block in the heap. The `xStart` block itself is the only block outside the heap. It is globally accessible (within the file) such that you have a starting point to run through the whole linked list.

```
xStart->size = 0
xStart->p_nextFreeBlock = xFirstFreeBlock
```

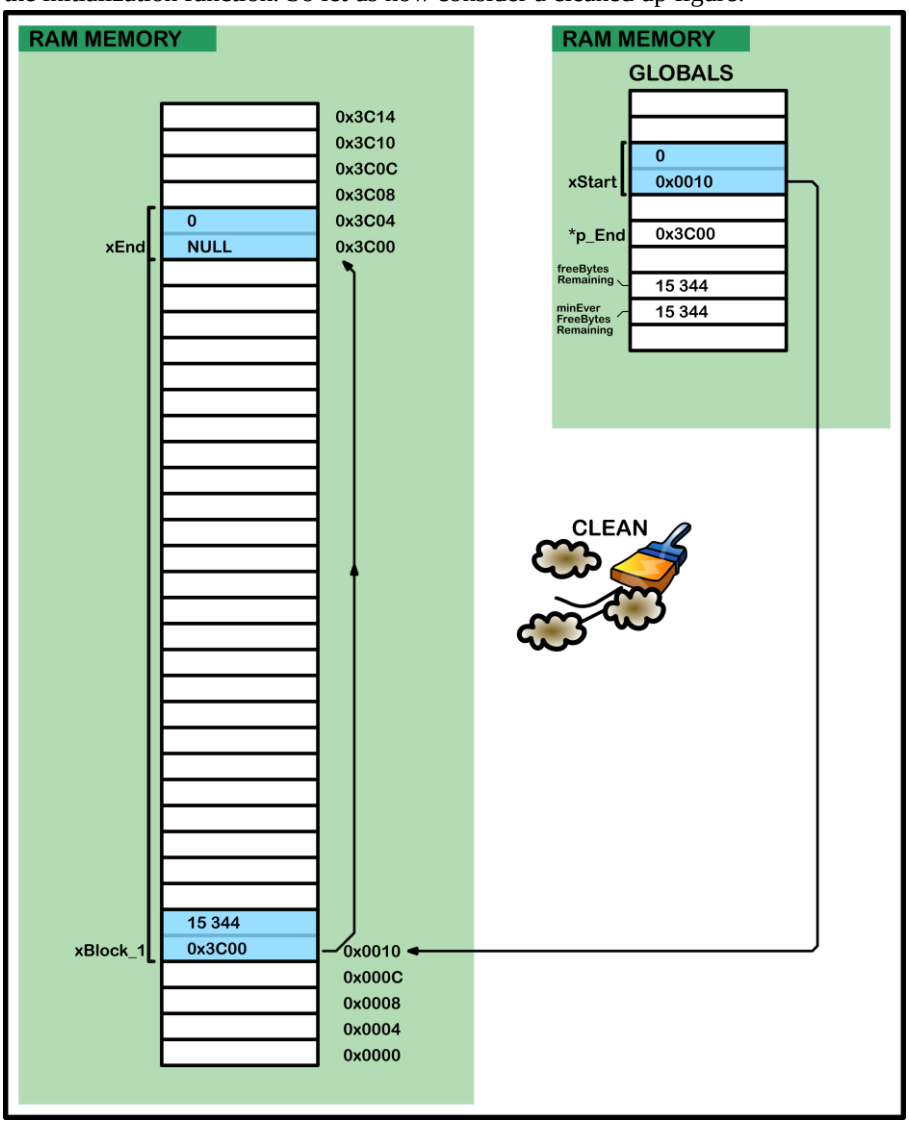
`xFirstFree-Block`: This `xFirstFreeBlock` spans the whole available heap. It is the only block (so far) intended to contain real data.

```
xFirstFreeBlock->size = 15 344
xFirstFreeBlock->p_nextFreeBlock = xEnd
```

`xEnd`: Just like `xStart`, the `xEnd` block is empty. It holds no data, and its sole purpose is to be the terminating point in the linked list.

```
xEnd->size = 0
xEnd->p_nextFreeBlock = NULL
```

The previous figure was good to get an insight in the heap initialization function. But the figure gets a bit complicated. Especially if you consider that those variables on the stack are deleted anyway when you return from the initialization function. So let us now consider a cleaned up figure:



Notice that I've changed the name of the `xFirstFreeBlock` into `xBlock_1`. The names of the blocks in the middle do not really matter, since your only access point is running through the linked list starting at `xStart`.

1.3 FreeRTOS heap `malloc()` function

Assume that an application requires 16 bytes of data from the heap. The application calls the function:

```
*pvPortMalloc(16)
```

This function returns a `(void *)` pointer to a memory block of 16 bytes that the application can use. We know how the initialized heap looks like. Let us now analyse what happens when 16 bytes are allocated through this `*pvPortMalloc()` function.
The `*pvPortMalloc()` function first increases the '`wantedSize`' parameter. The parameter gets increased such that it can span the needed number of bytes and the block header. So in our example the `wantedSize` parameter increases from 16 to 24.
Next, the `*pvPortMalloc()` function creates a few pointers:

`*p_prevBlock`: The block before `*p_curBlock`.

`*p_curBlock`: The 'current block'. `*p_curBlock` is initialized to `xStart->p_nextFreeBlock` and iterates until `( p_curBlock->size >= wantedSize )`.

The iteration has the purpose to find a block that is sufficiently large. This will be the block that gets returned to the application in the end. In our example there is no need for iteration, since the condition `( p_curBlock->size > wantedSize )` applies right from the start.

We know that `*p_curBlock` is the block that will be returned. But the application doesn't need the header. So the `*pvPortMalloc()` function provides another pointer to return only the 'payload':

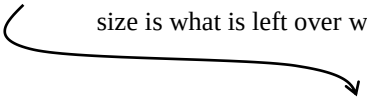
`*p_return`: A pointer to the 'data segment' of `*p_curBlock`. This is what the application needs.

If the current block `*p_curBlock` would have the exact size that the application needs – or maybe slightly larger – the algorithm would be finished right now. Not completely finished though. The following code would be necessary to effectively remove the `*p_curBlock` from the linked list of free blocks.:

```
/* This block is being returned for use, so it must be   */
/* taken out of the list of free blocks.                 */

p_prevBlock->p_nextFreeBlock = p_curBlock->p_nextFreeBlock;
```

In our example it is clear that there is a huge size mismatch. The block `*p_curBlock` is 15 344 bytes in size. The application only requested 16 bytes. It doesn't make sense to return a huge block to the application. So we need a way to cut the `*p_curBlock` in two pieces. The algorithm will trim down the size of `*p_curBlock` such that it corresponds to the '`wantedSize`' parameter. Next the algorithm creates a new block:

`*p_newBlock`: The 'new block' created by the algorithm. The algorithm puts the header at this position. The size is what is left over when `*p_curBlock` shrinked.

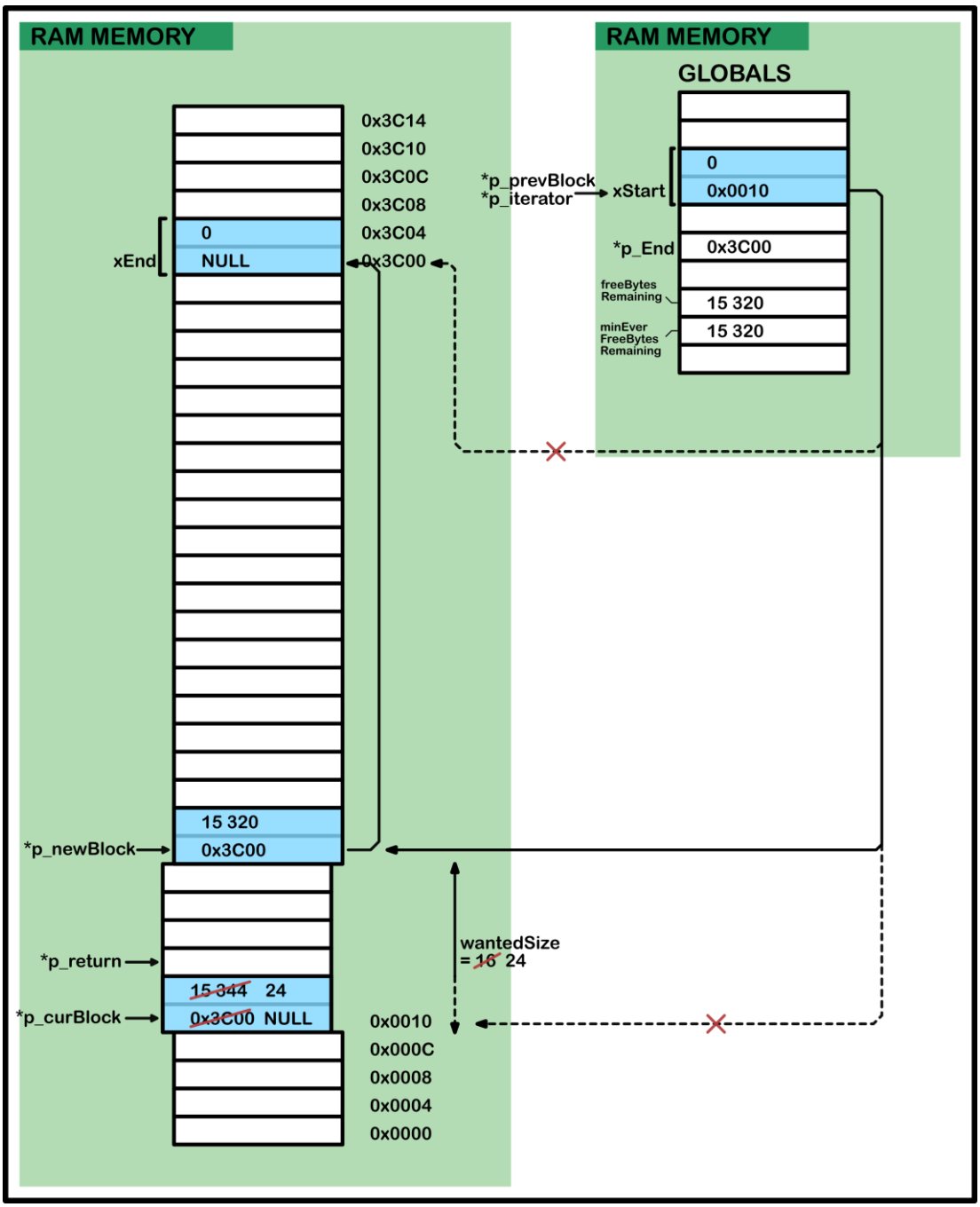


```
prvInsertBlockIntoFreeList( p_newBlock )
```

This subroutine will do the right bookkeeping such that `*p_newBlock` is now part of the linked list. After this update, the algorithm is finished. The pointer to the requested 16 bytes can now be returned:

```
return p_return;
```

The following figure is a visual help to understand how the algorithm works, and you can observe what the heap looks like after the 16 bytes got returned to the application:



heap_4.c

```c
/* ------------------------------------------------------------------------------------------ */
/*                                                                                            */
/*                     _ _            ____                                                   */
/*        _ __   ___ _| || | ___  ___  / /\ \                                                 */
/*       | '  \ / _` || || |/ _ \/ _|| |  | |                                                 */
/*       |_|_|_|\__,_||_||_|\___/\__|| |  | |                                                 */
/*                                    \_\/_/                                                  */
/* ------------------------------------------------------------------------------------------ */

void *pvPortMalloc( uint32_t wantedSize )
{

    vTaskSuspendAll();
    BlockLink_t *p_curBlock, *p_prevBlock, *p_newBlock;
    void *p_return = NULL;

    /* ---------------------------------------------------------------------------- */
    /*           C H E C K S                                                        */
    /*                                                                              */
    /* ---------------------------------------------------------------------------- */

    if( p_End == NULL )   // Initialize the heap.
    {
        prvHeapInit();
    }

    if( ( wantedSize & xBlockAllocatedBit ) != 0 )  // Top bit must be free.
    {
        ( void ) xTaskResumeAll();
        return NULL;
    }

    if( wantedSize == 0 )
    {
        ( void ) xTaskResumeAll();
        return NULL;
    }

    /* ---------------------------------------------------------------------------- */
    /*           A D J U S T    W A N T E D S I Z E                                 */
    /*                                                                              */
    /* ---------------------------------------------------------------------------- */
    /* The wanted size is increased so it can contain a BlockLink_t                 */
    /* structure in addition to the requested amount of bytes.                      */
    /* Ensure that blocks are always aligned to the required number of bytes.       */
    /*                                                                              */

    wantedSize += heapStructSize;
    if( ( wantedSize & (0x0007) ) != 0x00 )
    {
        // Byte alignment required.
        wantedSize += ( 8 - ( wantedSize & (0x0007) ) );
        configASSERT( ( wantedSize & (0x0007) ) == 0 );
    }


    if( wantedSize > freeBytesRemaining )
    {
        ( void ) xTaskResumeAll();
        return NULL;
    }

    /* ---------------------------------------------------------------------------- */
    /*           F I N D   B L O C K   O F   A D E Q U A T E   S I Z E              */
    /*                                                                              */
    /* ---------------------------------------------------------------------------- */
    /* Traverse the list from the start    (lowest address) block until one of      */
    /* adequate size is found.                                                      */
    /*                                                                              */

    p_prevBlock = &xStart;
    p_curBlock = xStart.p_nextFreeBlock;
    while( ( p_curBlock->size < wantedSize ) && ( p_curBlock->p_nextFreeBlock != NULL ) )
    {
        p_prevBlock = p_curBlock;
        p_curBlock = p_curBlock->p_nextFreeBlock;
    }

    if( p_curBlock == p_End )
    {
        // End marker was reached --> a block of adequate size was not found.
        ( void ) xTaskResumeAll();
        return NULL;
    }
```

```
    /* ------------------------------------------------------------------------ */
    /*          P R E P A R E   R E T U R N   P O I N T E R                     */
    /*                                                                          */
    /* ------------------------------------------------------------------------ */
    /* Return the memory space pointed to - jumping over the BlockLink_t        */
    /* structure at its start.                                                  */
    /*                                                                          */
    /* This block is being returned for use so must be taken out                */
    /* of the list of free blocks.                                              */
    /*                                                                          */

    p_return = ( void * ) ( ( ( uint8_t * ) p_prevBlock->p_nextFreeBlock ) + heapStructSize );
    p_prevBlock->p_nextFreeBlock = p_curBlock->p_nextFreeBlock;


    /* ------------------------------------------------------------------------ */
    /*          S P L I T   B L O C K   I F   T O O   L A R G E                 */
    /*                                                                          */
    /* ------------------------------------------------------------------------ */

    if( ( p_curBlock->size - wantedSize ) > heapMINIMUM_BLOCK_SIZE )
    {

        p_newBlock = ( void * ) ( ( ( uint8_t * ) p_curBlock ) + wantedSize );

        configASSERT( ( ( ( uint32_t ) p_newBlock ) & (0x0007) ) == 0 );

        p_newBlock->size = p_curBlock->size - wantedSize;
        p_curBlock->size = wantedSize;


        prvInsertBlockIntoFreeList( ( p_newBlock ) );
    }


    /* ------------------------------------------------------------------------ */
    /*          F I N I S H                                                     */
    /*                                                                          */
    /* ------------------------------------------------------------------------ */

    freeBytesRemaining -= p_curBlock->size;
    if( freeBytesRemaining < minEverFreeBytesRemaining )
    {
        minEverFreeBytesRemaining = freeBytesRemaining;
    }

    /* The block is being returned - it is allocated and owned      */
    /* by the application and has no "next" block.                  */
    p_curBlock->size |= xBlockAllocatedBit;
    p_curBlock->p_nextFreeBlock = NULL;

    ( void ) xTaskResumeAll();


    configASSERT( ( ( ( uint32_t ) p_return ) & (0x0007) ) == 0 );
    return p_return;
}
```

1.4 FreeRTOS heap `free()` function

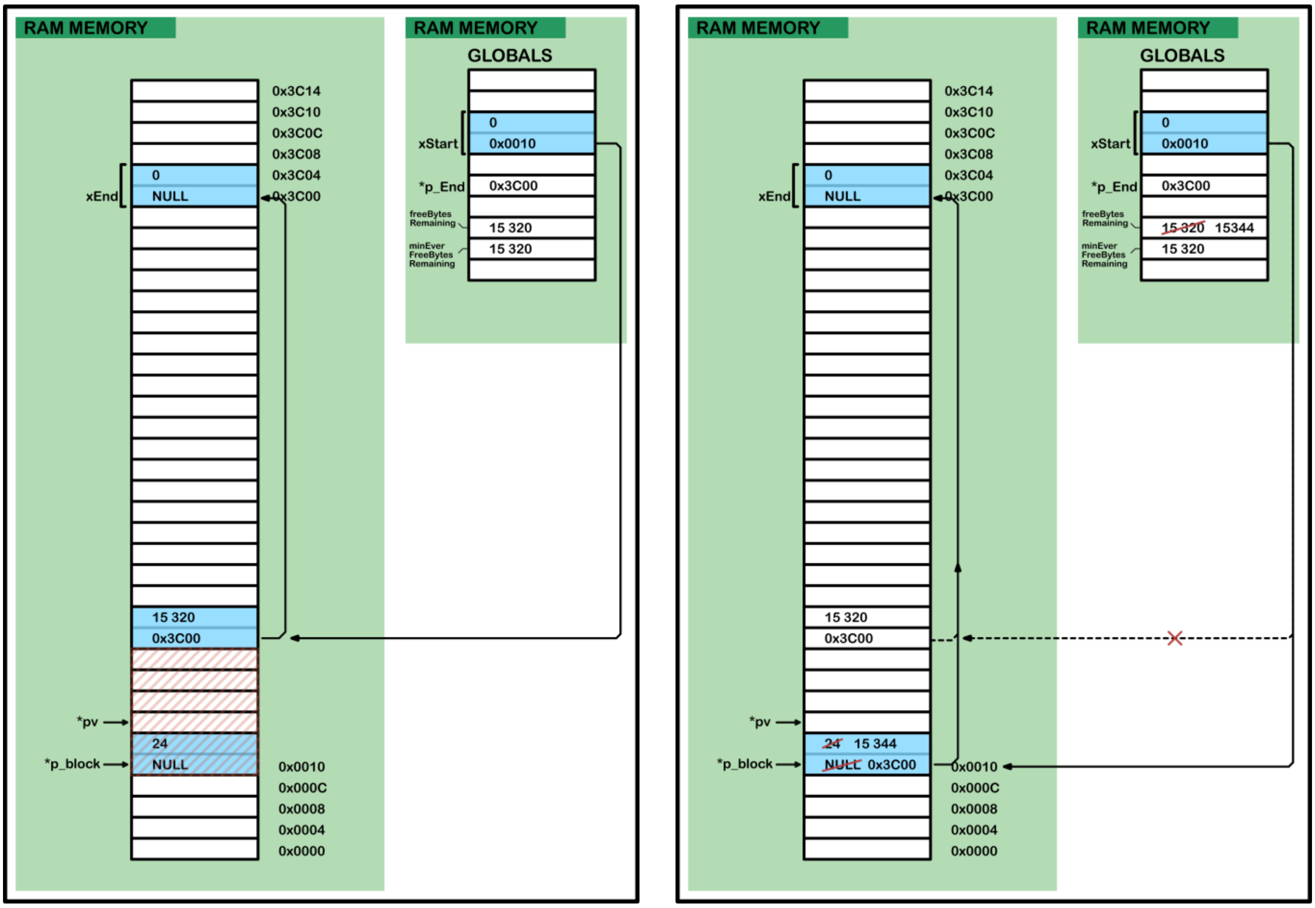
The freeRTOS `free()` function is relatively easy. It expects a pointer to the memory block that was allocated when the application called the `malloc()` function. The function knows that the header of that block is situated just before the payload. It retrieves the header and finds out the size of the memory block. Then it will insert the whole memory block back into the linked list of free blocks.

heap_4.c

```
/* ---------------------------------------------------------------------------------------- */
/*                                                                                          */
/*       __                      ____                                                       */
/*      / _| _ _  ___  ___  / /\ \                                                          */
/*     |  _|| '_|/ -_)/ -_)| |  | |                                                         */
/*     |_|  |_|  \___|\___|| |  | |                                                         */
/*                          \_\/_/                                                          */
/* ---------------------------------------------------------------------------------------- */

void vPortFree( void *pv )
{

    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *p_block;

    /* ------------------------------------------------------------------------------ */
    /*           C H E C K S                                                          */
    /*                                                                                */
    /* ------------------------------------------------------------------------------ */

    if( pv == NULL )
    {
        return;
    }

    // The memory being freed should have a block header immediately before it.
    puc -= heapStructSize;
    p_block = ( void * ) puc;

    // Check the block is actually allocated.
    configASSERT( ( p_block->size & xBlockAllocatedBit ) != 0 );
    configASSERT( p_block->p_nextFreeBlock == NULL );


    /* ------------------------------------------------------------------------------ */
    /*           R E T U R N   B L O C K   T O   T H E   H E A P                      */
    /*                                                                                */
    /* ------------------------------------------------------------------------------ */

    p_block->size &= ~xBlockAllocatedBit;

    vTaskSuspendAll();
    {
        // Add this block to the list of free blocks.
        freeBytesRemaining += p_block->size;
        prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) p_block ) );
    }
    ( void ) xTaskResumeAll();


}
```

Let us continue with the previous example. The `malloc()` function was called to make 16 bytes of heap-data available to the application. Now we want to `free()` that memory. We want to give it back to the ‘free heap space’.

The following figure illustrates how the `free()` algorithm works:



The freeRTOS `malloc()` and `free()` functions use a subroutine to insert a block into the 'free heap space' (or linked list of free blocks). Let us now take a deeper look at this subroutine:

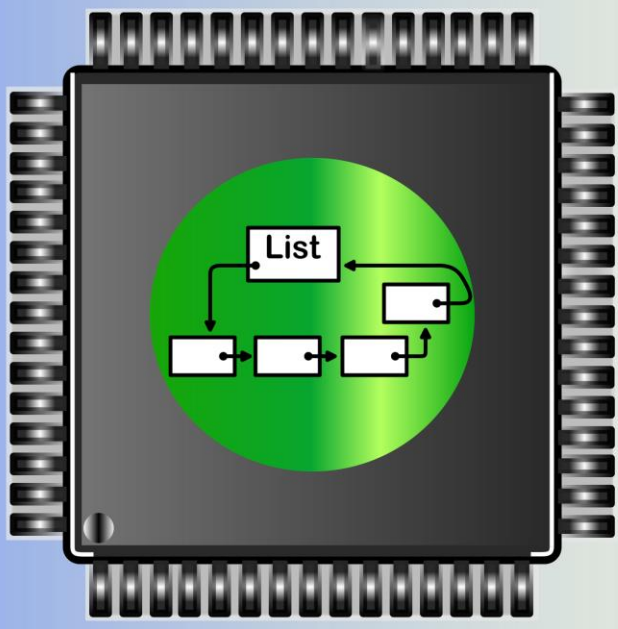
```
/* ------------------------------------------------------------------------------------------------- */
/*                                                                                                   */
/*        _                   _       _    _           _                                              */
/*       (_) _ _  ___ ___ _ _| |_    | |__| |___  __ | |__                                           */
/*       | || ' \ (_-</ -_)| '_||  _|   | '_ \| |/ _ \/ _|| / /                                          */
/*       |_||_||_|/__/\___||_|   \__|   |_.__/|_|\___/\__||_\_\                                          */
/* ------------------------------------------------------------------------------------------------- */
/*                                                                                                   */
/* Insert a block of memory that is being freed into the correct position in the list of free memory */
/* blocks.  The block being freed will be merged with the block in front of it and/or the block      */
/* behind it if they are adjacent to each other.                                                     */
/*                                                                                                   */
static void prvInsertBlockIntoFreeList( BlockLink_t *p_newBlock )
{
    BlockLink_t *p_iterator;
    uint8_t *puc;

    /* --------------------------------------------------------------------------- */
    /*          I T E R A T E                                                      */
    /*                                                                             */
    /* --------------------------------------------------------------------------- */
    /* Iterate through the list until:                                             */
    /*                                                          */
    /*      +----------+                                        */
    /*      |          |<--.                                    */
    /*      +----------+   |                                    */
    /*                     ^                                    */
    /*                     |                                    */
    /*      +----------+   |                                    */
    /*      |          | --´       p_iterator->next             */
    /*      +----------+   \                                    */
    /*                     |                                    */
    /*                     ^       <= p_newBlock                */
    /*      +----------+   |                                    */
    /*      |          | --´       p_iterator                   */
    /*      +----------+                                        */
    /*                                                          */
```

```
for( p_iterator = &xStart;
     p_iterator->p_nextFreeBlock < p_newBlock;
     p_iterator = p_iterator->p_nextFreeBlock   )
{
    // Iterate until the given p_newBlock is squeezed between p_iterator and p_iterator->next
}


/* ------------------------------------------------------------------------- */
/*   M E R G E   I F   P O S S I B L E                                       */
/*   A N D   M A K E   p_newBlock->next  P O I N T   T O   N E X T   B L O C K */
/* ------------------------------------------------------------------------- */


/* case A:                                                                   */
/* p iterator and p newBlock make a contiguous memory block.                 */
/* ---------------------------------------------------------                 */
/*                                                                           */
/*   +------------+                                +------------+            */
/*   |            |<-.(p_iterator->next)           |            |<-.         */
/*   +------------+  |                             +------------+  |         */
/*                   |                                           |         */
/*   +------------+  |                             +------------+  |         */
/*   |            |  |                             |            |  |         */
/*   | size = 16  |  |<- p_newBlock                |            |  |         */
/*   +------------+  |                             |            |  |         */
/*   |            |  |                             |            |  |         */
/*   | size = 16  | / <- p iterator                | size = 32  | /<-p newBlock */
/*   +------------+                                +------------+    p iterator */
/*                                                                           */
/*                                                                           */
/*                                                                           */

puc = (uint8_t *) p_iterator;
if(   ( (puc + p_iterator->size) == (uint8_t *) p_newBlock ) && (p_iterator != &xStart)   )
{
    p_iterator->size += p_newBlock->size;
    p_newBlock = p_iterator;
}


/* case B:                                                                   */
/* p newBlock and (p iterator->next) make a contiguous memory block.         */
/* ----------------------------------------------------------------          */
/*                                                                           */
/*                                                                           */
/*   +------------+                                +------------+            */
/*   |            |                                |            |                                 */
/*   | size = 16  |<-.  (p iterator->next->next)   | size = 16  |<-. (p iterator->next->next)     */
/*   +------------+  |                             +------------+  |                              */
/*                   |                                           |         */
/*   +------------+  |                             +------------+  |         */
/*   |            | /                              |            |  |         */
/*   | size = 16  |  <- (p iterator->next)         |            |  |         */
/*   +------------+                                |            |  |         */
/*   |            |                                |            | /         */
/*   | size = 16  |  <- p_newBlock                 | size = 32  | <- p_newBlock */
/*   +------------+                                +------------+           */
/*                                                                           */
/*                                                                           */
/*   Note: case A and case B can coexist. This doesn't affect                */
/*   anything in the code for case B.                                        */
/*                                                                           */

puc = (uint8_t *) p_newBlock;
if( ( puc + p_newBlock->size ) == ( uint8_t * ) p_iterator->p_nextFreeBlock )
{
    if( p_iterator->p_nextFreeBlock != p_End )
    {
        p_newBlock->size += p_iterator->p_nextFreeBlock->size;
        p_newBlock->p_nextFreeBlock = p_iterator->p_nextFreeBlock->p_nextFreeBlock;
    }
    else
    {
        p_newBlock->p_nextFreeBlock = p_End;
    }
}
```

```
    /* case C:                                                                  */
    /* p_newBlock and (p_iterator->next) are not contiguous                     */
    /* -------------------------------------------------------                  */
    /*                                                                          */
    /*                                                                          */
    /*   +------------+                                +------------+          */
    /*   |            |                                |            |                                */
    /*   | size = 16  |<-.  (p_iterator->next->next)   | size = 16  |<-. (p_iterator->next->next) */
    /*   +------------+  |                             +------------+  |                             */
    /*                   |                                           |                             */
    /*   +------------+  |                             +------------+  |                             */
    /*   |            | /                              |            | /                             */
    /*   | size = 16  |  <- (p_iterator->next)         | size = 16  |<-.  (p_iterator-> next)     */
    /*   +------------+                                +------------+  |                             */
    /*                                                               |             */
    /*   +------------+                                +------------+  |          */
    /*   |            |                                |            | /           */
    /*   | size = 16  |  <- p_newBlock                 | size = 16  | <- p_newBlock  */
    /*   +------------+                                +------------+             */
    /*                                                                          */
    /*                                                                          */
    /*  Note: case A and case C can coexist. This doesn't affect                */
    /*  anything in the code for case C.                                        */
    /*                                                                          */

    else
    {
        p_newBlock->p_nextFreeBlock = p_iterator->p_nextFreeBlock;
    }


    /* ------------------------------------------------------------------------ */
    /*       M A K E   p_iterator->next   P O I N T   T O   p_newBlock          */
    /*                                                                          */
    /* ------------------------------------------------------------------------ */
    /*                                                                          */
    /* Normally p_iterator is located before p_newBlock. So to keep the linked  */
    /* list consistent, p_iterator should point to p_newBlock as the next element */
    /* in the list.                                                             */
    /* There is one exception. If 'case A' was true, then p_iterator and p_newBlock */
    /* refer to the same block (they are identical).                            */
    /*                                                                          */
    /*                                                                          */

    if( p_iterator != p_newBlock )
    {
        p_iterator->p_nextFreeBlock = p_newBlock;
    }
}
```

Chapter 2

FreeRTOS Linked Lists



Our first encounter with linked lists was in the previous paragraph when we studies the FreeRTOS memory structures. FreeRTOS divides the heap in blocks of “free memory”. Each block is a member of the “free memory linked list”. You can run through all of them if you know where to find the first one. That’s the basic principle of a linked list.
FreeRTOS not only uses a linked list for structuring the memory. The kernel uses linked lists to manage the Tasks – also called Threads – that run in the program. These linked lists are slightly more complex than the one used for the heap. So FreeRTOS dedicates a separate `list.c` and `list.h` file to define a them. We shall study these two files in detail. The linked list defined by FreeRTOS is tailored to the needs of the kernel. But that doesn’t exclude the possibility to use it in your own application for non-kernel purposes.

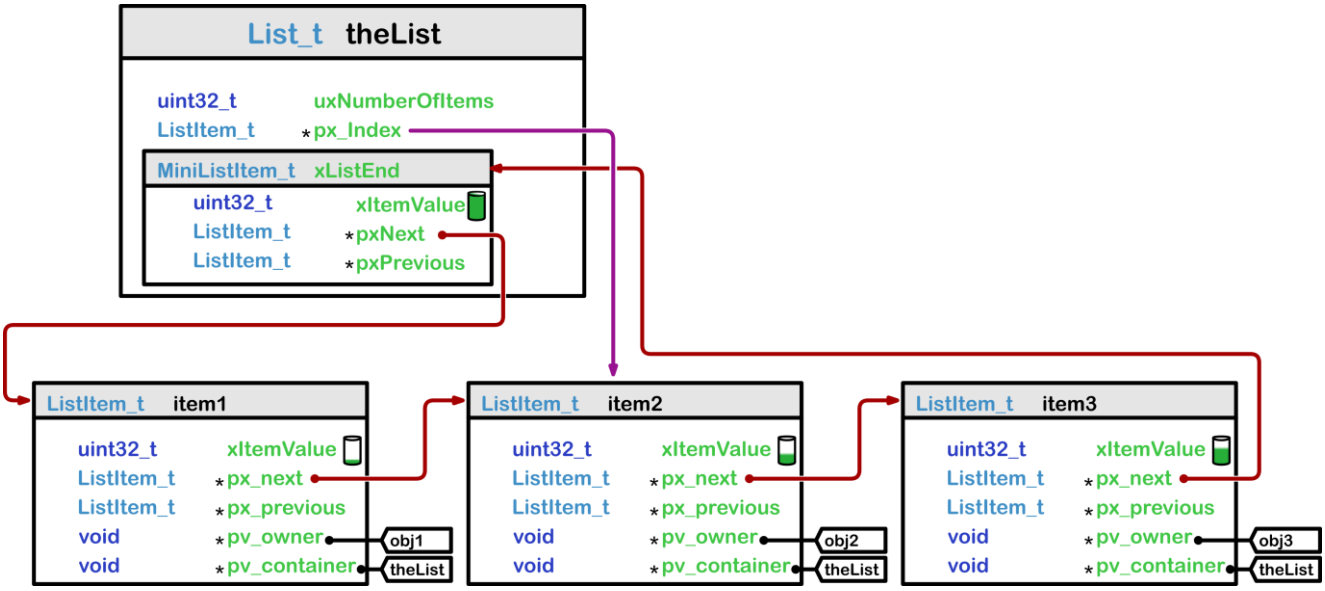
2.1 The FreeRTOS linked list

This is how the linked list is defined in FreeRTOS:

list.h

```
/*****************************/
/*      ListItem_t           */
/*****************************/
struct LIST_ITEM_t
{
    volatile uint32_t      xItemValue;
    struct LIST_ITEM_t   * volatile pxNext;
    struct LIST_ITEM_t   * volatile pxPrevious;
                 void    * pvOwner;
                 void    * volatile pvContainer;
};
typedef struct LIST_ITEM_t ListItem_t;

/*****************************/
/*    MiniListItem_t         */
/*****************************/
struct MINI_LIST_ITEM_t
{
    volatile uint32_t      xItemValue;
    struct LIST_ITEM_t   * volatile pxNext;
    struct LIST_ITEM_t   * volatile pxPrevious;
};
typedef struct MINI_LIST_ITEM_t MiniListItem_t;


/*****************************/
/*         List_t            */
/*****************************/
typedef struct LIST_t
{
    volatile uint32_t      uxNumberOfItems;
    ListItem_t           * volatile pxIndex;
    MiniListItem_t         xListEnd;
} List_t;
```

Let us create a simple example. Imagine that we have a `List_t` named `theList`. It contains 4 items. The first item is a `MiniListItem_t`. The first item is also the last one, because the list is circular. We call this item **`xListEnd`** to follow the FreeRTOS convention. But the name **`xListStart`** would be equally suited.
The three other items are **`item1`**, **`item2`** and **`item3`**. The figure on the next page shows how they are located in the circular linked list. Their position is not random though. Each item has a `uint32_t` value. The list is sorted in ascending order – low values come first. The **`xListEnd`** item got the highest possible value to fix its location.



2.2 Walking through the linked list

Going through the linked list is easy. Let us examine an example. We want to collect the values that are stored in each item. We will use the **`*pxIndex`** pointer to walk through the list. So we need to initialize this pointer such that it points to the first element. Remember that the list is circular. The first element is also the last one. So the pointer should contain the address of the item **`xListEnd`**. Store the value held by this item into `val0`:

```
uint32_t val0,val1,val2,val3;

theList.pxIndex = (ListItem_t *) &theList.xListEnd;
val0 = xDelayedTaskList1.pxIndex->xItemValue;
```

Increment the pointer **`*pxIndex`** such that it points to the next item. Extract the value and store it into `val1`:

```
theList.pxIndex = theList.pxIndex->pxNext;
val1 = xDelayedTaskList1.pxIndex->xItemValue;
```

Continue to do so until all items have been served. Of course in a real application you want to do that in a loop. But we keep it simple now:

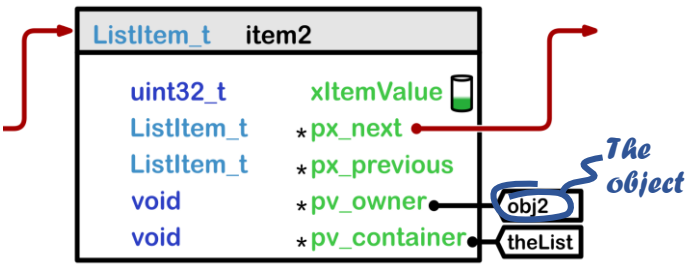
```
theList.pxIndex = theList.pxIndex->pxNext;
val2 = xDelayedTaskList1.pxIndex->xItemValue;

theList.pxIndex = theList.pxIndex->pxNext;
val3 = xDelayedTaskList1.pxIndex->xItemValue;
```

If you increment the pointer **`*pxIndex`** one more time, it should point again to the beginning (= end) of the list. We want to check that:

```
theList.pxIndex = theList.pxIndex->pxNext;
assert(theList.pxIndex == ((ListItem_t *) &theList.xListEnd) );
```

2.3 Objects

An item in the linked list is a fairly simple thing. It contains a numeric value (used for sorting) and some pointers to facilitate walking through the list. It is quite clear that such item is only useful in very simple applications. Therefore FreeRTOS allows each item to be linked to another object. The pointer is of type `(void *)` so it can point to any object you want. By convention we say that such an object "owns" the item.



