Embedded Programming

Ranim Tom

August 26, 2025

Contents

Todo list

Chapter 1

Data Representation

1.1 Introduction

The goal of this chapter is review and explain some common programming concept used in embedded context, such as:

- Data representation: binary, hexadecimal,$\cdots$
  - Construct memory map table of hardware peripherals
- Bitwise operation
  - To manipulate registers (set or clear some bits)

1.2 Source

- chapter 1 and 2 from [1],

1.3 Bit, Byte Word

- Computers handle info in terms of ***group of bits*** → the idea of ***byte*** ↔ a grouping of 8 bits
- There are also othre grouping in terms of ***multiple of bytes*** ↔ ***larger unit called word***
  - The word size is machine dependent ↔ on a x-86 archi word = 64 bits = 8 byetes, whereas arm-cortex M archi where word = 32 bits = 4 bytes
- Memory of computers and processor are ***byte addressable*** ↔ every byte in the memory has its unique addresse
  - Example in a micrcontroller: in stm32f407 discovery board, each peripheral registers has its unique addresse

1.4 Data Types in C

a types in

- *Read section 2.1 from [1], and try with other books*
- *To insert a table about the range of values*
- *Maybe see if there are some codes about limit value, min and max*

1.5 Number System and Bases

see section 2.2 in [1]

- Binary values are represtented in base 16 in modern computers
- Convert from base 2 → base 16: separate each 4 bits and take the equivalent hex digit, as shown in the conversion table

version ta-

Insert the conversion table later

1.5.1 Base Representation in C code

- There is what we call ***suffixes*** → force the compiler to treat a constant as an explicitly specified data type
- Used suffixes in embedded programming:
  - `0x` for base 16.
    Example: `uint p = 0xFFFF`
  - `UL`: `U` for unisigned and `L` for long int.
    Example: `0xFFFFUL`

Chapter 2

Introduction

Goal of this Reader: The purpose of this pseudo book is to document my programming simulation with STM32 microcontroller family.

Micrcontroller Used: The microcontroller will be used in STM32F103 B (blue pill).

Adapting Introduction

Chapter 3

Review of C

3.1 Goal

Purpose of this chapter is to review the C language, but with context of embedded programming $\leftrightarrow$ we will focus on features of C which are used in embedded programming (pointer to access registers, bit wise operation,$\cdots$).

3.2 Github and Repository

Course Repository: https://github.com/niekiran/Embedded-C/

3.3 GNU tools for windows

Installing gcc compilter (which is from GNU tools) is not straightforward on windows. The steps are:

1. Install the `msys` executable
   - Go to https://www.msys2.org/ to install the .exe installer
2. In the command prompt type `msys2`, and it will open an msys shell
3. In the msys shell type the commands:
   (a) `pacman -Syu`
   (b) `pacman -S --needed base-devel mingw-w64-x86_64-toolchain`

3.4 Board

We will use STM32F407G-DISC1 Figure 3.1.

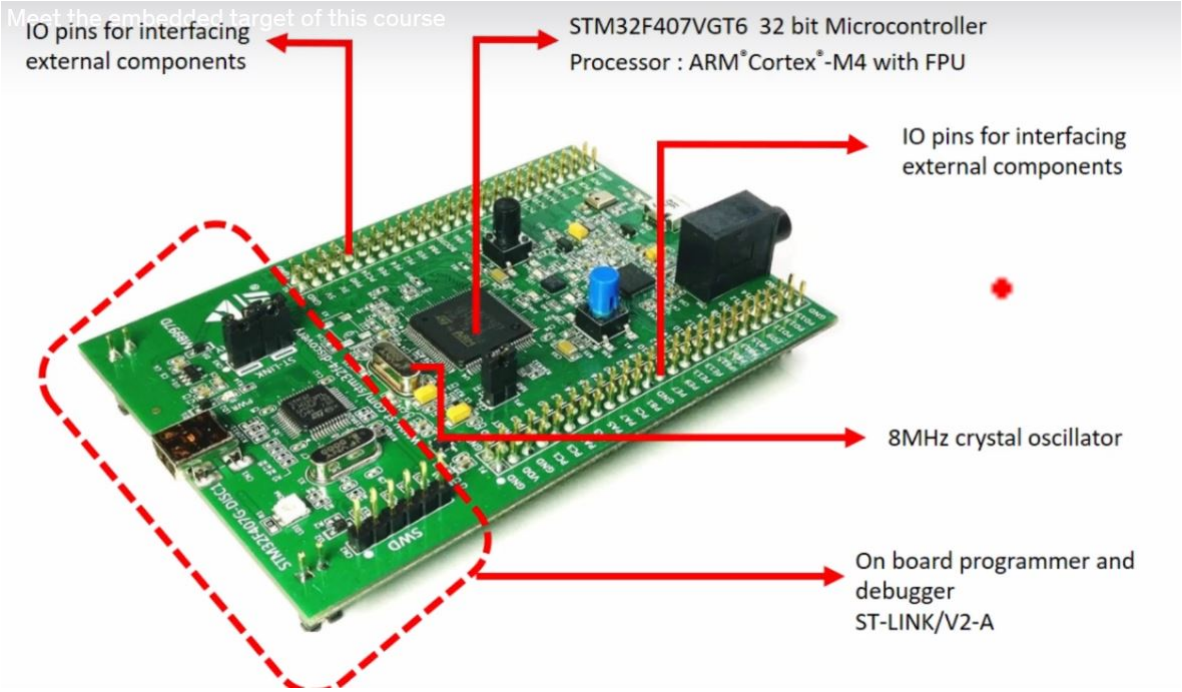


Figure 3.1: STM32F407G-DISC1 Main Components

3.4.1 Cable Type

We also needs usb cable: type A plug to mini type B plug.

3.5 Creating Project in STMCube IDE

In this section we are going to create C projects on 2 type of machines: the host (our computer) and the stm chip (as shown in Figure 3.2)

Figure 3.2: C Project types: either on a host machine or the embedded target platform

In StmCube IDE, we need to create 1st what is called a *workspace* for the projects, then we can create the source file codes (`main` file, function,$\cdots$) either for the host or the stm chip. Let's see the steps of creating a workspace first.

3.5.1 Creating a Workspace

The steps are:

1. We choose some directory (mine is called `Stm32f40g Projects`), and inside the directory we create 2 directory also called: `Host` and `Target`(for the stm)
2. We open the IDE, it will give us a prompt to choose the path for the workspace
   - For each type of project (`Host` and `Target`) we create a workspace (so we will have 2 workspace)
3. After we select the path for each workspace, we follow the steps in 3.5.3 or 3.5.4

Note that when finishing creating a workspace for some directory, we will a `.metadata` folder associated with this workspace.

3.5.2 Importing Existing Projects

Another useful thing to learn is how to import some existing project into our workspace. This can be useful if for example:

- We are cleaning up some existing directory which contains allot of projects, and we need to move them to some new created (clean) directory.

  This new clean directory will have its own workspace (see 3.5.1)

- Or if we download some new code from others to learn from them, and we need to use them from our code

Now concerning the steps for importing an existing projects are:

1. Launch the IDE and open to created workspace (this should be created before as discussed in 3.5.1)
2. From the tabs, click on `File --> Import`
3. A new window will appear, select on `General --> Existing Project into Workspace`
4. A new window will appear. Using this window we need to do 2 steps:
   (a) From `Select root directory`, browse to the path containing the project (this should lead to root folder containing all the projects)
   (b)
   (c) In the option, select `Copy project into Workspace`

3.5.3 Creating a project for Host

The steps are:

1. Go to `File --> New --> C/C++ Porject`
2. A pop window appear, choose `C Manage Build`
3. Give the project a name
   - *Note*: when giving a name for a source file or a function file, we must include the `.c` extension to STM32Cube IDE (as example :`name_example.c`), so it can create the file
4. Choose from Toolchaine `MinGWGCC` for windows machine (or `LinuxGCC` for Linux machines)
5. Click `Next` then `Finish`
6. A project will appear on the right hand side, then what we do is right click and `New --> source` and give it a name.
7. Building: after finishing writing the code, we need to build the project.
   Right click on the project name, and select `Build`
8. Running the project: right click on the project, select `Run as --> Local C/C++ project`

3.5.4 Project on Board

Steps are:

1. `File --> New Stm32 Project`
    - It will open *Target selection menu*
2. Select board, and type the stm board name *on the Board Selector menu*
3. Click on Next, then on the project setup,
    - Give a name to the project
    - Options:
      Target lanaguage: C
      Targeted binary type: executable
      Targeted project type: Empty (this is important, we don't select STM32Cube unless we know how to work on smt32cube MX software)

3.6 Data Types in C

In Figure 3.3, we have different data types along with their ranges

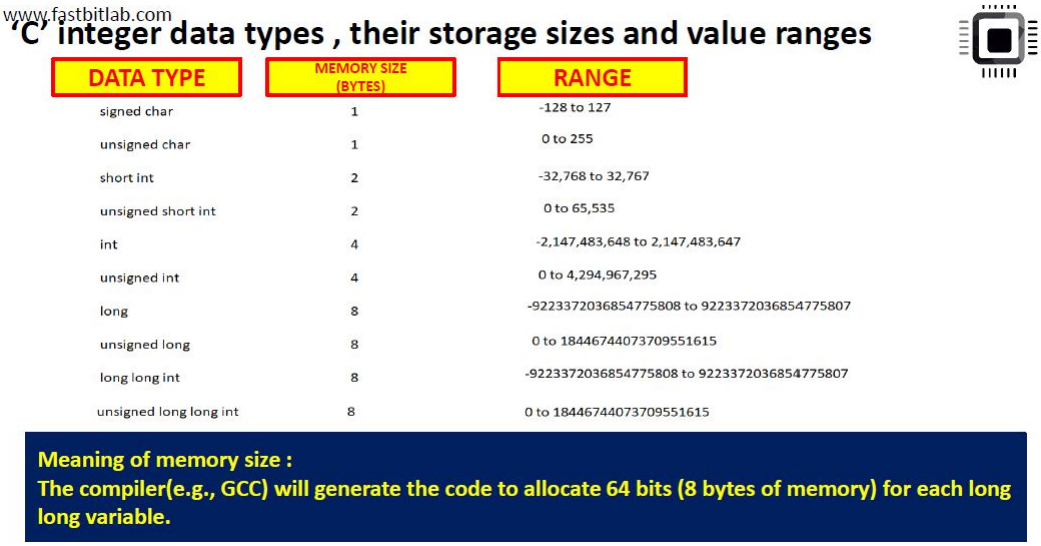


Figure 3.3: Data Types in C

- The important thing is to know that memory sizes are compiler dependent: C standard only specify the max and minimum range, then every compiler designer specify the correspondent size for each data type.
- However, there are some data types which remains the same for whatever type of compiler we use
  - `short int` and `unsigned short int`: consume 2 bytes of memory

3.7 Variables in C program

In C program, variables names are just giving meaning to use as programmer, whereas in the code, the compiler won't use these names, instead it will use ***addresses*** to put each variable in its correspondent location with its correspondent value (see Figure 3.4)

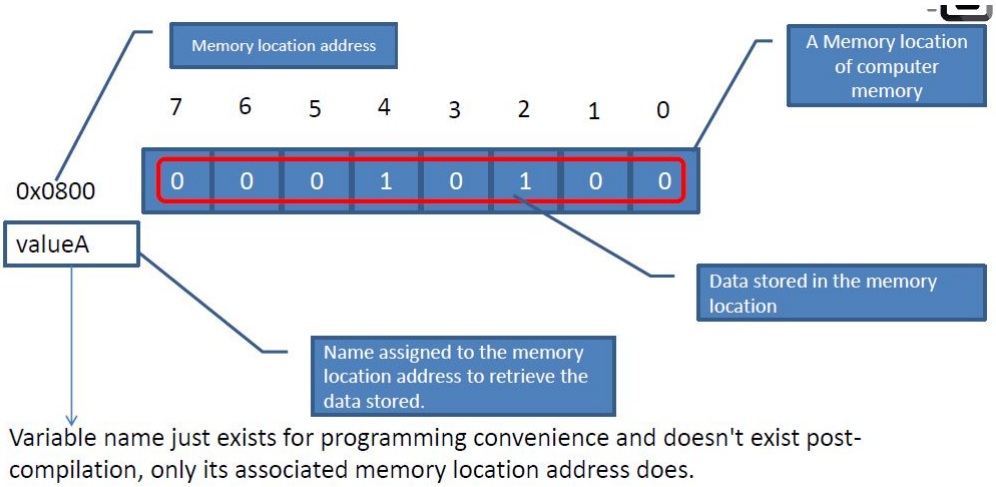


Figure 3.4: Variables and Memory

3.7.1 Declaration Vs Definition

- variable defined ↔ compiler allocate storage for this variable
- variable declared ↔ compiler informed that variable exist but no allocation is done yet (see Figure 3.5)

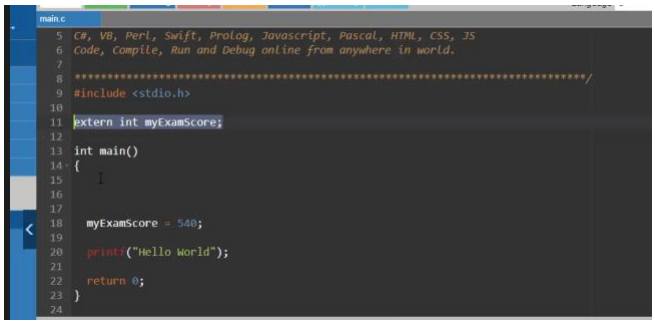


Figure 3.5: Variables and Memory

The `extern` keyword tell the compiler that the variable is defined outside the `main.c` file

3.8 Type Casting

Casting: *To redo when reviewing the arithmetic system later in logic design*

Big idea: when not paying attention to data types, we may ***loose information***. Example when doing division between 2 integer, the result will give an integers.
The solution is either:

1. Type cast 1 of the variable to float: `float result = (float) int a / int b`
2. Or use directly float data type on the variables `a` and `b`

There are also other examples for loss of information when manipulating binary/hex numbers (*to redo them later*)

3.9 Hello Stm32

In this section we will see how to write code on stm32 board.

Note: Review subsection 3.5.4 to create a project.

About the hierarchy of the project:

- stm32 project include many folder
  - It will give us an `include` folder where we put our header files, and also a `main.c` file to write our source code
- Every microcontroller include a ***startup folder***
  - We will explore the startup code later once we understand more about microcontrollers

3.9.1 Printing Hello world

Now we will try to display a message hello world on the microcontroller. However, we don't have a scree on the STM32F407G-DISC1 (see Figure 3.1).

The solution for this problem comes from ARM cortex, specifically ARM cortex M3/M4/M7 or higher processors.

Let's take a look at Figure 3.6.

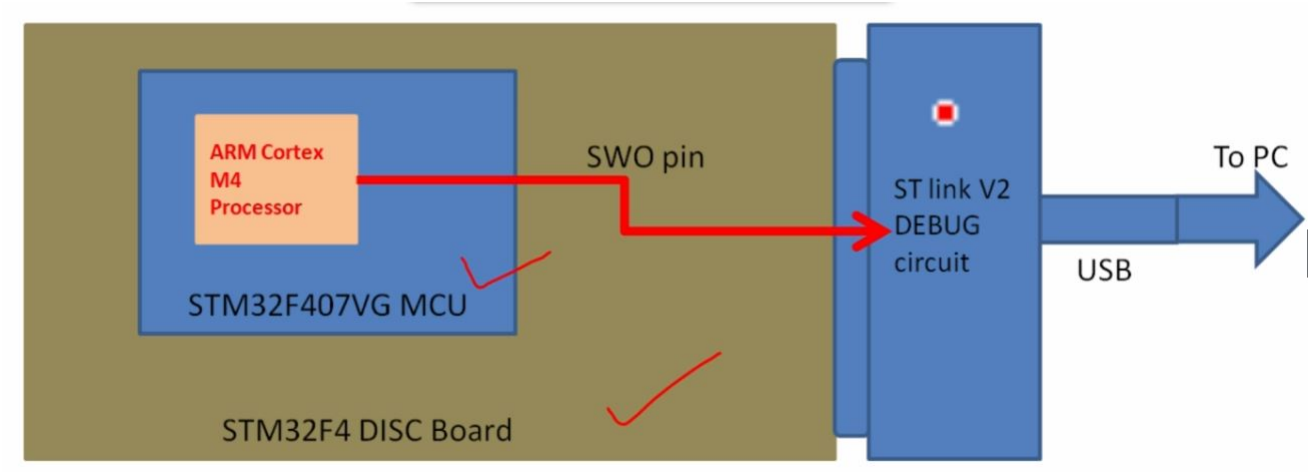


Figure 3.6: Stm discovery board: the SWO pin and st-link debugger

- We have the Stm discovery board, and inside the board we have the ARM processor
- At the front end of the board, we have what we call ***st-link***: this is a circuitry in which we use to communicate our pc/programm to the micronctroller.

  Through the st-link, we can write program to the internal flash of the mircrontroller, read various memory location from the mircrontoller, make the micro run and stops.

  All debug action are done via the st-link debug circuitry.

- The st-link debug circuitry talks to our pc through a USB connection

Now we will zoom in inside the ARM cortex M4 only, shown in Figure 3.7.

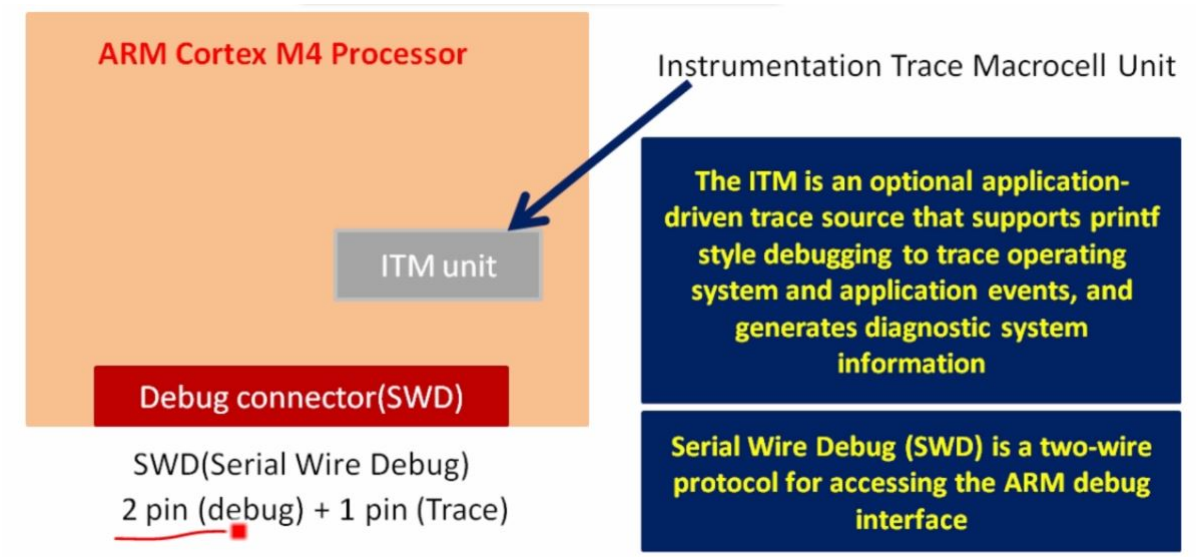


Figure 3.7: Stm discovery board: ARM cortex zoomed

- We have 2 units: the ITM unit and the debug interface, *SWD interface*
- The SWD interface: it is a debug interface which consists of 2 lines:
  - The SWDIO line: bidirectional line responsible for the debug information (like when putting a break point,$\cdots$)
  - The SWCLK: a clock driven by the host. In our case, the host is the st-link debug circuitry

Understanding the printf tracing debugging:

Now in order to understand how the `printf` for debugging works, Figure 3.8 shows the steps:

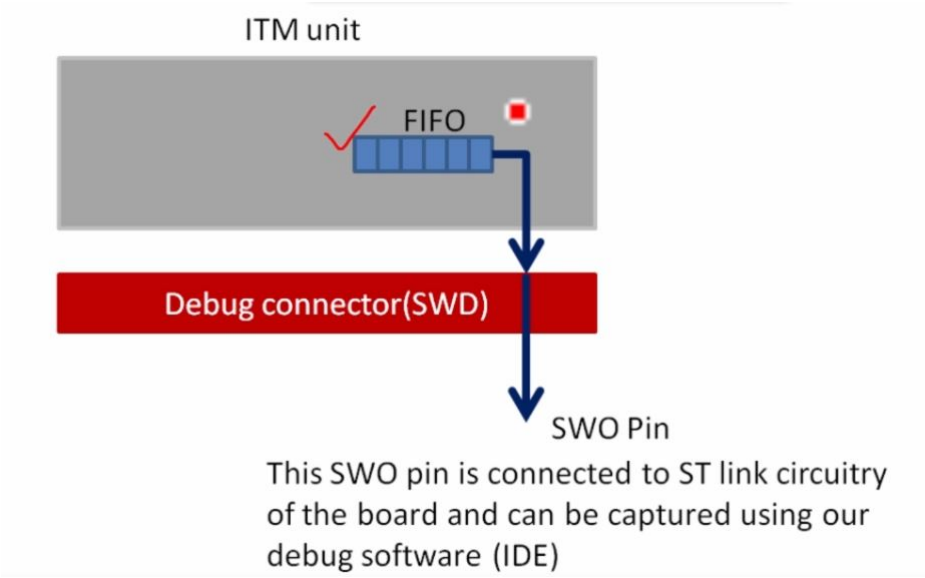


Figure 3.8: Stm discovery board: ARM cortex zoomed

- The data inside the `printf` will go inside the ITM unit, using the FIFO register/buffer
- Then it will be communicated using the SWO pin, which in turn connected to the ST-link and it will be captured by the IDE

Note: after creating the project, we need to go to and insert the code in `Src --> syscall.c`.

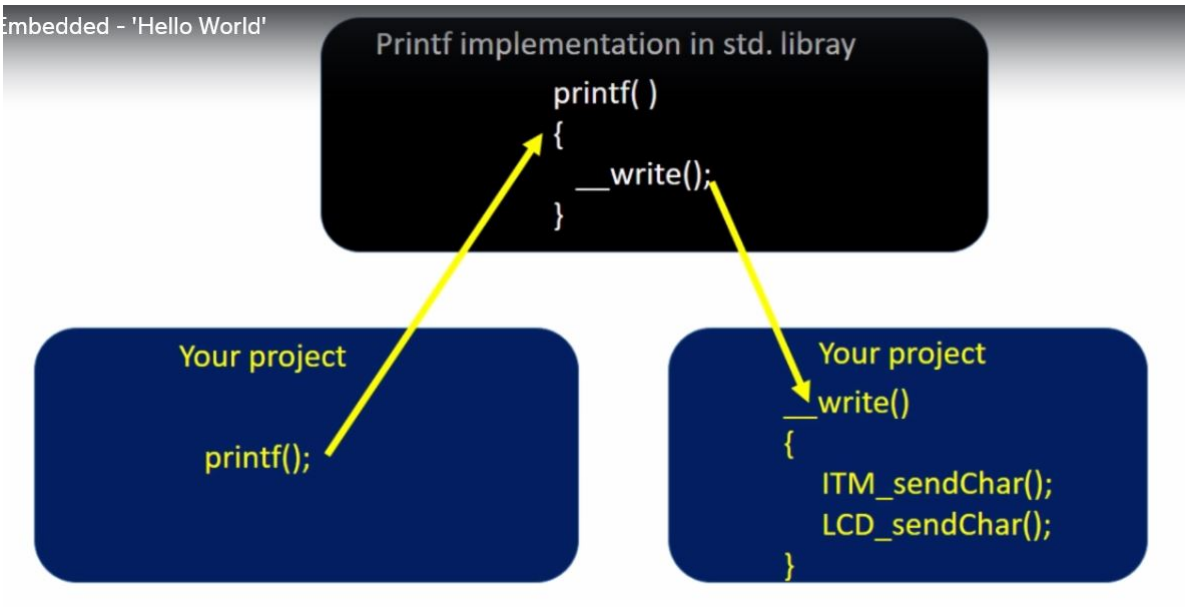


Figure 3.9: Printf call function stack

3.9.2 Cross Compilation

Now after saving the files of the source code, we need to build the project, and make what we call a ***cross compilation***, as shown in Figure 3.10.

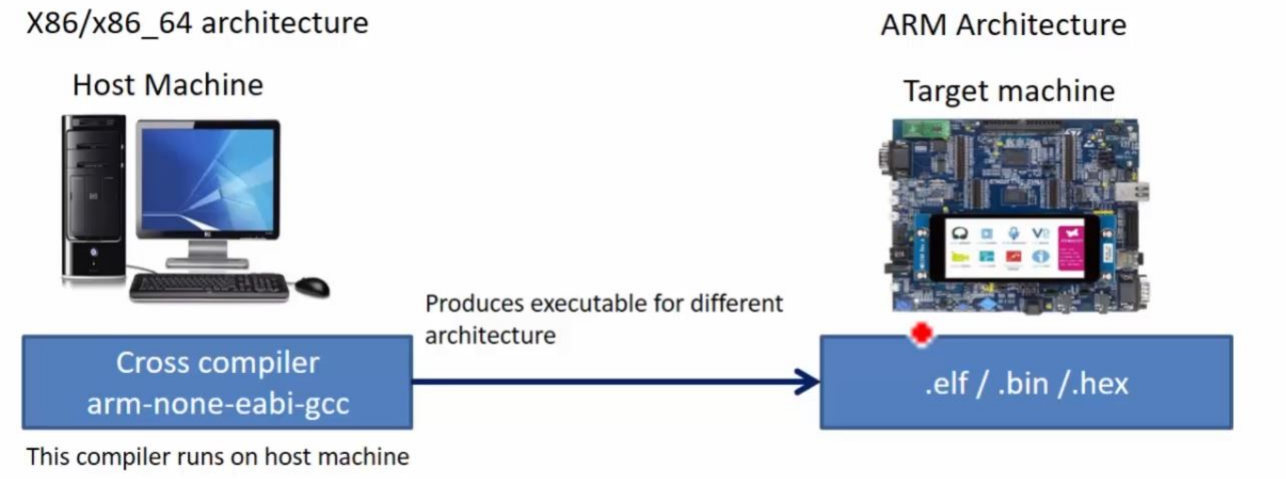


Figure 3.10: Cross Compilation: From host → to target stm32f40g

- Cross compilation means that the executable code will be uploaded or done for a different architecture
  - In our case, the cross compiler name is called arm-none-eabi-gcc, which is installed in the IDE and the toolchain
  - Many type of executable format will be produced during the cross compilation:

* `.elf`: executable for link format, used in debugging
* `.bin` and `.hex`, used for production

The opposite of cross compilation is ***native compilation***, shown in Figure 3.11.