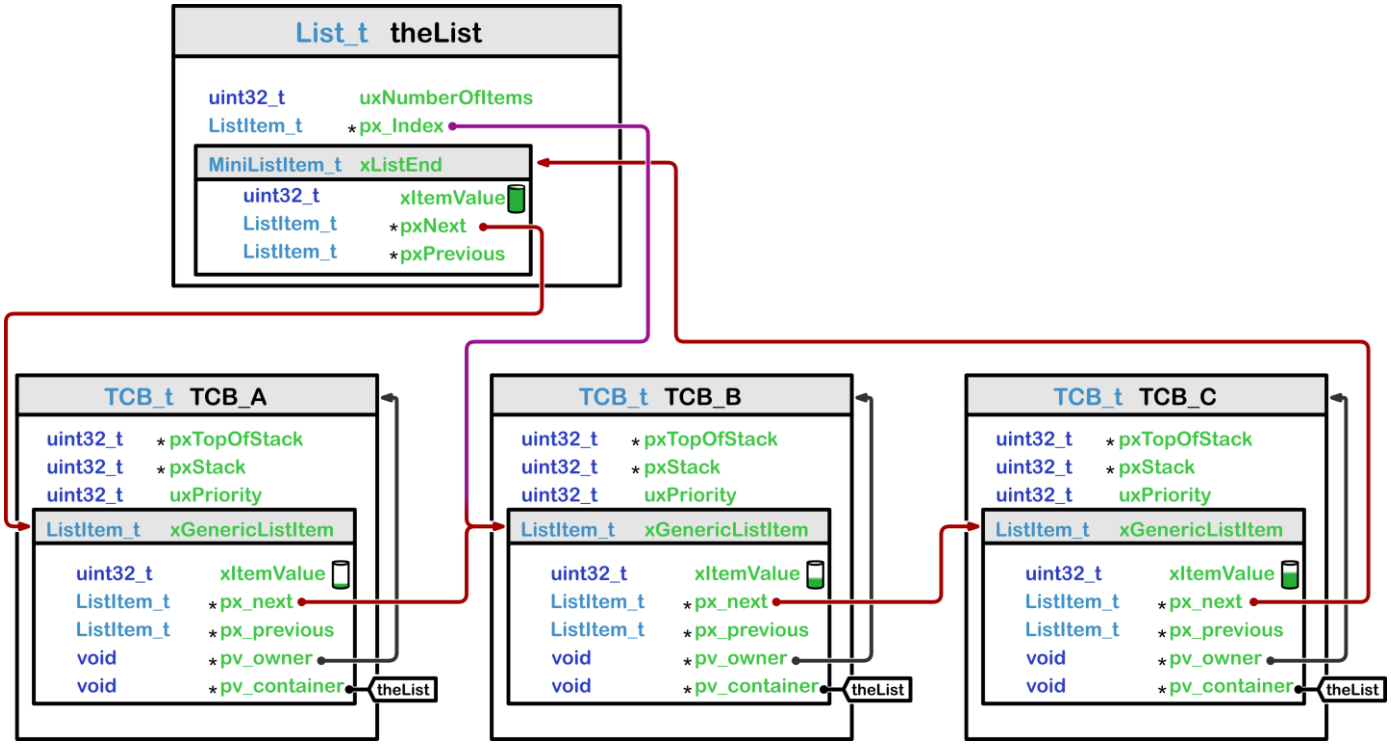
Now imagine you are dealing with an object. But you don't have a clue if your object belongs to a linked list. Let alone which linked list in particular. There is no problem if your object has a two-way link to its item in the linked list:

2.4 Tasks and linked lists

The FreeRTOS kernel uses linked lists to manage Tasks – or Threads – efficiently. We will dig into Tasks in later paragraphs, but we can reveal now that each Task holds a "Task Control Block". The TCB contains important information about its Task – such as priority, stack pointer, … The TCB can also own an item from a linked list. The term "own" can be taken quite literally. The TCB does not keep just a reference (pointer) to the item. But the item is created within the TCB. Take a look at the TCB struct:

tasks.c

```
typedef struct
{
    volatile uint32_t         *pxTopOfStack;
    ListItem_t                 xGenericListItem;
    ListItem_t                 xEventListItem;
    uint32_t                   uxPriority;
    ...
} TCB_t;
```



2.5 Linked list initialization

FreeRTOS provides two initialization functions. One for the list and one for each item you want to insert. The two functions are quite straightforward:

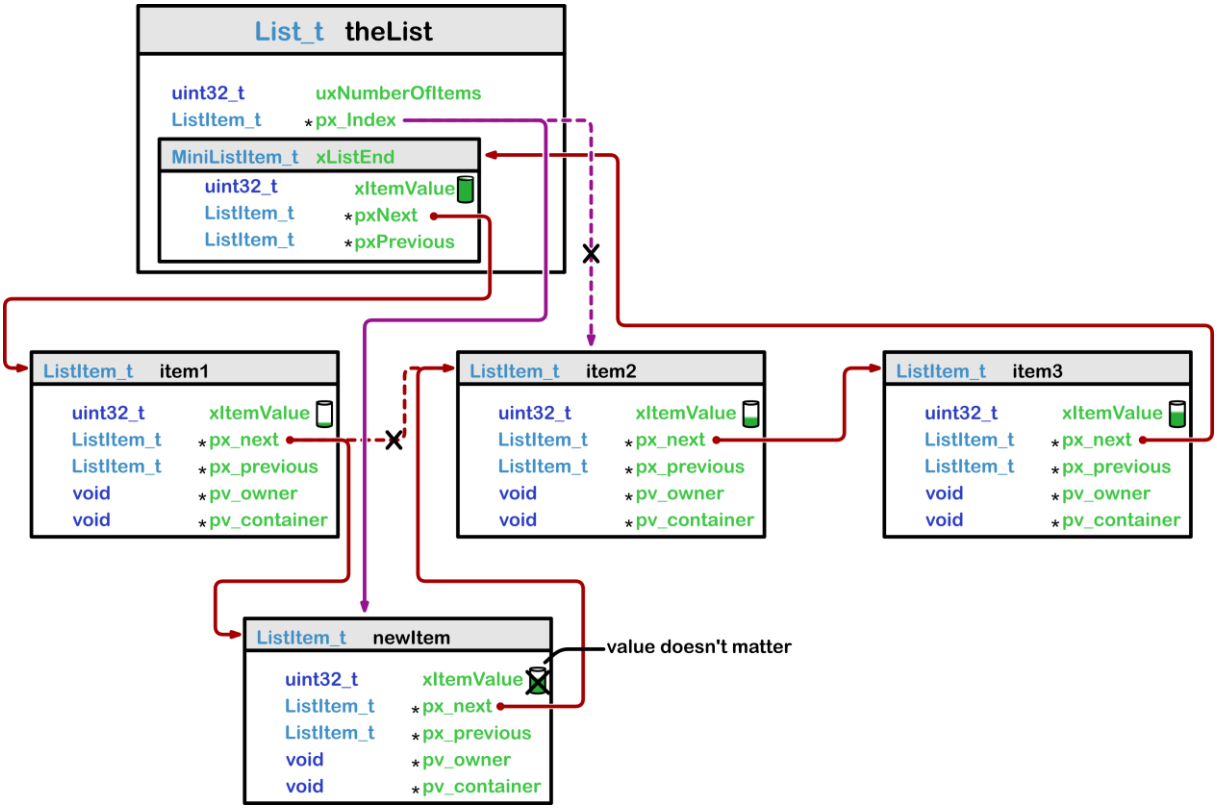
```
void vListInitialise(List_t * const p_List)
{

        p_List->pxIndex = (ListItem_t *) &( p_List->xListEnd);

        p_List->xListEnd.xItemValue = MAX_UINT32;
        p_List->xListEnd.pxNext     = ( ListItem_t * ) &( p_List->xListEnd );
        p_List->xListEnd.pxPrevious = ( ListItem_t * ) &( p_List->xListEnd );

        p_List->uxNumberOfItems = (uint32_t) 0U;
}

void vListInitialiseItem(ListItem_t * const p_item)
{
    /* Must be called before a list item is used.  This sets the list container to   */
    /* null so the item does not think that it is already contained in a list.       */
        p_item->pvContainer = NULL;
}
```

2.6 Insert items

FreeRTOS gives you two choices for inserting a new item. Either you insert it at the end of the list. Or you let FreeRTOS determine the position based on the value of the item. Let us now take a look at the first option. The name **`vListInsertEnd(..)`** can be a bit misleading. Where is the end of the list? A natural choice would be to place the item just before the `MiniListItem_t` **`xListEnd`**. But for some reason, FreeRTOS defines "end of list" as "the position where the item will be accessed last by the `listGET_OWNER_OF_NEXT_ENTRY(pxTCB, pxList)` macro". So it depends on where the "current item" is (the item pointed at by **`*pxIndex`**). The code is given below:

```
void vListInsertEnd(List_t * const pxList, ListItem_t * const pxNewListItem)
{
    ListItem_t * const pxIndex = pxList->pxIndex;

                                                        /*                          */
    pxNewListItem->pxNext = pxIndex;                    /*  |                       */
    pxNewListItem->pxPrevious = pxIndex->pxPrevious;    /*  |                       */
    //                                                  /*   > CODE TO INSERT       */
    pxIndex->pxPrevious->pxNext = pxNewListItem;        /*  |  THE ITEM            */
    pxIndex->pxPrevious = pxNewListItem;                /*  |                       */
    pxNewListItem->pvContainer = (void *) pxList;       /*  |                       */
                                                        /*  |                       */
    (pxList->uxNumberOfItems)++;

    /* BUG FOUND                                                                    */
    /* The documentation for this function mentions: "Placing an item in a list using      */
    /* vListInsertEnd effectively places the item in the list position pointed to by       */
    /* pvIndex."                                                                    */
    /* (1) The correct field name is not "pvIndex" but "pxIndex"                    */
    /*    (that's a documentation bug)                                              */
    /* (2) The pxList->pxIndex does not point to the NewListItem. The following code line  */
    /*     was forgotten:              */
    pxList->pxIndex = pxNewListItem;

    /* Calling listGET_OWNER_OF_NEXT_ENTRY increments pxIndex to the next item in the list.*/
    /* Placing an item in a list using vListInsertEnd effectively places the item in the   */
    /* list position pointed to by pxIndex.  This means that every other item within the   */
    /* list will be returned by listGET OWNER OF NEXT ENTRY before the pxIndex parameter   */
    /* again points to the item being inserted.                                     */

}
```

Te following figure illustrates the **`vListInsertEnd(..)`** algorithm. The value in the `newItem` doesn't matter since the algorithm will put it at the end of the list anyway.



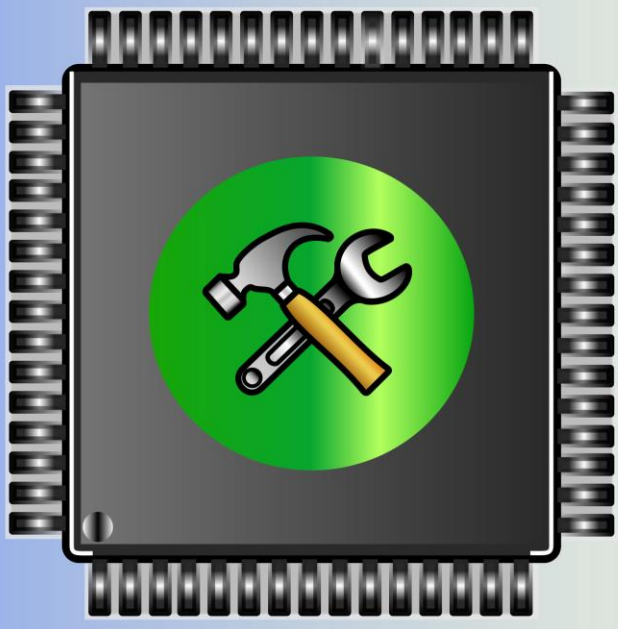
The **`vListInsert(..)`** function puts the item in the linked list according to its value. Remember that FreeRTOS uses linked lists in ascending value order (low values come first).

```
                                                                        */
/* Insert a list item into a list.  The item will be inserted into the list in      */
/* a position determined by its item value (ascending value order).                 */
/*                                                                                  */
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
{
    ListItem_t *pxIterator;
    const uint32_t xValueOfInsertion = pxNewListItem->xItemValue;


    /* Insert the new list item into the list, sorted in xItemValue order. If the    */
    /* list already contains a list item with the same item value, then the new list */
    /* item should be placed after it.  This ensures that TCB's which are stored in  */
    /* ready lists (all of which have the same xItemValue value) get a share of the  */
    /* CPU. However, if the xItemValue is the same as the back marker (xListEnd) the */
    /* iteration loop below will not end.  Therefore the value gets checked first,   */
    /* and the algorithm is slightly modified if necessary.                          */

    if( xValueOfInsertion == MAX_UINT32 )
    {
        pxIterator = pxList->xListEnd.pxPrevious;
    }
    else
    {

        for( pxIterator = (ListItem_t *) &(pxList->xListEnd);
             pxIterator->pxNext->xItemValue <= xValueOfInsertion;
             pxIterator = pxIterator->pxNext )
        {
            /* There is nothing to do here, just iterating to the wanted  */
            /* insertion position.                                        */
        }
    }
```

```
                                                    /*  _                       */
    pxNewListItem->pxNext = pxIterator->pxNext;     /*   |                      */
    pxNewListItem->pxNext->pxPrevious = pxNewListItem;  /*   |                  */
    pxNewListItem->pxPrevious = pxIterator;         /*    > CODE TO INSERT     */
    pxIterator->pxNext = pxNewListItem;             /*   |  THE ITEM           */
    pxNewListItem->pvContainer = (void *) pxList;   /*   |                      */
    (pxList->uxNumberOfItems)++;                    /*  _|                      */
}
```

Chapter 3

FreeRTOS Tasks



3.1 Function pointers

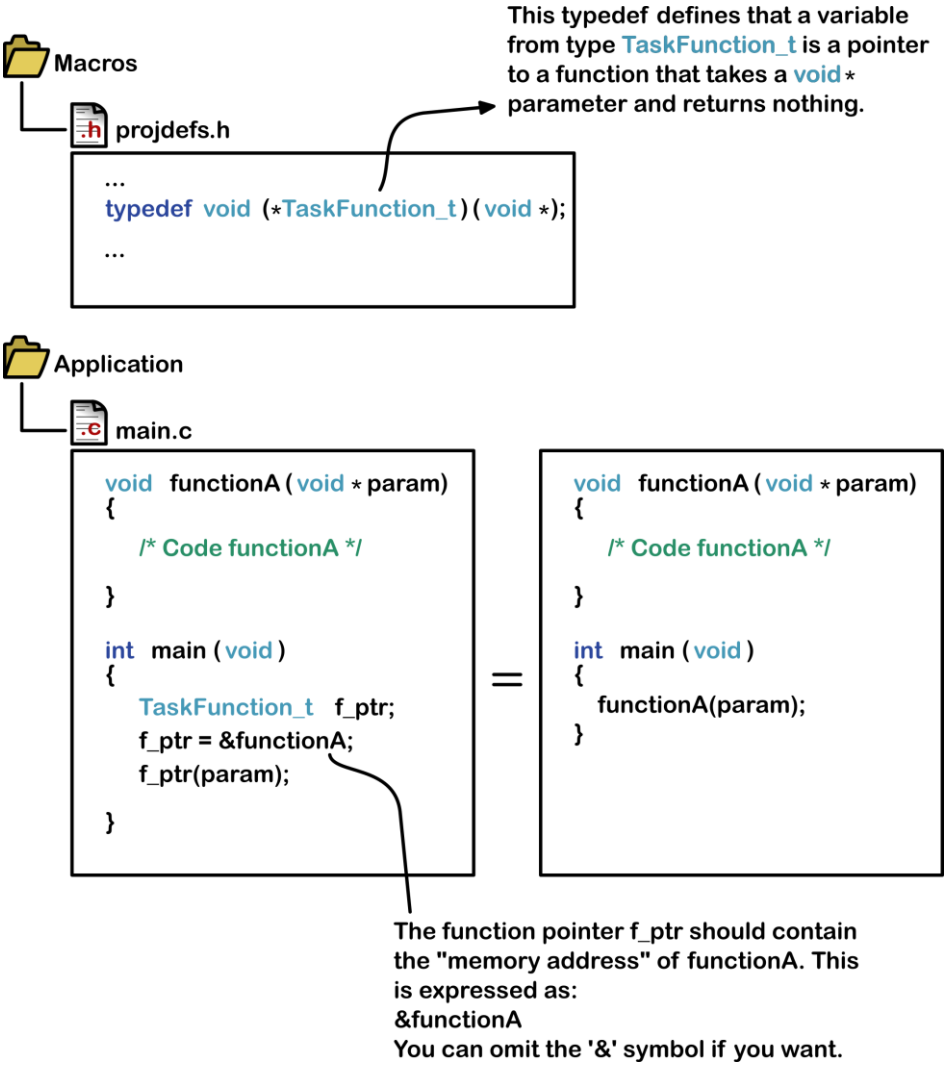
A very important concept to understand how Tasks in FreeRTOS work is the 'function pointer'. Let us compare the function pointer concept to the normal data pointer.

A normal *data pointer* contains the memory address of the data block it points to. Usually that memory address is a location in RAM. The type of the pointer tells the compiler what kind of data block it can point to. For example, if the type is `uint32_t`, then the compiler knows that the pointer points to a 32-bit value in memory.

A *function pointer* contains the memory address of the function it points to. Usually the function code lives in the onboard FLASH memory of the chip. So the memory address is the address of the first instruction in that function. The type of the function pointer tells the compiler what kind of function it can point to. The 'kind of function' refers to:

- number of parameters and type of each parameter
- return type

Any function that complies with these parameter and return type conditions is a possible candidate. The following figure gives an example:



The example in the figure illustrates very well the function pointer mechanism. But it is also very relevant for our study of FreeRTOS. Every Task in FreeRTOS corresponds to a function. That is a very important concept you should keep in mind.

3.2 The Task and the TaskFunction

Every **Task** corresponds to one **TaskFunction**. This is the actual function that gets executed when the scheduler selects the task to run. The function must comply with the following prototype:

projdefs.h

```
/* ------------------------------------------------------------------------- */
/*                          THE TASK PROTOTYPE                               */
/* ------------------------------------------------------------------------- */
typedef void (*TaskFunction_t)( void * );
```

So the function should take a `void*` parameter and return nothing.

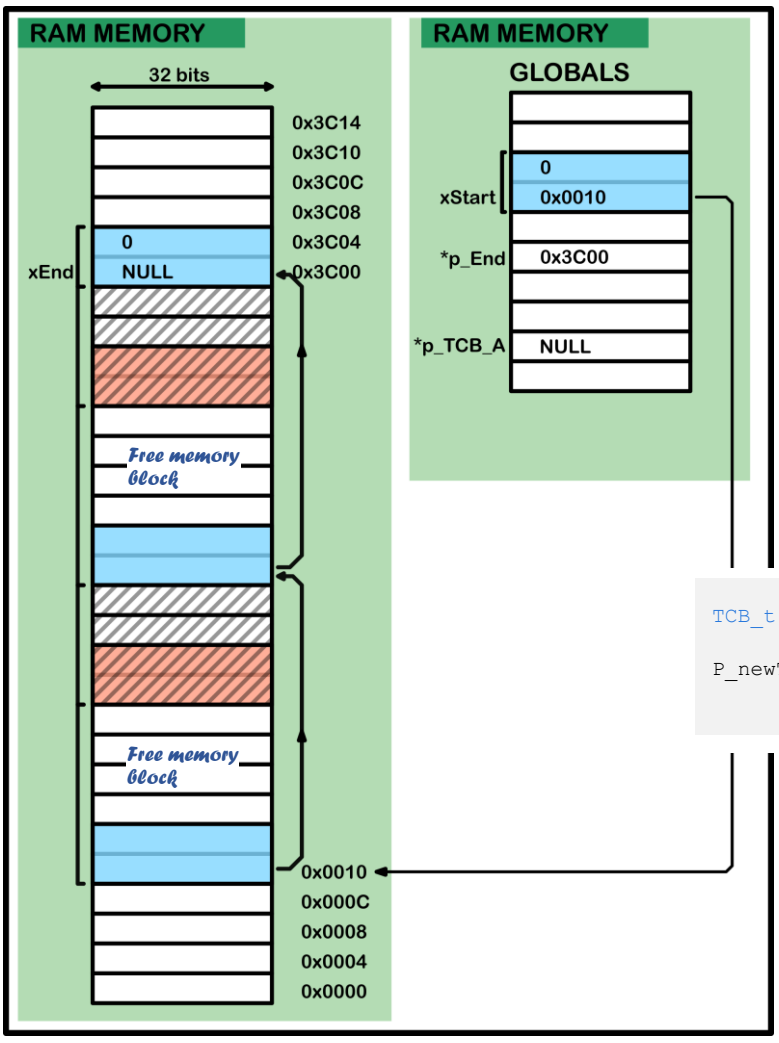
3.3 Memory allocation for a newly created Task

When creating a new Task, FreeRTOS will first allocate memory. This memory is taken from the heap and given to the newly created Task. The Task needs memory for its stack and also for its TCB.
I guess you know what a stack is. So I will only say a few words about it. If you don't use an RTOS for embedded programming, you only use one stack. One big stack for the entire program. All functions called will put their temporary variables on it. If you use an RTOS, you will probably have several stacks. One stack for every Task – or Thread – that runs in your software. Every stack needs some RAM memory to exist.
I guess you don't know what a TCB (Task Control Block) is. We will dig into that later. For the moment you can consider a TCB as the header of a Task. It contains critical information about the Task. Like a pointer to the stack of the Task. And much more.
The memory allocation for the newly created Task is done in the function **`prvAllocateTCBAndStack(..)`**. Notice that the function returns a TCB. That's correct. This function will allocate memory and create a brand new (and empty) TCB for your Task. But to allocate memory for the stack, the function needs to know how deep you want it to be. You can even give a pointer to some free memory block in RAM. If you do that, the stack will end up in that spot. But you can also give a **NULL** pointer, such that a free memory block of the desired lenght is searched for on the heap.

tasks.c

```
static TCB_t *prvAllocateTCBAndStack( const uint16_t stackDepth, uint32_t * const
p_stackBuffer )
{
    TCB_t *p_newTCB;
    uint32_t *p_stack;

    /* The Cortex-M7 has a descending stack. So allocate first the stack, then the  */
    /* TCB, so the stack does not grow into the TCB.                                */

    /*-----------------------------------------------------------------------------*/
    /*          A L L O C A T E    S T A C K                                       */
    /*-----------------------------------------------------------------------------*/
    /* Allocate space for the stack used by the task being created.                 */
    /*                                                                              */
    if(p_stackBuffer == NULL)
    {
        p_stack = (uint32_t *) pvPortMalloc( ((uint32_t)stackDepth) * sizeof(uint32_t) );
    }
    else
    {
        p_stack = (uint32_t *) p_stackBuffer;
    }

    if(p_stack == NULL)
    {
        return NULL;
    }
```

```
    /*-----------------------------------------------------------------------------*/
    /*          A L L O C A T E    T C B                                           */
    /*-----------------------------------------------------------------------------*/
    /* Allocate space for the TCB.  Where the memory comes from depends on the     */
    /* implementation of the port malloc function.                                 */
    /*                                                                             */
    p_newTCB = (TCB_t *) pvPortMalloc( sizeof(TCB_t) );

    if( p_newTCB != NULL )
    {
        /* Store the stack location in the TCB. */
        p_newTCB->pxStack = p_stack;
    }
    else
    {
        /* The stack cannot be used as the TCB was not created.  Free it again. */
        vPortFree( p_stack );
    }

    /*-----------------------------------------------------------------------------*/
    /*          F I L L    S T A C K                                               */
    /*-----------------------------------------------------------------------------*/
    if( p_newTCB != NULL )    /* Just to help debugging. */
    {
        ( void ) memset( p_newTCB->pxStack,
                         ( int ) tskSTACK_FILL_BYTE,
                         ( uint32_t ) stackDepth * sizeof( uint32_t ) );
    }
    return p_newTCB;
}
```



We can apply this code on an example. Imagine a heap like in the figure on the left. This heap has two only two free memory blocks. They are 24 bytes each (16 bytes data + 4 bytes header) in size. The rest of the heap is already allocated.
The figure is not very realistic. Normally the heap is way way larger. But it's enough to illustrate how the code works.

We call the function **prvAllocateTCBAndStack(..)** to allocate memory for the new Task A:

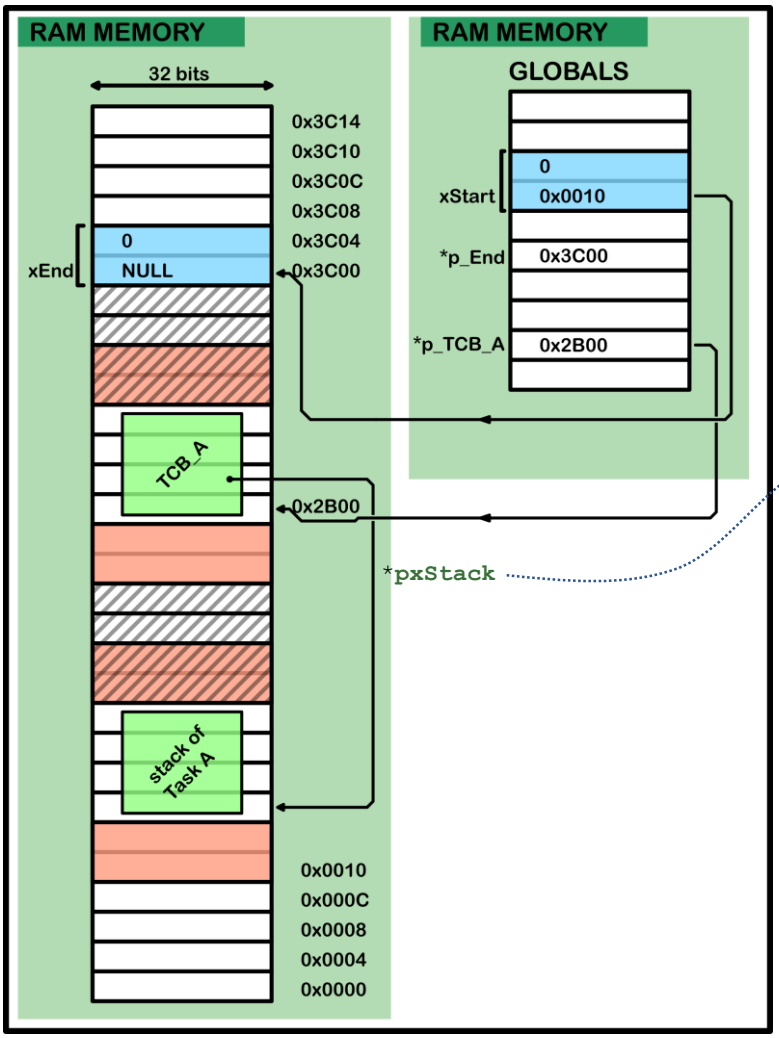
```
TCB_t *p_newTCB = NULL;

P_newTCB = prvAllocateTCBAndStack(16, NULL);
```

We want a stack of 16 bytes deep

We give parameter NULL such that the algorithm will look on the heap to allocate memory

Of course this is quite stupid. A stack of only 16 bytes is almost nothing. But this is just an example.

The algorithm searches on the heap for a free memory block that is large enough to hold the stack. After that, it explores the heap again to find a memory block that can hold the TCB. In reality the TCB is more than 80 bytes in size. So our example provides not enough free heap memory. But let us presume for a moment that a TCB is just 16 bytes long, such that it fits neatly into the free memory block. The memory looks like the figure below after calling the **`prvAllocateTCBAndStack(..)`** function:



The pointer `p_newTCB` points to `TCB_A`. This `TCB_A` is a C-structure (of type `TCB_t`) and one of its member fields points to the stack:

```
typedef struct
{
    uint32_t  *pxTopOfStack;
    uint32_t  *pxStack;
    ...
} TCB_t;
```

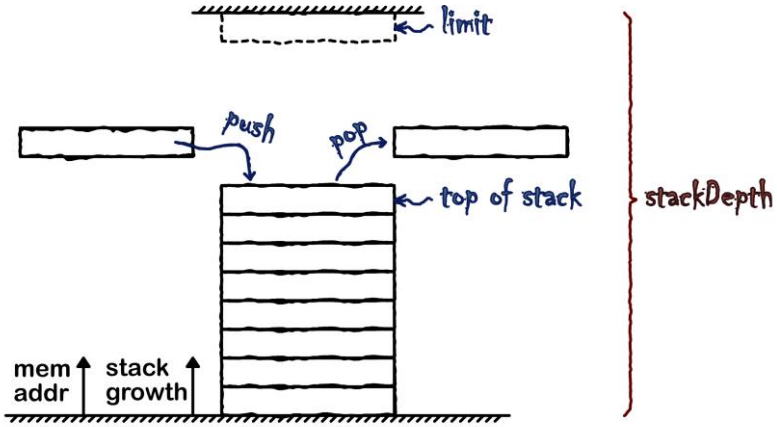
3.4 Stack and 'top of stack'

We have seen in the previous paragraph that the TCB of a Task contains some pointers to the stack. Let's take a look again:

```
typedef struct
{
    uint32_t   *pxTopOfStack;
    uint32_t   *pxStack;
    ...

} TCB_t;
```

The figure shows that `*pxStack` points to the 'beginning' of the stack. The lowest memory address. Now we shall take a deeper look at stacks on a Cortex-M7 microcontroller to get the terminology right. We will also see what the other field `*pxTopOfStack` points at.

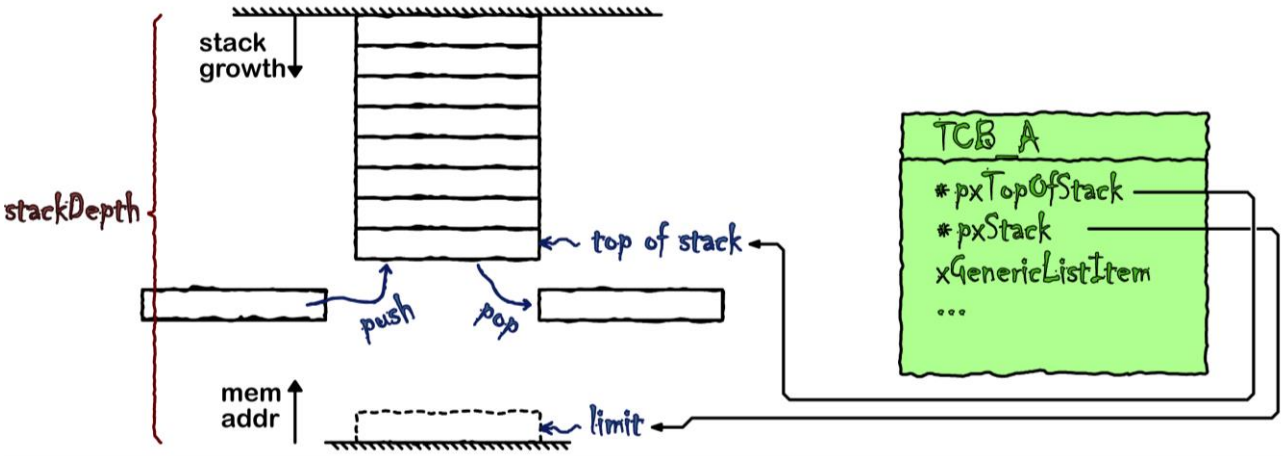
Many processors use an **ascending stack** architecture. The stack grows from low to high memory addresses.



Unfortunately memory is not endless. So you've got to define a certain limit on your stack. This limit determines how deep your stack can be.

Other processors – like the Cortex-M series – use a **descending stack** architecture. The stack grows in the reverse direction: from high to low memory addresses. The `*pxStack` field points to the lowest memory address, that is the lower limit of the stack. So actually `*pxStack` is not the beginning but rather the end of the stack. I've noticed that the comments in FreeRTOS are sometimes a bit confusing.

The field `*pxTopOfStack` points at the last item that was put on the stack. So 'top of stack' is a bit misleading, since it is not really the top but rather the item with the lowest memory address if compared to all other items on the stack. I believe that the confusing names given to the variables in FreeRTOS can be explained as follows. FreeRTOS is a general operating system ported to a large set of processors. Some of them use ascending stacks, others descending. The Cortex-M series is just one of the targeted microcontrollers. So we cannot expect the names of the variables to be consistent for both architectures. If you ever get confused, just get back to the figure below:



3.5 Initializing the TCB

After allocating the memory for the stack and the TCB, they are still empty[1]. An empty TCB is a problem. The TCB is the 'header' of the newly created Task. So we've got to initialize it. To do that, we call the function **`prvInitialiseTCBVariables(..)`**:

tasks.c

```
    /*-----------------------------------------------------------------------------*/
    /*      INITIALIZE TCB                                                         */
    /*-----------------------------------------------------------------------------*/
    /* Setup the newly allocated TCB with the initial state of the task.           */
    prvInitialiseTCBVariables( p_newTCB, p_name, uxPriority, xRegions, stackDepth );
```

Let us now examine the code from that function.