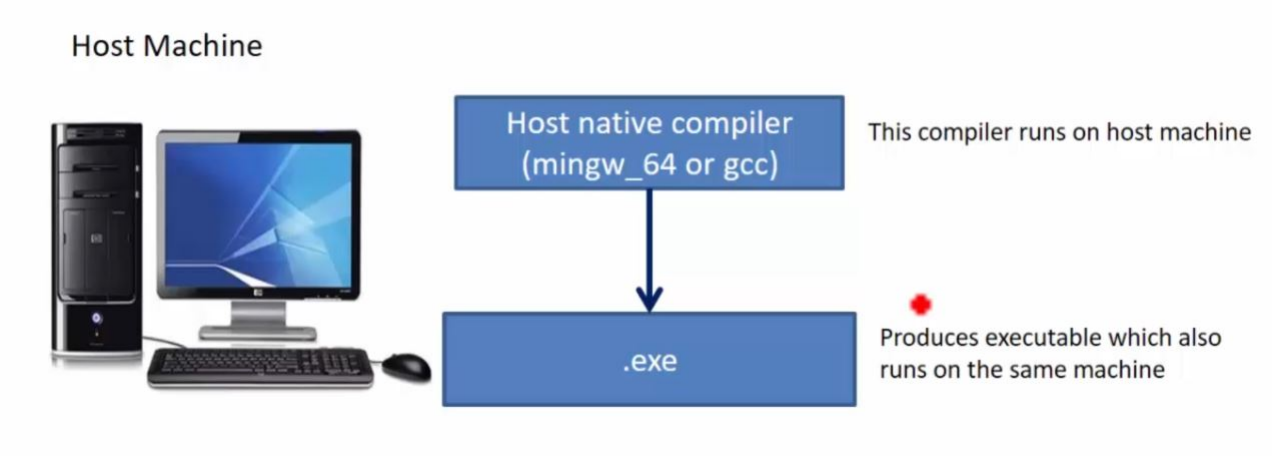


Figure 3.11: Native compilation

3.9.3 Steps of building the project

The steps for printing hello world on stm32 board:

1. Create a new project on the ***target board*** (review subsection 3.5.4 to create a project)
2. From github https://github.com/niekiran/Embedded-C/blob/master/All_source_codes/target/stm32f407_disc/001HelloWorld/Src/syscalls.c, add the snippet code about the implementation of `void ITM_SendChar(uint8_t ch)`
   - this function will send data to the board using ITM unit (see subsection 3.9.1)
3. Build the project
4. Now we need to load the project to the board.

   Connect the board to the pc
5. Right click on the project, `Debug As --> Debug Configuration`
6. A pop window will appear , click on the left side `STM32 MCU Debugging`, it will create a debug configuration
   - On the tabs, click on `Debugger`, then `Debug probe` select the option `ST-LINK GDB Server`
   - In `Interface`: it is `SWD`
   - In `SWV`: we click for `Enable` and we leave the clock options as it is
   - Finally we close it

7. The right click on the project again, select `Debug As --> STM32 MCU C/C++ Application`
   - It will load the project on the board, and it will tell us to switch to debug perspective (we select `ok`)
8. Now in order to see the `printf`, we go to the `Window Tab --> Show View --> SWV --> SWV ITM`
   (a) It will open the ITM console
   (b) Click on the configure button (the tool logo), and select from `ITM Stimulus port` port 0
   (c) Click from the ITM console start trace (red circle) to accept data
   (d) From the tab bar under `Navigate`, click on `run the code`

3.9.4 Back to Editing mode

Once we finish up debugging, we need to go back to editing mode. This can be done using the button `Teminate` (red square under `Search Tab`)

- Note: In new version of stmcube IDE, there is `terminate` button in the ITM console also, but it produces some bug and error, so don't use it and use the `terminate` button under the `Search` tab (see video 56 in section 8 to handle this error)

3.9.5 Open OCD

For older version of ARM cortex, we can use the Open OCD, which stand for *open on chip debugger*.

See video 57 on section 8 later to write it (for now I won't use the open OCD since the SWD works).

3.9.6 Some Extra Settings

Let's see now how to explore data location and variable in memory.

Right click on the project name,select `properties --> C/C++ Build --> Settings`, and it will open to us multiple option concerning the IDE and MCU.

Note: there is nothing important for now, see video 60 in section 8 to hear their description, we will examine each of the feature and the settings later.

3.10 Build Process

Now let's try to understand the build process. It is composed from various steps:

- Preprocessing
- Parsing
- Producing Object files
- Linking object files
- Producing final executable
- Post processing on final executable

The above steps can be divided into 2 major steps: compilation and linking.

Compilation:

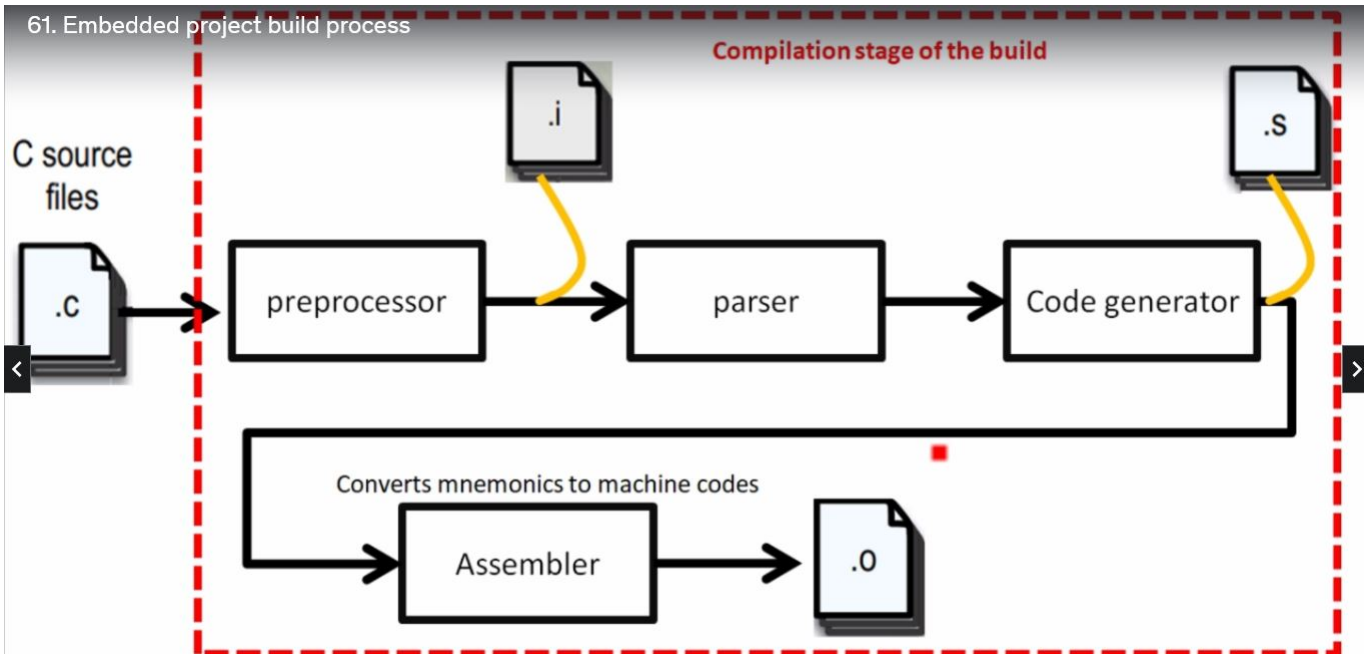


Figure 3.12: Compiler Steps

- The preprocessor execute the # statements
- Parser will ensure that syntax is correct
- Code generation: every C line code will turn into an assembly code
- Then the assembly code will turn to machine code `.o`
  - Note: every C file will have its associated assembly file code `.o`

Then it comes the linking stage.

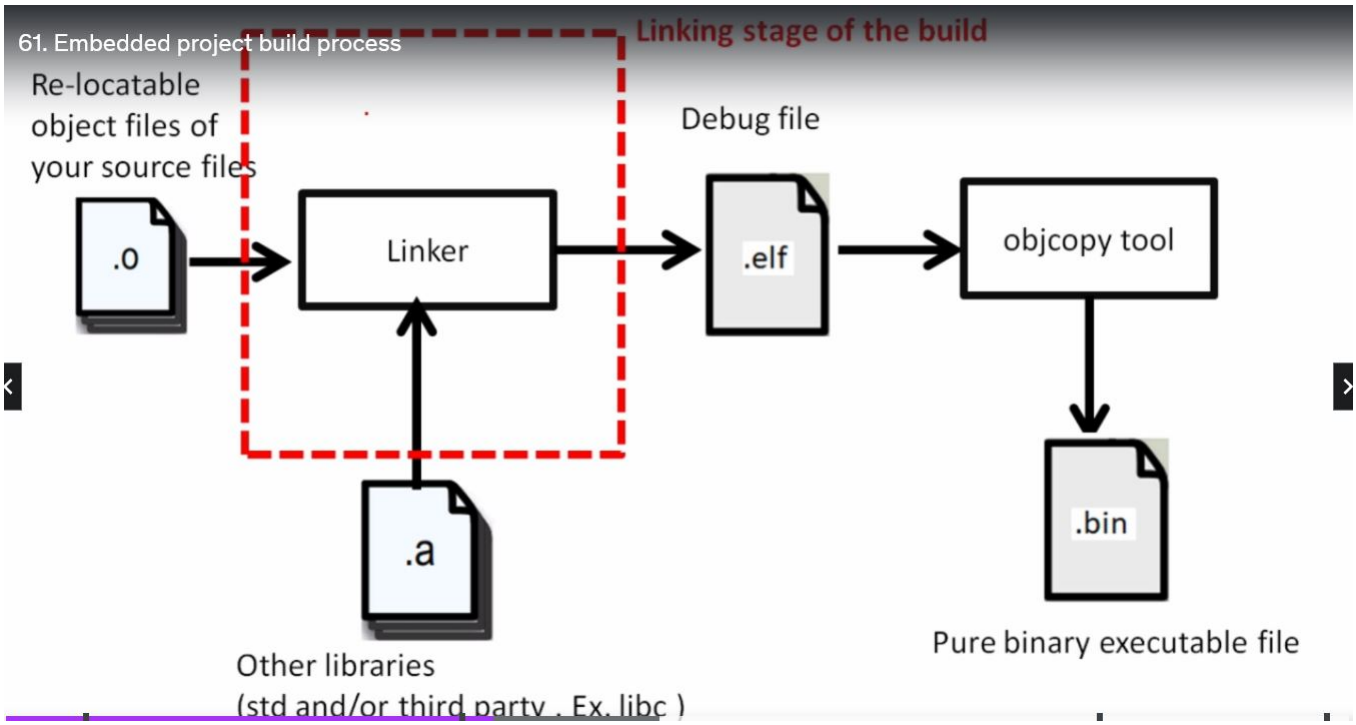


Figure 3.13: Linker Steps

- All the `.o` files will merge into 1 executable file: a `.elf` file
- Then a postprocessing step using tools like objcopy will turn the `.elf` to a 1 final binary file

3.11 Analyzing C embedded Code

Now we will take some C embedded code written for a MCU, and try to understand it. The list of points we try to uderstand are:

- Anatomy of microcontroller
- Identifying code and data parts of the program
  - Place of code in memory and data in memory
- Disassembly feature using the IDE
- Analyzing the executable file `.elf` using GNU tool such as objdump and size

3.11.1 Anatomy of microcontroller

A microcontroller is a small computer system on a ship, but with small resources compared to a desktop computer, because a microcontroller target embedded application. In Figure 3.14, we have the different resources found on the ship of a microcontroller.

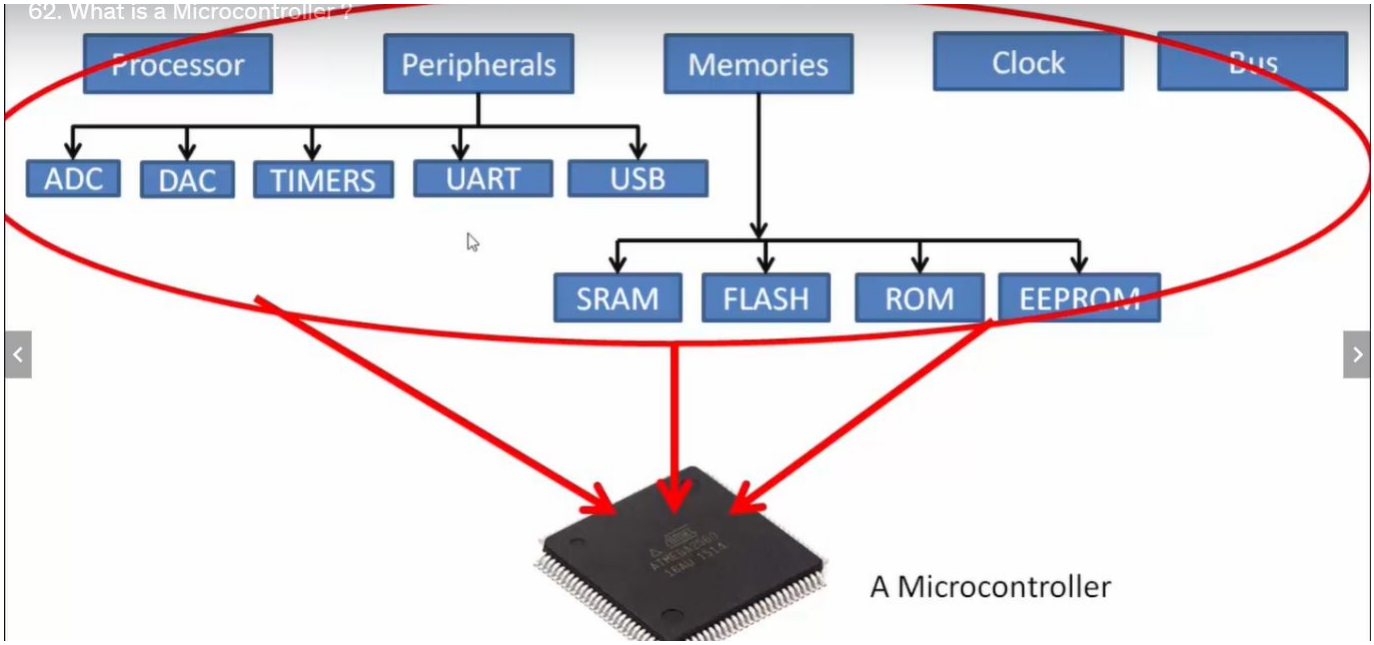


Figure 3.14: Resources implemented inside a microcontroller ship

- In peripharls: some modules for *connectivity* like UART and USB, for *time* like TIMERS
- Memories: Volatile memory like SRAM, non-volatile like FLASH,ROM,EEPROM

In Figure 3.15, we have a simple block diagram of a MCU.