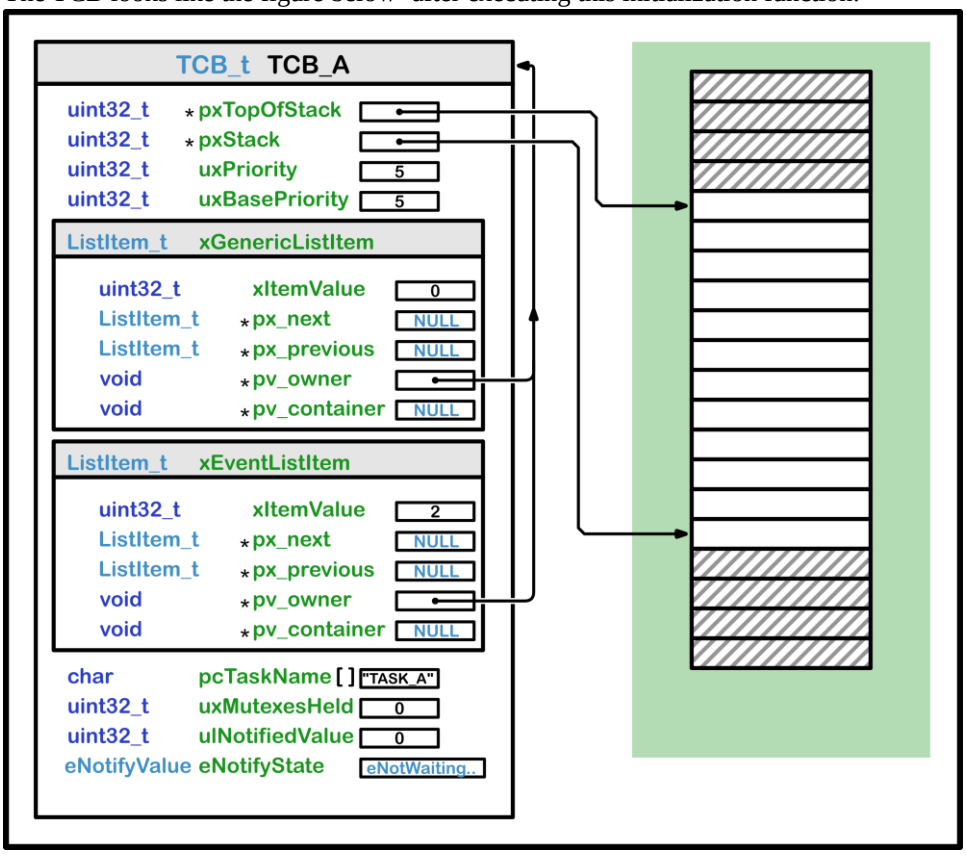
```
static void prvInitialiseTCBVariables(TCB_t * const p_TCB,
                                       const char * const p_name,
                                       uint32_t taskPriority,
                                       const MemoryRegion_t * const xRegions,
                                       const uint16_t usStackDepth )
{
    uint32 t x;

    /*-------------------------------------------------------------------------*/
    /*      TCB NAME                                                           */
    /*-------------------------------------------------------------------------*/

    for( x = 0; x < (uint32_t) configMAX_TASK_NAME_LEN; x++ )
    {
        p_TCB->pcTaskName[ x ] = p_name[ x ];

        /* Don't copy all configMAX TASK NAME LEN if the string is shorter  */
        /* just in case the memory after the string is not accessible       */
        /* (extremely unlikely).                                            */
        if( p_name[ x ] == 0x00 )
        {
            break;
        }
    }

    /* Ensure the name string is terminated in the case that the string     */
    /* length was greater or equal to configMAX TASK NAME LEN.              */
    p_TCB->pcTaskName[ configMAX_TASK_NAME_LEN - 1 ] = '\0';


    /*-------------------------------------------------------------------------*/
    /*      TASK PRIORITY                                                      */
    /*-------------------------------------------------------------------------*/

    if( taskPriority >= (uint32_t) configMAX_PRIORITIES )
    {
        taskPriority = (uint32_t) configMAX_PRIORITIES - (uint32_t) 1U;
    }

    p_TCB->uxPriority = taskPriority;
    p_TCB->uxBasePriority = taskPriority;

    /*-------------------------------------------------------------------------*/
    /*      MUTEXES                                                            */
    /*-------------------------------------------------------------------------*/

    p_TCB->uxMutexesHeld = 0;
```

Footnotes:

[1] The TCB is not completely empty. The memory allocation function **`prvAllocateTCBAndStack(..)`** makes the field `*pxStack` to point at the lower limit of the stack. But apart from that, all other fields are **`NULL`**.

```
    /*-----------------------------------------------------------------------*/
    /*      LIST ITEMS                                                       */
    /*-----------------------------------------------------------------------*/
    /* Set the list container to NULL such that the item doesn't think it    */
    /* is already contained in a list.                                       */
    /*                                                                       */
    vListInitialiseItem( &( p_TCB->xGenericListItem ) );
    vListInitialiseItem( &( p_TCB->xEventListItem ) );

    /* Make this TCB the owner of the list items.                            */
    listSET_LIST_ITEM_OWNER( &( p_TCB->xGenericListItem ), p_TCB );
    listSET_LIST_ITEM_OWNER( &( p_TCB->xEventListItem ), p_TCB );

    /* Give a value to the xEventListItem.                                   */
    listSET_LIST_ITEM_VALUE( &( p_TCB->xEventListItem ),
                             (uint32_t) configMAX_PRIORITIES - (uint32_t) taskPriority );

    /*-----------------------------------------------------------------------*/
    /*      MPU                                                              */
    /*-----------------------------------------------------------------------*/
    /* The Memory Protection Unit is not used                                */

    ( void ) xRegions;
    ( void ) usStackDepth;


    /*-----------------------------------------------------------------------*/
    /*      NOTIFICATIONS                                                    */
    /*-----------------------------------------------------------------------*/
    p_TCB->ulNotifiedValue = 0;
    p_TCB->eNotifyState = eNotWaitingNotification;

}
```

The TCB looks like the figure below[2] after executing this initialization function:



Footnotes:

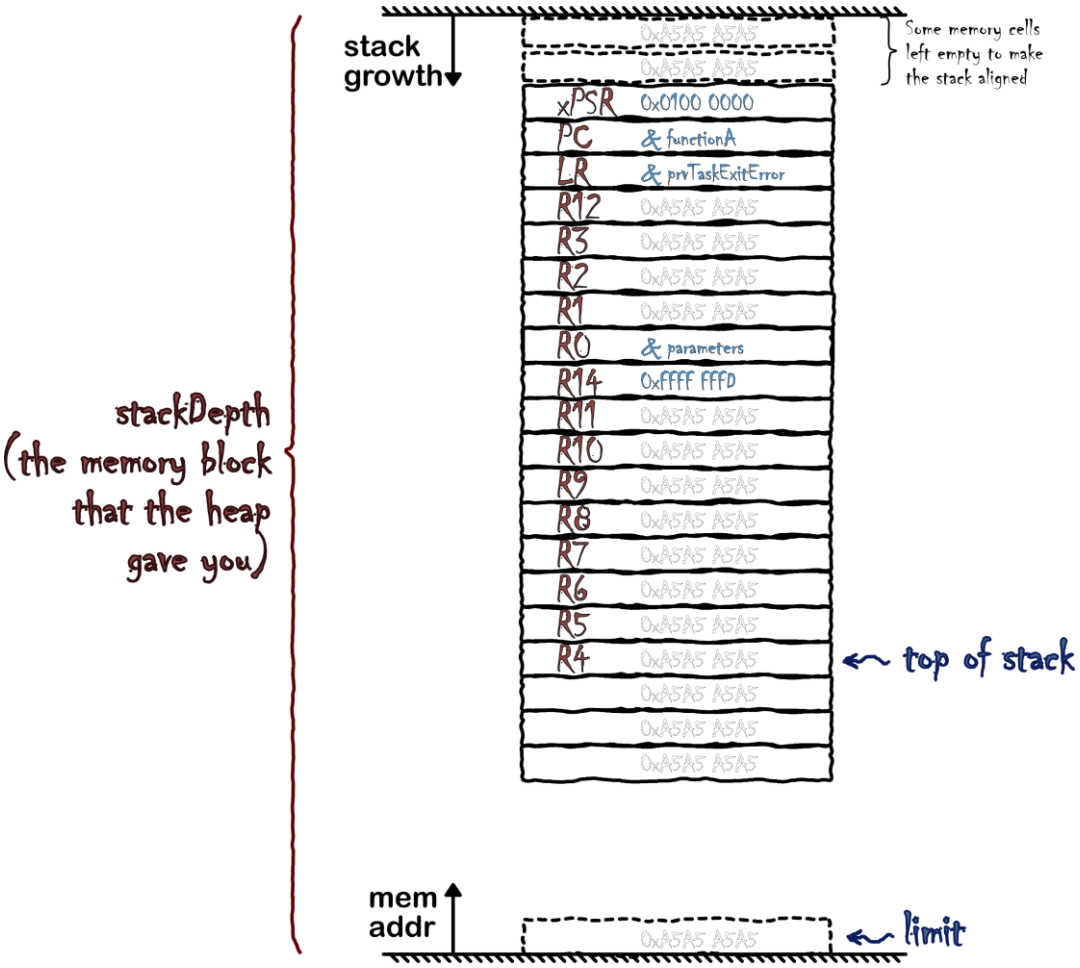
[2] To be 100% correct, I should mention that the initialization of the `*pxTopOfStack` pointer requires some more code that doesn't appear in this function. But we will dig into that soon.

3.6 Initializing the stack

Now that we have initialized the TCB, we can focus on the stack of the Task. But does a stack really needs initialization? The stack is nothing more than piece of memory on which the Task can store temporary variables. Since the Task has not yet started to run, there are no variables at this moment to store. So we should leave it empty for the moment, no?

To get an answer, we should explore first how the FreeRTOS kernel switches from one Task to the next. This kernel functionality is called 'context switching'. It happens every millisecond[3]. We will dig into the context switching later on, but I can reveal right now that the stack plays a vital role. When the kernel switches from Task A to B, it will first save the "context" of Task A onto its stack before moving on to Task B.

Now let us presume that the kernel is running the Idle Task. And we create `TASK_A` here. Don't we want a smooth context switch from `IdleTask` to `TASK_A`? An empty stack would crash the kernel when it attempts a context switch to the new Task. The function **`pxPortInitialiseStack(..)`** fills the stack with a few values such that the kernel can smoothly switch to `TASK_A`. This is what the stack looks like after executing **`pxPortInitialise-Stack(..)`** on it[4]:



After reading the further chapter about "context switching", you will completely understand why the stack has to look like this. The code to reach this stack state is given on the next page. The function **`pxPortInitialise-Stack(..)`** not only initializes the stack, but also returns the memory address from the "top of stack".

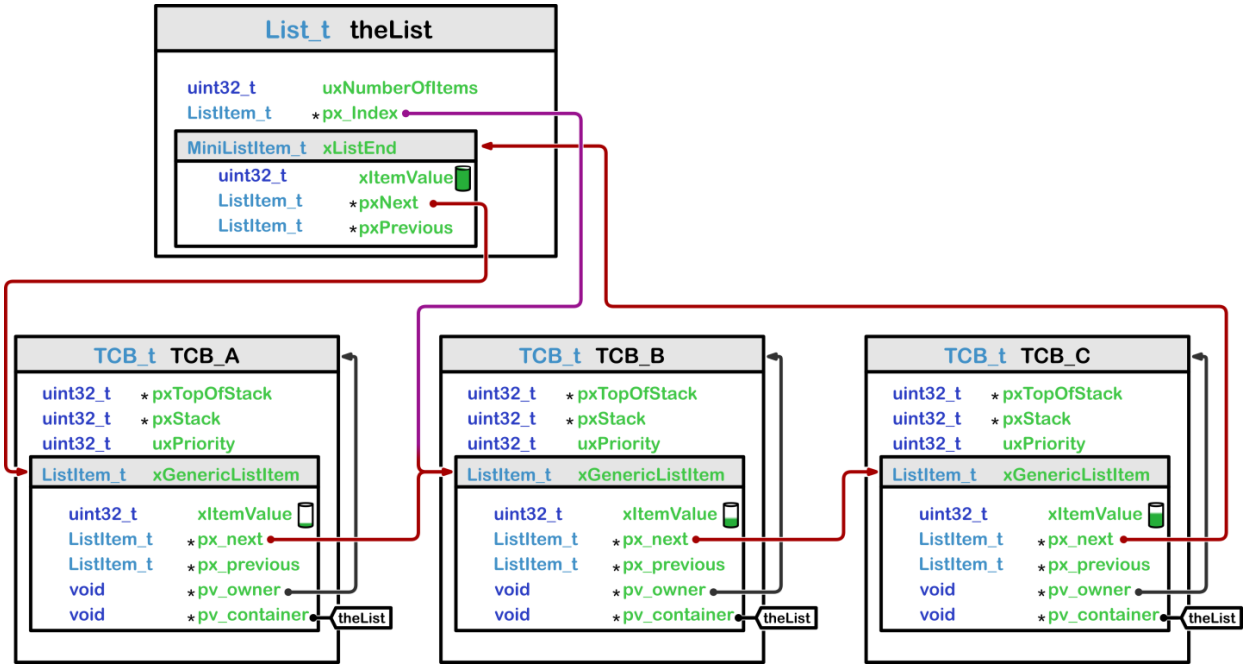
Footnotes:
[3] Once every millisecond is the default configuration. You can change it if you like.
[4] All memory cells are prefilled with `0xA5A5 A5A5` when the memory block is allocated from the heap.

port.c

```c
/*-------------------------------------------------------------------------------*/
/*                                                                               */
/*       ___      _ _   _      _ _             ___ _           _                 */
/*      |_ _|_ _ (_) |_(_)__ _| (_)___ ___    / __| |_ __ _ __| |__              */
/*       | || ' \| |  _| / _` | | (_-</ -_)   \__ \  _/ _` / _| / /              */
/*      |___|_||_|_|\__|_\__,_|_|_/__/\___|   |___/\__\__,_\__|_\_\              */
/*-------------------------------------------------------------------------------*/
/*                                                                               */
/*   Simulate the stack frame as if it would be created by a context             */
/*   switch interrupt.                                                           */
/*                                                                               */

#define init_xPSR              ( 0x01000000 )
#define init_EXEC_RETURN       ( 0xfffffffd )

uint32_t *pxPortInitialiseStack( uint32_t *pxTopOfStack,
                                 TaskFunction_t pxCode, void *pvParameters )
{


    /* Offset added to account for the way the MCU uses the stack on            */
    /* entry/exit of interrupts, and to ensure alignment.                       */
    pxTopOfStack--;

    /*-------------------xPSR-------------------------*/
    *pxTopOfStack = init_xPSR;              /* xPSR */
    pxTopOfStack--;

    /*--------------------PC--------------------------*/
    *pxTopOfStack = ( uint32_t ) pxCode;    /* PC <- This is where the interrupt    */
    pxTopOfStack--;                         /*       returns to when the ISR        */
                                            /*       completes.                     */


    /*--------------------PC--------------------------*/
    *pxTopOfStack = ( uint32_t ) portTASK_RETURN_ADDRESS;   /* LR <- This is where the    */
                                                            /*       function returns to  */
                                                            /*       when the function    */
                                                            /*       call completes.      */

    /*---------------R12, R3, R2, R1 and R0--------------*/
    pxTopOfStack -= 5;                           /* R12, R3, R2, R1 should be left empty.  */
    *pxTopOfStack = ( uint32_t ) pvParameters;   /* R0 should contain a ptr to parameters. */


    /*---------------R14: EXEC_RETURN--------------------*/
    pxTopOfStack--;
    *pxTopOfStack = init_EXEC_RETURN;  /* Return to Thread mode, exception return uses     */
                                       /* non-floating-point state from the PSP and        */
                                       /* execution uses PSP after return.                 */

    /*--------R11, R10, R9, R8, R7, R6, R5 and R4--------*/
    pxTopOfStack -= 8;        /* R11, R10, R9, R8, R7, R6, R5 and R4 should be left empty. */

    return pxTopOfStack;
}
```

3.7 Lists of Tasks

How does the Kernel keep track of all Tasks in your program? Using Lists! We already touched this subject briefly in the previous chapter. Do you remember this figure? Tasks A, B and C 'own' an item from a certain List. In this way, you could consider Tasks A, B and C to be part of that List.

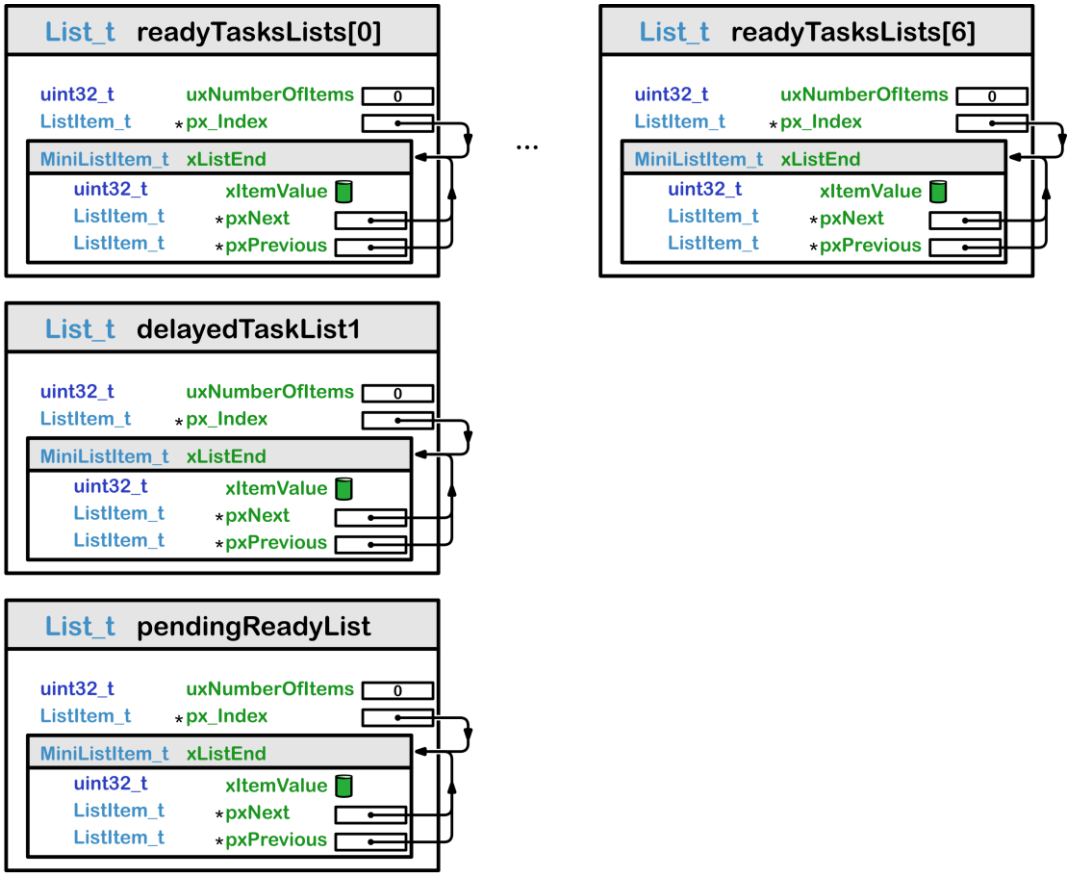


The freeRTOS kernel has several lists. A list for the Tasks that are ready. A list for the Tasks that are delayed. A list for those that should be deleted, etc...
At the top of the `tasks.c` file you can find all of them:

tasks.c

```
/*-----------------------------------------------------------*/
/*                        TASK LISTS                         */
/*-----------------------------------------------------------*/

static List_t readyTasksLists[ configMAX_PRIORITIES ];   /*< Prioritised ready tasks. */

static List_t delayedTaskList1;                     /*< Delayed tasks. */
static List_t delayedTaskList2;                     /*< Delayed tasks (two lists are used,
                                                        one for delays that have overflowed
                                                        the current tick count. */

static List_t * volatile p_delayedTaskList;         /*< Points to the delayed task list
                                                        currently being used. */

static List_t * volatile p_overflowDelayedTaskList; /*< Points to the delayed task list
                                                        currently being used to hold tasks
                                                        that have overflowed the current
                                                        tick count. */

static List_t pendingReadyList;                     /*< Tasks that have been readied while
                                                        the scheduler was suspended. They
                                                        will be moved to the ready list when
                                                        the scheduler is resumed. */


static List_t tasksWaitingTermination;              /*< Tasks that have been deleted - but
                                                        their memory is not yet freed. */

static List_t suspendedTaskList;                    /*< Tasks that are currently suspended. */
```

The same file also provides a function to initialize all of them: **`prvInitialiseTaskLists`**`(void)`. An initalized List has only one element: **`xListEnd`**. Here are a few examples of how these Lists look like after initialization:




```
#define prvAddTaskToReadyList( p_TCB )                                          \
  scheduler.topReadyPriority |= ( 1UL << ( (p_TCB)->priority ) );               \
  vListInsertEnd( &( coreLists.readyTasks[ (p_TCB)->priority ] ),
                                                  &( (p_TCB)->genericListItem ) )
```

3.8 Creation of a new Task

At this moment we do have enough background to study the creation of a new Task. We know how to allocate memory for the TCB and the stack. We know how to initialize them, etc. To create a new Task you can simply call the function **`xTaskGenericCreate(..)`**. Or you can use the macro xTaskCreate(..). This macro is a wrapper for the function:

```
#define xTaskCreate( p_taskFunc, p_name,
                     stackDepth, p_parameters,
                     priority, p_taskHandle )    xTaskGenericCreate( (p_taskFunc), (p_name),
                                                                     (stackDepth), (p_parameters),
                                                                     (priority), (p_taskHandle),
                                                                     (NULL), (NULL) )
```

Let us now dive into the code of **`xTaskGenericCreate(..)`**:

```
/* -----------------------------------------------------------------------------------------*/
/*                                                                                          */
/*      _____         _                     _                                              */
/*     |_   _|_ _ ___| |__  __ _ _ _ ___ __ _| |_ ___                                      */
/*       | |/ _` (_-< / / / _| '_/ -_) _` |  _/ -_)                                        */
/*       |_|\__,_/__/_\_\ \__|_| \___\__,_|\__\___|                                        */
/* -----------------------------------------------------------------------------------------*/

int32_t xTaskGenericCreate( TaskFunction_t p_taskFunc,
                            const char * const p_name,
                            const uint16_t stackDepth,
                            void * const p_parameters,
                            uint32_t priority,
                            TaskHandle_t * const p_taskHandle,
                            uint32_t * const p_stackBuf,
                            const MemoryRegion_t * const xRegions )
{
    int32_t returnVal;
    TCB_t * p_newTCB;
    uint32_t *p_topOfStack;

    configASSERT( p_taskFunc );


    /*----------------------------------------------------------------------------------*/
    /*       ALLOCATE MEMORY FOR STACK AND TCB                                          */
    /*----------------------------------------------------------------------------------*/
    p_newTCB = prvAllocateTCBAndStack(stackDepth, p_stackBuf);
    if(p_newTCB == NULL)
    {
        returnVal = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
        return returnVal;
    }

    /*----------------------------------------------------------------------------------*/
    /*       INITIALIZE TCB                                                             */
    /*----------------------------------------------------------------------------------*/
    /* Setup the newly allocated TCB with the initial state of the task.              */
    prvInitialiseTCBVariables( p_newTCB, p_name, priority, xRegions, stackDepth );

    /*----------------------------------------------------------------------------------*/
    /*       INITIALIZE STACK                                                           */
    /*----------------------------------------------------------------------------------*/
    /* Initialize the stack as if the task was already running, but had been          */
    /* interrupted by the scheduler. The return address is set to the start of the    */
    /* task function.                                                                 */
    /* Before initializing the stack, make sure that the pointer to the top is 8-byte */
    /* aligned.                                                                       */
    /*                                                                                */

    p_topOfStack = p_newTCB->p_stack + ( stackDepth - (uint16_t) 1 );
    p_topOfStack = (uint32_t *) ( ((uint32_t)p_topOfStack) & ( ~((uint32_t)0x0007) ) );
    configASSERT( ( ( (uint32_t)p_topOfStack & (uint32_t)0x0007 ) == 0UL ) );

    p_newTCB->p_topOfStack = pxPortInitialiseStack( p_topOfStack, p_taskFunc, p_parameters );
```

```
    /*-----------------------------------------------------------------------------------*/
    /*      CONNECT THE TASK HANDLE                                                      */
    /*-----------------------------------------------------------------------------------*/

    if( (void *) p_taskHandle != NULL )
    {
        /* Pass the TCB out - in an anonymous way.  The calling function or task can    */
        /* use this as a handle to delete the task later if required.                   */
        *p_taskHandle = ( TaskHandle_t ) p_newTCB;
    }

    /*-----------------------------------------------------------------------------------*/
    /*      ADD THE TASK TO THE READY LIST                                               */
    /*-----------------------------------------------------------------------------------*/

    taskENTER_CRITICAL();   // Task lists will get updated. Interrupts should not
    {                       // access them now!
        currentNumberOfTasks++;
        if( p_currentTCB == NULL )
        {
            /* There are no other tasks, or all the other tasks are in          */
            /* the suspended state -> make this the current task.               */
            p_currentTCB = p_newTCB;

            if( currentNumberOfTasks == (uint32_t) 1 )
            {
                /* This is the first task to be created so do the preliminary   */
                /* initialisation required.  We will not recover if this call   */
                /* fails, but we will report the failure.                       */
                prvInitialiseTaskLists();
            }

        }
        else if( (schedulerIsRunning == false) && ( p_currentTCB->priority <= priority ) )
        {
            /* The scheduler is not running and this Task has a higher priority */
            /* than the current task. -> Make this Task the current one.        */
            p_currentTCB = p_newTCB;
        }


        taskNumber++;
        prvAddTaskToReadyList( p_newTCB );
        returnVal = pass;
    }
    taskEXIT_CRITICAL();


    /*-----------------------------------------------------------------------------------*/
    /*      CALL FOR CONTEXT SWITCH IF PRIORITY IS HIGH                                  */
    /*-----------------------------------------------------------------------------------*/

    if( (returnVal == pass)&&(schedulerIsRunning)&&(p_currentTCB->priority < priority) )
    {
        /* The created task is of a higher priority than the current task       */
        /* so it should run now.                                                */
        invokePendSV();
    }

       return returnVal;
}
```

The macro `prvAddTaskToReadyList( p_newTCB )` is expanded as follows:

```
{
    scheduler.topReadyPriority |= ( 1UL << ( p_TCB->priority ) );

    vListInsertEnd( &( coreLists.readyTasks[p_TCB->priority] ), &( p_TCB->genericListItem ) );
}
```

The kernel scheduler keeps track of the highest priority Task that is present in the system[5]. The variable ***scheduler*.topReadyPriority** is used for that purpose. Let us take an example. Imagine that a Task with priority 5 is added to the ready lists[6]. The kernel could notice that the variable **topReadyPriority** equals 4. Since 5 > 4, the kernel can update **topReadyPriority**. This approach would be perfectly fine. But there is a faster alternative to win a few CPU cycles. We are dealing with kernel code here, so winning CPU cycles is always a good thing. Notice how the kernel updates its **topReadyPriority** variable:

```
scheduler.topReadyPriority |= ( 1UL << ( p_TCB->priority ) );
```

This OR operation takes only one instruction. To continue our example, we can assume that the variable gets the value:

```
scheduler.topReadyPriority = 0b0000 0000 0011 xxxx
```

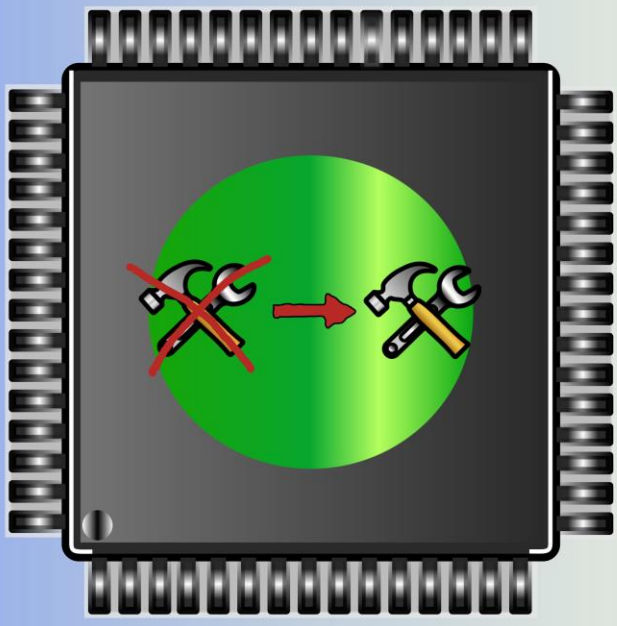
Only the leftmost '1' bit is important. The position of that bit tells you what the highest priority level in the system is.

[5] Only for Tasks that are in the ready lists.

[6] Do not forget that Task priority levels and interrupt priority levels are two very different things! Even the conventions in numbering do not correspond. A high priority interrupt has a low numerical priority value. A Task with high priority has a high numerical priority value.

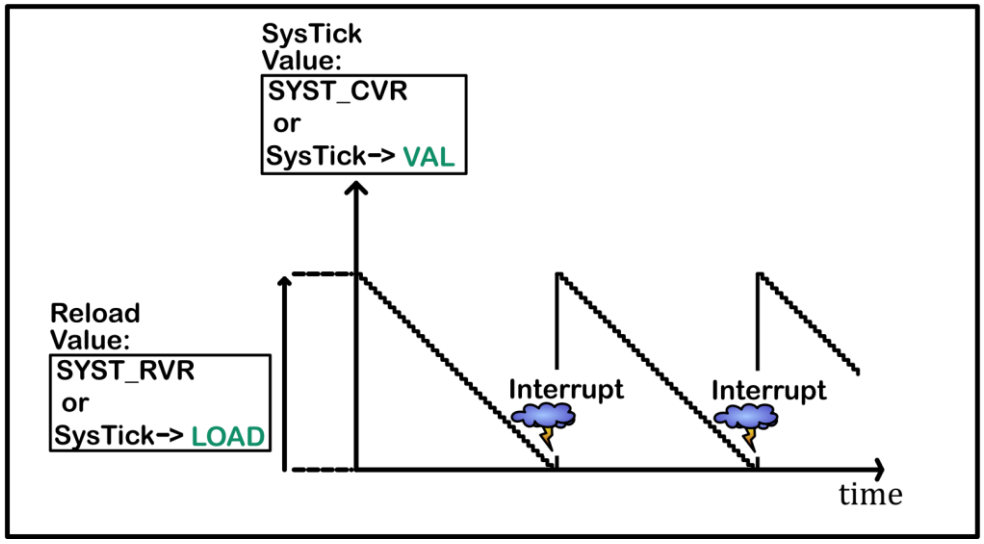
Chapter 4

The Context Switch



4.1 The RTOS heartbeat

The Cortex-M7 has a built-in 24-bit downcounting timer. This timer is specifically designed for an RTOS. You can consider it as the heartbeat of the system. Every millisecond[7] the timer generates an interrupt.



Suppose the CPU is processing `TASK_A`. The `SysTick` timer generates an interrupt. Below is the code of the `SysTick_Handler` ISR (slightly simplified). The target of `SysTick_Handler` is to find out if `TASK_A` is really the highest priority task that can run. If so, then `SysTick_Handler` will exit without doing anything. `TASK_A` continues to run as if nothing happened.

port.c

```
void SysTick_Handler( void )
{
    {
        /* Increment the RTOS tick. */
        if( increment_Tick() != false )
        {
            /* A context switch is required.  Context switching is performed in
               the PendSV interrupt.  Pend the PendSV interrupt. */
            invokePendSV();
            /* To my understanding, the PendSV interrupt and the SysTick interrupt
             * have both the same priority. So this SysTick interrupt cannot be
             * preempted by PendSV. It will smoothly finish before PendSV is
             * executed.
             */
        }
    }
}
/*-----------------------------------------------------------*/
```

The subroutine **`increment_Tick`**`()` takes the actual decision if a context switch is necessary. We will cover that in the chapter about the Kernel. Now we only want to investigate how the context switch from Task A to Task B is executed.

[7] One millisecond is the default configuration. Of course you can change it.

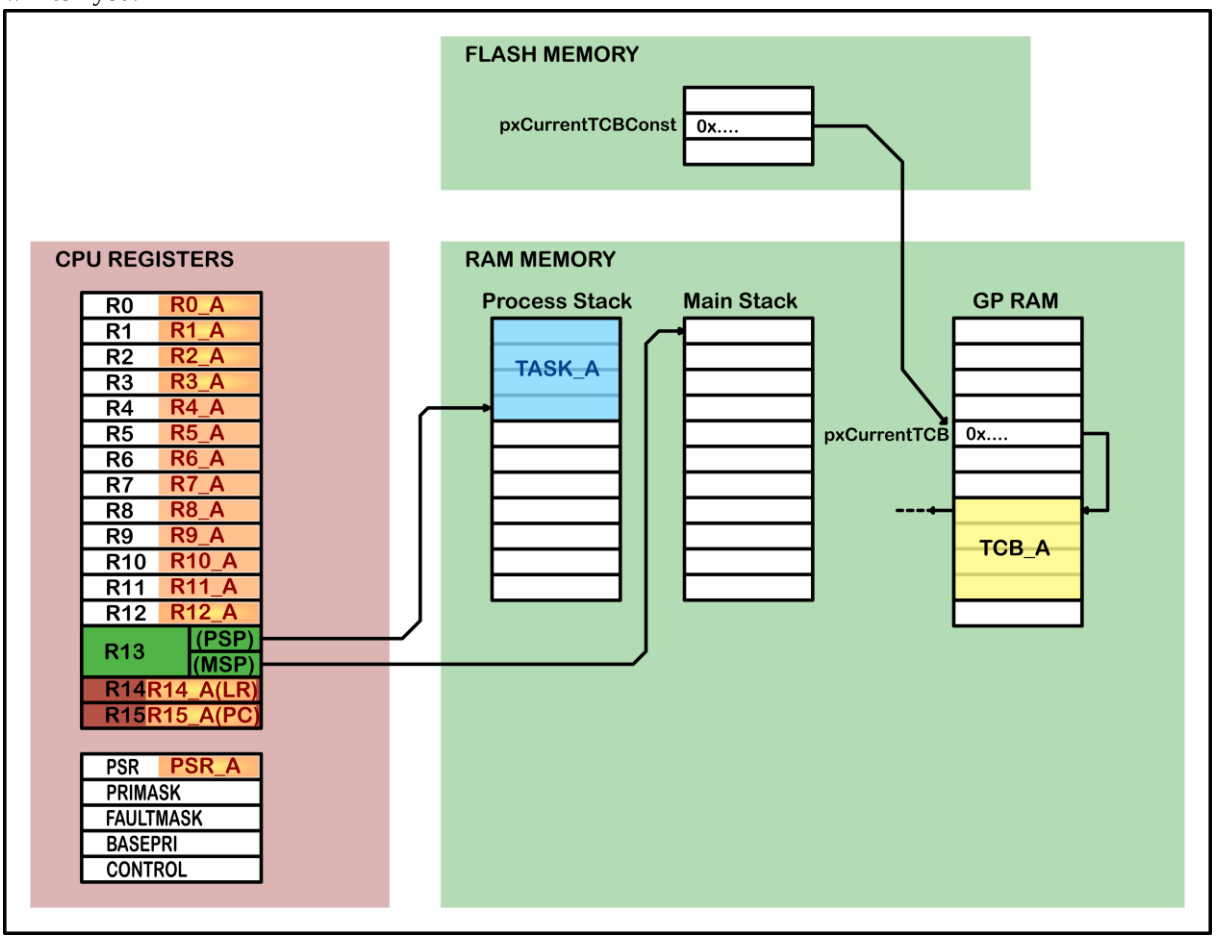
4.2 The Context Switch

Let us presume that `TASK_B` has a higher priority and is ready to execute. The subroutine **`xTaskIncrementTick()`** will notice this and returns `true`. As a result **`invokePendSV()`** gets called. `PendSV` is an interrupt-driven request for system-level service. That's what the ARM Cortex-M7 datasheet says about it. But do not worry, it only means that the interrupt `PendSV` is now "pending". It is waiting for its turn to get serviced by the CPU. When `SysTick_Handler` is finished – and there is no other more important interrupt pending – the CPU will start to service the `PendSV_Handler()` ISR.

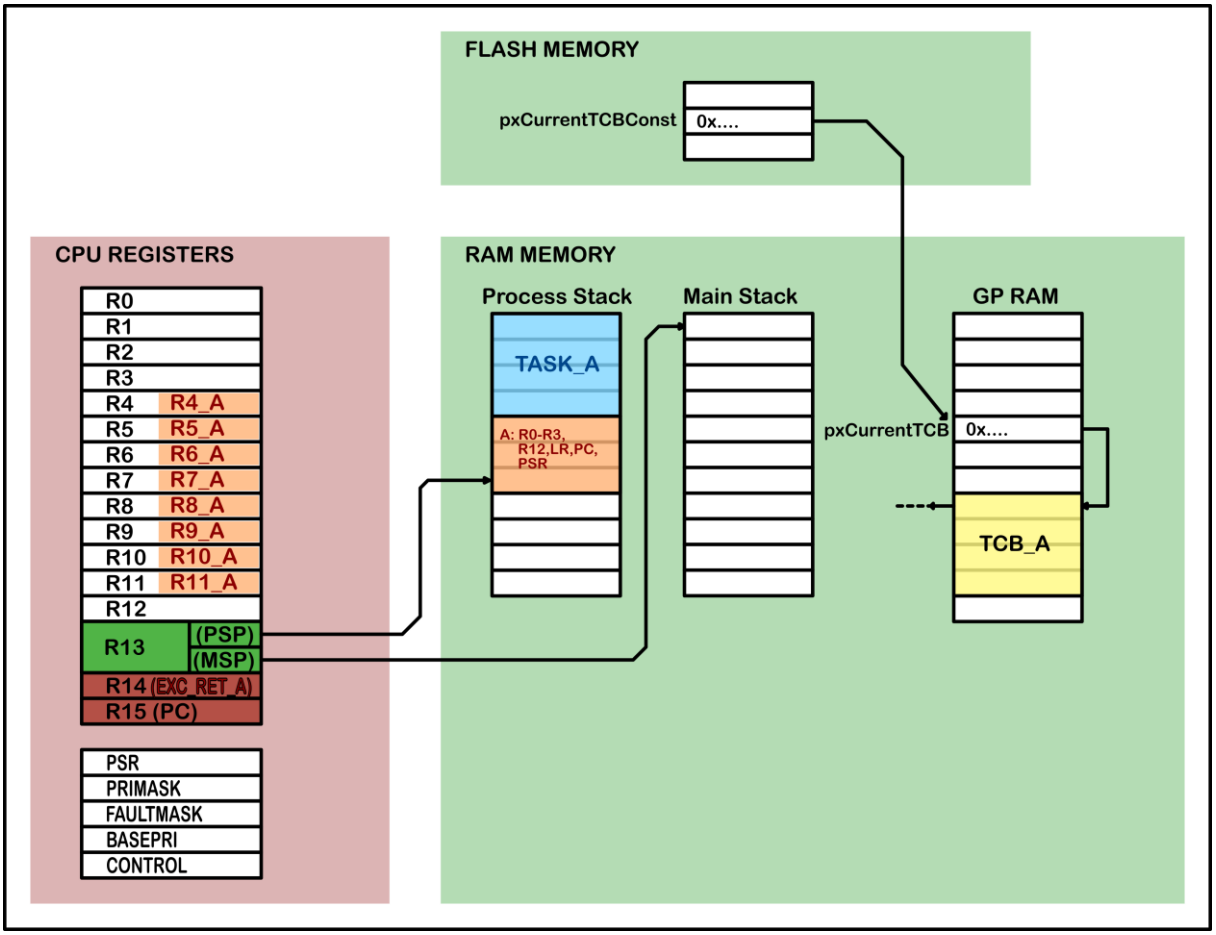
Let us now study the `PendSV_Handler()` ISR. The figure below shows the state of the system when it is still running `TASK_A` *just before* the `PendSV` interrupt fires. All the CPU registers can have important data that should not get lost. Maybe the CPU was in the middle of an important calculation in `TASK_A`. I colored all these registers orange.

The Cortex-M7 has two stack pointers (the `CONTROL` register defines which one is used). When `TASK_A` is running the CPU uses the `PSP` (Process Stack Pointer). The blue colored memory cells represent data that is on the stack at this very moment. Retaining this data is important to correctly continue the execution of `TASK_A`.

Somewhere in the Flash memory is a memory cell labeled "`pxCurrentTCBConst`". It points to the variable `pxCurrentTCB` that lives in RAM. And `pxCurrentTCB` points to `TCB_A` – the "`Task Control Block`" of `TASK_A`. Maybe this is a good moment to say something important about tasks. In FreeRTOS every task has a "`Task Control Block`". This datastruct holds vital information about the task. The variable `pxCurrentTCB` is a global variable (!) that points to the TCB of the task that is currently running. So if you ever get confused and want to know which task is running at the very moment, you simply take a look at where `pxCurrentTCB` is pointing at. But what if you don't know where to find this `pxCurrentTCB` pointer? Well, `pxCurrentTCBConst` will tell you.