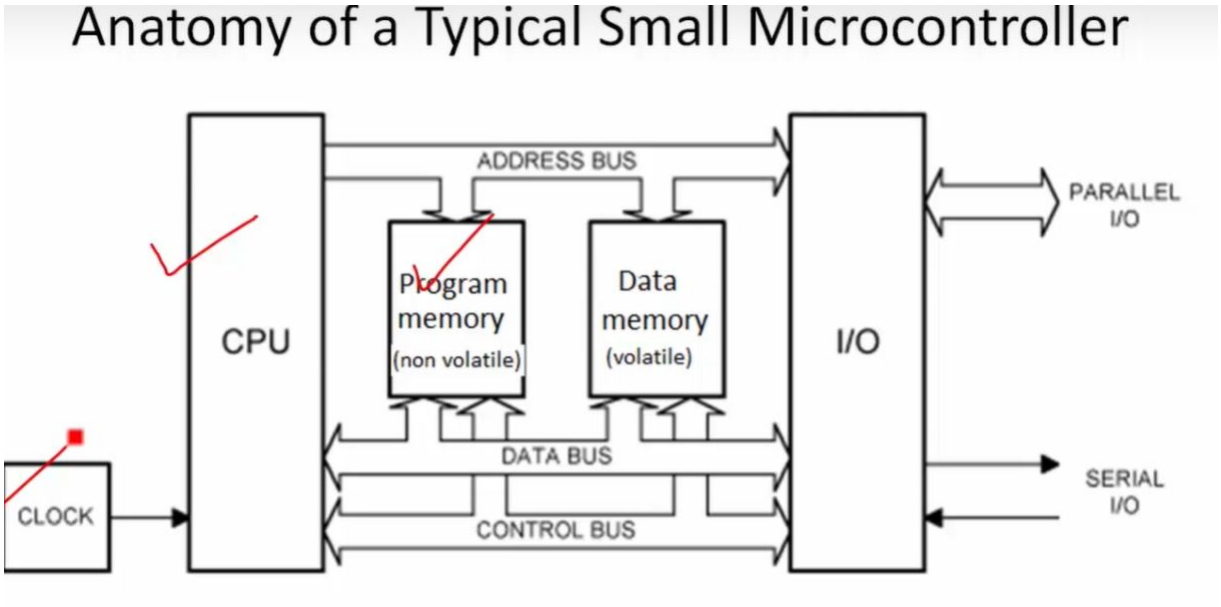


Figure 3.15: Block diagram of MCU

- Every MCU contains a CPU (a processor) which produce instructions
  - These instruction are stored inside the program memory (which are non-volatile)
  - The speed of instruction handling depends on the clock
- It contains I/O pins to communication with external world

Now we take some MCU used in industry.

- STM32

  The CPU of STM32 MCU are produced by arm company: it is an arm cortex CPU based, so it is not stm microelectronics who synthesised the CPU

3.11.2 Code Memory

Code memory is where we store our code and constant data, and the memory type is non-volatile.

There are many type of code memory circuity, like ROM (alos ROM have different types such as EEPROM,MPROM,$\cdots$), FLASH (which is used in stm32 MCU units, and they are very cheap).

Another type of memory is , which is very fast in access time, thus making the price of microncontroller to shoot up.
In stm32 reference manual, chapter 3, the FLASH is of size 512 KB, and it is also divided into several sectors/block.

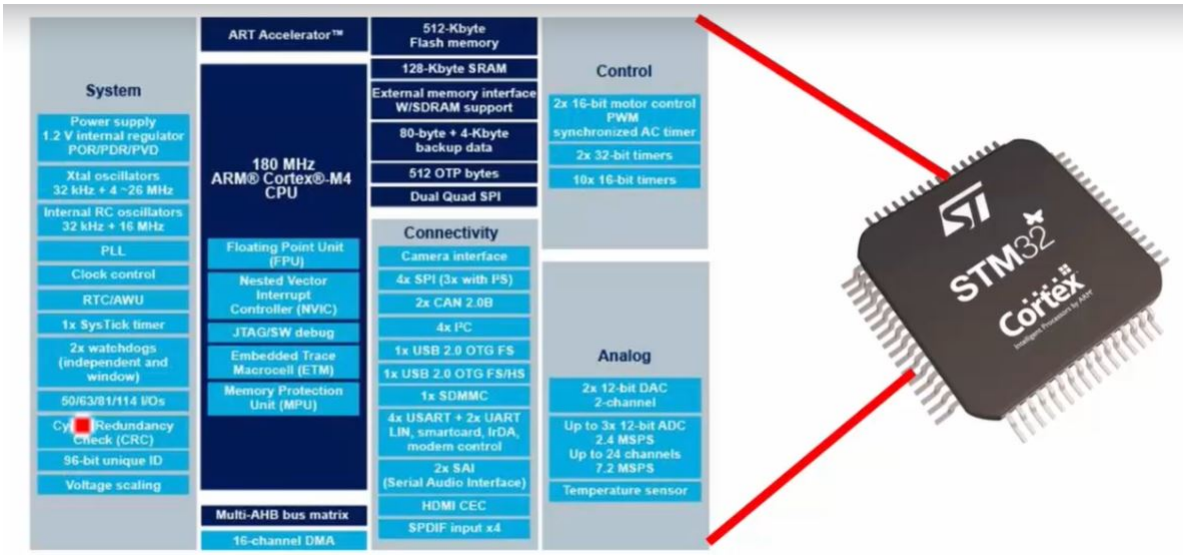


Figure 3.16: Block diagram of MCU: STM32 MCU units

3.11.3 Tracking variable in the memory

Refer to the `main` of the project named `Tracking-variables-memory`.

1. Once we finish writing the code, we need to ***build the project*** in order to have at the end the `.elf` file
2. To load the code to the microncontroller, we need to

   `Debug as --> STM32 MCU C/C++ Application`

   - Review setting of 3.9.3.
   - A console log window will appear and tell us that `Download verified successfully` More specifically, the data will go to the `FLASH`

3. First, we need what is called ***the base address*** of the `FLASH` memory.

   We go to the reference manual, Table 5 in section 3.3, shown in Figure 3.17.

Table 5. Flash module organization (STM32F40x and STM32F41x)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	. . .	. . .	. . .
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Figure 3.17: Flash Memory organization

   - It is a total of 512 KByte memory
   - It is a read only memory, also we call it ***system memory***

   The base address start from the 1st line `0x0800 0000`, and we can use the memory till sector 11, of final address is `0x080F FFFF`

4. To see the content of the `FLASH` (or any other type of memory), we go to `Window --> Memory` or `Memory content`, then type the base address `0x0800 0000`.

5. Now the `FLASH` is for the program code.

   For the data code, we need the address of `SRAM`

   - In stm32 we have 2 `SRAM`: `SRAM1` the main one, and `SRAM2` an auxiliary one
   - There addresses can be found in chapter 2 in the reference *to search for it later*

AM

Note: the data get transferred from `FLASH --> SRAM` by routines and handler written in assembly in the `Startup` folder of the project.
See video 65 for more details to navigate through the content (but it is not important for now).

3.11.4 Disassembly

Now we will see how to disassembler work in the IDE. First we need to go into debug mode, by `Debug --> Stm32 C/C++ Application`, then we go to `Window --> Show View --> Disassembly`.

It will open to us an disassembly window, showing us the assembly code of our program. The goal of such debugging is if we want to optimize at a instruction level, or understand what is happening at the assembly level.

We can do 2 types of debugging:

1. C code debugging: by inserting break points
2. Assembly code: by inserting break points in the disassembly window or by enable instruction mode (the i logo with $\rightarrow$ under the `Run tab`)

Review video 66 and 67 from section 10 for more details.

3.12 Floating Point numbers

Now we move to how real numbers are stored in the memory.
In computers, real numbers (with comma) are stored according to IEEE75 floating point system.

Goal of floating system: in case we are working with very small (charge of electron) or very big (distance between earth and sun), we can't use type `long long` for example, that's why we use floating point number.

3.12.1 IEEE75 floating

Suppose for example we have to store $+7.432 \times 10^{48}$. We can't store its binary equivalent because it will consume allot of memory.
Instead, we store the important part as shown in Figure 3.18

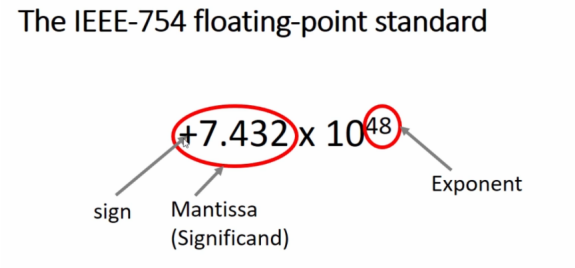


Figure 3.18: Floating Point Example

By this way, we will *approximate* the number and store only the important part in the memory, which are the essential components to give the value of the number.

How we store:
There are 2 ways to store the main part of Figure 3.18:

1. Single precision shown in Figure 3.19.

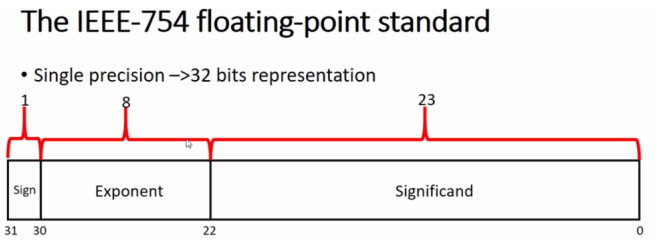


Figure 3.19: IEEE-745 Single Precision

For float: storage is 4 bytes, precision up to 6 decimal places, and range from $1.2 \times 10^{-38} \rightarrow 3.4 \times 10^{38}$

2. Double precision (more accurate but consume more memory) shown in Figure 3.20.

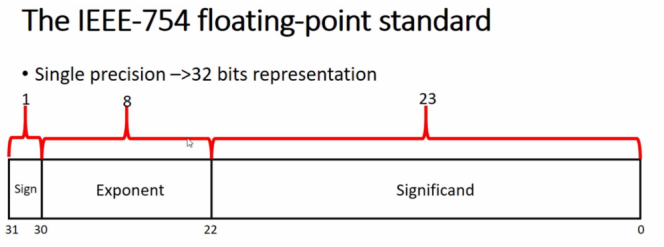


Figure 3.20: IEEE-745 Double Precision

For double: storage is 8 bytes, precision up to 15 decimal places, and range from $2.3 \times 10^{-308} \rightarrow 1.7 \times 10^{308}$

Format Specifier: `%lf` for double and `%f` for float.
An example for handling floating point using charge of electron is shown in Figure 3.21.

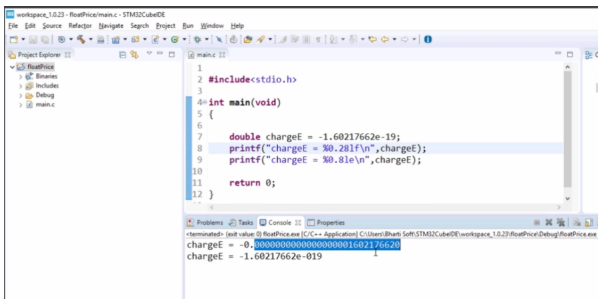


Figure 3.21: Example Code for floating point

Some nice thing for small number is the formatting using scientific notation, using the format specifier `e`

3.13 Scanf

When using `scanf`, in some console it causes the console to store in the output buffer. In other words, the output stream get stucked because we are waiting for user command. This is because when using the `printf` function, the stream doesn't go directly to the console, but it goes to some output buffer, where we have a buffer API which output the stream to the console, as shown in Figure 3.22.

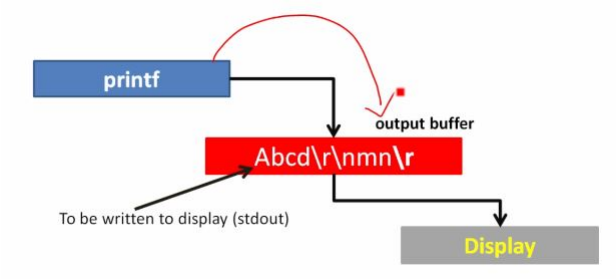


Figure 3.22: Flash Memory organization

In order to output the stream from the output buffer to the screen, we use the function `fflush(stdout)`.
Note: video 72 contain nice explanation about input buffer for storing strings. To understand it later.