Upon entering the `PendSV_Handler()` ISR, the CPU will automatically push registers `R0 – R3`, `R12`, `LR`, `PC` and `PSR` onto the stack. After doing this, the CPU automatically writes an `EXC_RETURN` value into `R14(LR)`. When first reading this in the datasheet of the M7-core, I thought that this value is the return address such that the CPU can resume `TASK_A` after exiting the interrupt. But I was wrong. The return address is the `PC` that is pushed onto the stack. The `EXC_RETURN` value written into `R14` is a standardised pattern with almost all its bits set to `'1'`. The lowest 5 bits give some information about which stack was used (Process Stack or Main Stack) and if the FPU (Floating Point Unit) was active. At the end of the interrupt ISR the `EXC_RETURN` value should be written into the `PC`. The CPU will recognize the pattern and initiate the appropriate exception return sequence.
So to summarise the story, just take a look at the figure below. This is the system state when entering the `PendSV_Handler` ISR:

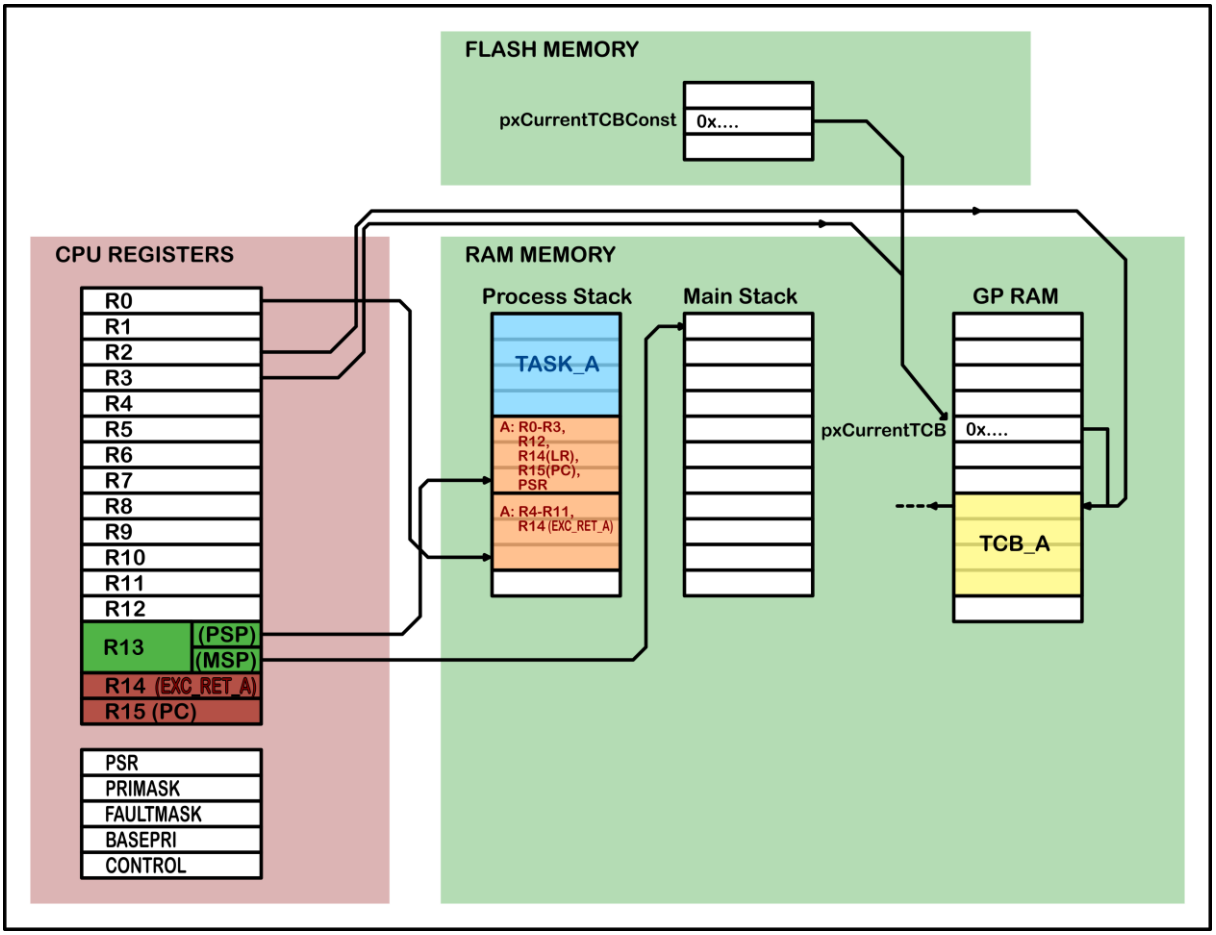


Your sharp mind has noticed that registers `R4 – R11` did not get stored. That's correct. By convention the ISR routine should not touch them. Or if the ISR wants to use them, it has the responsibility to restore them in their initial state before exiting.
We want to save the complete context of `TASK_A`. So let's store these registers on the stack. We will do that in a minute. First things first. Let us look at the very first code-lines in the ISR. Because until now, not even one instruction of the ISR has been executed. Everything discussed so far happens automatically – it is burned in the silicon.

This is how the ISR starts:

port.c

```
void PendSV_Handler( void )
{
    /* This is a naked function. */   <- note: I do not know what this means

    __asm volatile
    (
    " mrs r0, psp                   \n" /* Move PSP (process stack ptr) into r0.                    */
    " isb                           \n" /* ISB: Instruction Synchronization barrier.                */
    "                               \n"
    " ldr r3, pxCurrentTCBConst     \n" /* Load r3 with the address of the variable 'pxCurrentTCB'  */
    "                               \n" /*                                                          */
    " ldr r2, [r3]                  \n" /* Load r2 with the address of 'TCB_A'                      */
    "                               \n" /*                                                          */

    /* Some code to store floating   */
    /* point registers on the stack. */
    /* I omitted it for simplicity.  */

    "                               \n" /* -------------------------------------------------------- */
    "                               \n" /*    PUSH CORE REGISTERS ONTO THE STACK                    */
    "                               \n" /* -------------------------------------------------------- */
    "                               \n"
    " stmdb r0!, {r4-r11, r14}      \n" /* Push the core registers onto the process stack. Write    */
    "                               \n" /* back the final address into r0.                          */
    "                               \n" /* So now:                                                  */
    "                               \n" /* - r0 contains the addr of the stack 'top'.               */
    "                               \n" /*                                                          */
```



Well done! All the registers are now on the (process) stack! So now it is about time to start the switch to `TASK_B`. But one more thing needs to be done first. `R0` points to the 'stack top' of `TASK_A` and all of its saved context. If

we are not careful, R0 gets overwritten and no one knows where the stack top of TASK_A and its context is! So what we need to do is storing that value into the TCB (Task Control Block) from TASK_A.
This is done in the following code line:

```
    " str r0, [r2]                \n" /* Save the new 'top of stack'                          */
    "                             \n" /* into the first member TCB_A.                        */
    "                             \n" /*                                                     */
```

R2 contains the address of the TCB_A data-structure. In practice this boils down to "R2 contains the address of the first member of data-structure TCB_A". The instruction above will store the value from R0 into the memory cell addressed by R2. After doing that, the first member of TCB_A points to the top of the stack! And that is indeed what we envision the first member to do. Take a look at the struct definition:

tasks.c

```
typedef struct
{
    volatile uint32_t  *pxTopOfStack;  /*< Points to the location of the last item  */
                                       /*  placed on the tasks stack.               */
    ...
}
```

Now it is – finally – time to start the context switching. The context switch is performed by making the global variable pxCurrentTCB point to TCB_B instead of TCB_A. To make that switch, PendSV_Handler will call another subroutine: **vTaskSwitchContext**(). But one has to be careful when calling a subroutine. By convention you should assume that the subroutine will mess up registers R0-R3, R12 and R14. We might want to keep R3 and the MSP (Main Stack Pointer). So let us save them on the *main stack*. That's right, don't mess with the process stack now! You want to keep that one clean.

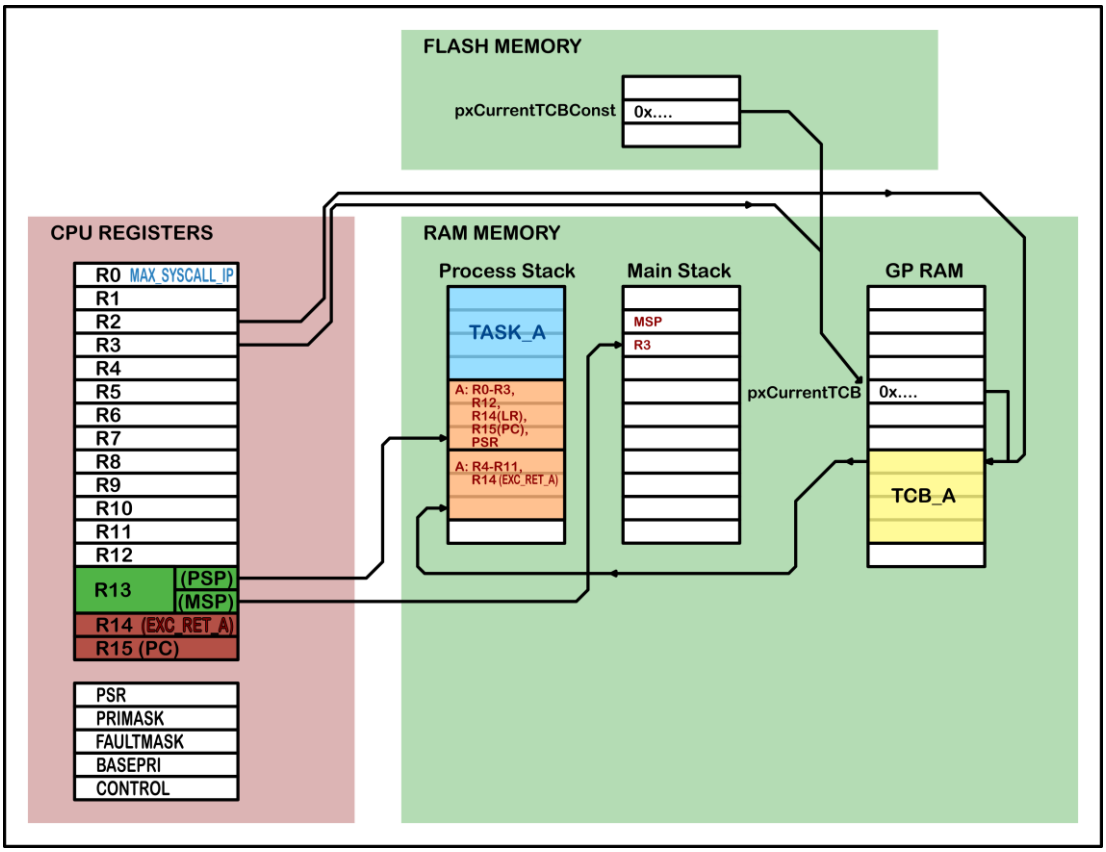
```
    "                             \n" /*                                                     */
    " stmdb sp!, {r3}             \n"
    "                             \n" /*                                                     */
```

And let us also block any interrupts that have a lower priority (higher numerical value) than configMAX_SYSCALL_INTERRUPT_PRIORITY.

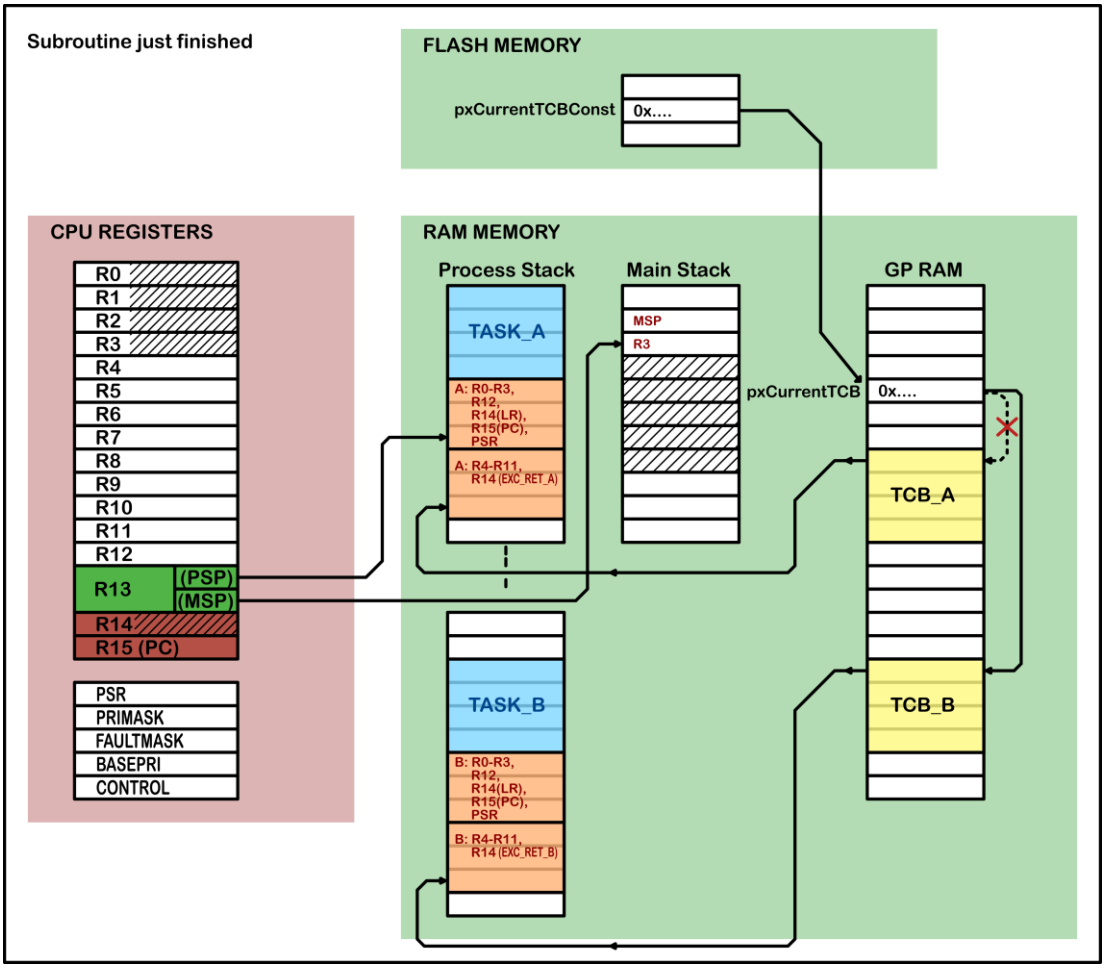
```
    " mov r0, %0                  \n" /* %0 corresponds to configMAX_SYSCALL_INTERRUPT_PRIORITY   */
    " cpsid i                     \n" /* Errata workaround. */
    " msr basepri, r0             \n" /* Block interrupts.                                   */
    " dsb                         \n"
    " isb                         \n"
    " cpsie i                     \n" /* Errata workaround. */
```

But what if the subroutine **vTaskSwitchContext**() messes with the process stack? Do not worry because the CPU is now running in Handler mode. That is the CPU mode when an interrupt routine is serviced. This means that standard push and pop instructions use the MSP (Main Stack Pointer) instead of the PSP (Process Stack Pointer).

Take a look at the following figure to get an overview of the system just before the CPU jumps to the subroutine **`vTaskSwitchContext()`**:



After finishing the subroutine, the system looks like this:

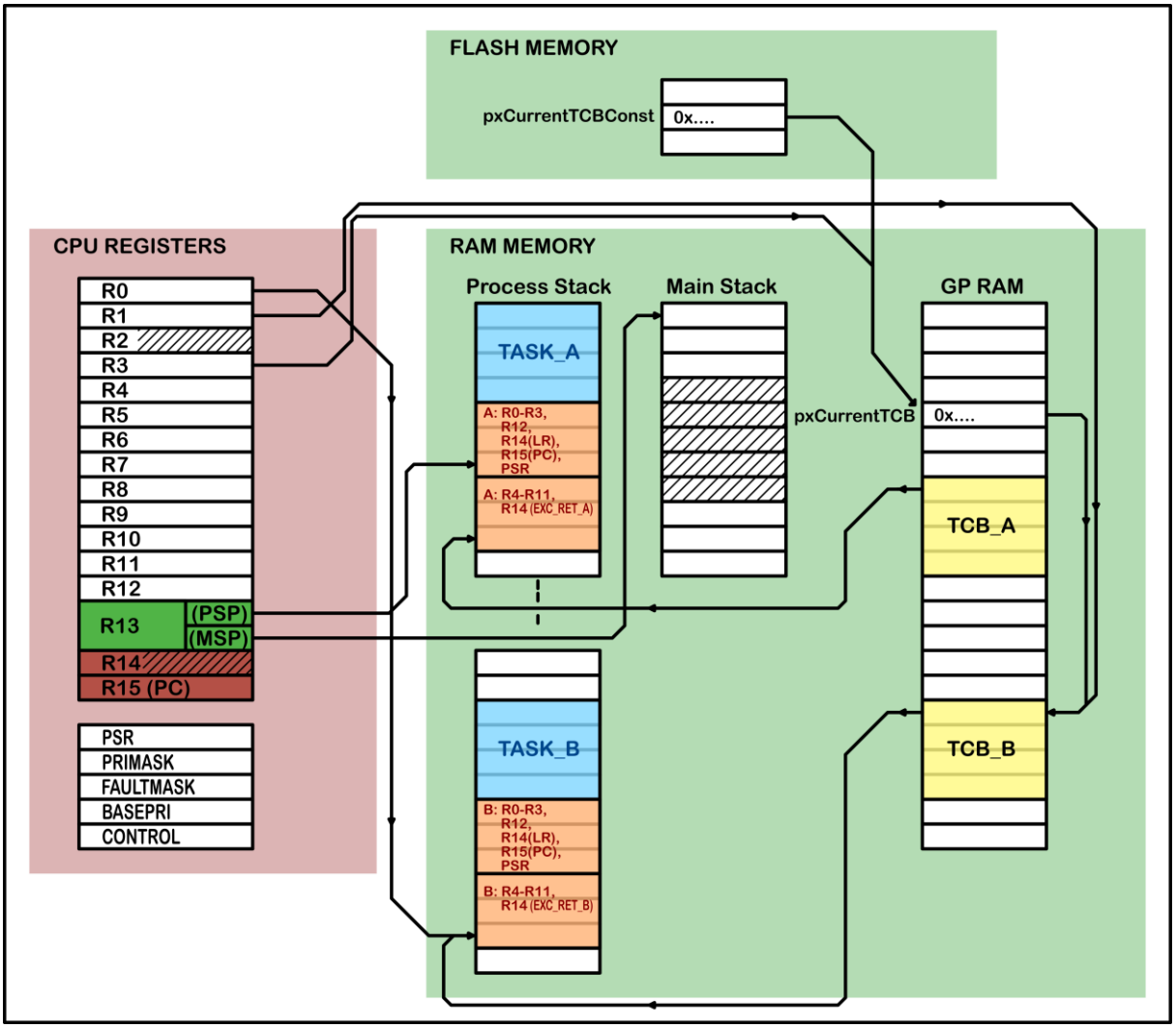


The subroutine did nothing more than making the `pxCurrentTCB` variable point to `TCB_B` instead of `TCB_A`. And as you can see on the figure, the first member of `TCB_B` points to the top of the stack from `TASK_B`. The subroutine might have messed up CPU registers `R0-R3`, `R12` and `R14` (I've put /// in the memory cells to indicate that). But that's okay because we knew that beforehand.
After exiting the subroutine, we will no longer block interrupts, and we will pop the saved registers from the main stack:

```
    " mov r0, #0                \n" /*                                          */
    " msr basepri, r0           \n" /* Allow interrupts                        */

    " ldmia sp!, {r3}           \n" /* Load r3 from the stack                  */
```

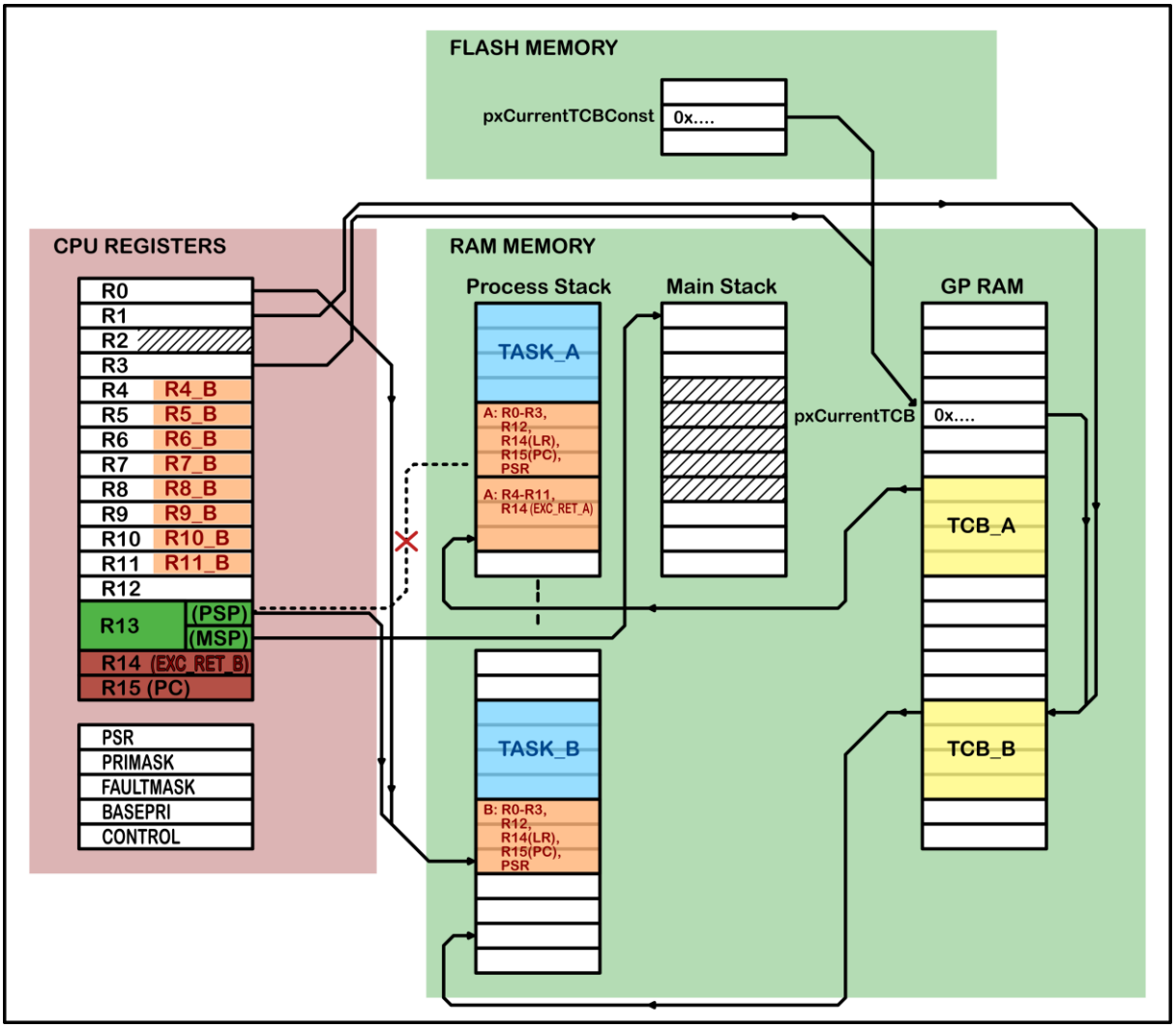
Okay, back to work. We know that the variable `pxCurrentTCB` points to the new TCB_B. And we know that the first member thereof contains the 'top of stack' from `TASK_B`. We want that! We want to track it down, and save the address of the "top of stack from `TASK_B`" into one of the CPU registers. Let us choose `R0` for that purpose. The following code will put the address of the "top of stack from `TASK_B`" into `R0`:

```
    "                           \n"
    " ldr r1, [r3]              \n" /*   REMEMBER: pxCurrentTCB points to the new one now   */
    " ldr r0, [r1]              \n" /* So now:                                              */
    "                           \n" /* - r0 holds the value of first member of TCB_B. This  */
    "                           \n" /*      value is the addr of the top of stack from Task B. */
    "                           \n" /* - r1 holds the address of TCB_B (so its first member) */
    "                           \n" /* - r3 holds the address of pxCurrentTCB               */
    "                           \n"
```



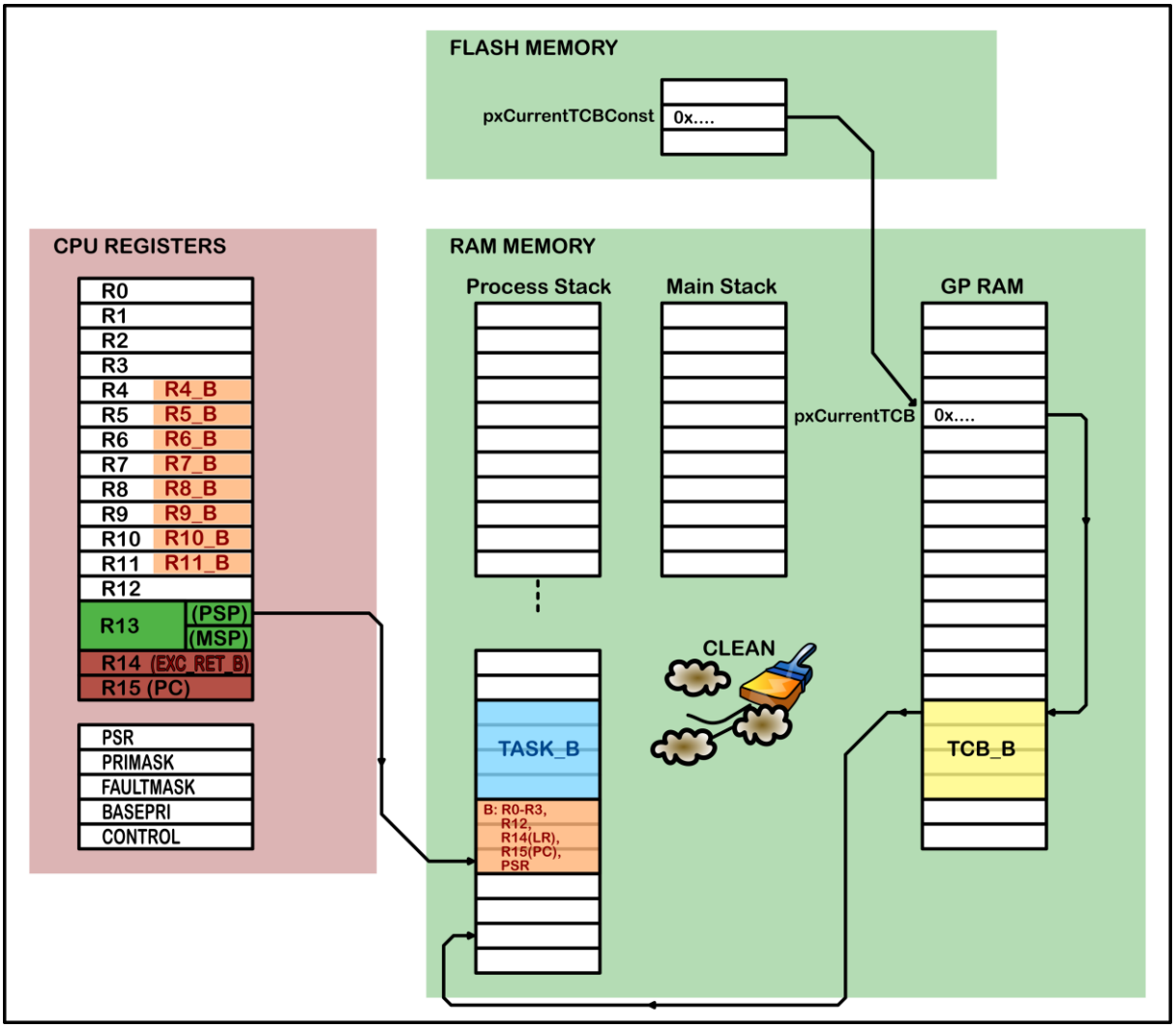
We are almost there. The time is ready to start unwinding the "saved `TASK_B` context" from the stack and loading it into the CPU registers. This should be done in two phases: a software phase and a hardware phase. Just take a look at how we have built the context of `TASK_A` on the stack to understand why. Part of it was pushed onto the stack automatically when entering the ISR. There was no code needed for that. Another part of the context was pushed onto the stack by the first code lines of the ISR.
So let us now unwind the 'software part' of the `TASK_B` context:

```
    " ldmia r0!, {r4-r11, r14}    \n" /* Pop the core registers.                  */
    " msr psp, r0                 \n" /* Update the PSP                           */
```



Unwinding the 'hardware part' is not our responsibility. But we should make sure that the system is in a correct state when the hardware does its work. Because the previous figure is a bit overwhelming, I've made a new figure for you. This new figure is just a cleaned-up version of the previous one. Only the necessary arrows are showed. The system itself has not changed!

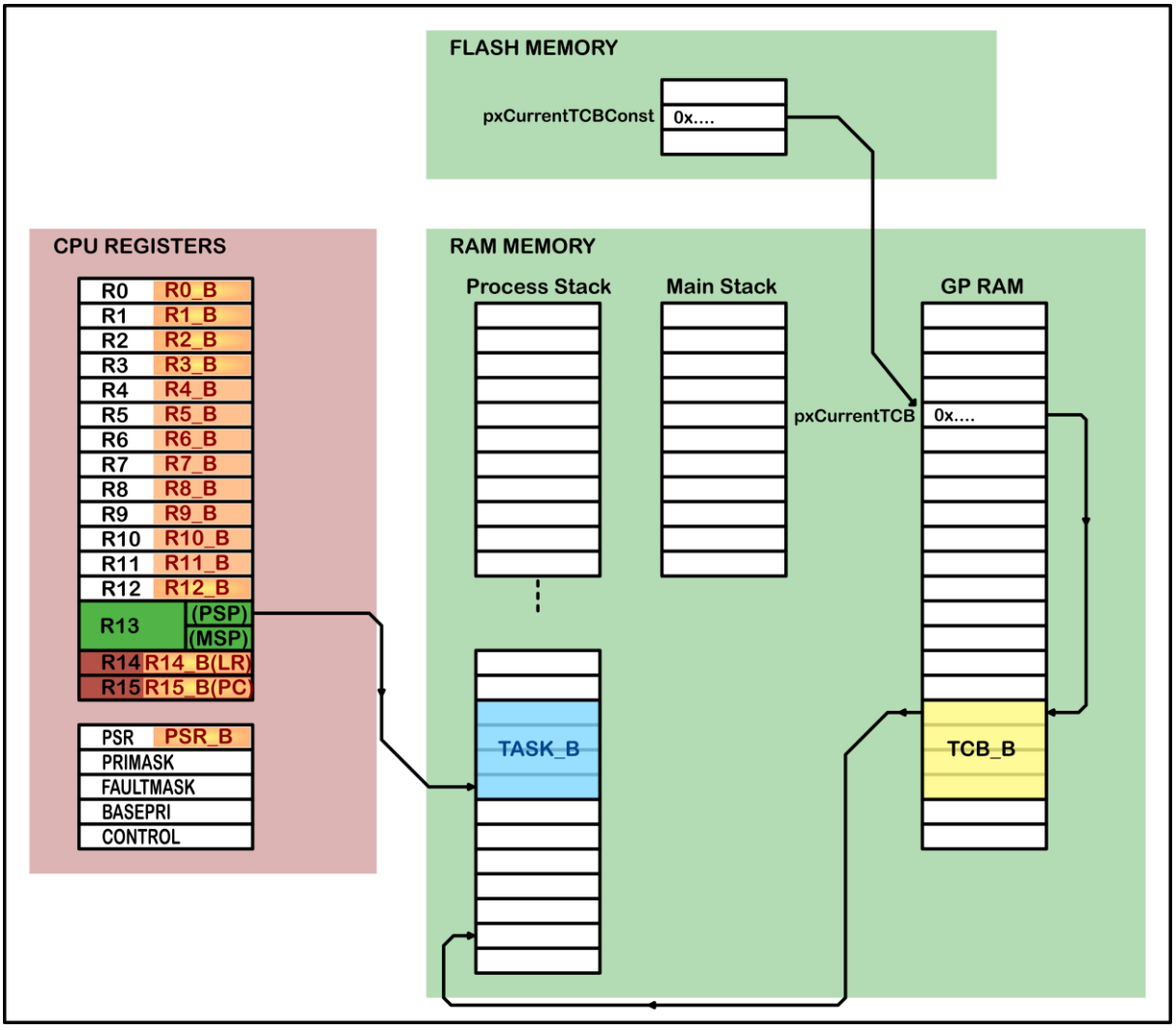
Now take a careful look at this figure. Do you notice it? That's right! It's exactly the same as the second figure in the story. Just go back a few pages. And look at the figure of the system when the interrupt routine is just entered by the CPU.
This figure is exactly the same, except that now the system has a `_B` instead of a `_A` suffix. You get the point? We can now safely exit the interrupt routine, and the CPU will restore the state of `TASK_B`, and proceed to run that task.



Return from the ISR.

```
    "                               \n"
    " bx r14                        \n" /* Branch indirect. This instruction will actually       */
    "                               \n" /* put r14 into the PC. r14 contains the EXC_RETURN      */
    "                               \n" /* value that is constructed at the entry of the        */
    "                               \n" /* exception. Loading it into the PC terminates it.     */
    "                               \n" /* The stack is automatically unwinded starting         */
    "                               \n" /* from PSP. The PSP points to the stack of Task B.     */
    "                               \n"
```

Finished!
Now the CPU resumes `TASK_B`. As you can see, the whole state of TASK_B is perfectly restored!



4.3 The selection of a new Task

In the previous paragraph we explained how the ISR routine `PendSV_Handler` implements the context switch procedure:

> "The context switch is performed by making the global variable `pxCurrentTCB` point to `TCB_B` instead of `TCB_A`. To make that switch, `PendSV_Handler` will call another subroutine: **`taskSwitchContext`**`()`."

So we can conclude that the context switch procedure is implemented in `PendSV_Handler`, but the selection of the new task – `TASK_B` – is done by the function **`taskSwitchContext`**`()`.