3.14 Pointers

Pointers are used in embedded programming in order to access peripheral, read/write data to them.

To recap, pointers are just addresses of variables, as shown in Figure 3.23.

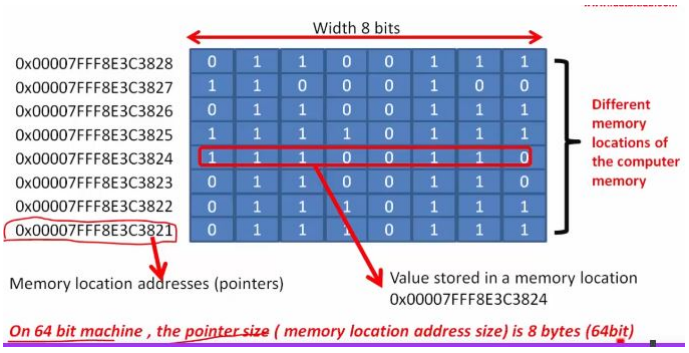


Figure 3.23: Pointers

Operations: we can decrement a pointer to reach memory addressee beneath current location (or increment, to go up in memory), and we can access the data in memory via the pointer.

Steps:

1. Initialize the data `int data = 5`
2. Pointer assignment to data `int* p_data = &data`
   - The pointer data type is the same as data type (pointer pf type int, so the data is of type int)
3. Access the content via the pointer `value = *p_data`, where `*` is the dereferencing operator

Important notes about pointers:

- Whatever the data type, the pointer content (addresses) have 8 byte
  - This means that `int*` and `char*` will have 8 bytes addresses
- But the value will be different
  - `int* p_data = &data`, `*p_data` will have 4 bytes because we have `int` data type
  - `char* p_data_2 = &data_2`, `*p_data_2` will have 1 byte because we have `char` data type

Pointer Arithmetic

: *To redo video 63 on pointer arithmetic*
Take away: when incrementing pointer by 1, we move by the size of the data type.

Example: if we have `int* p_data`, that is a pointer to `int`, the +1 will be an offset of 4 byte because size of int is of 4 bytes.

3.15 stdint and Portability

Suppose we wrote come C code, and we are testing our program with 2 compilers (because we are using 2 hardware for example, smt32 and PIC for example). Each of the hardware have its own compiler. Although the C code is correct, our code may have some buggy issues due to compiler design, that is each designer will design different byte size for data type according to the specific hardware architecture, which differ from hardware to hardware.

For example, for MPLAB compiler (for the PIC microcontroller), the `int` data type has 2 byte allocation (shown in Figure 3.24), whereas the in stm IDE the int data type has 4 byte allocation.

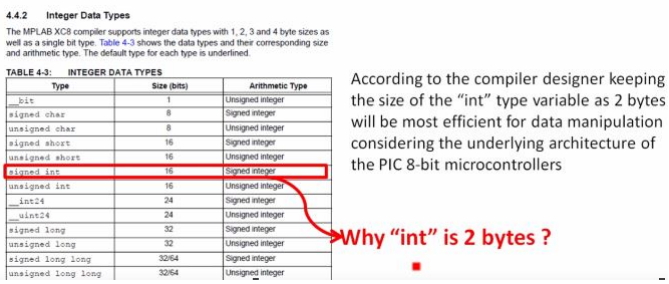


Figure 3.24: Data Allocation

The reason is also behind this freedom is that C standard have given the freedom to compiler designer to choose based on optimum hardware design for data type `int` and `long`
Take an example the code shown in Figure 3.25.

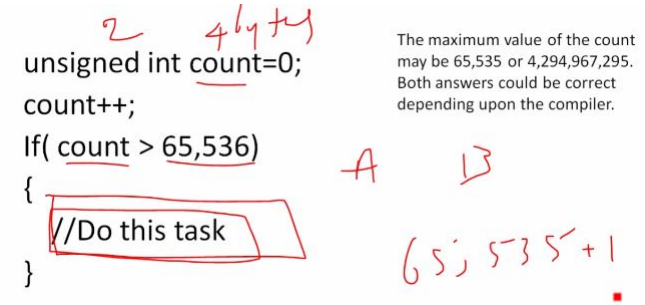


Figure 3.25: Code portability

If `unsigned int` have 2 bytes, then the max is $2^{16} - 1 = 65535$, and when reaching 65 535 +1, the `while` loop won't enter and the code won't work anymore.

To avoid such problems, we will instead of true variable data type, their aliases in the `stdint.h` file, as shown in Figure 3.26.

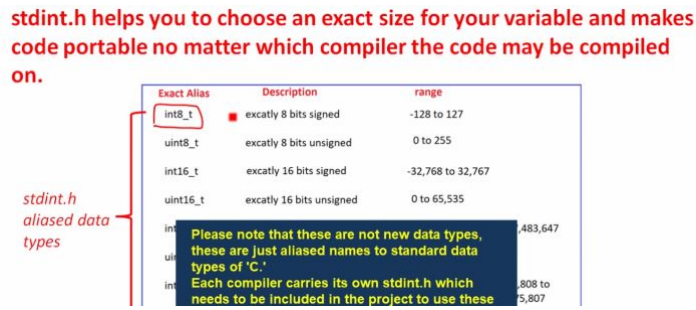


Figure 3.26: Code portability

Some of useful aliases data type which we will use in programming are shown in Figure 3.27

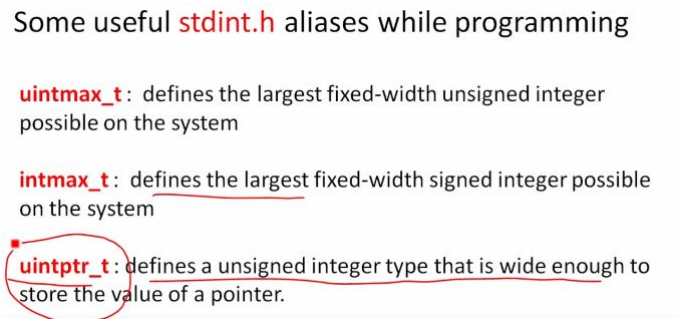


Figure 3.27: Code portability

3.16 Bitwise Operation

Bitwise operation plays an important role in embedded programming, accessing registers,· · · . They are built up upon our understanding of number systems and logic gates.

First a recap about logic gates:

Symbol	Description
|	OR gate: 1 if any operand contains 1
&	AND gate: 1 if the 2 operand contains 1 at the same time
^	XOR: 1 if 1 of the operand contains 1, but not both (we exclude this case)
~	NOT operation
<<	left shift operator: left shift by some number of bits
>>	right shift operator: left shift by some number of bits

Table 3.1: Bitwise Operation used in embedded programming

Note: the >> and << are important because they enable us to set bits in registers without affecting the other bit positions. The other bits maybe connected to other things or we don't know what they do.

Another point, that in C language, we have also logical AND (&&), logical OR (||), and logical NOT (!).
The difference between them and operator in Table 3.1, is that these operator act directly on non binary level, whereas bitwise are in bit level (example for OR is shown in Figure 3.28).

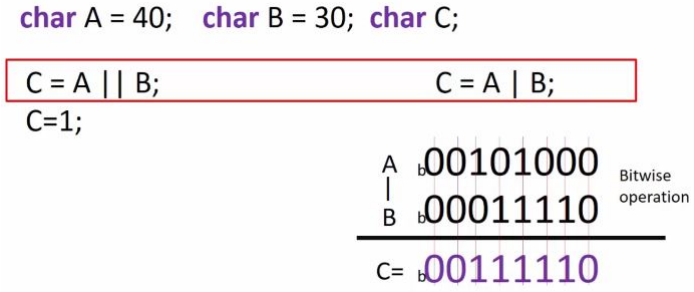


Figure 3.28: Difference between logical OR and bitwise OR

Usually we have 4 operations in embedded programming:

1. Testing bit (&)
2. Setting of bits (|)
3. Clearing of bits (~, &)
4. Toggling of bits (^)

3.16.1 Testing bit

Suppose we have some number, and we want to test some bits in some position x. An example is shown in Figure 3.29.

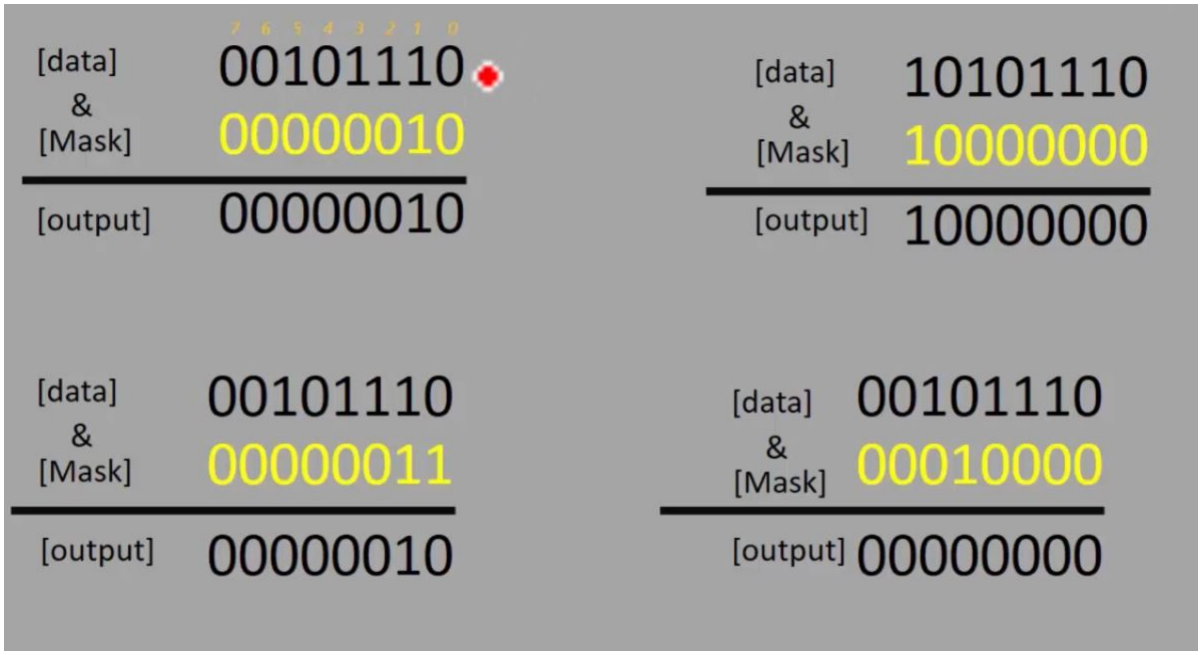


Figure 3.29: Testing bit using and operation and masking

- In the 1st example, we want to test bit in position 1, so we set the mask in this position to 1 and all other 0
- This is because x & $1 = x$, so we know the identity of x, and for other bit position we don't care, we set the mask to 0

3.16.2 OR

Now we will take some 8 bit register example, and we to say that we want to set 1 in the 1st bit field of this register.

The syntax for doing so: `PortB | mask`. An example is shown in Figure 3.30.

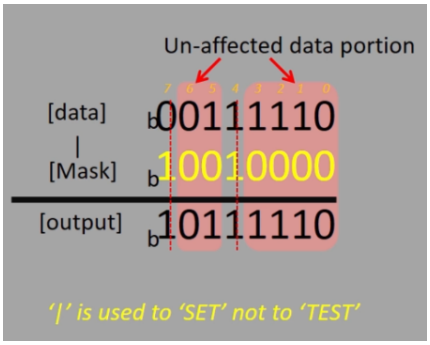


Figure 3.30: Setting bits using OR

OR masking to remember: The key thing about OR masking is that the mask 1 column will give us 1 no matter the content in the port, and the remaining will be unchanged. So the OR is a good operation to set some bits to 1 and keep the rest unchanged.

3.16.3 XOR

The XOR operation can be served a flipping, same as the OR: it flipping the bit in the column mask, and the other remains unchanged. An example is shown in Figure 3.31.

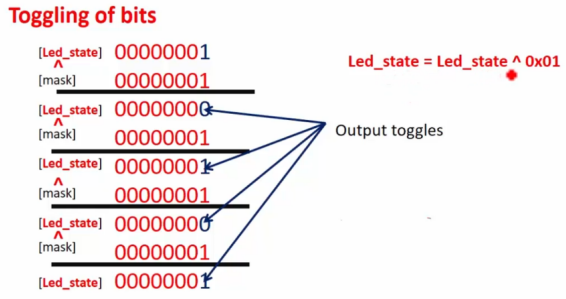


Figure 3.31: LED Toggling with XOR

3.17 LED Practice

Now we will use bitwise operation, pointers in order to turn on LED on our board.

3.17.1 Hardware GPIO,Bus

First, we need to understand hardware connection to our board. For this purpose, we download the *schematic board* for the STM40 discovery board form stm website, in resource section.

Figure 3.32 shows the LED in our SMT32F40 discovery board.

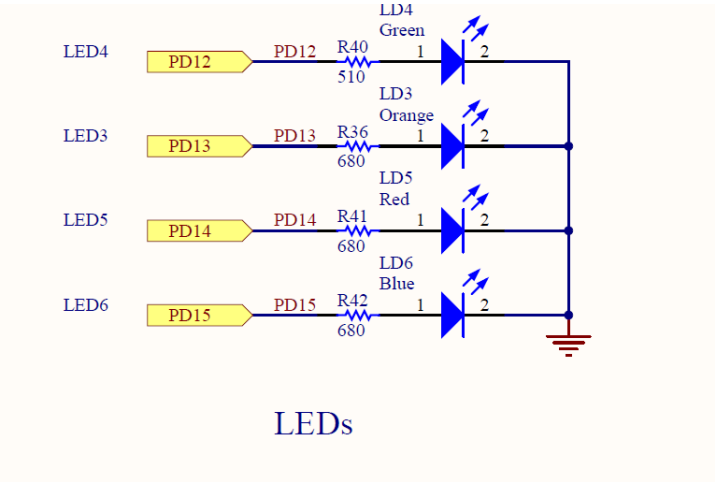


Figure 3.32: LED Connection to SMT32F40 discovery board

- `PD15` stands for port D, pin 15.

In the schematic also, we ***can find the ports in the board***, shown in Figure 3.33

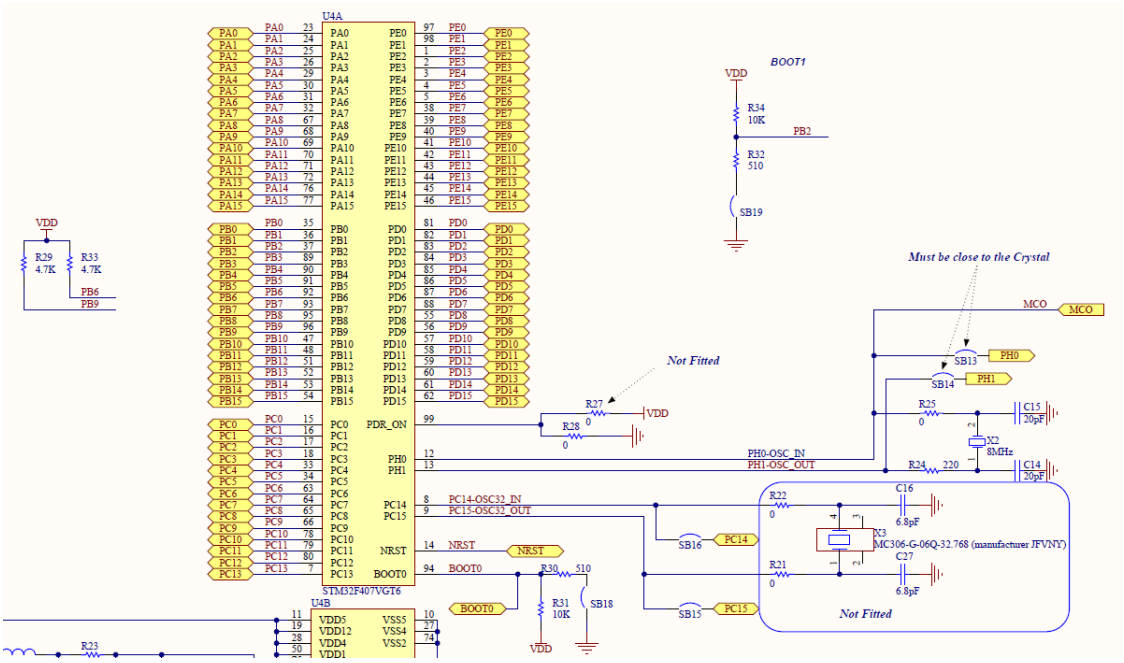


Figure 3.33: Ports of SMT32F40 discovery board

- In stm32Fx series, each port has 16 pins
- We can connect to them external peripheral (LED,display,button, Bluetooth transceiver,external memory,$\cdots$)

Now back to Figure 3.32. Suppose we want to access the green LED, so we need to control `PD12`.
How we can do so? Figure 3.34 contains the answer.

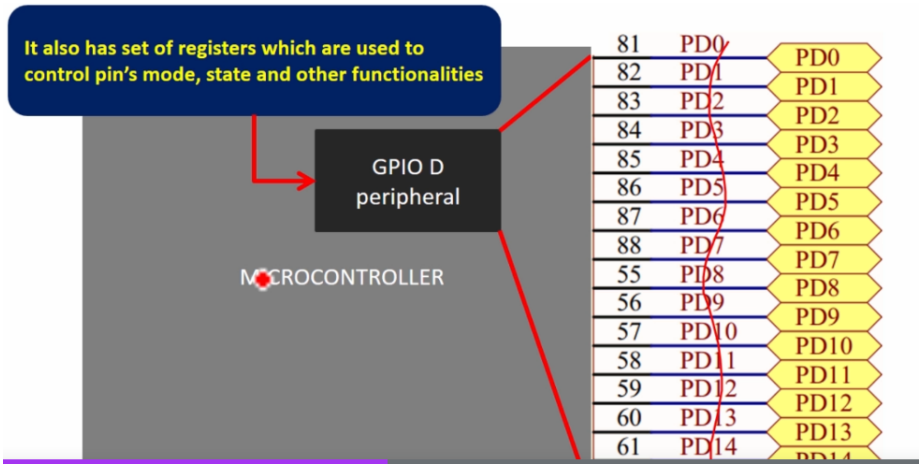


Figure 3.34: Controlling port D from GPIO D peripheral

Port D is controlled via a *peripheral* called GPIO D, which contains a register controlling the port D (mode of each pin,state,function,$\cdots$).

Each register in GPIO D ***contains its own address***, and if we want to use some register, we need to access via its unique address via the data bus as shown in Figure 3.35.

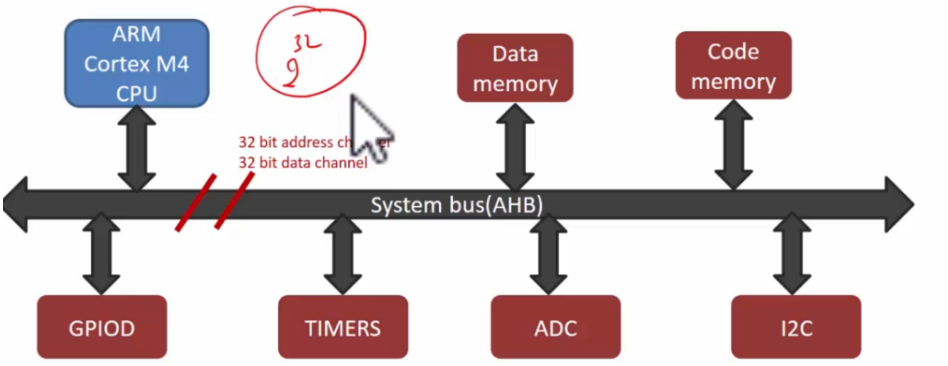


Figure 3.35: Data bus Connecting peripheral

- AHB stand for advanced High-performance Bus, created by st
- The width of the bus is 32 bits, which means we can put 2^{32} addresses as shown in Figure 3.36.

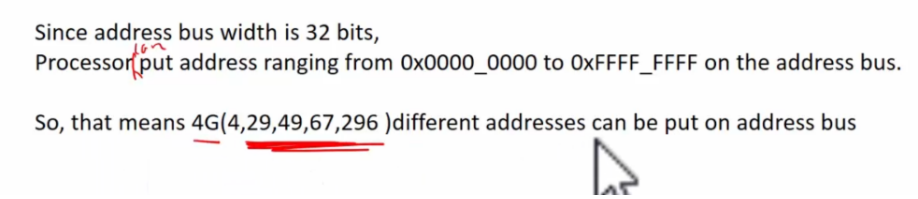


Figure 3.36: Data bus width and addresses range

The memory map of ARM cortex Mx processor is shown in Figure 3.37.

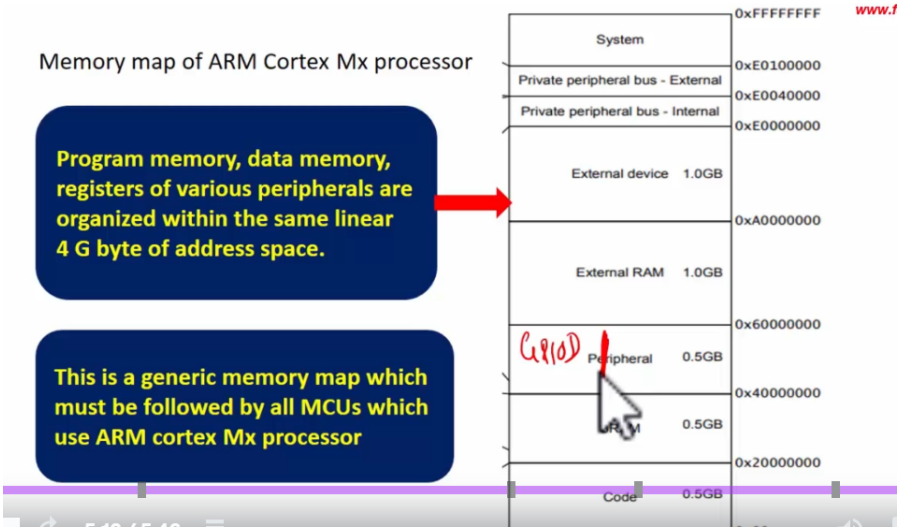


Figure 3.37: Memory Map

To know what is GPIO D address (or any other peripheral we want to access it), we go to the reference manual, section 2.3 (Memory map), Table 1, and we find for GPIO D `0x 4002 0C00` as base address (starting address) for GPIO D.

3.17.2 Peripheral Information

- All stm32 peripheral are 32 bit wide
- Different peripheral have different number of registers (GPIO D may have 10 register, and ADC may have 8 register)
- An example about GPIO D registers are shown in Figure 3.38.

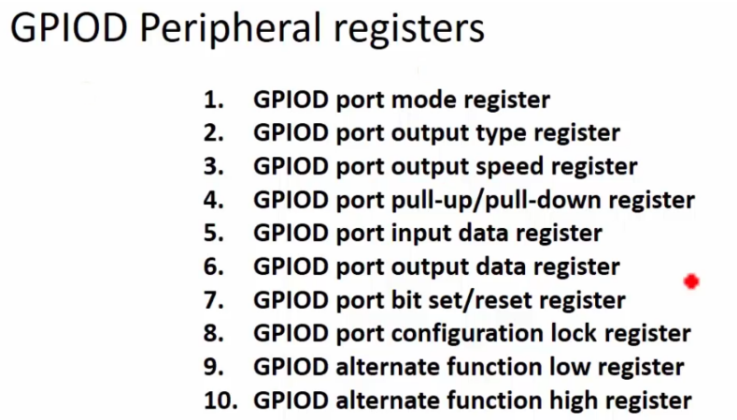


Figure 3.38: Different peripheral registers for GPIOD

Each of the 10 register is of 32 bit width, and has its own start and end addresses as shown in Figure 3.39

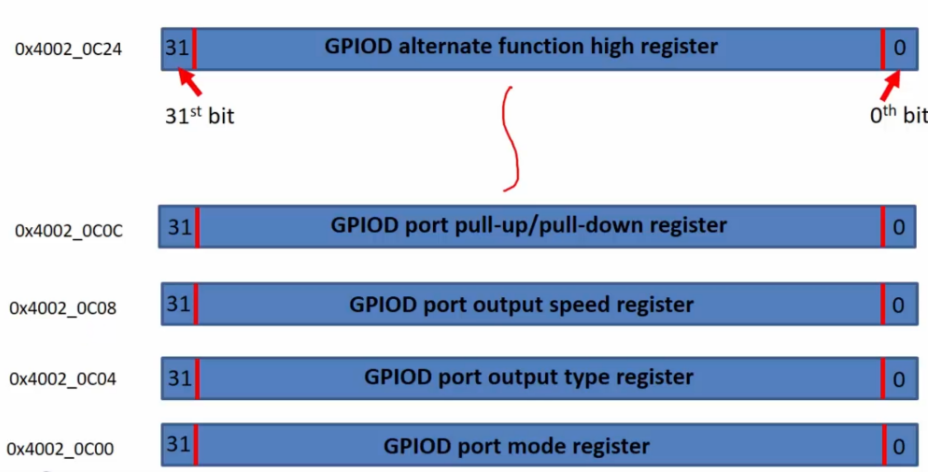


Figure 3.39: GPIOD Register Addresses

Each of these register has its own purpose, like GPIO mode register is used to configure the port pins as output or input. Also an example is shown in Figure 3.40, where the data register is used to turn on LED ON or OFF (by setting bits 0 for OFF or 1 for ON)

We will explore these registers later though coding and exercises.

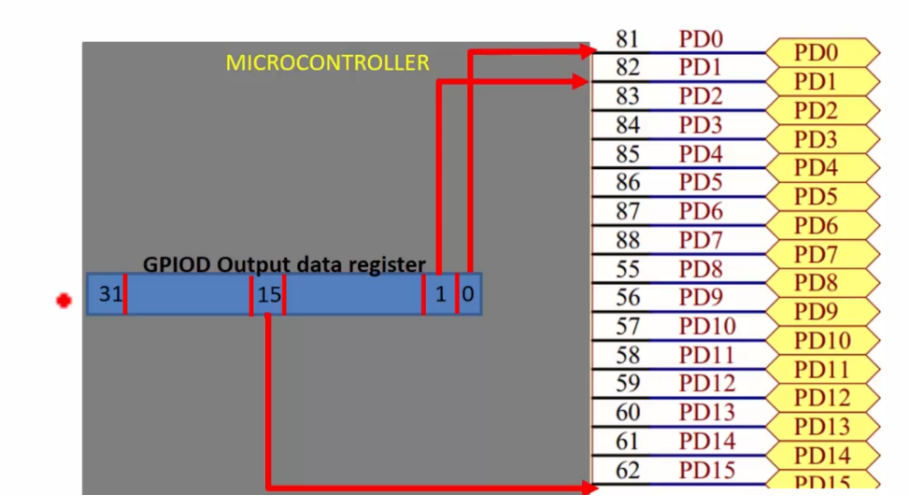


Figure 3.40: GPIOD Output Data Register

3.17.3 Procedure for turning a LED

To turn a LED ON or OFF in a microcontroller, several thing need to be done and checked.