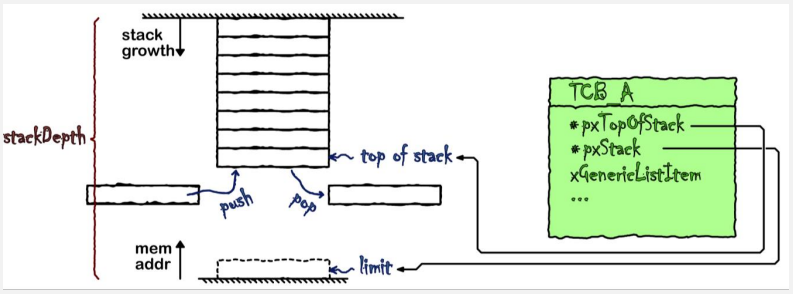
A slightly simplified version of this function is given below:

```
void taskSwitchContext( void )
{

    /* Check for stack overflow, if configured. */
    taskFIRST_CHECK_FOR_STACK_OVERFLOW();
    taskSECOND_CHECK_FOR_STACK_OVERFLOW();
```

```
    /* Select a new task to run using either the generic C or port
    optimised asm code. */
    taskSELECT_HIGHEST_PRIORITY_TASK();
}
```

The subroutine **`taskSwitchContext()`** uses a lot of macros. Most of them do not appear anywhere else in the Operating System. I believe that defining several layers of macros can sometimes make the code too complicated. Here you can find the **`taskSwitchContext()`** function with all macros expanded:

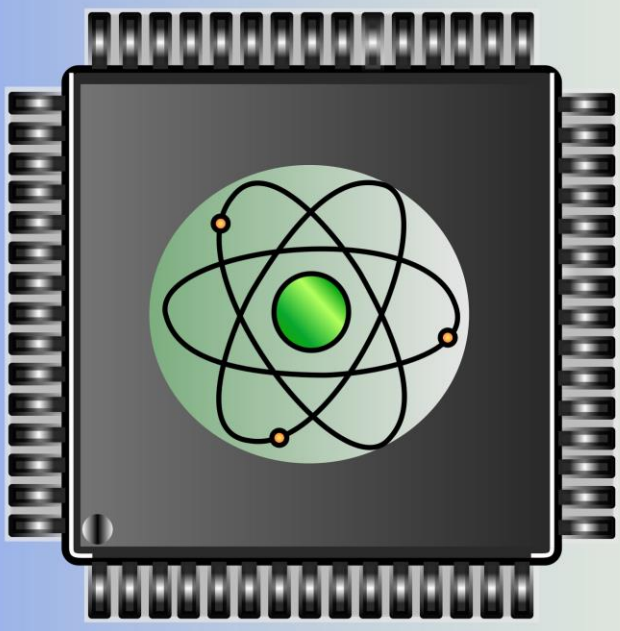
```
void taskSwitchContext( void )
{
    if( scheduler.isSuspended != ( uint32_t ) false )
    {
        /* The scheduler is currently suspended - do not allow a context switch. */
        scheduler.yieldPending = true;
    }
    else
    {
        scheduler.yieldPending = false;

        /*-----------------------------------------------------------------------------------------------*/
        /*                                Check for stack overflow.                                      */
        /*-----------------------------------------------------------------------------------------------*/
```




```
        if( p_currentTCB->p_topOfStack <= p_currentTCB->p_stack )
        {
            /* vApplicationStackOverflowHook( ( TaskHandle_t ) p_currentTCB, p_currentTCB->p_taskName ); */
        }


        static const uint8_t expStackBytes[] =
                    { tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE,
                      tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE,
                      tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE,
                      tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE,
                      tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE, tskSTACK_FILL_BYTE };

        /* Has the extremity of the task stack ever been written over? */
        if( memcmp( (void *) p_currentTCB->p_stack, (void *) expStackBytes, sizeof(expStackBytes) ) != 0 )
        {
            /*vApplicationStackOverflowHook( ( TaskHandle_t ) p_currentTCB, p_currentTCB->p_taskName );*/
        }

        /*-----------------------------------------------------------------------------------------------*/
        /*                                Find highest priority Task                                     */
        /*-----------------------------------------------------------------------------------------------*/
        /* We know that the position of the leftmost '1' bit in scheduler.topReadyPriority denotes       */
        /* the highest priority in the system.                                                           */

        uint32_t topPriority;
        topPriority = 31 - asm_countLeadingZeros( ( scheduler.topReadyPriority ) );


        assert( get_listLength( &( coreLists.readyTasks[ topPriority ] ) ) > 0 );


        /*-----------------------------------------------------------------------------------------------*/
        /*                        Make p currentTCB point to the right Task                              */
        /*-----------------------------------------------------------------------------------------------*/
        get_nextItem_owner( p_currentTCB, &( coreLists.readyTasks[ topPriority ] ) );

    }
}
```

Chapter 5

The FreeRTOS Kernel



5.1 The RTOS Kernel

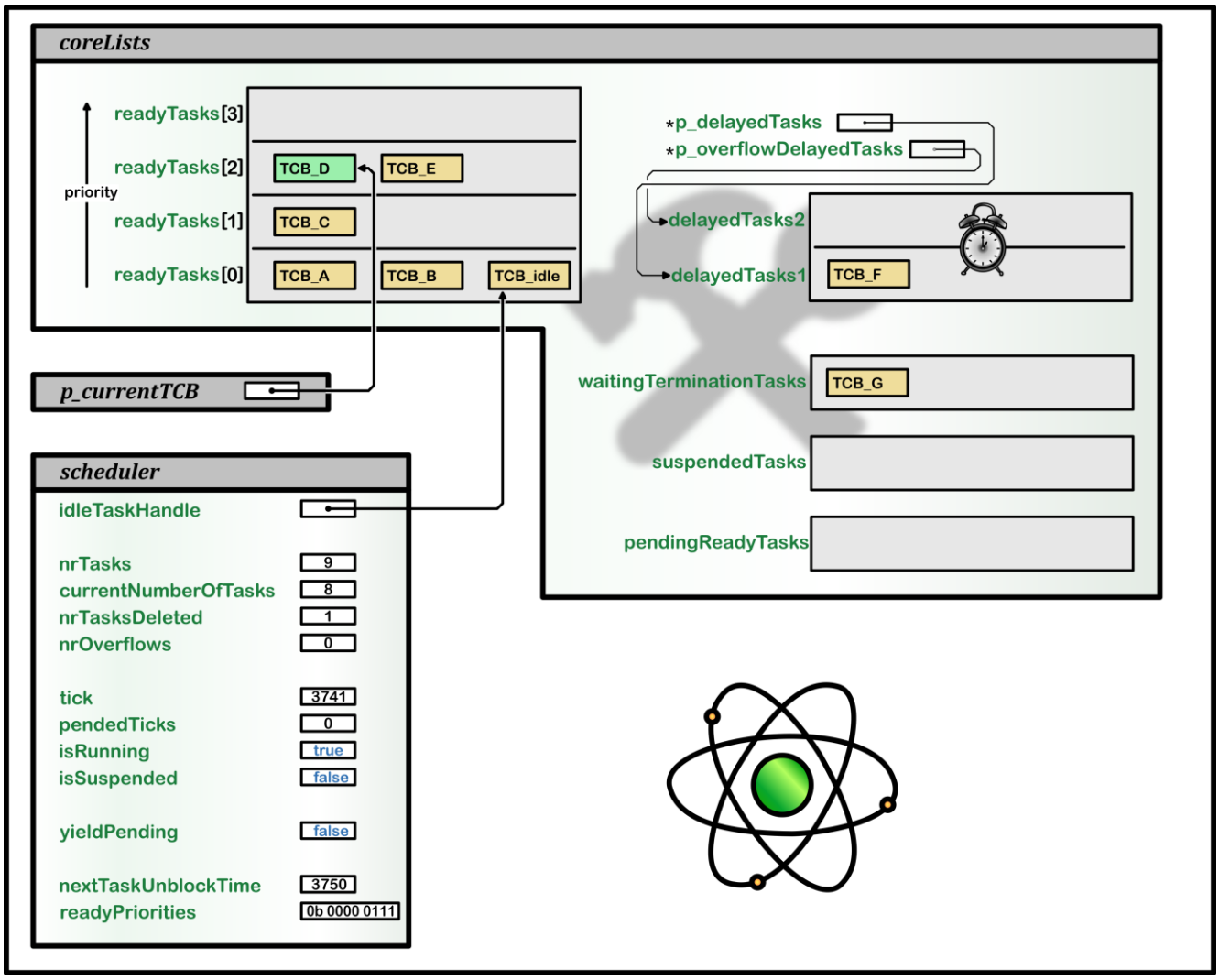
You could consider the whole FreeRTOS operating system as the kernel. But let us divide the operating system in a 'kernel' and some useful 'components'. Tasks, Lists and the Heap are all components. The kernel glues those components together in an intelligent way such that the whole system is your operating system.
This division between the kernel and its components really helped me to get a mental image of the whole operating system. But there is no division when you look at the C-code of FreeRTOS. Most of the kernel functionality can be found in the `Task.c` file and the `Port.c` file. I have cleaned up the code and put all the kernel functionality in `kernel.c`.

5.1.1 The Kernel status

I got also puzzeled when trying to figure out the kernel status variables. They are more or less scattered around. I've put them together in C-structs in the `kernel.h` file. So far I have created three structs containing kernel variables:

- ***coreLists***
- ***p_currentTCB*** `(is not a struct, but a global variable)`
- ***scheduler***

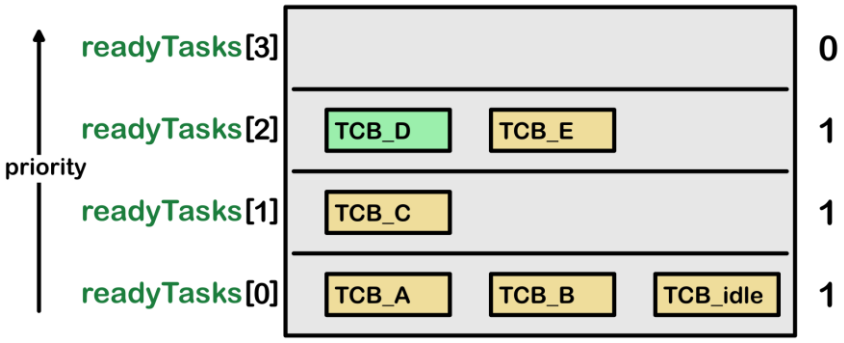
The ***coreLists*** struct contains all the lists of Tasks in the system. A few lists for Tasks that are in the ready state, a list for Tasks that are delayed, a list for Tasks that are waiting to get deleted, ...
The ***p_currentTCB*** variable points to the currently executing Task. This variable is global, since almost every part of the operating system should be able to know what Task is executing right here and right now.
The ***scheduler*** struct keeps track of the time. How many 'ticks' have passed? You can ask the scheduler. The scheduler also keeps an eye on how many Tasks are currently in the system, and what the top priority is.



```
/*-----------------------------------------------------------------------------------------------*/
/*                                                                                               */
/*                            | |    ( )     | |                                                 */
/*      / _|/ _ \| '_|/ -_)| |__ | |(_-<|  _|(_-<                                                 */
/*      \__|\___/|_|  \___||____||_|/__/ \__|/__/                                                */
/*-----------------------------------------------------------------------------------------------*/

typedef struct
{
    List_t readyTasks[ MAX_PRIORITIES ];  /*< Prioritised ready tasks. */

    List_t delayedTasks1;                 /*< Delayed tasks. */

    List_t delayedTasks2;                 /*< Delayed tasks (two lists are used - one for
                                              delays that have overflowed the current tick
                                              count. */

    List_t pendingReadyTasks;             /*< Tasks that have been readied while the scheduler
                                              was suspended.  They will be moved to the ready
                                              list when the scheduler is resumed. */

    List_t waitingTerminationTasks;       /*< Tasks that have been deleted - but their memory
                                              not yet freed. */

    List_t suspendedTasks;                /*< Tasks that are currently suspended. */


    List_t * volatile p_delayedTasks;     /*< Points to the delayed task list currently
                                              being used. */

    List_t * volatile p_overflowDelayedTasks;    /*< Points to the delayed task list currently
                                                     being used to hold tasks that have
                                                     overflowed the current tick count. */
} coreLists_t;

extern coreLists_t coreLists;



/*-----------------------------------------------------------------------------------------------*/
/*                                                                                               */
/*      /   |    | |           | |      | |                                                       */
/*      \__ \/ _|| ' \ / -_)/ _` || || || |/ -_)| '_|                                            */
/*      |___/\__||_||_|\___|\__,_| \_,_||_|\___||_|                                              */
/*-----------------------------------------------------------------------------------------------*/

/************************************************/
/*      POINTER TO THE CURRENT TASK             */
/************************************************/
/*                                              */
     extern TCB_t * volatile p_currentTCB;
/*                                              */
/************************************************/


#define tskIDLE_PRIORITY            ( (uint32_t)0U )

typedef struct
{
    TaskHandle_t idleTaskHandle;

    volatile uint32_t currentNumberOfTasks;

    uint32_t nrTasksDeleted;

    volatile uint32_t tick;

    volatile uint32_t readyPriorities;  /*< The position of the leftmost '1' bit tells you
                                            the highest priority level. */
    volatile int32_t isRunning;

    volatile uint32_t pendedTicks;

    volatile int32_t yieldPending;
```

```
    volatile int32_t nrOverflows;
    uint32_t nrTasks;
    volatile uint32_t nextTaskUnblockTime;
    volatile uint32_t isSuspended;
} scheduler t;
extern scheduler_t scheduler;
```

Most Kernel variables are quite straightforward. Their names reveal what they are built for. There is one variable that requires some extra attention. The **`scheduler.readyPriorities`** variable keeps track of all the priority levels in the system that are occupied by Tasks. Imagine the following situation:



The **`scheduler.readyPriorities`** variable has a '1' bit for every non-empty **`readyTasks`** list. A '0' bit corresponds to a priority level that has not yet been occupied by any Task. So for our example we can figure out that:

scheduler.readyPriorities = 0b 0000 0111

This variable is very convenient and efficient. Imagine that the Kernel needs to know what the highest occupied priority level in the system is. It could plough through all the **`readyTasks`** lists and check which one is empty and which one is not. This would return the correct result but requires quite some processing time. Instead the Kernel can take a quick look at the **`scheduler.readyPriorities`** variable. All it needs to know is the position of the leading '1' bit. There is an simple instruction in the ARM instruction set to do that! So it can be done extremely fast.

5.1.2 The 'invisible' lists

Every Task resides in one of the lists. Most Tasks live in the ***coreLists***. The Kernel has direct access to them. But some Tasks move to an ***eventList***. An eventList belongs to a queue, mutex, semaphore or other object. We will cover those objects later on. But let us take the queue as an example. A queue object gets created when Task_A wants to send information to Task_B. Task_B registers itself as a 'listener' to the queue. So the queue puts Task_B into its receive-***eventList***. When Task_A writes an item to the queue, it can wake up Task_B to take the item and process it.
Hopefully this short introduction makes clear that ***eventLists*** have a one-on-one relationship with their queue (or other object like semaphore, mutex, ...). The creation of a queue object results in the immediate creation of a receive-***eventList*** and a send-***eventList***.

Maybe now it gets clear why every Task has two `listItems`. The **`genericListItem`** is used to 'sit' in one of the ***coreLists***. The **`eventListItem`** is used to 'sit' in one of the ***eventLists***.

5.1.3 How Tasks are ordered in the lists

Let us first focus on the **`genericListItem`**. Upon creation of a new Task – say Task_A – this item does not get a value! So how does the Operating System know where to put Task_A? Shouldn't the position correspond to the priority of the Task?
The answer to this question is simple. Task_A ends up in one of the ready Lists from the Kernel. The priority dictates in which of the seven[8] ready Lists Task_A gets stored. The position within the ready List itself does not matter, since all other Tasks in that list have equal priority.
Once Task_A gets moved to another list – for example the `delayedTasks` list – the situation changes. The **`genericListItem`** value is set to `timeToWake`. Task_A will end up somewhere at the tail of the list if that value is big. Remember that FreeRTOS linked lists are assembled in ascending order.

The story of an **`eventListItem`** is different. It gets a value immediately when the Task is created. The value is the inverse of the Tasks priority. A high priority Task will end up at the head of an ***eventList*** (and vice versa). That is exactly what we want. If a queue wakes up one of its listeners, it wants to access the highest priority one first.

[8] By default there are seven priority levels for Tasks in FreeRTOS: levels 0 to 6.

5.2 The RTOS heartbeat

The heartbeat of the RTOS is its system clock. It is a 24-bit timer that generates an interrupt every millisecond (depending on its configuration). The previous chapter explained that the timer interrupt can invoke a Context Switch between Tasks. In this chapter we will dive into the configuration settings of the timer. How to configure the heartbeat of your system.

5.1.1 Configuration of the heartbeat

The first prerequisite to setting up the `SysTick` timer is getting knowledge about the CPU clock frequency. Therefore we need to take a look in the `system_stm32f7xx.c` file located in the CMSIS folder. This file also contains the `SystemInit()` function. Remember that this function gets executed at the very beginning in the startup sequence – before the `main()` routine is called. It also modifies the clock settings, so it is logic that the CPU clock frequency settings can be found in the same file:

system_stm32f7xx.c

```
uint32_t SystemCoreClock = 16000000;
```

The variable `SystemCoreClock` is indeed 16 Mhz at startup. It quickly changes to a much higher value, up to 216 Mhz maximum when the initialization code gets executed. But do not worry. This variable gets updated accordingly.
Now that we have a variable containing the CPU frequency (in Hz), we can proceed to configuring the `SysTick` timer. The startup code for the FreeRTOS scheduler calls the function **`vPortSetupTimerInterrupt()`**. Read carefully the code below:

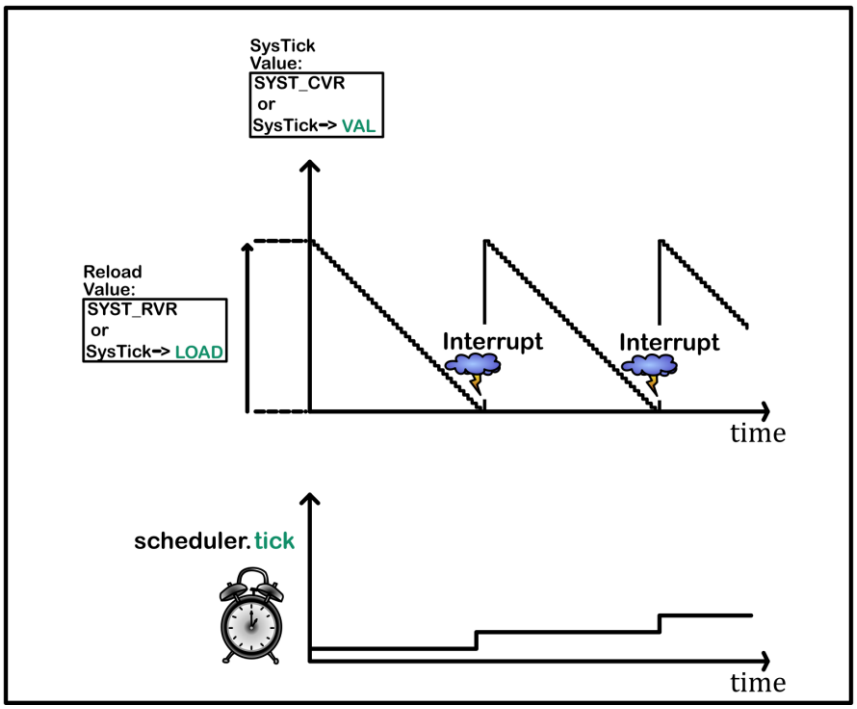
FreeRTOSConfig.h

```
#define configTICK_RATE_HZ                      ((uint32_t)1000)
```

port.c

```
int32_t xPortStartScheduler( void )
{
    ...
    vPortSetupTimerInterrupt();
    ...
    return 0;
}


/* SYSTICK CONTROL REGISTER                                          */
/* Datasheet Core-M7: SYST CSR register is at address 0xE000E010     */
#define portNVIC_SYSTICK_CTRL_REG         ( * ( ( volatile uint32_t * ) 0xe000e010 ) )
#define portNVIC_SYSTICK_ENABLE_BIT       ( 1UL << 0UL )
#define portNVIC_SYSTICK_INT_BIT          ( 1UL << 1UL )
#define portNVIC_SYSTICK_CLK_BIT          ( 1UL << 2UL )

/* SYSTICK RELOAD VALUE REGISTER                                     */
/* Datasheet Core-M7: SYST RVR register is at address 0xE000E014     */
#define portNVIC_SYSTICK_LOAD_REG         ( * ( ( volatile uint32_t * ) 0xe000e014 ) )


void vPortSetupTimerInterrupt( void )
{

    /* Configure SysTick to interrupt at the requested rate. */
    portNVIC_SYSTICK_LOAD_REG = ( SystemCoreClock / configTICK_RATE_HZ ) - 1UL;
    portNVIC_SYSTICK_CTRL_REG = ( portNVIC_SYSTICK_CLK_BIT | portNVIC_SYSTICK_INT_BIT |
                                                    portNVIC_SYSTICK_ENABLE_BIT );
}
```

The `Reload Value` from the `SysTick` timer is now configured correctly such that the timer will generate an interrupt every millisecond. Adjust the **`#define configTICK_RATE_HZ`** statement in the `FreeRTOSConfig.h` file to change the interrupt frequency of the timer.



The function **`vPortSetupTimerInterrupt()`** could be written differently:

```
void vPortSetupTimerInterrupt( void )
{
    SysTick->LOADbits.RELOAD = ( SystemCoreClock / configTICK_RATE_HZ ) - 1UL;
    SysTick->CTRLbits.CLKSOURCE = 1;
    SysTick->CTRLbits.TICKINT = 1;
    SysTick->CTRLbits.ENABLE = 1;
}
```

I believe this programming style is much more intuitive. Although it requires some adaptations in the `core_cm7.h` file to make the bitfields from the registers accessible.

5.3 The Context Switch decision

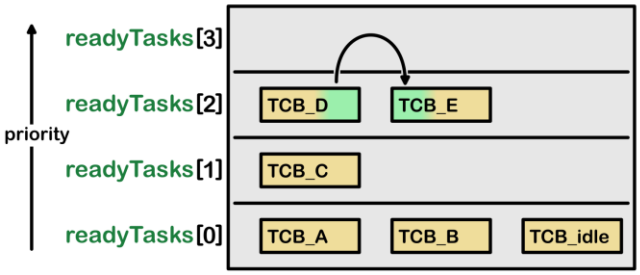
Let us take a look at the code in the interrupt routine of the `SysTick` timer. We already studied how the Context Switch from Task A to Task B takes place in the `invokePendSV()` function. Even the procedure of finding Task B (among the other ready Tasks) was covered. But we did not yet investigate how the kernel decides that a Context Switch should happen. This paragraph is entirely devoted to that decision making process. How does the Kernel decide that Task A should stop and another one should run?

```
void SysTick_Handler( void )
{
    // Moved from port.c to kernel.c

    (void) SET_INTERRUPT_MASK_FROM_ISR();
    {
        if( increment_Tick() != false )
        {
            invokePendSV();
        }
    }
    CLEAR_INTERRUPT_MASK_FROM_ISR( 0 );
}
```

The decision is made in the **`increment_Tick()`** subroutine. This subroutine is rather big and complicated. But let us first examine a simplified version of it. Little by little we will extend the function until we have the full version.

```
/*-----------------------------------------------------------------------------------------*/
/*                        int32_t increment_Tick(void)                                     */
/*-----------------------------------------------------------------------------------------*/
/* This function is called on every SysTick interrupt. Here the decision is taken if a     */
/* Context Switch should happen.                                                           */
/*                                                                                         */

int32_t increment_Tick( void )
{
    int32_t switch_required = false;


    ++scheduler.tick;


    /* Tasks of equal priority to the currently running task will share  */
    /* processing time (time slices)                                     */

    if( get_listLength( &( coreLists.readyTasks[p_currentTCB->priority] ) ) > 1 )
    {
        switch_required = true;
    }

    return switch_required;
}
```

The code simply looks at the proper ***`coreLists`***`.readyTasks[`***`n`***`]` list to find out if other Tasks are ready to run. Imagine that Task D was running. Task D has priority level 2. So the code investigates the `readyTasks[`***`2`***`]` list to find out if there are other Tasks with the same priority. Task E is found, and the return variable `switch_required` is set.



Why doesn't the code investigate the lists `readyTasks[`***`1`***`]` and `readyTasks[`***`0`***`]`? The answer is simple. The fact that Task D is running at priority level 2 implies that all Tasks of lower priority levels

are not entitled to run. They simply have to wait until the **readyTasks[2]** list – and all other lists above that level – are empty.
The answer to the next question requires a bit more thought. Why doesn't the code investigate the higher priority lists **readyTasks[3], readyTasks[4], ..** ? After all, a Task sitting in one of those lists has a higher priority and should run!
The answer is that we can be confident that all those lists are empty. There is no need to check them. Anytime that a process adds a new Task to one of the ready lists, it has the responsibility to take the following actions:

- Check if the priority of the new Task is higher than the priority of the running Task.
- If that is the case, force a context switch immediately. Or set the variable ***scheduler*.yieldPending** to **true**. That will force a context switch on the next tick interrupt.

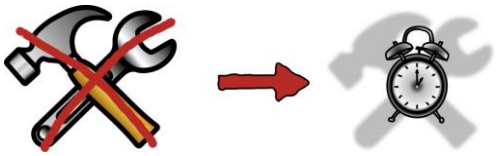
So if these rules are consistently obeyed, there is no need to check once more if any of the higher priority lists contains a Task that should run. All we need to do now is add a simple line of code that checks if a context switch is forced through the ***scheduler*.yieldPending** variable.

```
int32_t increment_Tick( void )
{
    ...

    if( scheduler.yieldPending != false )
    {
        switch_required = true;
    }

    return switch_required;
}
```

The context switch decision in the **increment_Tick**(void)subroutine is very simple. Things get a bit more complicated when we add extra functionalities. Like Tasks that are delayed or suspended.

5.4 Delayed Tasks

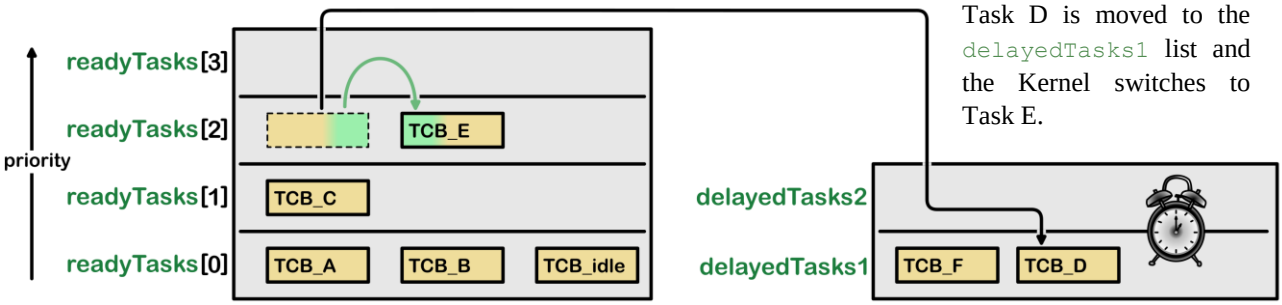
Sometimes you want to delay a Task for a few milliseconds. This can entitle other Tasks to more CPU power for a while. FreeRTOS provides an easy way to do this. Just call the **`taskDelay(..)`** function and tell it how many 'ticks' you want the currently running Task to get delayed. Check out the code of the function.



5.4.1 Put a Task to sleep (delay)

```
void taskDelay( const uint32_t ticksToDelay )
{
    uint32 t timeToWake;
    int32_t alreadyYielded = false;

    if( ticksToDelay == (uint32_t)0U )
    {
        /* A delay time of zero just forces a reschedule. */
        invokePendSV();
        return;
    }


    assert( scheduler.isSuspended == 0 );
    vTaskSuspendAll();
    {
        /* Calculate the time to wake - this may overflow but this is   */
        /* not a problem.                                               */
        timeToWake = scheduler.tick + ticksToDelay;

        /* This task cannot be in an event list as it is the currently  */
        /* executing task.                                              */
        /* We must remove ourselves from the ready list before adding   */
        /* ourselves to the blocked list as the same list item is used  */
        /* for both lists.                                              */
        /*                                                              */
        if( listRemove( &( p_currentTCB->genericListItem ) ) == ( uint32_t ) 0 )
        {
            reset_readyPriorities( p_currentTCB->priority, scheduler.readyPriorities );
        }

        prvAddCurrentTaskToDelayedList( timeToWake );
    }
    alreadyYielded = xTaskResumeAll();


    /* Force a reschedule if xTaskResumeAll has not already done so. We may */
    /* have put ourselves to sleep.                                         */
    if( alreadyYielded == false )
    {
        invokePendSV();
    }
}
```

The following figure shows what happens if Task D (the running task) gets delayed:

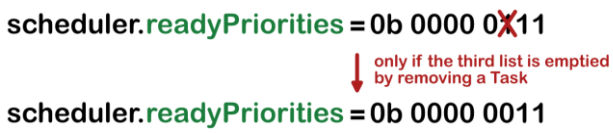


Task D is moved to the `delayedTasks1` list and the Kernel switches to Task E.

The **`taskDelay(..)`** function removes the Task from the ready list and puts it inside the delayed Tasks list. But where exactly in that list will the Task end up? The position depends on the Tasks '`timeToWake`' variable. It holds the absolute tick count at which the Task should wake up. The lower this value, the closer the Task is to the beginning of the list. If the Operating System wants to know which Task will wake up next, it does not need to process all the entries in the list. It should only look at the first element, since that one will wake up first.
The code for the subroutine **`prvAddCurrentTaskToDelayedList(..)`** is given below:

```
static void prvAddCurrentTaskToDelayedList( const uint32_t timeToWake )
{
    set_listItem_value( &( p_currentTCB->genericListItem ), timeToWake );

    if( timeToWake < scheduler.tick )
    {
        /* Wake time has overflowed.  Place this item in the overflow list.         */
        listInsert( coreLists.p_overflowDelayedTasks, &( p_currentTCB->genericListItem ) );
    }
    else
    {
        /* The wake time has not overflowed, so the current list is used.           */
        listInsert( coreLists.p_delayedTasks, &( p_currentTCB->genericListItem ) );


        /* If the task entering the blocked state was placed at the head of the     */
        /* list of blocked tasks then scheduler.nextTaskUnblockTime needs to be     */
        /* updated too.                                                             */
        /*                                                                          */
        if( timeToWake < scheduler.nextTaskUnblockTime )
        {
            scheduler.nextTaskUnblockTime = timeToWake;
        }
    }
}
```

The Task should stop running as soon as it gets transferred to the delayed Tasks list. So we need to force an immediate context switch. The context switch requires no extra thought if the ready Tasks list still holds some Tasks of the same priority.
But if the list is emptied by removal of the currently running Task, the **`scheduler.readyPriorities`** bits should be updated! You will mess up the Kernel state if you forget this. The Kernel will think that there are still some ready Tasks available at that priority level, whereas in fact there is none.

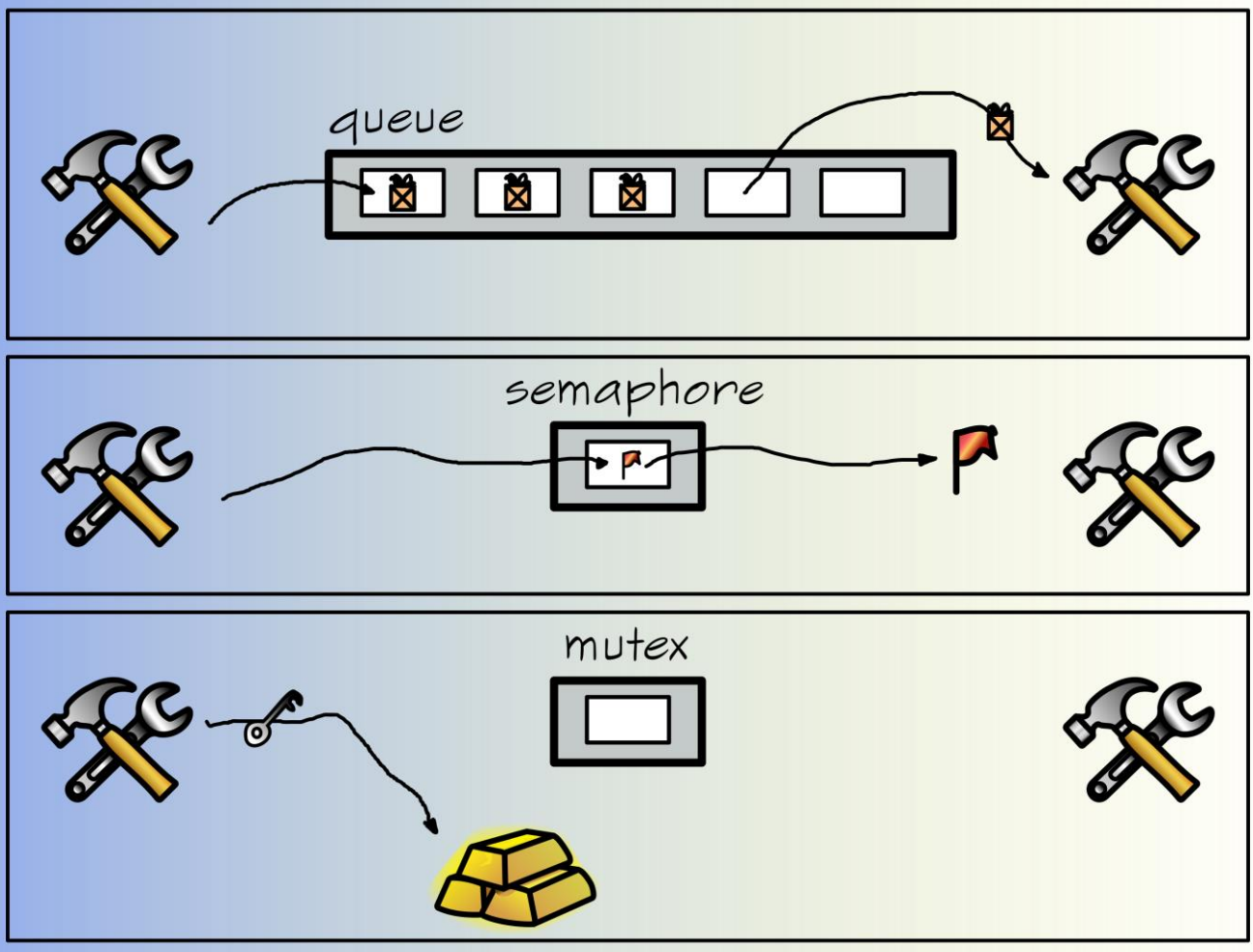


5.4.2 The Task wakes up!

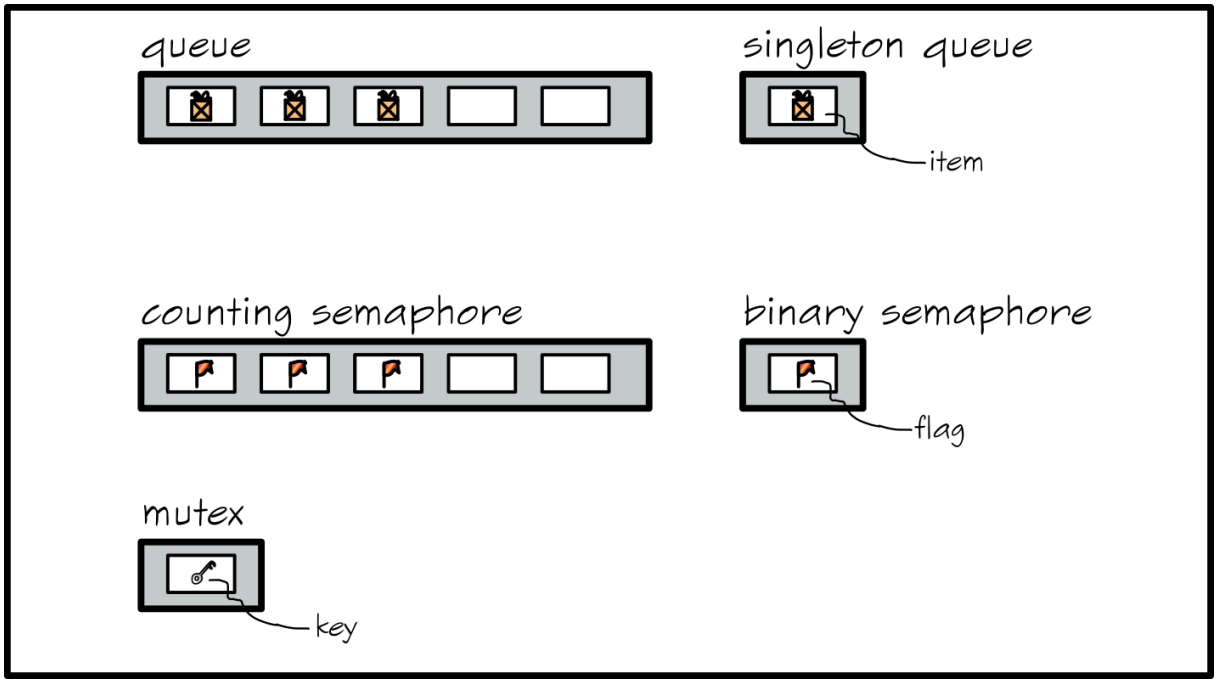
UNDER CONSTRUCTION

Chapter 6

Queues, semaphores, mutexes and all that



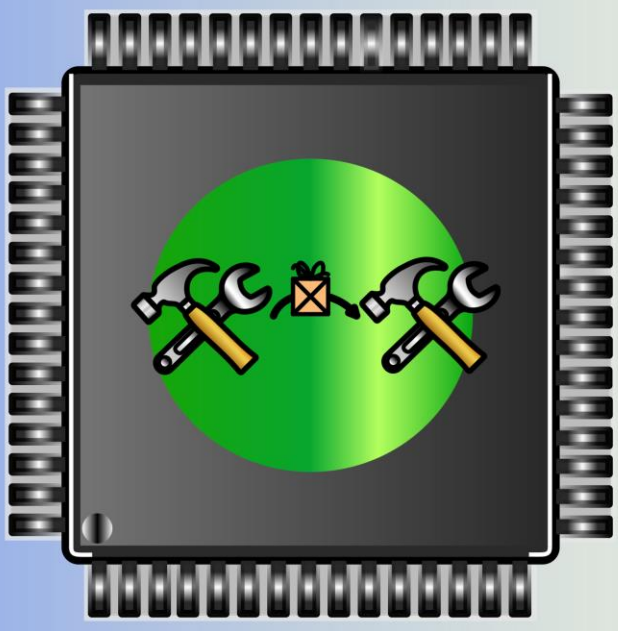
6.1 Queues



CHAPTER UNDER CONSTRUCTION

Chapter 7

Passing information through queues

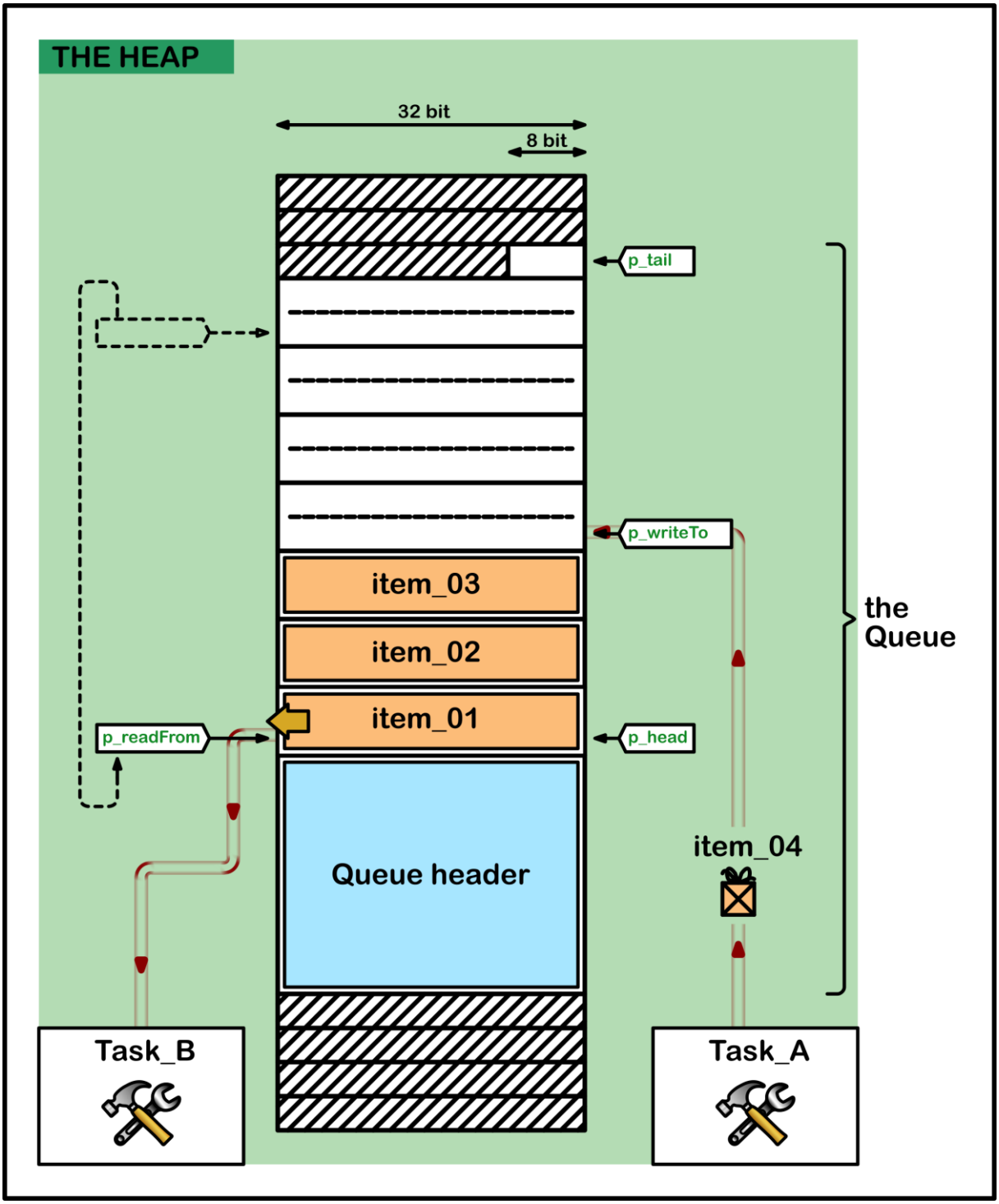


7.1 Passing information through a queue

Queues in FreeRTOS are a mechanism to transfer data from one Task to another. A queue is a separate object. It does not belong to a specific Task. A queue object can therefore be used by any Task in your system to send or receive data.