1. Identify the GPIO port (peripheral) which LED is connected: port D
2. In this port, which pin we will use
3. ***Important: Enabling the pins***

   By default, in most microncontrollers, most pins are dead (in sleep mode). We must first activate the clock into them, otherwise no configuration or any operation can be performed.

   - In some microcontrollers it can be otherwise, so we must check the data sheet before
   - In stm microcontroller, all peripheral are in sleep mode and clock need to be configured
4. Configure its mode (input pin or output pin)
5. Write data to it

It is important to note that step 3 (enabling the clock) comes before step 4 (configuring the mode), otherwise we can't do anything. So in the next section we will learn how to configure the clock.

3.17.4 Configuring Clocks

In stm32 microcontroller, most clock are configured through the `RCC` register, which stands for reset and clock control. It can be found in chapter 6 in the reference manual.

Now the `RCC` has many registers, and we are interested in configuring the clock relative to our peripheral, which is `GPIO D`.

For `GPIO D`, we need to use the `RCC AHB1` bus register, because `GPIO D` uses `AHB1` bus to talk with the ARM microprocessor as shown in Figure 3.41.

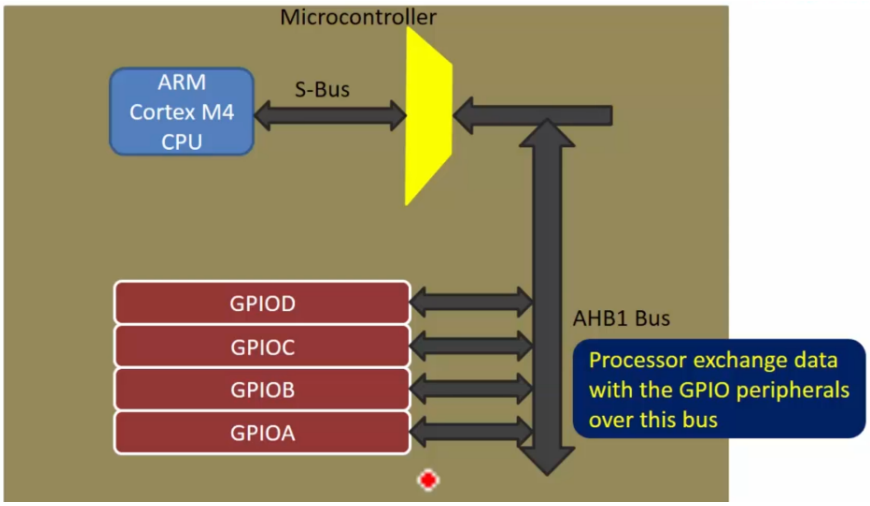


Figure 3.41: GPIOD and AHB1 bus

This information we can know it from the ***data sheet***, section 2.2 functional overview page 19. In Figure 3.42, we have a screen shot of the diagram contained in the data sheet.

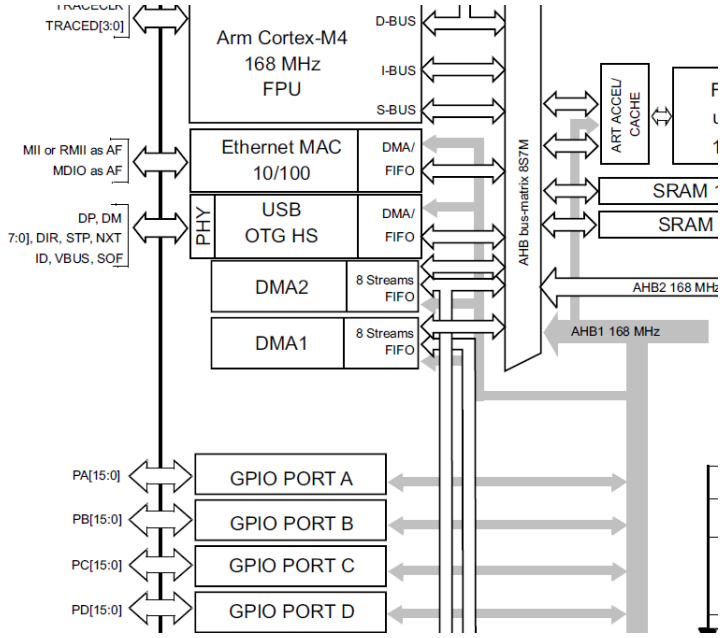


Figure 3.42: GPIOD and AHB1 bus

As we can see, the GPIO D uses AHB 1 bus to talk the ARM processor.

Now we understood which RCC register is used for GPIO D, we open the *reference manual* to see how to set the RCC AHB1. RCC AHB1 register is composed from several register (from section 6.3.5 → 6.3.10). Since we are interested in a clock configuration, we go to section 6.3.10, where we have clock options.
For GPIO D, we need to set bit 3 (because it is written GPIO D EN).
Address offset 0x30 means we need to add 30 to the base address of the RCC to get to the RCC AHB1ENR register.
The base address of the RCC can be found the memory map section of the reference manual

3.17.5 Configuring GPIO

After setting the clock using the RCC register, we need now to configure GPIO registers. For this application, we need to access 2 register:

1. GPIOx MODER (see section 8.4.1 in reference manual).
    - Each 2 bits in this register is associated to 1 pin. This is because we have 4 state for each pin
    - We are using LED on pin 12 of port D, so we need to assign 01 to bits 24 and 25 (responsible for pin 12).
    - To do this, we can first clear bits using & masking, then set the 24^{th} bit to 1 using a bit wise OR mask
2. After configuring pin type, we need to send output so the LED blink, and this is done using GPIOx ODR rigister (see section 8.4.6)
    - Here we have 16 bits, which are enough to control the pins (either ON or OF, so 1 bit is enough to code these 2 states)
    - We do it using a bit wise OR mask at bit 12

3.17.6 Monitoring Registers in Real time

The smt32 cube IDE let us view real time application of the register.

To track the register in real time:

1. We go in debug mode `Debug As --> Stm32 C/C++ Application`
   Debugger mode will be on.
2. From the tab menu, we go to `Window --> Show View --> SFR`. A new window will appear at the right containing the registers as shown in Figure 3.43

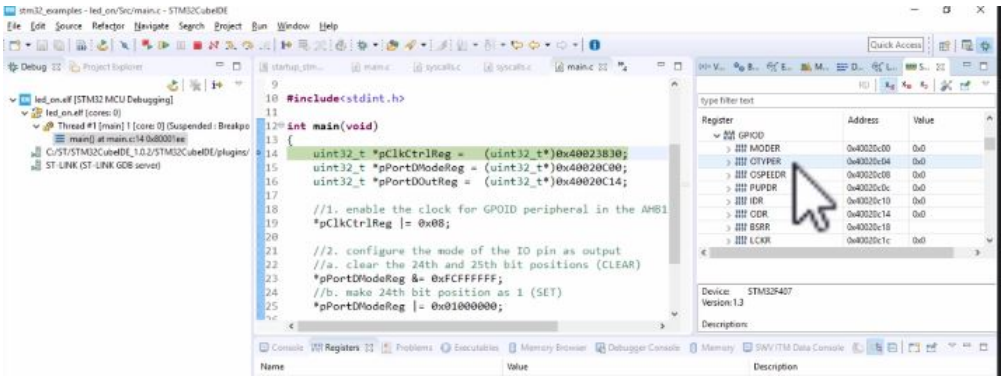


Figure 3.43: Tracking registers in real time

3.18 Bitwise Operation: shifting

In section 3.17, we have done several register manipulation, that is set and clear bits in registers using port masking. This approach become very tedious, because each time ***we need to harcode the mask ourself***.
To avoid such a complication, we can use shifting operation.
Note: 3.18.4 contains the complete example wrap up.

3.18.1 Bitwise Right

Take the number a = $111_{10} = 6F_{16} = 01101111$. Shifting by 4 bits is shown in Figure 3.44.

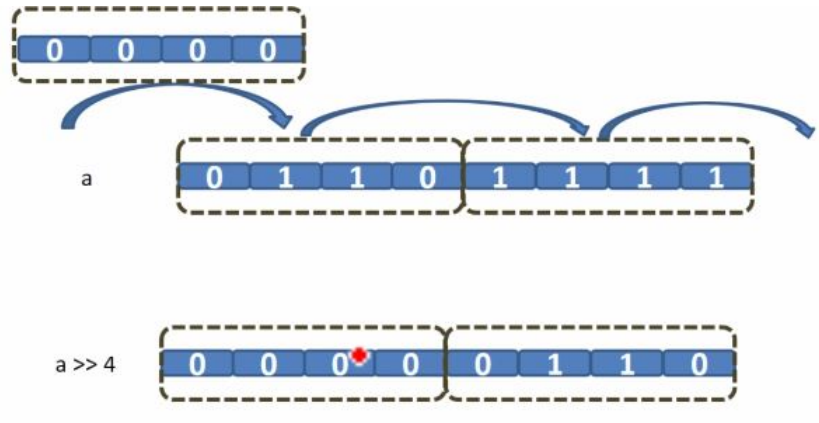


Figure 3.44: Bitiwise shift: right direction by 4 bits

The result is $06_{16} = 6_{10}$.

3.18.2 Bitwise Left

We take same example in Figure 3.44, but with left direction this time. Example is shown in Figure 3.45.

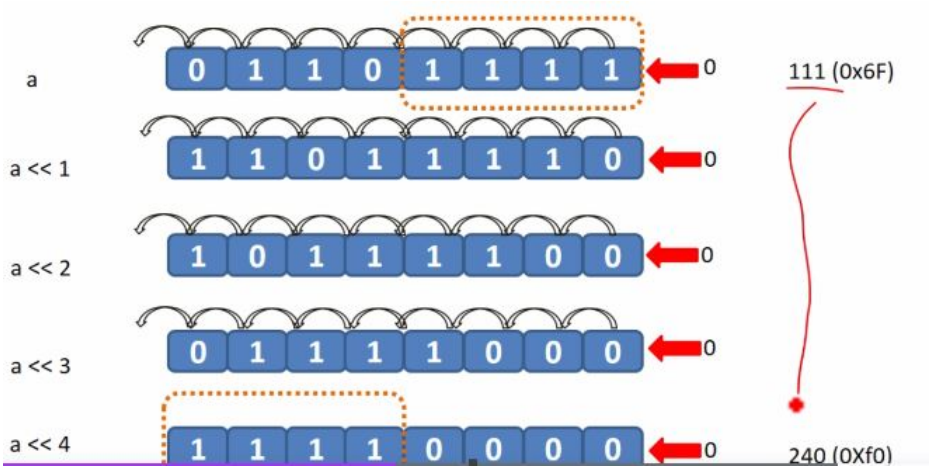


Figure 3.45: Bitiwise shift: left direction by 4 bits

3.18.3 Some notes about bitwise shift

In Figure 3.46,we have a comparison table between bitwise shift left and right.

<< vs >>

<< (left shift)	>> (right shift)
4 << 0 => 4	128 >> 0 => 128
4 << 1 => 8	128 >> 1 => 64
4 << 2 => 16	128 >> 2 => 32
4 << 3 => 32	128 >> 3 => 16
4 << 4 => 64	128 >> 4 => 8
4 << 5 => 128	128 >> 5 => 4

Figure 3.46: Bitiwise shift: Left and Right Comparison

Shifting left increase the value, whereas shifting right decrease the value by 2.
The reason behind this is the weight changing: when shifting to the left, the weights value increase, whereas in shifting to the right, the weight value of the number decrease.

3.18.4 Applicability of Bitwise in Embedded Programming

Bitwise shift are very useful when clearing and setting bits in a given data register. We give 2 example for setting and clearing a bit.

Note:

- from now on, we use the new approach instead of the old one, that is figure out the value of the mask
- See code `LED Bitshift` version to see the modification done.

Now we start the examples.