A queue is an object in its own right. It is not owned by, nor assigned to any particular Task. Any Task can write to a given queue, and any Task can receive items from it.

The following figure illustrates how Task_A can send items to Task_B. The items in this figure are 64-bit in size. The item size is fixed. You determine the size when creating a new queue object, and it remains constant throughout its lifetime.



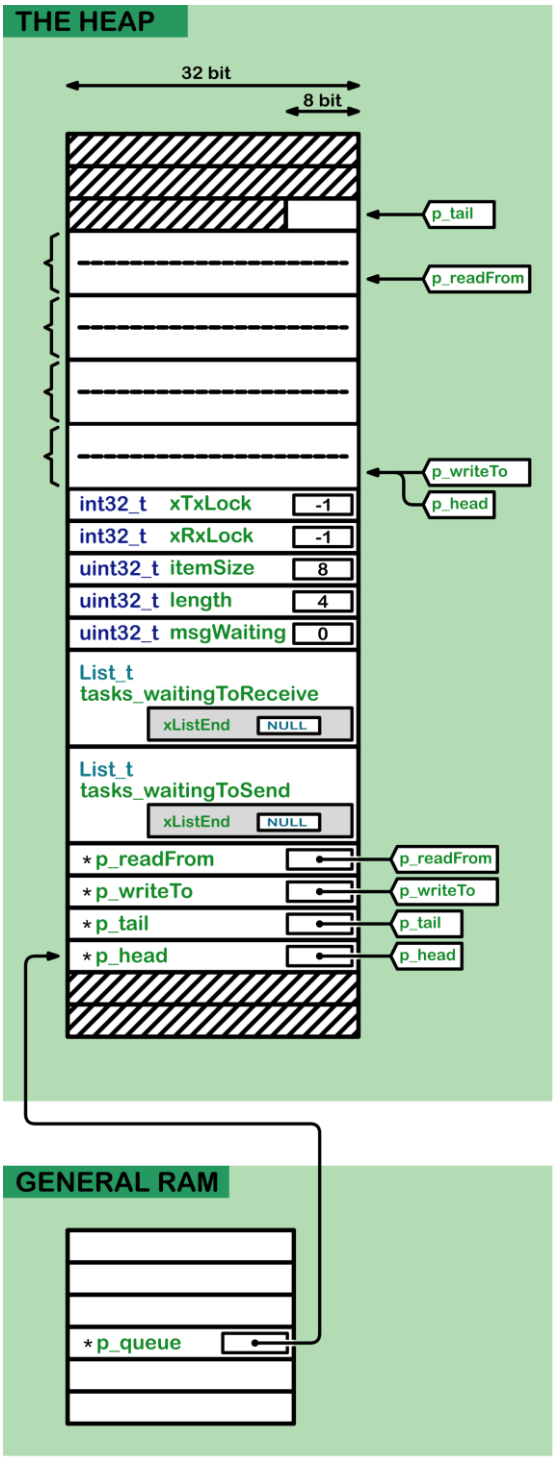
7.2 Creation of a Queue

The following code creates a new Queue object and returns its handle. This code is simplified to keep your focus on the Queue object creation. A lot of checks and special case treatments are omitted.

```
QueueHandle_t queueGenericCreate( const uint32_t queueLength, const uint32_t itemSize,
{                                                                const uint8_t queueType )
    Queue_t *p_newQueue;
    uint32_t queueSizeInBytes;
    QueueHandle_t queueHandle = NULL;
    int8_t *p_allocatedBuf;


    queueSizeInBytes = ( uint32_t ) ( queueLength * itemSize ) + 1;
    p_allocatedBuf = ( int8_t * ) pvPortMalloc( sizeof( Queue_t ) + queueSizeInBytes );

    p_newQueue = ( Queue_t * ) p_allocatedBuf;
    p_newQueue->p_head = p_allocatedBuf + sizeof( Queue_t );
    p_newQueue->length = queueLength;
    p_newQueue->itemSize = itemSize;
    queueGenericReset( p_newQueue, true );

    queueHandle = p_newQueue;

    assert( queueHandle );

    return queueHandle;
}
```

```
void queueGenericReset( QueueHandle_t queueHandle, bool isNew )
{
    Queue_t * const p_queue = ( Queue_t * ) queueHandle;

    ENTER_CRITICAL();
    {
        p_queue->p_tail = p_queue->p_head + ( p_queue->length * p_queue->itemSize );
        p_queue->messagesWaiting = 0;
        p_queue->p_writeTo = p_queue->p_head;
        p_queue->p_readFrom = p_queue->p_head + ((p_queue->length - 1) * p_queue->itemSize);
        p_queue->xRxLock = queueUNLOCKED;
        p_queue->xTxLock = queueUNLOCKED;

        if( isNew )
        {
            vListInitialise( &( p_queue->tasks_waitingToSend ) );
            vListInitialise( &( p_queue->tasks_waitingToReceive ) );
        }
        else
        {
            /* If there are tasks blocked waiting to read from the queue, then           */
            /* the tasks will remain blocked as after this function exits the queue      */
            /* will still be empty.                                                      */
            /* If there are tasks blocked waiting to write to the queue, then one of     */
            /* them should get unblocked, as after this function exits it will be        */
            /* possible to write to the empty queue                                      */
            if( is_empty( &( p_queue->tasks_waitingToSend ) ) == false )
            {
                if( xTaskRemoveFromEventList( &( p_queue->tasks_waitingToSend ) ) == true )
                {
                    // The unblocked Task has higher priority than the current one.
                    invokePendSV();
                }
            }
        }

    }
    EXIT_CRITICAL();
}
```

Let us analyse the code with an example. Imagine you create a Queue object that can hold 4 items at most. Each item is 64 bits in size (two 32-bit words or eight bytes). You pass these specifications to the creation function and a Queue like the figure on the left is returned.



The Queue header is based on the following struct definition[9]:

```
typedef struct QueueDefinition
{
    int8_t *p_head;
    int8_t *p_tail;
    int8_t *p_writeTo;
    int8_t *p_readFrom;

    List_t tasks_waitingToSend;
    List_t tasks_waitingToReceive;

    uint32_t messagesWaiting;
    uint32_t length;
    uint32_t itemSize;
    int32_t xRxLock;
    int32_t xTxLock;

} Queue_t;
```

At first this might seem a bit misleading. The name of the struct is `Queue_t`, so you might think this struct contains everything – the header and data. But in fact it only contains the header. Luckily retreiving the data is a piece of cake. You know that the data is positioned right after the header. Moreover, the header indicates the start and end of the data block, as well as the size of each item in it.

The queue handle is a mere pointer to the start of the queue (which is actually the start of the header).

```
typedef void * QueueHandle_t;
```

The whole queue is located in RAM memory allocated from the heap.

[9] I have omitted some unions and volatile keywords to improve the readability of this text. Refer to the actual code of FreeRTOS for a complete implementation.

7.3 Sending to and receiving from a Queue

7.3.1 Copying data

FreeRTOS provides a function to store an item in the Queue, and a function to take an item. A simplified version of the code is given below:

```
static bool copyData_toQueue( Queue_t * const p_queue, const void *p_newItem )
{
    bool mutex_yieldRequired = false; // Used if queue is also a mutex.
    ...

    (void) memcpy( (void *) p_queue->p_writeTo, p_newItem, (uint32_t) p_queue->itemSize );
    p_queue->p_writeTo += p_queue->itemSize;

    if( p_queue->p_writeTo >= p_queue->p_tail )
    {
        p_queue->p_writeTo = p_queue->p_head;
    }

    ++( p_queue->messagesWaiting );

    return mutex_yieldRequired;
}
```

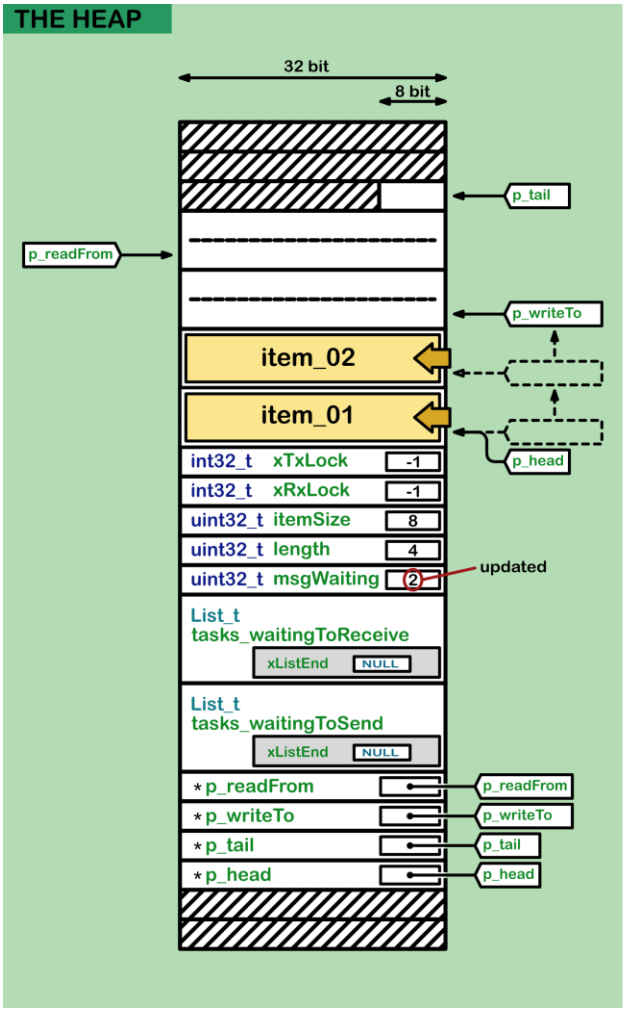
```
static void copyData_fromQueue( Queue_t * const p_queue, void * const p_buf )
{
    p_queue->p_readFrom += p_queue->itemSize;

    if( p_queue->p_readFrom >= p_queue->p_tail )
    {
        p_queue->p_readFrom = p_queue->p_head;
    }

    (void) memcpy( (void *) p_buf,(void *) p_queue->p_readFrom,(uint32_t) p_queue->itemSize );

}
```

These functions are quite straightforward. The first one copies an item into the queue. After that the `p_writeTo` pointer increases and wraps around if necessary. The other function takes an item from the queue and copies it into a given buffer. Eventually the pointer `p_readFrom` increases.
Note that all items stored to and taken from the queue are *copied*. This approach is very important to remember:

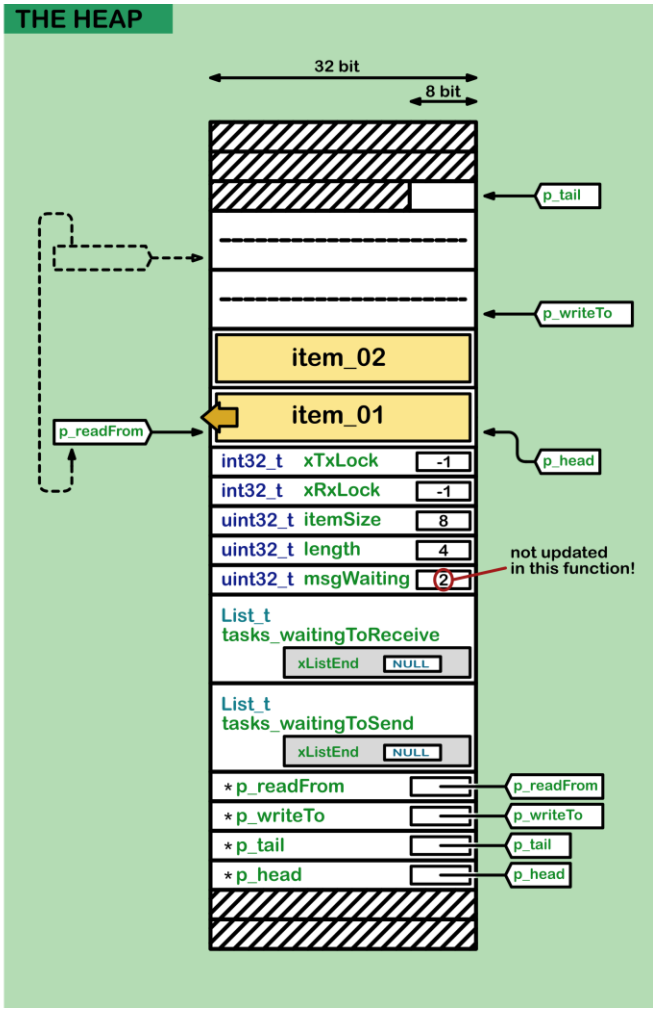
All items stored into a queue are *copied*. A copy is taken from the item you pass as a parameter.
All items read from the queue are *copied* as well. They get copied into the buffer you provide.

The next page shows a figure that illustrates very well the actions of **`copyData_toQueue(..)`** and **`copyData_fromQueue(..)`**.

Copy two items to the queue. The pointer `p_writeTo` increments twice. The `msgWaiting` field increments as well.



Read an item from the queue. The pointer `p_readFrom` increments *before* the read operation. The `msgWaiting` field is not updated.



These copy-functions do not check if the queue is full. They don't check if Tasks are on the waiting list, such that one of them might need to wake up when the queue gets a new element. The copy-functions are rather 'dumb copiers' and should not be used directly by your Tasks!

What you need are the functions **`queueGenericSend(..)`** and **`queueGenericReceive(..)`**. They do all the necessary checking and can even wake up Tasks that are on the waiting lists. The copy-functions are used within these `genericSend` and `genericReceive` functions. But before we will study them, we shall first take a look at the event lists.

7.3.2 The event lists

A newly created queue has two empty event lists. One can hold Tasks that are waiting to receive data. The other holds Tasks that are waiting to send data to the queue. The queue does not use its event lists in an ideal situation. Imagine that Task_A wants to send an item to the queue. The queue is not full and accepts the item immediately. There is no reason to put Task_A on its waiting list. Task_B requests data from the queue. The queue is not empty, so it can serve Task_B right away. Again there would be no valid reason to put Task_B on a waiting list.
The event lists get in the picture when the queue cannot serve Task_A or Task_B immediately. The queue has an event list for Tasks that are waiting to receive items, and one for those that are waiting to send:

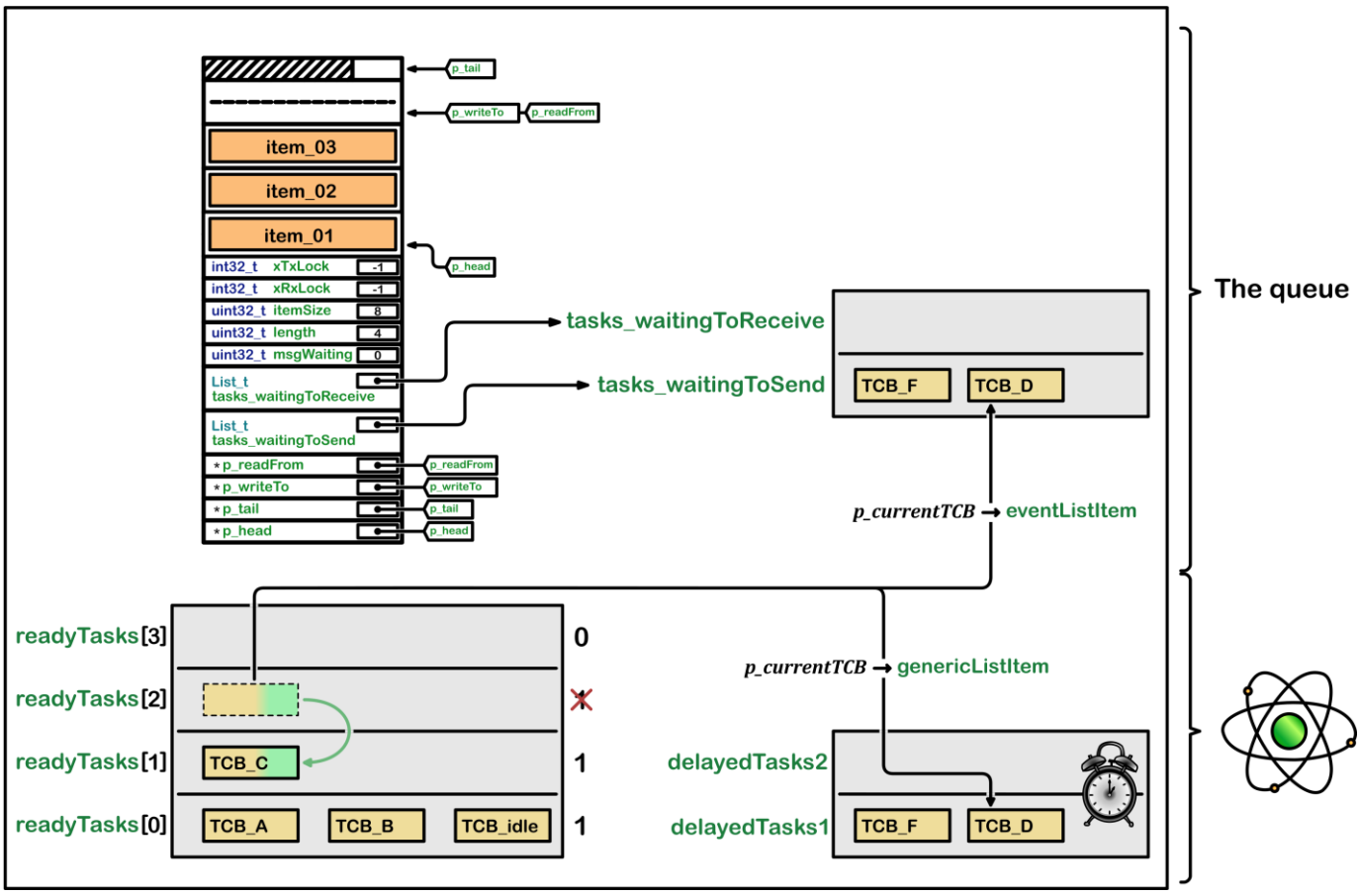
```
List_t tasks_waitingToReceive
List_t tasks_waitingToSend
```

We will now examine the code how a Task is placed on such event lists.

```
void placeTask_onEventList( List_t * const p_eventList, const uint32_t ticksToWait )
{
    /*************************************************************************/
    /* THIS FUNCTION MUST BE CALLED WITH EITHER INTERRUPTS DISABLED OR THE   */
    /* SCHEDULER SUSPENDED AND THE QUEUE BEING ACCESSED LOCKED.              */
    /* A locked queue prevents simultaneous access from interrupts.          */
    /* A locked scheduler guarantees exclusive access to the ready lists.    */
    /*************************************************************************/
    assert( p_eventList );


    listInsert( p_eventList, &( p_currentTCB->eventListItem ) );

    if( listRemove( &( p_currentTCB->genericListItem ) ) == ( uint32_t ) 0 )
    {
        // Clear the '1' bit that corresponds to the emptied ready list.
        reset_readyPriorities( p_currentTCB->priority, scheduler.readyPriorities );
    }

    {
        if( ticksToWait == MAX_UINT32 )
        {
            /*-----------------------------------------*/
            /* Add the task to the suspended task list  */
            /*-----------------------------------------*/
            listInsertEnd( &coreLists.suspendedTasks, &( p_currentTCB->genericListItem ) );
        }
        else
        {
            /*-----------------------------------------*/
            /* Add the Task to the delayed list         */
            /*-----------------------------------------*/
            addCurrentTask_toDelayedList( scheduler.tick + ticksToWait );
        }
    }
}
```

The code removes the current Task from the ready lists. The Task ends up in two lists: the **`delayedTasks`** list (belongs to the Kernel) and the **`tasks_waitingToSend`** list[10] (belongs to the queue).
Remember that every Task has two '`ListItems`' such that it can live in two lists simultaneously. Notice that there is no immediate context switch forced by the **`placeTask_onEventList(..)`** function itself. Nevertheless a context switch is crucial since the code removes the currently running Task from the ready lists! The calling function has the responsibility to request a context switch.

[10] Or the tasks_waitingToReceive list – depending on the intentions of the Task.



Position in the delayedTasks list

The position of the Task in the `delayedTasks` list depends on the timeout. The time that it will wake up in case the queue keeps being inaccessible. The sooner it times out, the closer it will be to the head of the `delayedTasks` list. This is why the addCurrentTask_toDelayedList(..) function begins with the following code line:

```
set_listItemValue( &( p_currentTCB->genericListItem ), timeToWake );
```

Position in the tasks_waitingToSend list

The `eventListItem` has a value related to the Task its priority level. The higher the priority, the lower this value. The value is fixed during the initialization of the Task.

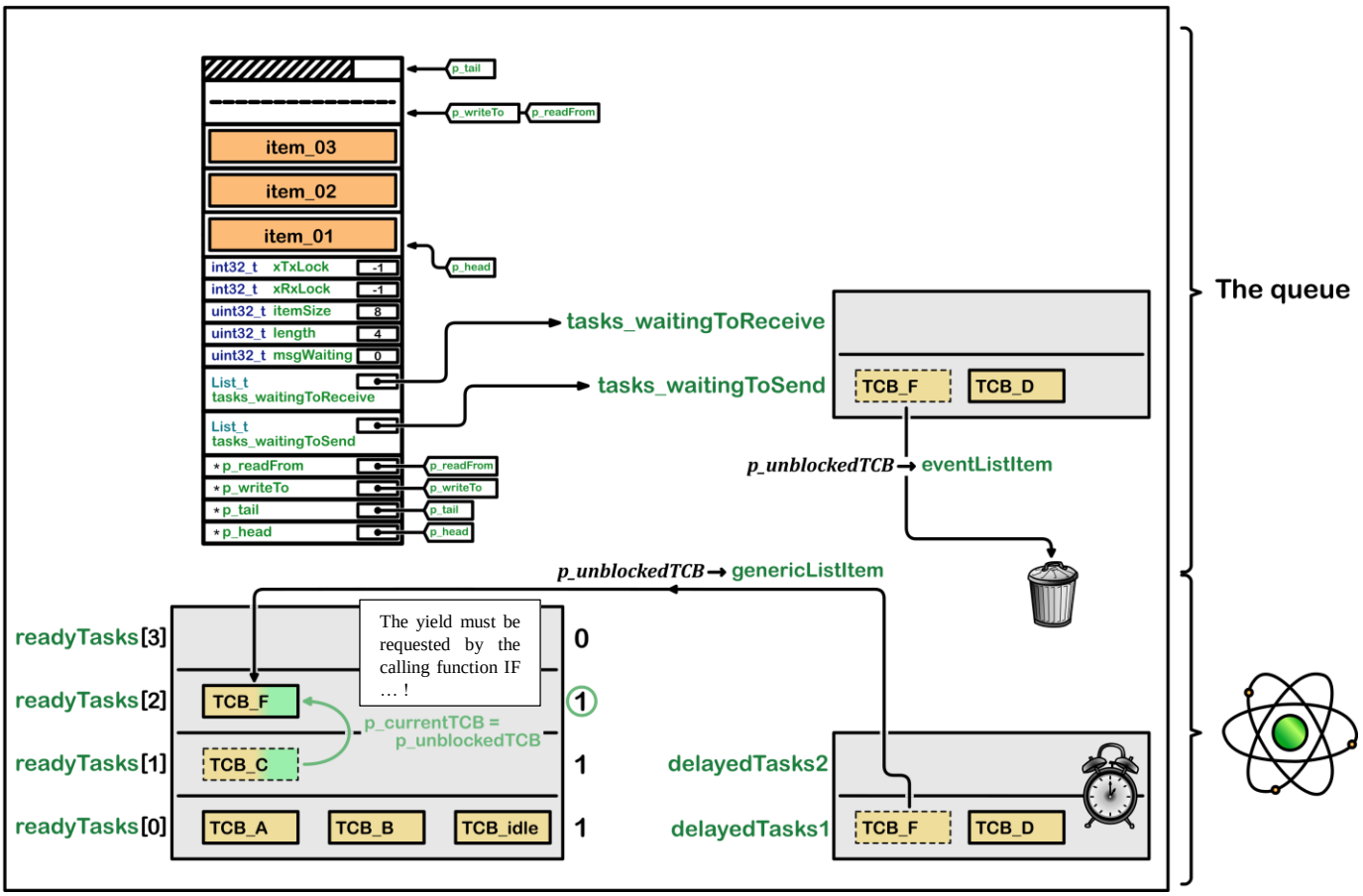
As you know, lists get ordered in ascending value order. So the queue can be sure that at any given time the first Task in its `tasks_waitingToSend` list has the highest priority.

```
set_listItemValue( &( p_TCB->eventListItem ), MAX_PRIORITIES - taskPriority );
```

Let us now examine the code that removes a Task from an event list:

```
bool removeTask_fromEventList( const List_t * const p_eventList )
{
    /*************************************************************************/
    /* THIS FUNCTION MUST BE CALLED FROM A CRITICAL SECTION.                 */
    /* IT CAN ALSO BE CALLED FROM A CRITICAL SECTION WITHIN AN ISR.          */
    /* If an event is for a queue that is locked then this function will     */
    /* never get called - the lock count on the queue will get modified      */
    /* instead. This means exclusive access to the event list is guaranteed */
    /* here.                                                                 */
    /*************************************************************************/
    TCB_t *p_unblockedTCB;
    bool yieldRequired;
    p_unblockedTCB = ( TCB_t * ) get_headItem_owner( p_eventList );
    assert( p_unblockedTCB );

    /*-----------------------------------*/
    /* 1. Remove Task from eventList     */
    /*-----------------------------------*/
    ( void ) listRemove( &( p_unblockedTCB->eventListItem ) );

    if( scheduler.isSuspended == ( uint32_t ) false )
    {
        /*-------------------------------------------------------*/
        /* 2a. Remove Task from suspended / delayed Tasks list   */
        /*-------------------------------------------------------*/
        ( void ) listRemove( &( p_unblockedTCB->genericListItem ) );
        prvAddTaskToReadyList( p_unblockedTCB );
    }
    else
    {
        /*-------------------------------------------------------*/
        /* 2b. Put Task on pending ready Tasks list              */
        /*-------------------------------------------------------*/
        /* The delayed and ready lists cannot be accessed, so hold this task    */
        /* pending until the scheduler is resumed.                              */
        listInsertEnd( &( coreLists.pendingReadyTasks ), &( p_unblockedTCB->eventListItem ) );
    }

    if( p_unblockedTCB->priority > p_currentTCB->priority )
    {
        /*-----------------------------------------*/
        /* 3a. Unblocked Task has higher priority  */      -> Request a context switch!
        /*-----------------------------------------*/
        yieldRequired = true;
        /* Mark that a yield is pending in case the user is not using the   */
        /* "xHigherPriorityTaskWoken" parameter to an ISR safe FreeRTOS     */
        /* function.                                                        */
        scheduler.yieldPending = true;
    }
    else
    {
        /*-----------------------------------------*/
        /* 3b. Unblocked Task has lower priority   */
        /*-----------------------------------------*/
        yieldRequired = false;
    }

    return yieldRequired;
}
```

The **`removeTask_fromEventList(..)`** function removes the highest priority Task from the list. It also moves that Task from the delayed list back into the proper ready list. A context switch should occur immediately if the unblocked Task has higher priority than the current running one. Again, the yield is not implemented in the function itself, but should be requested by the calling function!



7.2.3 The complete send function

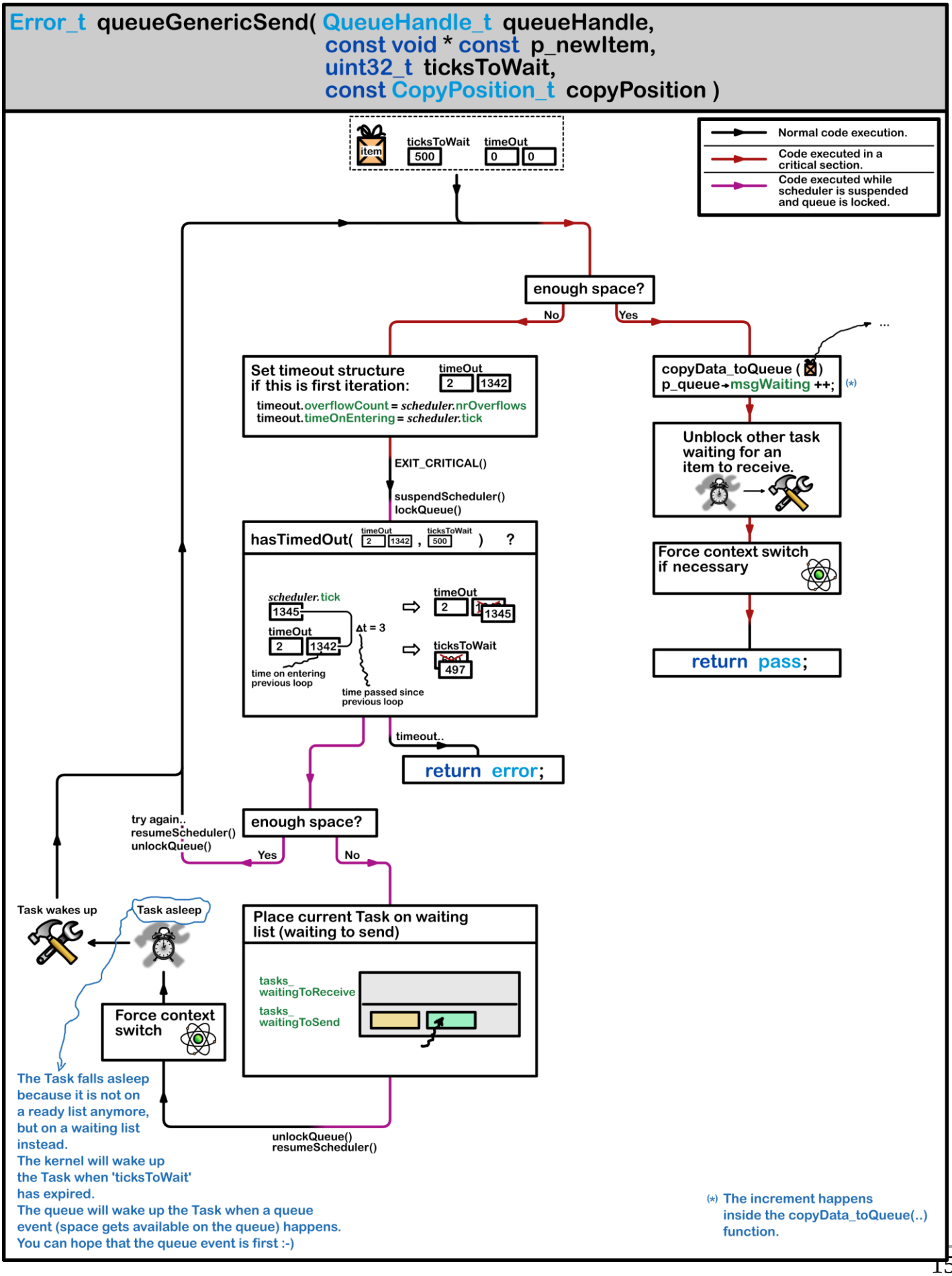
Now we have all the building blocks to proceed to the complete send function. We have a subroutine to copy an item to the queue. We have a subroutine to put a Task on a waiting list. All we need to do now is putting everything together.

```
Error_t queueGenericSend( QueueHandle_t queueHandle,
                          const void * const p_newItem,
                          uint32 t ticksToWait,
                          const CopyPosition_t copyPosition )
{

    bool entryTimeSet = false;
    bool yieldRequired = false;
    TimeOut t timeOut;
    Queue_t * const p_queue = ( Queue_t * ) queueHandle;

    assert( p_queue );
    assert( !( (p_newItem == NULL) && (p_queue->itemSize != (uint32_t) 0U)              ) );
    assert( !( (copyPosition == queueOVERWRITE) && (p_queue->length != 1)               ) );
    assert( !( (xTaskGetSchedulerState() == SCHEDULER_SUSPENDED) && (ticksToWait != 0)  ) );


    for( ;; )
    {
        ENTER_CRITICAL();
        {
            if(  (p_queue->messagesWaiting < p_queue->length) ||
                 (copyPosition == queueOVERWRITE)                    )
            {
            /*----------------------------------------------------------------------------*/
            /*                          ENOUGH SPACE ON THE QUEUE                         */
            /*----------------------------------------------------------------------------*/
                // *----------------------*
                // *  1. Write to queue   *
                // *----------------------*
                yieldRequired = copyData_toQueue( p_queue, p_newItem, copyPosition );


                // *-----------------------------------*
                // *  2. Check if other Task is waiting *
                // *-----------------------------------*
                {
                if( is_empty( &( p_queue->tasks_waitingToReceive ) ) == false )
                {
                    if( removeTask_fromEventList(&(p_queue->tasks_waitingToReceive)) == true )
                    {
                        /* The unblocked task has a higher priority. Force context  */
                        /* switch. Yes it is ok to do this from within the critical */
                        /* section.                                                 */
                        invokePendSV();
                    }
                }
                else if( yieldRequired )
                {
                    /* This path is a special case that will only get               */
                    /* executed if the task was holding multiple mutexes and        */
                    /* the mutexes were given back in an order that is              */
                    /* different to that in which they were taken.                  */
                    invokePendSV();
                }
                }

                EXIT_CRITICAL();

                // *----------------------*
                // *  3. Success! End now *
                // *----------------------*
                return pass;
            }
```

```
        else
        {
        /*-----------------------------------------------------------------------*/
        /*                       NOT ENOUGH SPACE ON THE QUEUE                   */
        /*-----------------------------------------------------------------------*/
            // *----------------------*
            // *  1. Set timeout      *
            // *----------------------*
            if( ticksToWait == ( uint32_t ) 0 )
            {
                /* The queue was full and no block time is specified (or        */
                /* the block time has expired) so leave now.                    */
                /* Return to the original privilege level before exiting        */
                /* the function.                                                */

                EXIT_CRITICAL();
                return errQUEUE_EorF;
            }
            else if( entryTimeSet == false )
            {
                /* Configure the timeout structure.                             */

                setTimeOutState( &timeOut );
                entryTimeSet = true;
            }
        }
    }
    EXIT_CRITICAL();    // Interrupts and other tasks can now send and receive
                        // from the queue.

    // Note: we are still in the for-loop!

    suspendScheduler();
    prvLockQueue( p_queue );
    if( hasTimedOut( &timeOut, &ticksToWait ) )
    {
    // *----------------------*
    // *  2a. Timed out       *
    // *----------------------*

        prvUnlockQueue( p_queue );
        ( void ) resumeScheduler();

        return errQUEUE_EorF;
    }
    else
    {
    // *----------------------*
    // *  2b. Not timed out   *
    // *----------------------*
        bool queueIsFull = false;
        ENTER_CRITICAL();
        {
            if( p_queue->messagesWaiting == p_queue->length ) { queueIsFull = true;  }
                                                         else { queueIsFull = false; }
        }
        EXIT_CRITICAL();

        if( queueIsFull )
        {
        // *-------------------------*
        // *  3a. Full -> Wait       *
        // *-------------------------*
            placeTask_onEventList( &( p_queue->tasks_waitingToSend ), ticksToWait );

            /* Unlocking the queue means queue events can affect the        */
            /* event list.  It is possible  that interrupts occurring now   */
            /* remove this task from the event list again - but as the      */
            /* scheduler is suspended the task will go onto the pending     */
            /* ready list instead of the actual ready list.                 */
            prvUnlockQueue( p_queue );

            /* Resuming the scheduler will move tasks from the pending      */
            /* ready list into the ready list - so it is feasible that this */
            /* task is already in a ready list before it yields - in which  */
            /* case the yield will not cause a context switch unless there  */
```

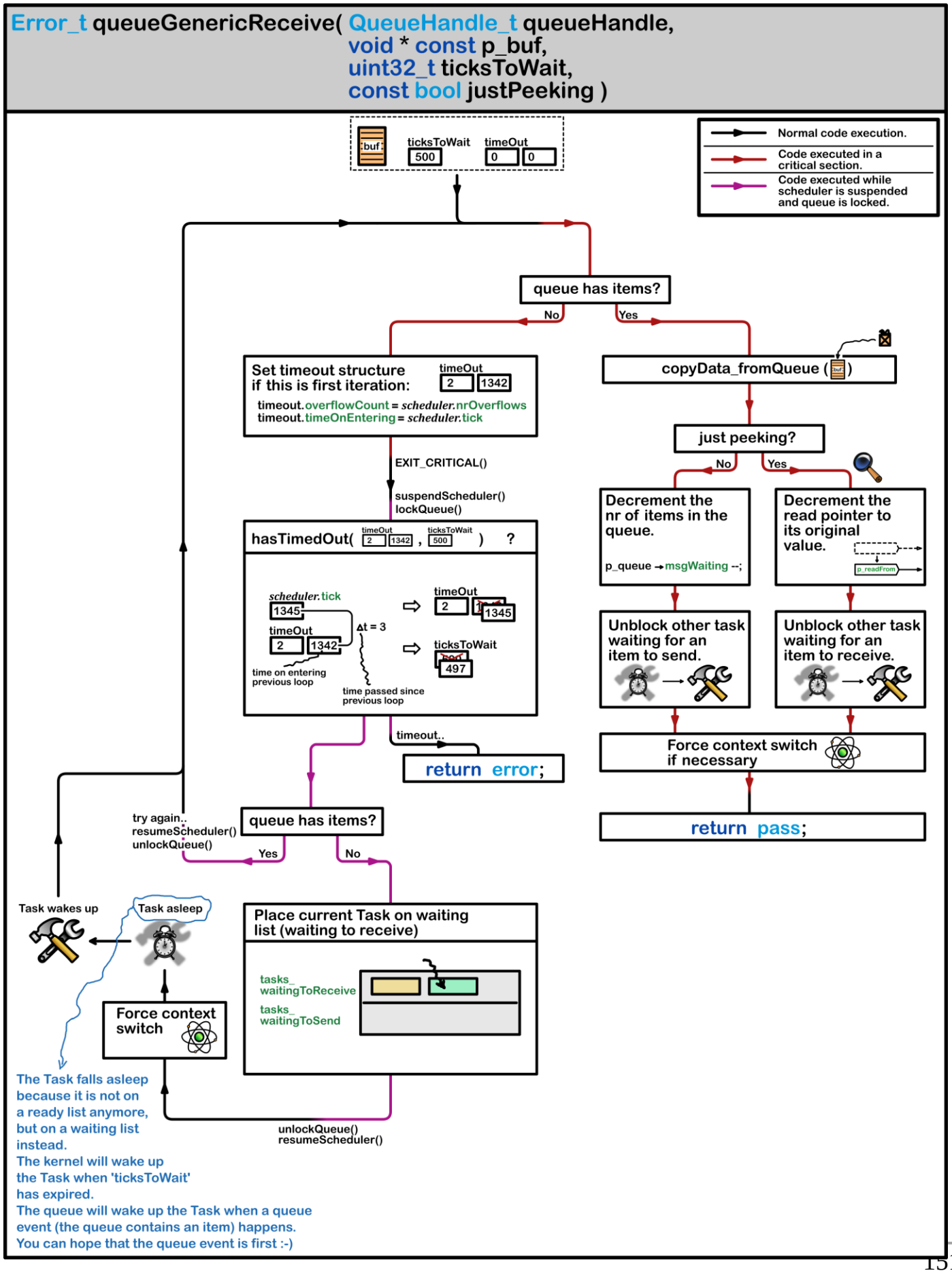
```
                /* is also a higher priority task in the pending ready list.    */
                if( resumeScheduler() == false )
                {
                    invokePendSV();
                }

            }
            else
            {
            // *-----------------------------*
            // *  3b. Not Full -> Try again *
            // *-----------------------------*
                prvUnlockQueue( p_queue );
                ( void ) resumeScheduler();

            }
        }

    }
}
```

The figure below gives a complete overview on how the send function works. You enter the for loop with 3 objects: The item you want to send, a variable that holds the 'ticks to wait' and an empty `TimeOut_t` object.



7.3.4 The complete receive function

The **`queueGenericReceive(..)`** function is very similar to the Send function. The figure below can help to understand how it works.



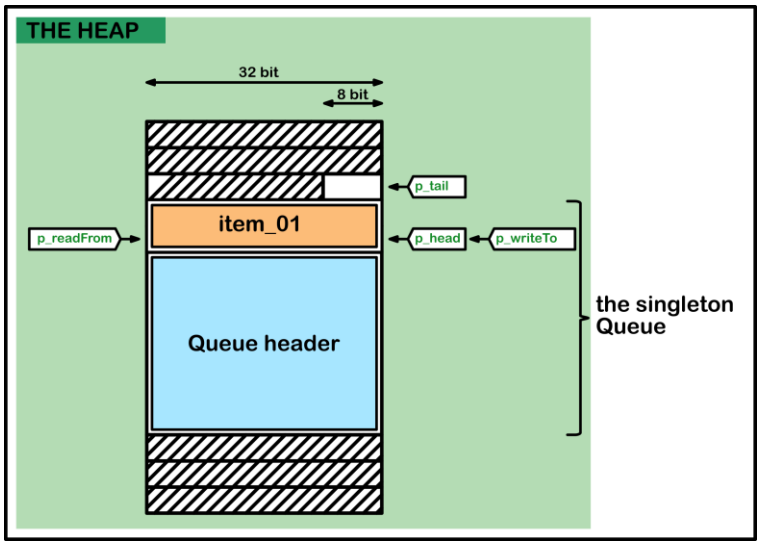
When you decide to just peek the queue, nothing will change. Read the following comments:

```
/*-----------------------------------------------------------------------*/
/* PEEK A QUEUE                                                          */
/* Receive an item from a queue without removing the item from the       */
/* queue. The item is received by copy so a buffer of adequate size      */
/* must be provided.  The number of bytes copied into the buffer was     */
/* defined when the queue was created.                                   */
/*-----------------------------------------------------------------------*/
```

7.4 The singleton queue

Sometimes you need a simple queue that holds no more than one item. Let us create such a queue that can hold one numeric value:

```
QueueHandle_t queue = NULL;
queue = queueCreate(1, sizeof(uint32_t));
```



The singleton queue does not behave different from other queues. Even its creation is completely the same. You only have to fill in '1' as the length – that's it.

FreeRTOS provides a function that cannot be used on any queue, except the singleton type:

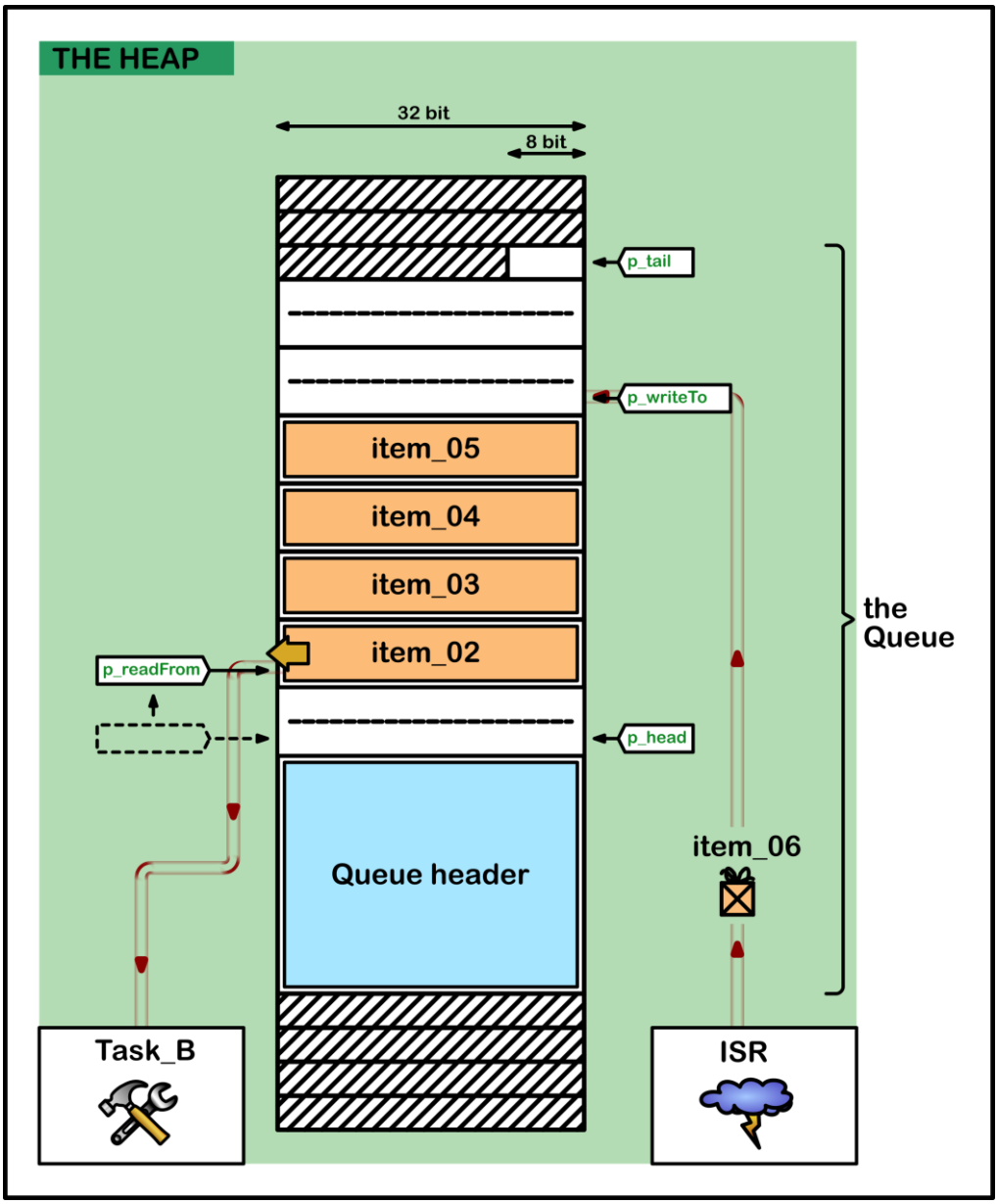
```
queueOverwrite( queue, p_newItem )
```

Here is an example:

```
uint32_t valToSend;
uint32_t valReceived;
QueueHandle_t myQueue = NULL;

myQueue = queueCreate(1, sizeof(uint32_t));
valToSend = 10;
queueOverwrite(myQueue, &valToSend);
valToSend = 100;
queueOverwrite(myQueue, &valToSend);
queueReceive(myQueue, &valReceived, 0);

if(valReceived != 100)
{
    //ERROR!
}
```

7.5 Interrupt routines accessing a queue

Queues provide a communication channel between Tasks. This is their first and foremost purpose. But there is more. An interrupt routine[11] can access a queue as well! The figure below shows an interrupt routine writing `item_06` tot the queue while Task_B is reading `item_02`.

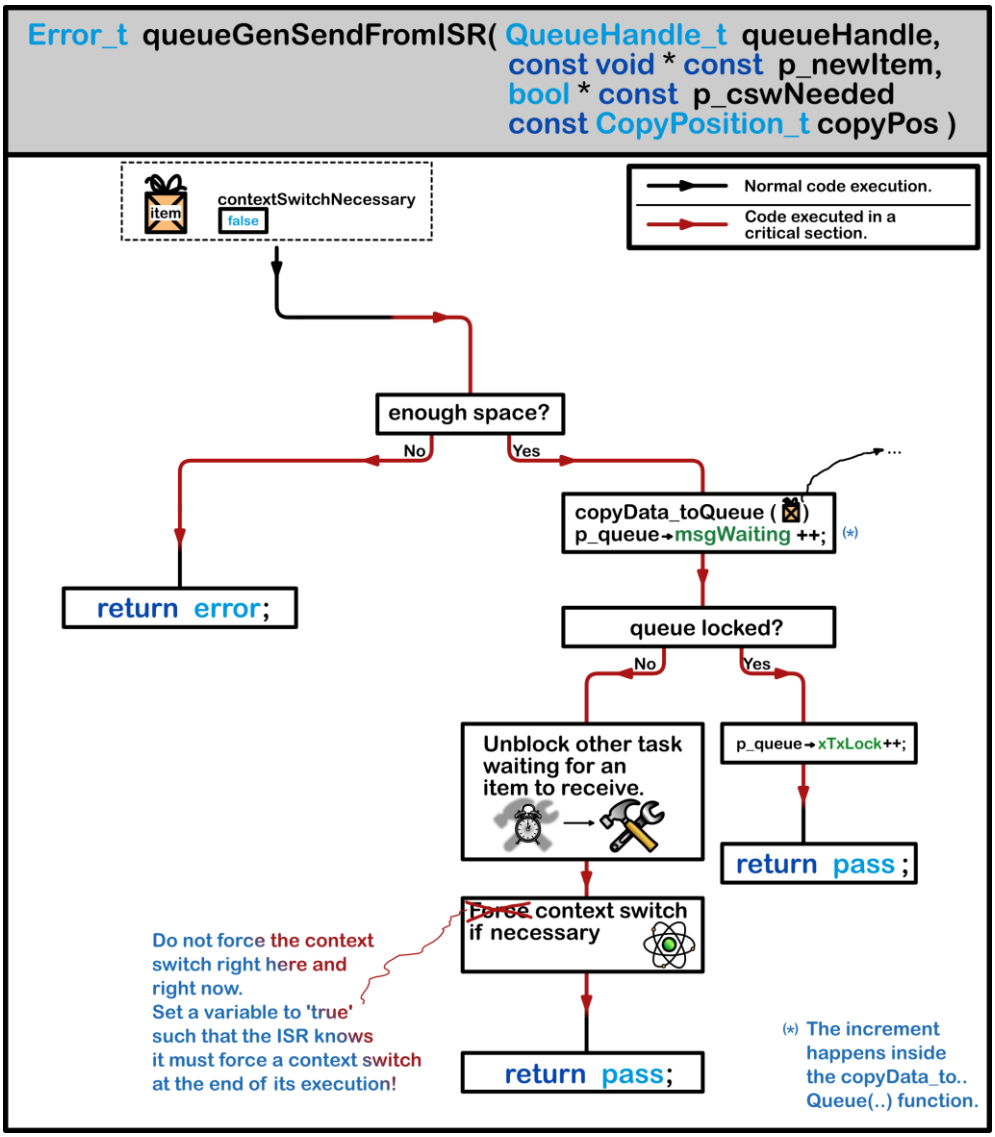


There are a few things to keep in mind here. Would it make sense to define a timeout? An interrupt cannot go to sleep. You want your interrupt to complete in a fraction of a millisecond. Good interrupts do not last longer than a few microseconds. There is no time to sleep!
FreeRTOS provides the following functions to send and receive items to/from a queue:

- **`queueGenericSendFromISR(..)`**
- **`queueGenericReceiveFromISR(..)`**

[11] In fact, only interrupts whose interrupt priority is equal to or logically below `MAX_SYSCALL_IP` (numerically above) are allowed to use a FreeRTOS queue. All other interrupts have a priority which is too high (numerically too low) and are not allowed to interfere in any way with FreeRTOS components.

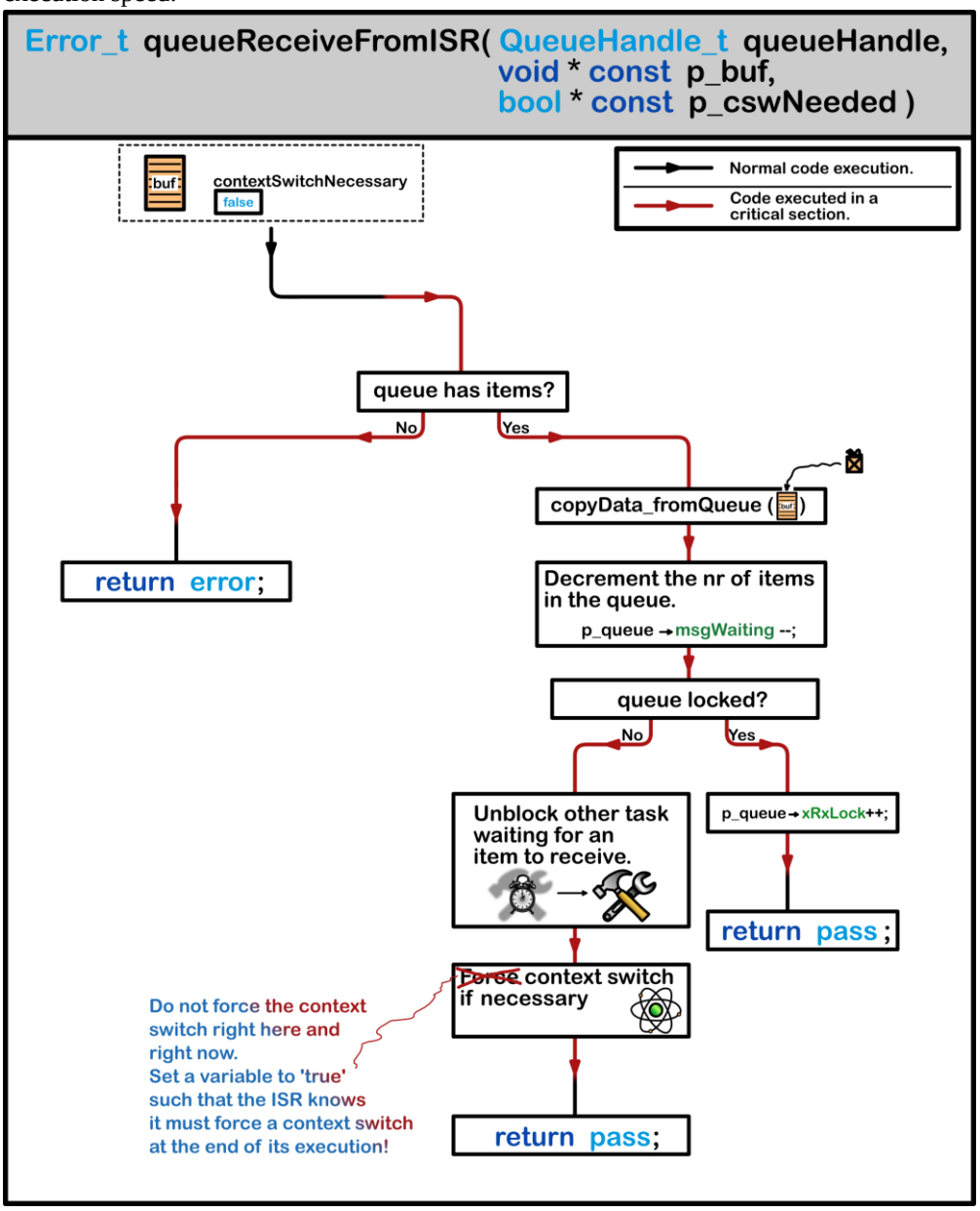
They are simpler and faster. The figure below shows how it works.



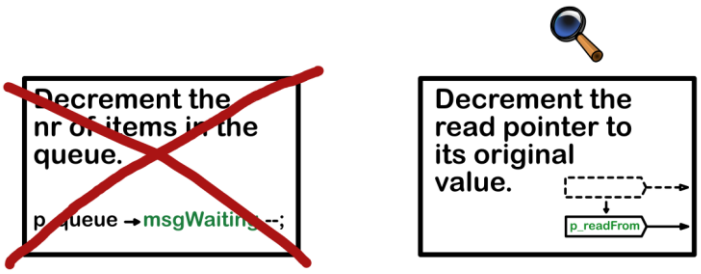
The function checks if the queue is locked after copying the item to it. If the queue is locked, then the field **`p_queue->xTxLock`** should increment. In this way the queue memorizes the fact that an item was added whilst being locked. Later on – when the queue gets unlocked – it will examine if some waiting Tasks on its event list should get unblocked.
If the queue was not locked, then waiting Tasks can get unblocked immediately. Sometimes this can result in a need for context switching. But it is best to wait for the actual context switch till the end of the interrupt routine.

What if there is simply not enough space on the queue? The non-ISR function would put the calling Task asleep and try again later. An error gets returned only after the timeout expires. But we know that an interrupt routine cannot sleep. So this function returns an error code immediately.

The following figure illustrates the receive function. Note that this function does not ask the question if you just want to peek. FreeRTOS provides a separate function for that purpose. This approach slightly improves the execution speed.



The peek function is very similar. Only the following block differs:



Example usage:

```
* Example usage:
   <pre>

 QueueHandle_t xQueue;

 // Function to create a queue and post some values.
 void vAFunction( void *pvParameters )
 {
   char cValueToPost;
   const uint32_t xTicksToWait = ( uint32_t )0xff;

        // Create a queue capable of containing 10 characters.
        xQueue = xQueueCreate( 10, sizeof( char ) );
        if( xQueue == 0 )
        {
                // Failed to create the queue.
        }

        // ...

        // Post some characters that will be used within an ISR.  If the queue
        // is full then this task will block for xTicksToWait ticks.
        cValueToPost = 'a';
        xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
        cValueToPost = 'b';
        xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

        // ... keep posting characters ... this task may block when the queue
        // becomes full.

        cValueToPost = 'c';
        xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
 }

 // ISR that outputs all the characters received on the queue.
 void vISR_Routine( void )
 {
    int32_t xTaskWokenByReceive = false;
    char cRxedChar;

        while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &xTaskWokenByReceive) )
        {
                // A character was received.  Output the character now.
                vOutputCharacter( cRxedChar );

                // If removing the character from the queue woke the task that was
                // posting onto the queue cTaskWokenByReceive will have been set to
                // true.  No matter how many times this loop iterates only one
                // task will be woken.
        }

        if( xTaskWokenByReceive != ( char ) false;
        {
                portYIELD ();
        }
 }
```

7.6 Other Queue interactions

```
#define xQueueOverwrite(queue, p_newItem) queueGenericSend( (queue), (p_newItem), 0, queueOVERWRITE )
```

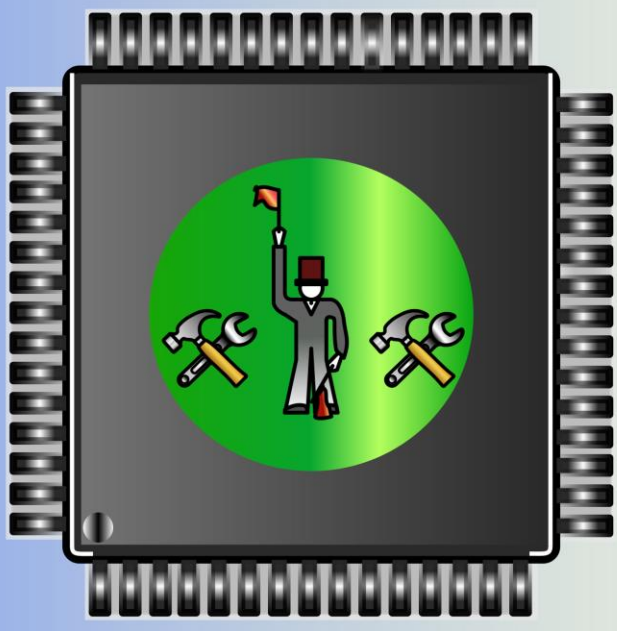
UNDER CONSTRUCTION

The example usage for an interrupt:

```
* Example usage:
   <pre>

 QueueHandle_t xQueue;

 void vFunction( void *pvParameters )
 {
        // Create a queue to hold one uint32_t value.  It is strongly
        // recommended *not* to use xQueueOverwriteFromISR() on queues that can
        // contain more than one value, and doing so will trigger an assertion
        // if assert() is defined.
        xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}

void vAnInterruptHandler( void )
{
// xHigherPriorityTaskWoken must be set to false before it is used.
int32_t xHigherPriorityTaskWoken = false;
uint32_t ulVarToSend, ulValReceived;

        // Write the value 10 to the queue using xQueueOverwriteFromISR().
        ulVarToSend = 10;
        xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

        // The queue is full, but calling xQueueOverwriteFromISR() again will still
        // true because the value held in the queue will be overwritten with the
        // new value.
        ulVarToSend = 100;
        xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

        // Reading from the queue will now return 100.

        // ...

        if( xHigherPrioritytaskWoken == true )
        {
                // Writing to the queue caused a task to unblock and the unblocked task
                // has a priority higher than or equal to the priority of the currently
                // executing task (the task this interrupt interrupted).  Perform a context
                // switch so this interrupt returns directly to the unblocked task.
                invokePendSVif(); // or invokePendSVif() depending on the port.
        }
}
```

Chapter 8

Synchronizing through semaphores



8.1 Binary semaphores

A binary semaphore can be considered conceptually as a queue with length one. The queue can only contain one item, so is always either empty or full.

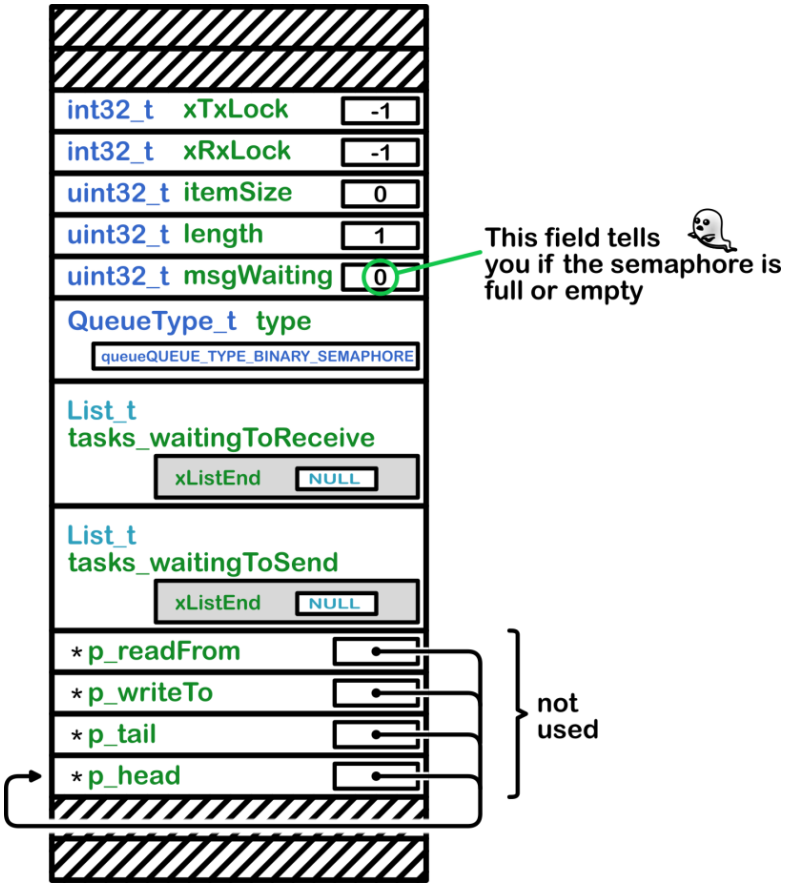
```
SemaphoreHandle_t semphrCreateBinary()
{
    QueueHandle_t xHandle;

    xHandle = queueGenericCreate(1, 0, queueQUEUE_TYPE_BINARY_SEMAPHORE );

    assert( xHandle );
    return ( SemaphoreHandle_t )xHandle;
}
```

queueLength itemSize queueType

So a binary semaphore is a queue with just one item – just like a singleton queue. Right? This is not entirely correct. A binary semaphore has indeed one item, but the item itself has no size (`itemSize = 0`) There is no space allocated for it. It is a ghost item.

Why on earth would you want to create such a strange object? Because the binary semaphore has no intentions to transfer any sort of data between Tasks. So it would be wasteful to allocate memory for the item. The zero-sized item item only serves one purpose. It's presence in the semaphore makes it 'full'. It's absence in the semaphore makes it 'empty'. So the ghost item corresponds conceptually to the semaphore flag.

The figure below shows a freshly created binary semaphore. Notice that there is no memory allocated for data. The messagesWaiting field tells you if the semaphore is either full or empty.



Attention: binary semaphores can cause confusion, as they do not follow the same rules as other semaphore usages (eg. mutexes), where a Task that takes a semaphore must always give it back.

8.2 Binary semaphore – give and take functions

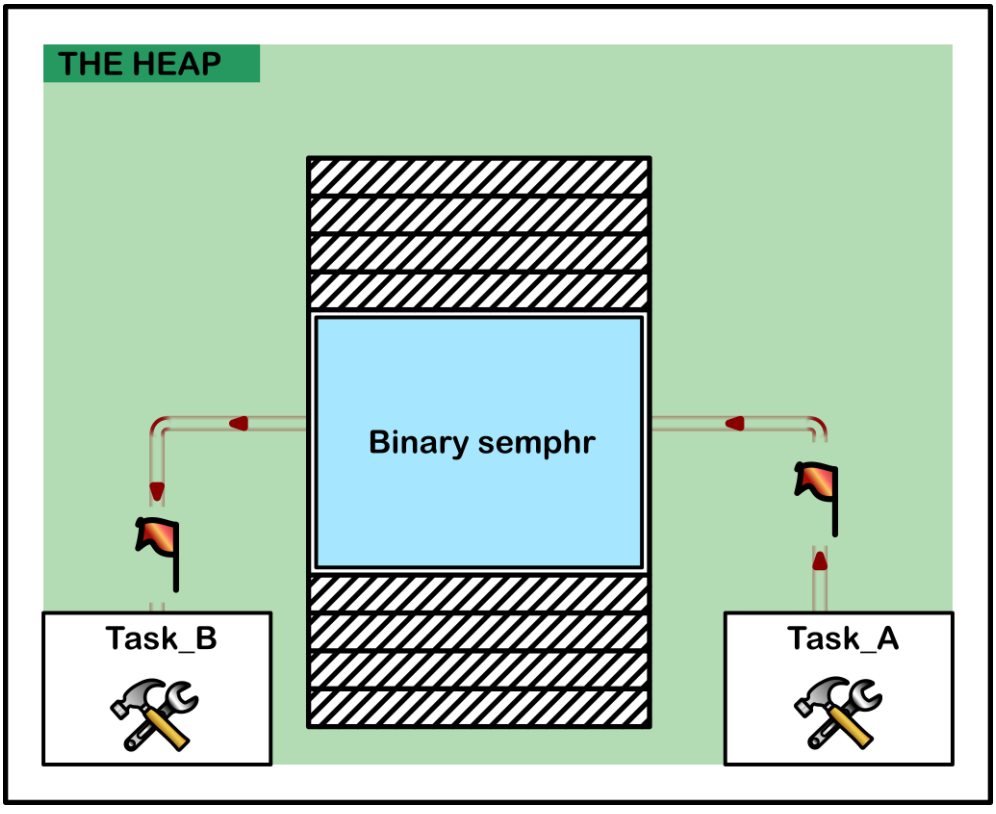
8.2.1 The semaphore give function

A Task can fill a semaphore by calling the **semphrGive** function[12]:

```
#define semphrGive(xSemaphore)    queueGenericSend(  (QueueHandle_t)xSemaphore,
                                                     NULL,
                                                     0,
                                                     queueSEND_TO_BACK                )
```

(pNewItem, ticksToWait, copyPosition)

The item passed on is **NULL**. Can the queue send function handle this? Yes, it is possible. The queue send function calls the **copyData_toQueue(..)** function, which attempts to copy the data to the queue. But before doing so, it checks the **itemSize** of the queue first. Simply to know how many bytes should get copied. If that **itemSize** is zero, no copy is made at all. But the **messagesWaiting** field does increment. That is exactly what we need.



Notice that the **semphrGive** function will never block. The **ticksToWait** field is zero. When the semaphore was already full, the **semphrGive** function returns immediately:

```
return errQUEUE_FULL;
```

[12] To be exact, this is not a function but a macro.

8.2.2 The semaphore take function

```
#define semphrTake( xSemaphore, xBlockTime )    queueGenericReceive( (QueueHandle_t)xSemaphore,
                                                                     NULL,
                                                                     xBlockTime,
                                                                     false                    )
```

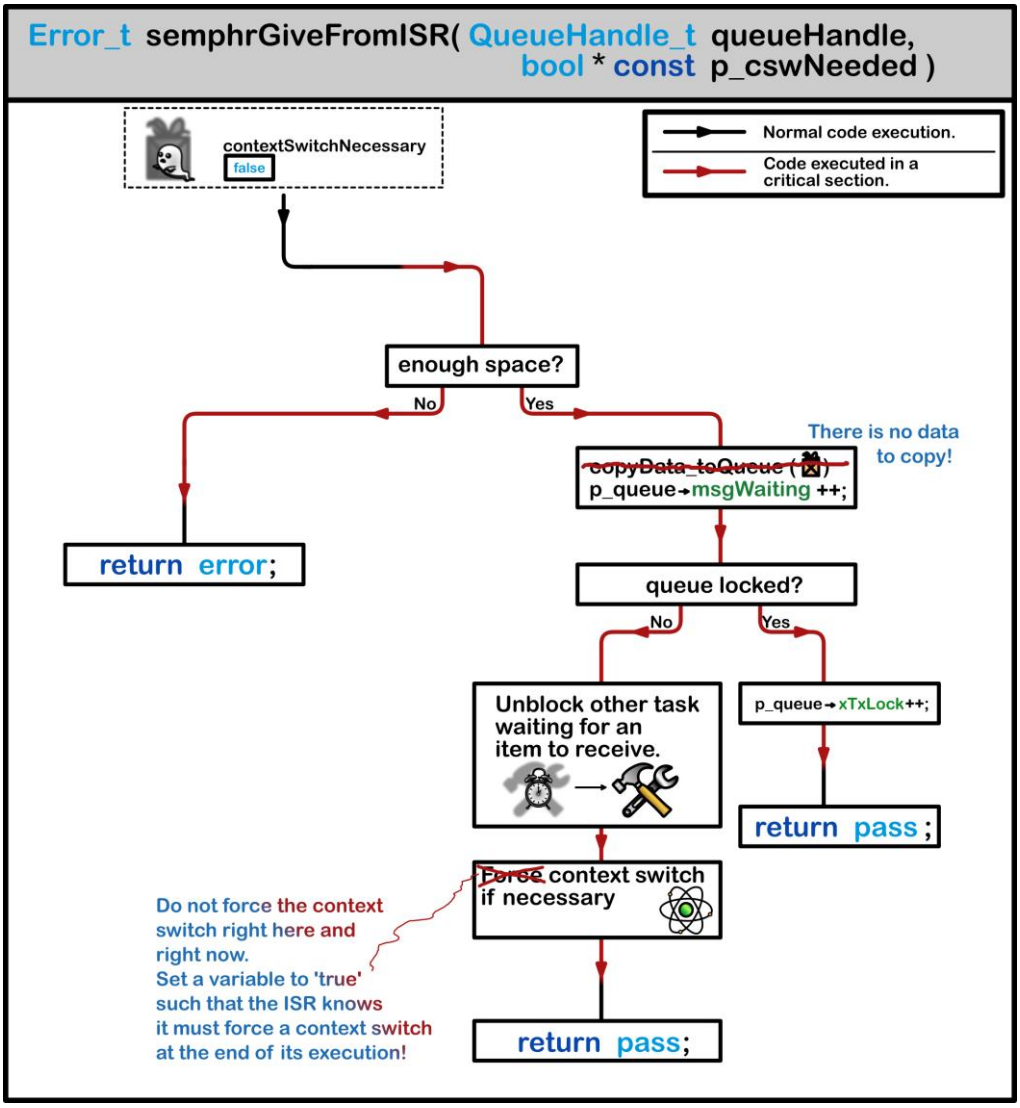
p_buf, ticksToWait, justPeeking

The `semphrTake` function uses the corresponding queue function as well. Of course it does not expect to get an actual item so it provides `NULL` as buffer. You can set a blocking time. The Task making this call will fall asleep for a while if the semaphore is not available immediately. It will wake up when the timeout has expired or when the semaphore is available – whichever happens first.

8.3 Binary semaphore access from an interrupt

8.3.1 The semaphore give function

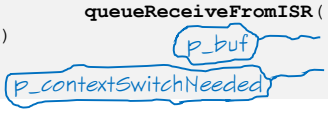
The **`semphrGiveFromISR(..)`** function is very similar to **`queueGenericSendFromISR(..)`**. In fact, it would be perfectly possible to define **`semphrGiveFromISR`** as a simple macro-wrapper. But doing so would lead to a small delay penalty. You would need to check a few things (like checking if you're dealing with a binary semaphore such that no actual data copy is needed, etc..). We want interrupts to be as short as possible. That's why a dedicated **`semphrGiveFromISR(..)`** makes sense. The following figure illustrates the inner workings of the function.



7.3.2 The semaphore take function

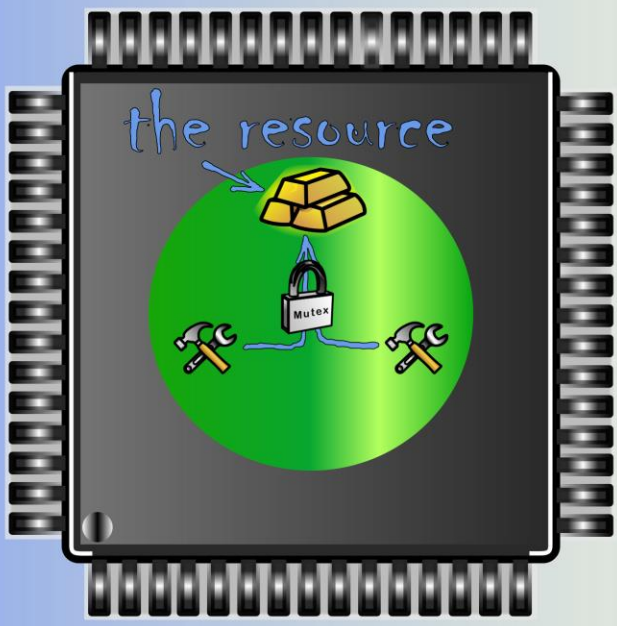
The **semphrTakeFromISR(..)** function is also very similar to **queueReceiveFromISR(..)**. So you are confronted with the same dilemma: design the **semphrTakeFromISR** function as a wrapper or make a dedicated function. For some reason, FreeRTOS chose the first option here.

```
#define semphrTakeFromISR(semphr,                    queueReceiveFromISR( (QueueHandle_t) semphr,
                          p_contextSwitchNeeded)                         NULL,
                                                                         p_contextSwitchNeeded )
```



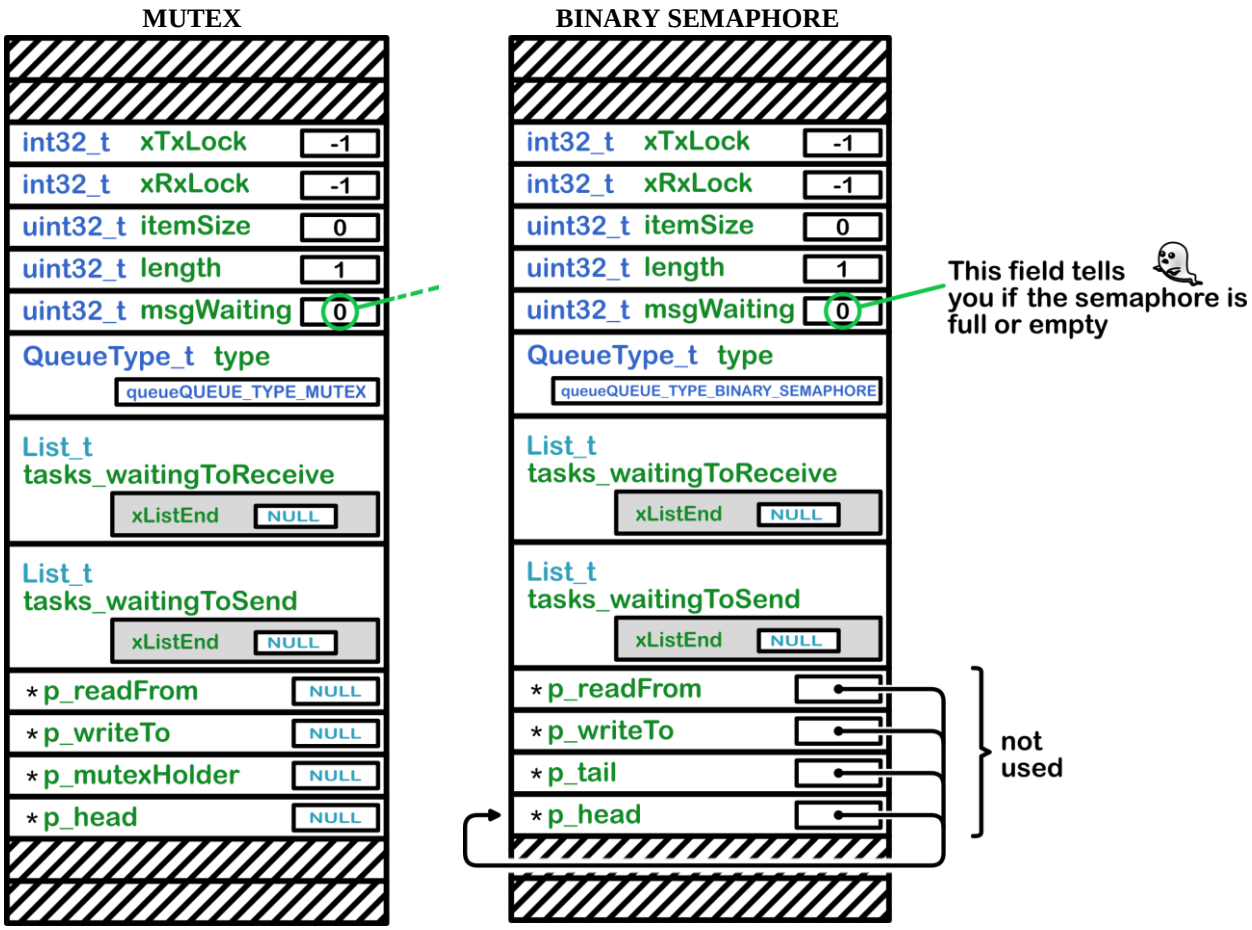
Chapter 9

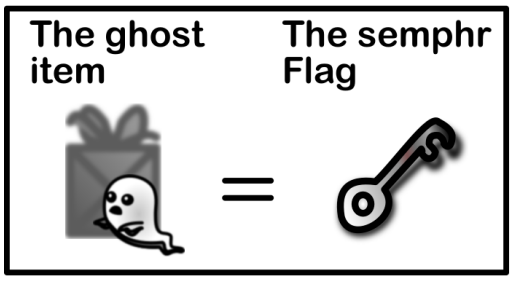
Sharing resources through mutexes



9.1 The mutex creation

The following figure illustrates the differences between a Mutex and a Binary semaphore object:

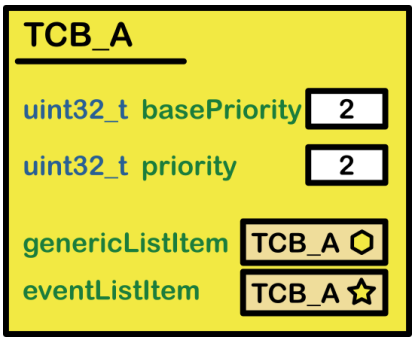




9.2 Priority inheritance

9.2.1 Review of Tasks and their list items

Remember that every Task has two list items:



- The Task uses its `genericListItem` to sit in one of the Kernel lists. That would be a **`readyTasks`** list if the Task is not blocked.
The value of the genericListItem is not fixed and in many cases not important. It does play a role when you put the item in the **`delayedTasks`** list. The algorithm gives a value to the item corresponding to the 'time to wake'. In this way, you can be sure that the first item in the delayedTasks list wakes up first.

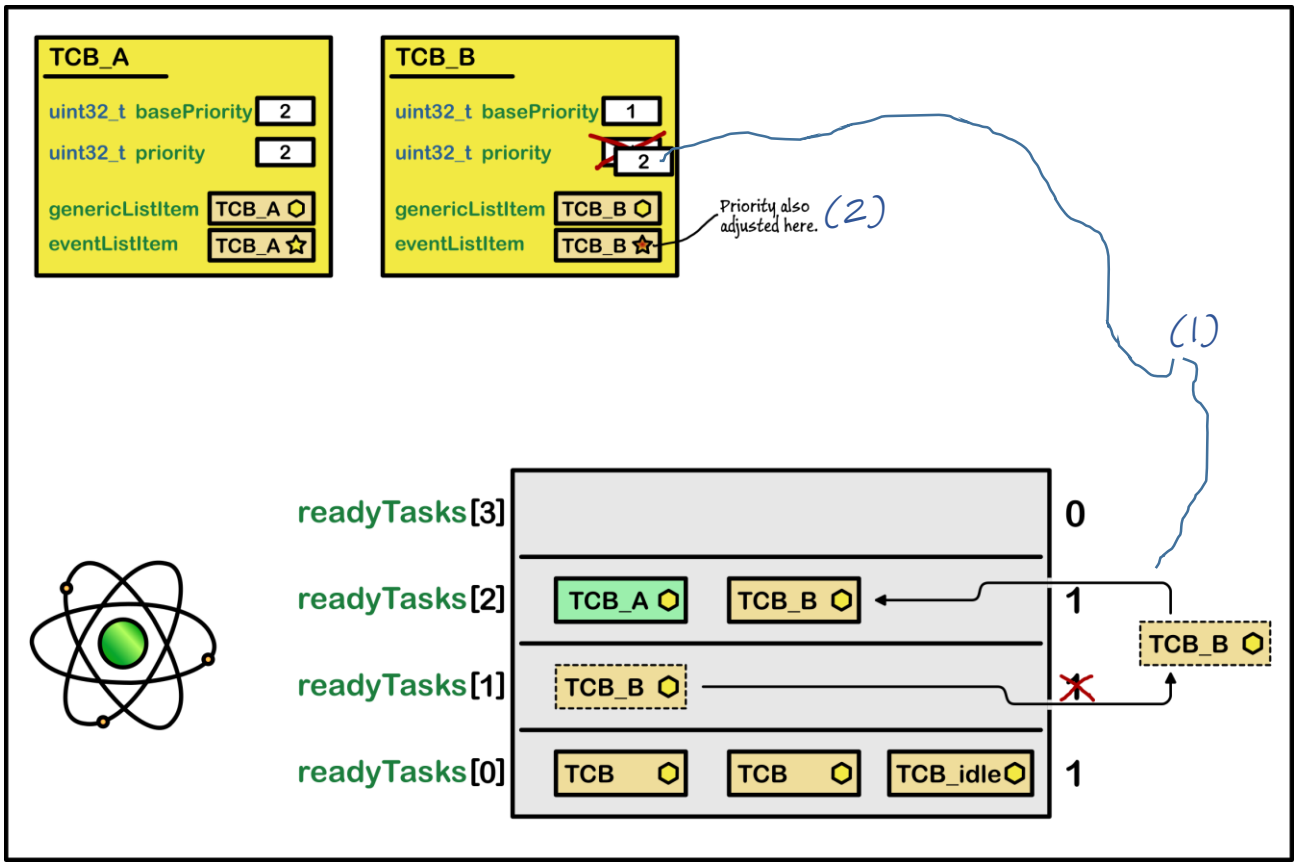
- The Task uses its `eventListItem` to sit in an **`eventList`**. Such list can belong to a binary semaphore, a mutex, …
The value of the eventListItem is important. It gets a fixed value at the very beginning – when the Task is created. The value is inversely proportional to the priority of the Task. Since lists are ordered in ascending value order, you can be sure that the first item in every eventList corresponds to the highest priority Task in that list.

9.2.2 The priority inheritance algorithm

The figure below shows what happens when you invoke the **`inheritPriorityFromCurrentTask(..)`** function.

```
ENTER_CRITICAL();
{
    inheritPriorityFromCurrentTask( p_taskB );
}
EXIT_CRITICAL();
```

We assume that `taskA` is the current Task.



The figure shows two things happening to `TaskB`:

(1) The priority level of `TaskB` increments. The **`genericListItem`** is no longer in the correct **`readyTasks`** list. So the algorithm shifts it one level up.

(2) The **`eventListItem`** holds a value inversely proportional to the priority of the Task. The algorithm updates that value[13].

You can conclude that `TaskB` has leveled up to the priority of `TaskA`. That is exactly what the priority inheritance algorithm aims to achieve. But does TaskB truly behave according to its newly adapted priority? Are there any shortcomings or pain points?

9.2.3 The shortcomings of the priority inheritance implementation

It is interesting to investigate the things that the inheritance implementation does not do. Let us consider a few examples:

(1) The **`genericListItem`** levels up in the **`readyTasks`** lists. But what if the Task is asleep? In that case you won't find the genericListItem in one of the **`readyTasks`** lists. The priority inheritance mechanism simply increments the priority level but won't intervene with the defined sleep time. The Task just keeps sleeping as if nothing special happened.

(2) The value in the Tasks **`eventListItem`** gets updated. This will make sure that the Task will end up closer to the head of any **`eventList`** you put it in. In other words, when you make this Task waiting for another semaphore, it will be put in the higher priority section of the waiting list.
But nothing changes if the Task is already in an **`eventList`** (waiting list). The priority inheritance algorithm does not dive into the existing **`eventLists`** to reorganize them.

[13] It is remarkable that the **`eventListItem`** only updates when a very specific condition is met. The condition states that the update must not occur when the bit corresponding to **`EVENT_LIST_ITEM_VALUE_IN_USE`** equals '1'. When I first encountered this condition, I thought that the bit is set whenever you put an **`eventListItem`** into an **`eventList`**. In other words, whenever you put a Task in the waiting list of a binary semaphore, mutex, …
Apparently the significance of the bit is somewhat more subtle. The bit is NOT set when you put the Task in a normal waiting list. The bit is only set when you put it in an "unordered waiting list". At the moment of writing this paragraph, I do not know what unordered **`evenLists`** are used for.

9.1 The mutex type confusion

9.1 The confusion explained

The way FreeRTOS distinguishes a mutex from other queue types (like the basic queue, binary semaphores, …) is confusing. Let us first start to examine the whole thing.
You can find the following definition in `queue.c` :

```
#define queueQUEUE_IS_MUTEX      NULL
```

The `queue.c` file also gives the following comment:

```
/*-----------------------------------------------------------------------------*/
/* When the Queue_t structure is used to represent a base queue, its p_head and */
/* p_tail members are used as pointers into the queue storage area.             */
/* When the Queue_t structure is used to represent a mutex, p_head and p_tail   */
/* pointers are not necessary, and the p_head pointer is set to NULL to         */
/* indicate that the p_tail pointer actually points to the mutex holder (if     */
/* any).                                                                        */
/* Map alternative names to the p_head and p_tail structure members to ensure   */
/* the readability of the code is maintained despite this dual use of two       */
/* structure members.                                                           */
/*-----------------------------------------------------------------------------*/
```

So we can conclude that the struct members **`p_head`** and **`p_tail`** have a dual function. For a normal queue, they point to the storage area. For a mutex, **`p_head`** equals `0` and **`p_tail`** points to the mutex holder. The comment later on mentions that there are two ways to deal with this situation. You can define two macros that represent an alias for these two members:

```
#define p_mutexHolder                   p_tail
#define type                            p_head
```

Or you can use a union when you define the struct:

```
typedef struct QueueDefinition
    union
    {
        int8_t *p_head;
        uint32_t type;
    }

    union
    {
        int8_t *p_tail;
        void *p_mutexHolder;
    };

    …

} Queue_t;
```

It doesn't matter which option you choose. The philosophy remains the same. When you are dealing with a normal queue, use the **`p_head`** and **`p_tail`** member names. When you are dealing with a mutex queue, use the **`type`** and **`p_mutexHolder`** aliases.
FreeRTOS chose the first option (macros instead of unions).

But now comes the funny part. When you create a mutex, FreeRTOS wants you to use a macro from the `semphr.h` file:

```
#define xSemaphoreCreateMutex() xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
```

This macro calls the function **xQueueCreateMutex(..)** from the queue.c file. But the argument passed on is not what you would expect!
You can find the following definition in queue.h :

```
#define queueQUEUE_TYPE_MUTEX    ((uint8_t)1U)
```

So the file queue.c contains a definition for **queueQUEUE_IS_MUTEX** and the file queue.h holds a definition for **queueQUEUE_TYPE_MUTEX**. Sadly they have a different numerical value.

So what is the function **xQueueCreateMutex(..)** going to do with this situation? Well, it turns out to be like this:

- Somewhere in the beginning of the function, the **queue_type** member is set to **queueQUEUE_IS_MUTEX**. This corresponds to numerical value 0.

- If configUSE_TRACE_FACILITY == 1 then the **queue_type** member (alias for **p_head**) is changed to **queueQUEUE_TYPE_MUTEX**. This corresponds to numerical value 1.

It is very strange – to say the least – that switchin on the configUSE_TRACE_FACILITY has an impact on the value in the **type** struct member! This is a big deal, because the value in that struct member plays an important role in many functions. It is the key for FreeRTOS to decide if a given queue should be treated as a normal or a mutex-type queue.

9.2 A possible solution

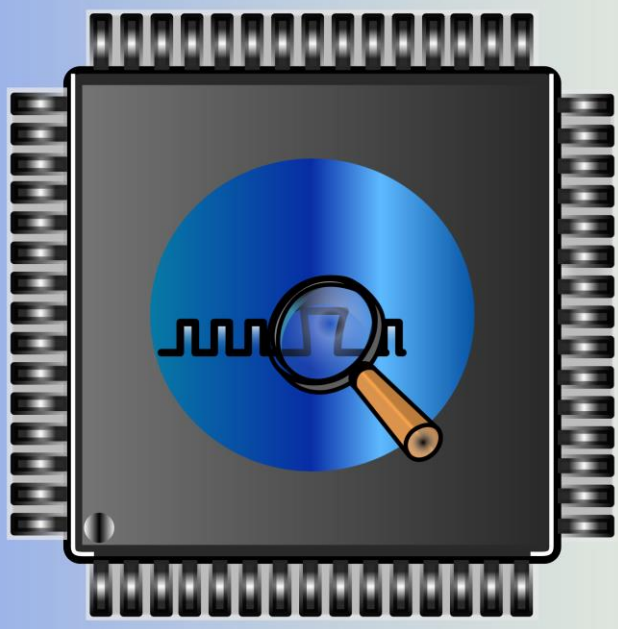
To clear up this mistake, I delete the **queueQUEUE_IS_MUTEX** definition in the queue.c file. Wherever this macro was used, I replace it with **queueQUEUE_TYPE_MUTEX**. So my code has only a single numerical value that represents a mutex, and a single macro-reference to it.

Next I decoupled the **p_head** struct member and its alias. In my code **type** is no longer an alias for **p_head**, but a separate member of the struct.
Then I make sure that the code for creating a new mutex behaves correctly. It should clear **p_head** to NULL, because that struct member is simply not used. And it should set **type** to **queueQUEUE_TYPE_MUTEX** such that freeRTOS can distinguish the mutex from a regular queue.

PART 3:
Peripheral Drivers

Chapter 1

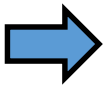
SPI bus

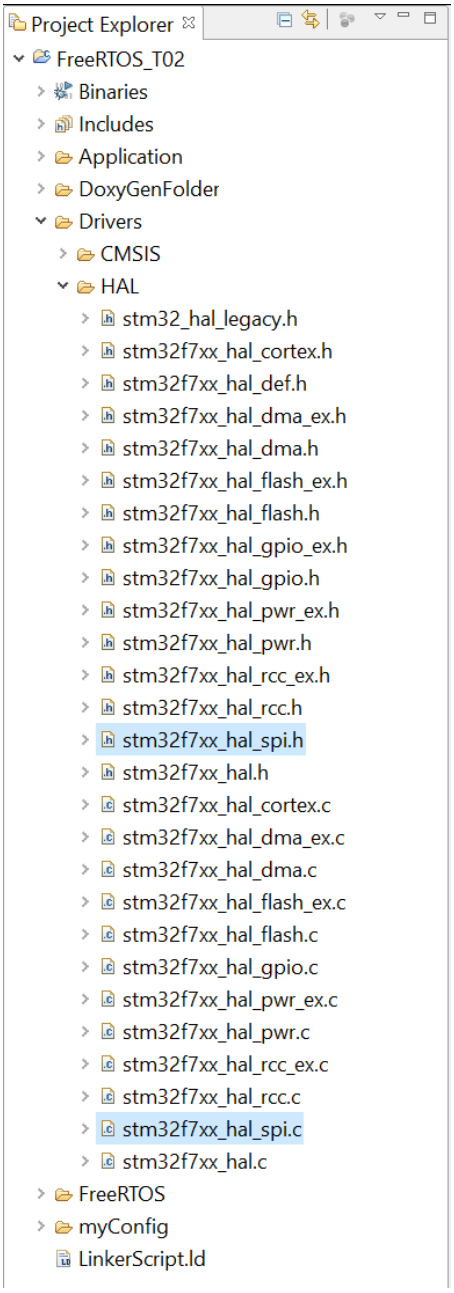


In this chapter we will examine the most popular communication protocol between chips: SPI. All modern sensor chips support this standard. SPI is faster and easier to use than I2C.
You have several options if you want to use SPI. You could start from scratch and write your own driver. Or you can rely on the driver software that STMicroelectronics provides. We opt for the second approach. We will first look at the generated driver software from the STM32CubeMX software and plug it into our project.

1.1 Import the HAL driver from STMicroelectronics

The STM32CubeMX software from STMicroelectronics generates the SPI driver if you select one (or more) of the six available SPI modules. That's right. Our microcontroller has six SPI modules! Let us examine the driver software for the first module. The other modules are similar of course.





Do not worry if you forgot to select the SPI module at the beginning of your project. It is very easy to plug the driver software at any moment into your project.
Generate a new project with CubeMX and copy the two following files into your own project:

- `stm32f7xx_hal_spi.h`
- `stm32f7xx_hal_spi.c`

These two files should end up in the right place. That is the folder `.../Drivers/HAL` in your project.

These two files contain almost the whole SPI driver. There are 3 things you should do to get the driver up and running:

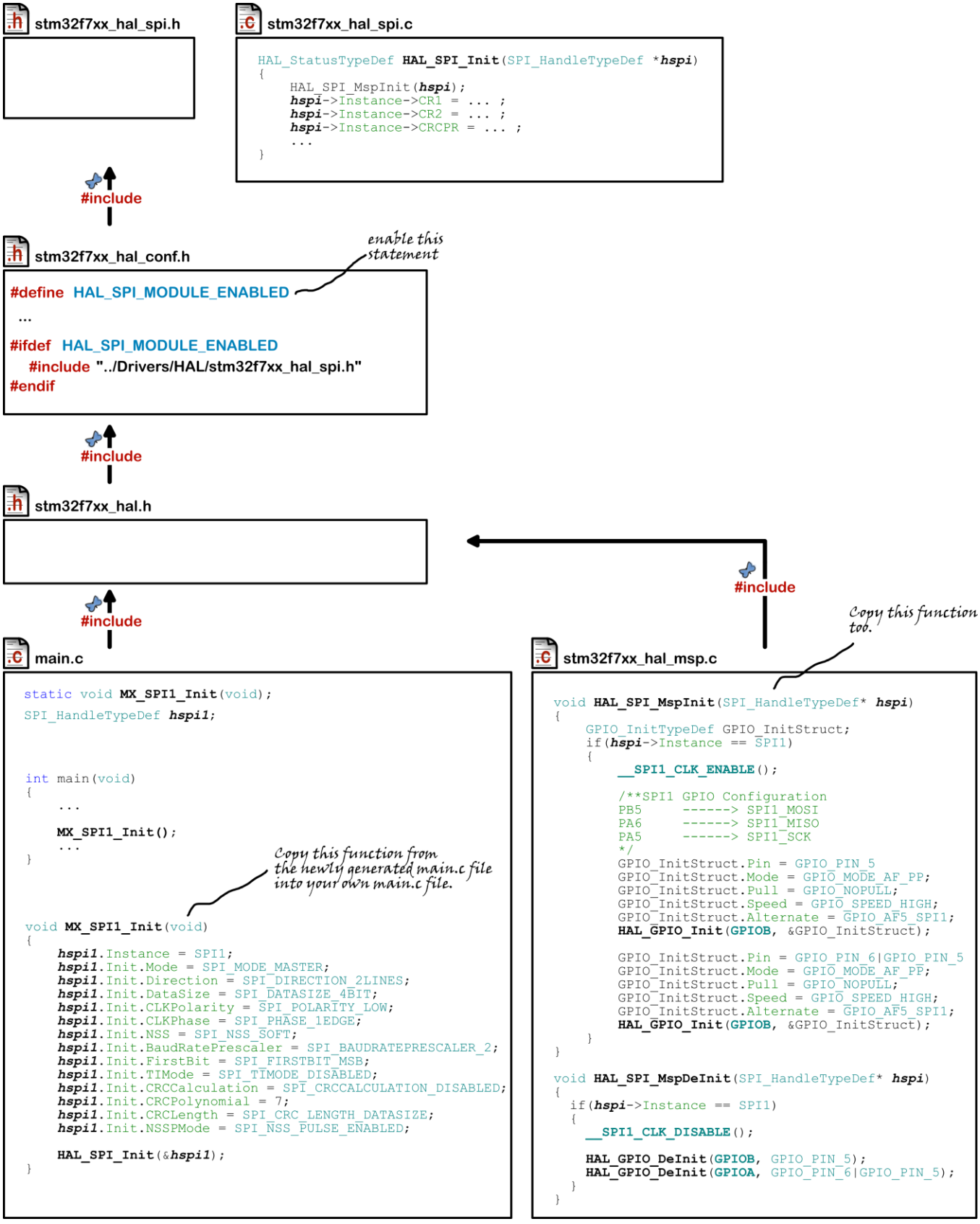
1. Enable the `#define` statement in the `stm32f7xx_hal_conf.h` file:

```
#define HAL_SPI_MODULE_ENABLED
```

2. Add the function `MX_SPI1_Init(void)` to the `main.c` file. You can copy the function from the newly generated `main.c` file (the one that CubeMX just generated).

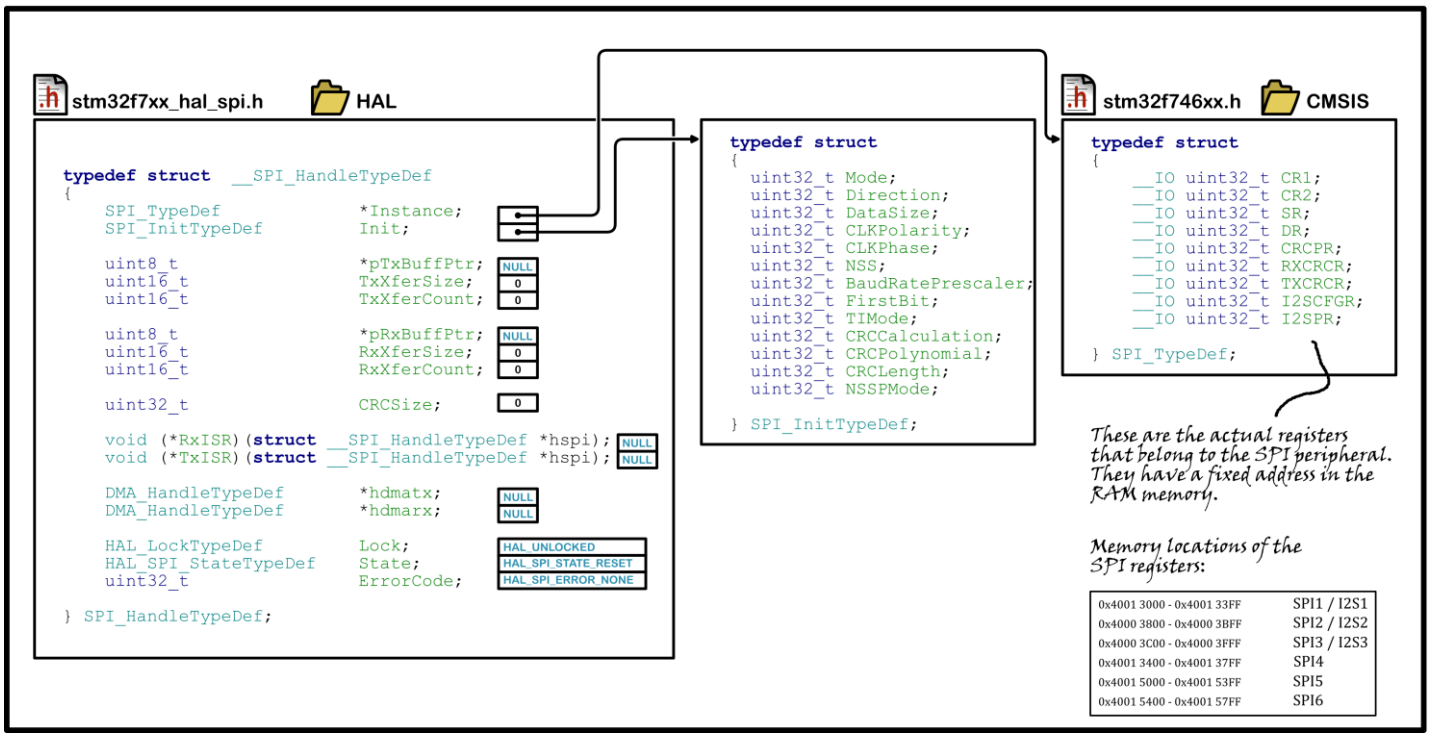
3. Add the function `HAL_SPI_MspInit(SPI_HandleTypeDef *hspi)` to the `stm32f7xx_hal_msp.c` file. The content of that function can be found in the `stm32f7xx_hal_msp.c` file just generated by CubeMX.

The next figure gives an overview.


stm32f7xx_hal_spi.h
stm32f7xx_hal_spi.c
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi)
{
HAL_SPI_MspInit(hspi);
hspi->Instance->CR1 = ... ;
hspi->Instance->CR2 = ... ;
hspi->Instance->CRCPR = ... ;
...
}
#include
stm32f7xx_hal_conf.h
enable this statement
#define HAL_SPI_MODULE_ENABLED
...
#ifdef HAL_SPI_MODULE_ENABLED
#include "../Drivers/HAL/stm32f7xx_hal_spi.h"
#endif
#include
stm32f7xx_hal.h
#include
#include
main.c
static void MX_SPI1_Init(void);
SPI_HandleTypeDef hspi1;
int main(void)
{
...
MX_SPI1_Init();
...
}
Copy this function from the newly generated main.c file into your own main.c file.
void MX_SPI1_Init(void)
{
hspi1.Instance = SPI1;
hspi1.Init.Mode = SPI_MODE_MASTER;
hspi1.Init.Direction = SPI_DIRECTION_2LINES;
hspi1.Init.DataSize = SPI_DATASIZE_4BIT;
hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
hspi1.Init.NSS = SPI_NSS_SOFT;
hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
hspi1.Init.TIMode = SPI_TIMODE_DISABLED;
hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLED;
hspi1.Init.CRCPolynomial = 7;
hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
hspi1.Init.NSSPMode = SPI_NSS_PULSE_ENABLED;
HAL_SPI_Init(&hspi1);
}
#include
Copy this function too.
stm32f7xx_hal_msp.c
void HAL_SPI_MspInit(SPI_HandleTypeDef* hspi)
{
GPIO_InitTypeDef GPIO_InitStruct;
if(hspi->Instance == SPI1)
{
__SPI1_CLK_ENABLE();
/**SPI1 GPIO Configuration
PB5 ------> SPI1_MOSI
PA6 ------> SPI1_MISO
PA5 ------> SPI1_SCK
*/
GPIO_InitStruct.Pin = GPIO_PIN_5
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF5_SPI1;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_5
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF5_SPI1;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
}
void HAL_SPI_MspDeInit(SPI_HandleTypeDef* hspi)
{
if(hspi->Instance == SPI1)
{
__SPI1_CLK_DISABLE();
HAL_GPIO_DeInit(GPIOB, GPIO_PIN_5);
HAL_GPIO_DeInit(GPIOA, GPIO_PIN_6|GPIO_PIN_5);
}
}

1.2 The SPI handle

The whole SPI driver is built around an **SPI_Handle** object. Think of it as an object that represents the driver. All your interactions with the driver are through this object.
The **SPI_Handle** object itself has no special location in memory. It can end up anywhere in the RAM. But its ***Instance** field points to the actual SPI peripheral registers[14]. These registers do have a fixed RAM address, burned in silicon.



It is very important that the ***Instance** field points to the right spot. Therefore the following code line is executed at the very beginning of the handle initialization:

```
hspi1.Instance = SPI1;
```

The file `stm32f746xx.h` in the CMSIS folder holds the definition for **SPI1**:

```
#define SPI1                    ((SPI_TypeDef *) SPI1_BASE)

#define SPI1_BASE               (APB2PERIPH_BASE + 0x3000)
#define APB2PERIPH_BASE         (PERIPH_BASE + 0x00010000)
#define PERIPH_BASE             ((uint32_t)0x40000000)
```

From this definition we can conclude that **SPI1** is of type `(SPI_TypeDef *)` and has value `0x40013000`. Basically **SPI1** is a pointer and holds the start address of the memory block it points to. We know from the datasheet of our microcontroller that the SPI registers are located at address `0x40013000`. At least for the first SPI module. The microcontroller has 6 SPI modules in total. But we only consider the first one[15] in this chapter.

[14] These registers are called SFRs: *Special Function Registers*.
[15] If you know how to configure the first module, you can configure easily all the others.

1.3 SPI initialization

The figure below clarifies the initialization procedure. The main routine goes first through the necessary general HAL initializations:

```
HAL_Init();
SystemClock_Config();
```

Later on the main routine invokes one by one the initialization functions for each peripheral. For the first SPI module this function is named **`MX_SPI1_Init()`**. The implementation of that function is also in the `main.c` file. The function basically has two main purposes:

- Make the **`Instance`** field point to the right spot – where the SPI registers are located.
- Fill the **`Init`** field with the desired configuration settings.

`MX_SPI1_Init()` passes the ***hspi*** handle on to the next function after completing its task: **`HAL_SPI_Init(..)`**. The next function can be found in the `stm32f7xx_hal_spi.c` file in the HAL folder. This function aspires to insert the correct values into the peripheral SPI registers (the SFRs). Therefore it extracts the necessary information from the **`Init`** field.

main.c — USER

```
int main(void)
{
    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_SPI1_Init();
    ...


    while (1)
    {
        /* User code */
    }
}
```

```
/*********************************************************/
/*            Create peripheral handles                  */
/*********************************************************/

SPI_HandleTypeDef hspi1;
DCMI_HandleTypeDef hdcmi;
QSPI_HandleTypeDef hqspi;
SPDIFRX_HandleTypeDef hspdif;
...

/*********************************************************/
/*              SPI1 Initialization                      */
/*********************************************************/
void MX_SPI1_Init(void)
{

    hspi1.Instance = SPI1;
    hspi1.Init.Mode = SPI_MODE_MASTER;
    hspi1.Init.Direction = SPI_DIRECTION_2LINES;
    hspi1.Init.DataSize = SPI_DATASIZE_4BIT;
    hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
    hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
    hspi1.Init.NSS = SPI_NSS_SOFT;
    hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
    hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi1.Init.TIMode = SPI_TIMODE_DISABLED;
    hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLED;
    hspi1.Init.CRCPolynomial = 7;
    hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
    hspi1.Init.NSSPMode = SPI_NSS_PULSE_ENABLED;

    HAL_SPI_Init(&hspi1);

}
```

stm32f7xx_hal_spi.c — HAL

```
/*********************************************************/
/*                  SPI Initialization                   */
/*********************************************************/

HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi)
{
    /* Overly simplified, the code */
    /* looks a bit like this:      */

    hspi->Instance->CR1 = ...
    hspi->Instance->CR2 = ...
    hspi->Instance->CRCPR = ...
    ...
    /* The code simply extracts the information      */
    /* from the SPI_Init struct and uses it to fill  */
    /* in the corresponding registers (which are     */
    /* accessible through the handle as well).       */
    /*                                               */

    HAL_SPI_MspInit(hspi);

    return HAL_OK;
}
```

stm32f7xx_hal_msp.c — USER(!)

```
void HAL_SPI_MspInit(SPI_HandleTypeDef* hspi)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(hspi->Instance==SPI1)
    {
        __SPI1_CLK_ENABLE();

        /**********************************/
        /* SPI1 GPIO Configuration        */
        /*                                */
        /* PB5      ------> SPI1_MOSI     */
        /* PB4      ------> SPI1_MISO     */
        /* PB3      ------> SPI1_SCK      */
        /**********************************/
        GPIO_InitStruct.Pin = GPIO_PIN_5|GPIO_PIN_4|
                              GPIO_PIN_3;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
        GPIO_InitStruct.Alternate = GPIO_AF5_SPI1;
        HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    }
}
```

The figure also shows the last step of the initialization procedure. The **`HAL_SPI_Init(..)`** function passes on the ***hspi*** handle to the **`HAL_SPI_MspInit(..)`** function. This is where the physical pins of your microcontroller get allocated.