1. Setting 4^{th} bit: In Figure 3.47, we have the example using the old method and bitwise method

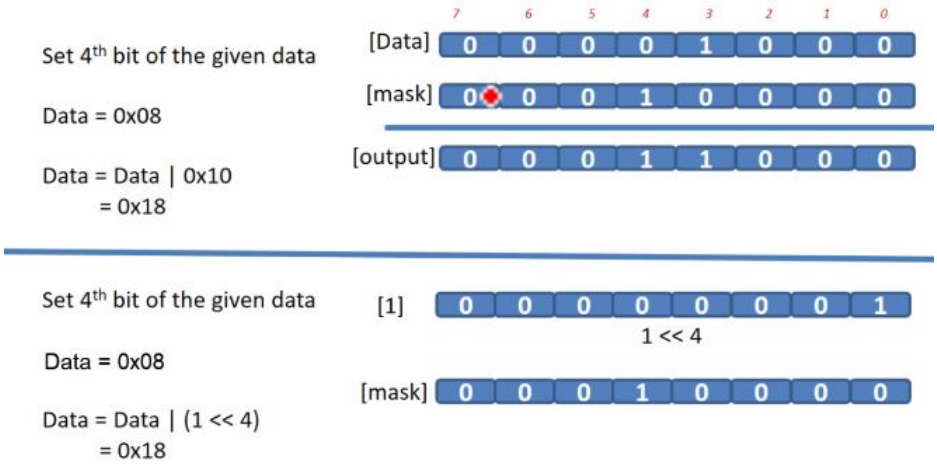


Figure 3.47: Bitiwise shift to set a bit

   - Note that using the `1<<4` method, we don't need to figure out the value of the mask as in the old approach

2. Clearing a bit:

   Notice that after negating, the value will be the same as in the 1^{st} method

Note:

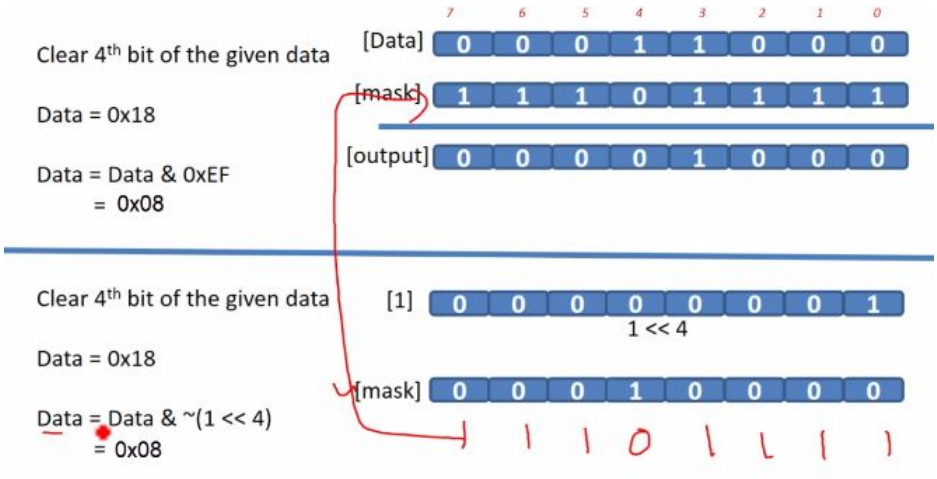


Figure 3.48: Bitiwise shift to clear a bit

3.19 Bit Extraction

Now before we continue we present another task usually done in embedded programming called *bit extraction*.

Problem Statement: Extract bit position from $9^{th} \rightarrow 14^{th}$ bit position. The example is shown in Figure 3.49, along with the method used.

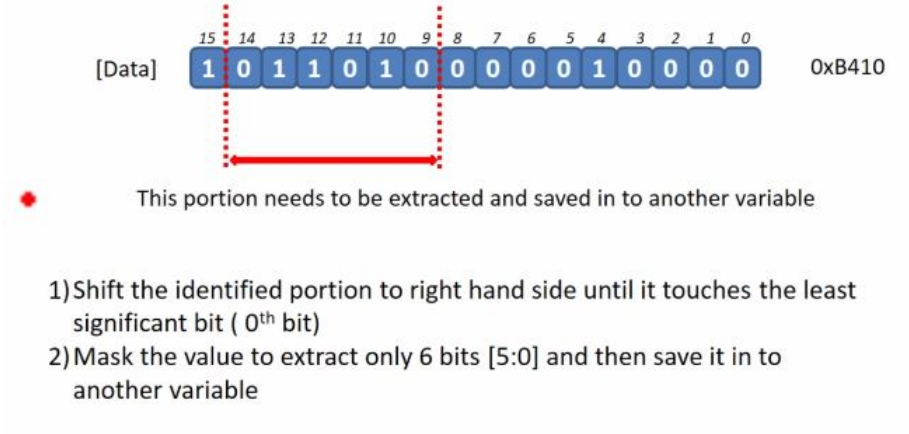


Figure 3.49: Bit Extraction

The shifting step is shown in Figure 3.50.

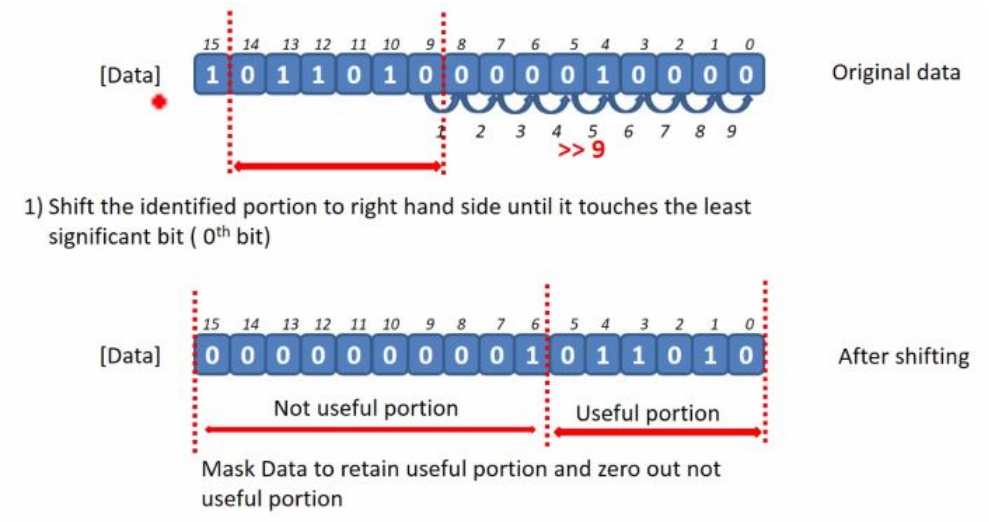


Figure 3.50: Bit Extraction: Shifting Step

Then after shifting we do masking to the shifted data as shown in Figure 3.51.

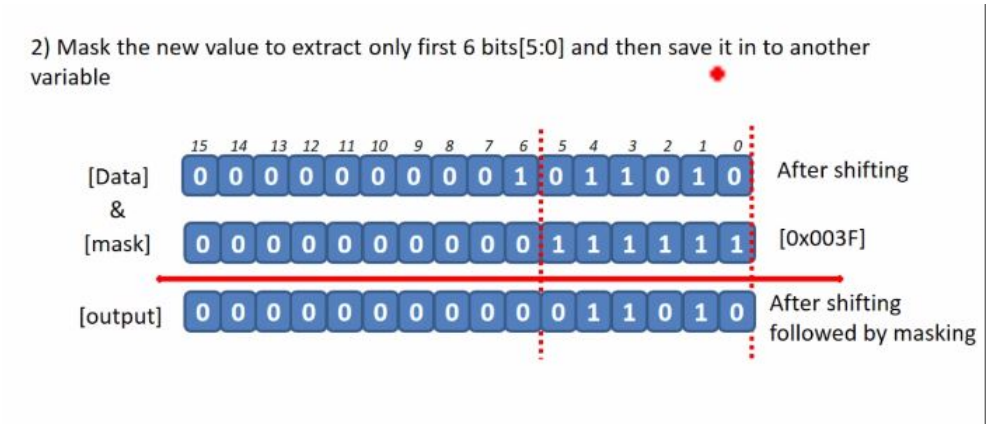


Figure 3.51: Bit Extraction: Masking the shifted data step

Code:

- `uint16 t = 0xB410`
- `uint8 t output`
- `output = (uint8 t)((Data >> 9)&0x003F)`

3.20 Loop in Embedded programming

In this section, we explain some point and some pitfalls about loop in embedded programming.

3.20.1 while loop

1. `while` loop and the `;`: We should pay attention in programming, when we put a `;` after a `while` loop.

   We have an example in Figure 3.52.

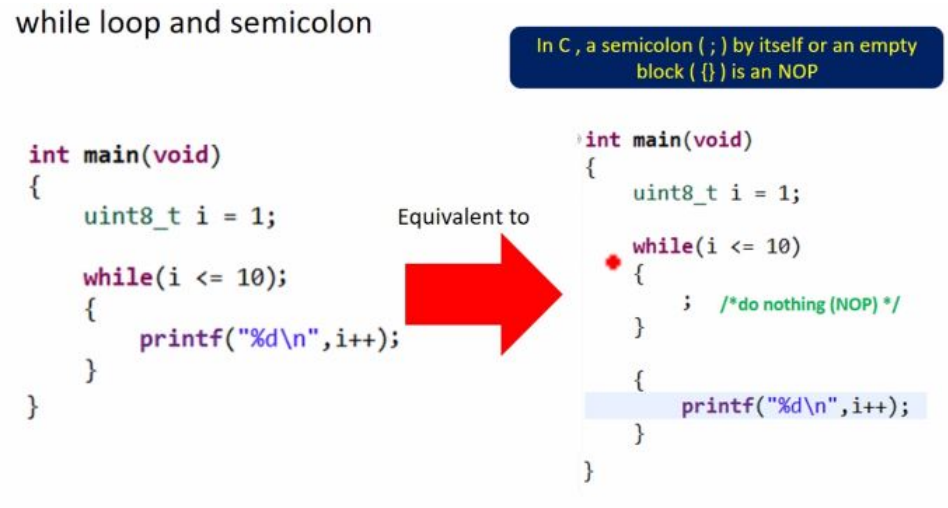


Figure 3.52: while loop and semicolone

   It is clear that inserting a `;` after the `while` loop in this case will not make the counter to increment the variable `i`.

2. Super loop (forever loop):

   In embedded programming unlike pc program, the program in the microcontroller need to run forever (or as long it is power up). In Figure 3.53, we have an example of `while` loop where the `;` is inserted in the right place.

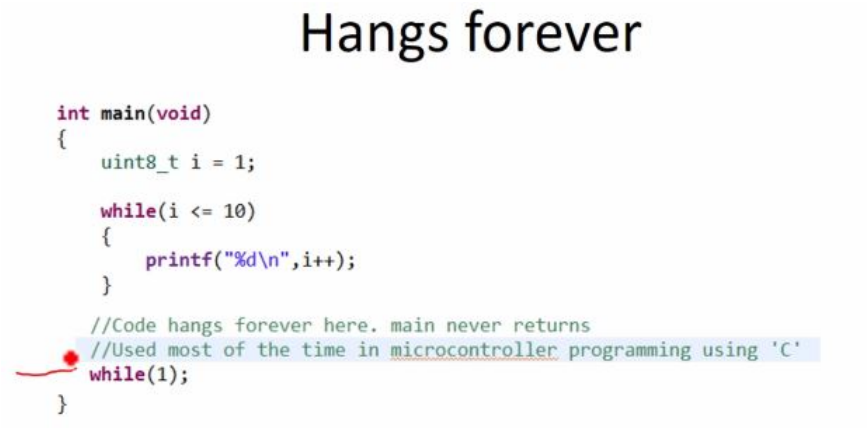


Figure 3.53: while loop and semicolone

Another example is shown in Figure 3.54

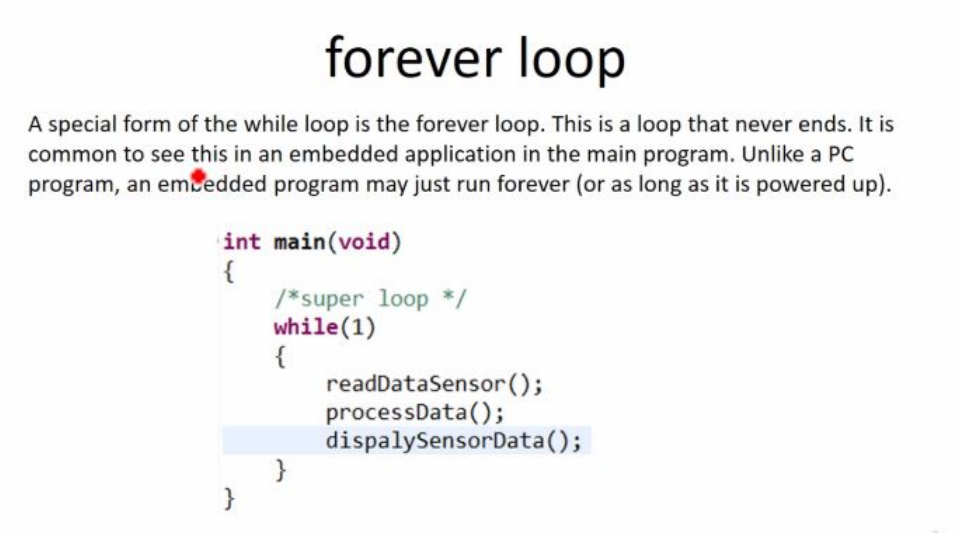


Figure 3.54: Super loop

3.20.2 do while

Unlike the `while` which can or can't have a `;`, a `do while` ***must have a*** `;` after it, otherwise it will be a compilation error.
In embedded programming, we use `do while` loop when writing *multiline C macros in a header file*. This part will be explored later when we talk about macros and header file in later sections.

3.21 LED Toggling

In this section, we will modify the LED ON program to produce a toggling LED. The purpose is to use start using loops in a embedded programming.

Problem Statement: write a program to make the LED toggle between ON and OFF state, where a certain *delay* is injected between these 2 states.

Delay types: to produce a certain delay, we have 2 methods:

1. software method: this done using `for` loop for example. However, this method is inaccruate and waist allot of cycles.
2. Hardware method: this is done using peripheral such as timers. This method is more accurate.

For the LED toggling application, we will use software method since accuracy is not important in this application. We are only interested to produce a human observable delay to see the toggling.

See project `LED Toggling`.

3.22 Type Qualifier: const

The first qualifier we encounter is `const`. When used in front of variable, this means we can real only but not modified. Nevertheless, we can still modify the value by accessing the address via pointer as shown in Figure 3.55.

```
8  int main(void)
9  {
10     uint8_t const data = 10;
11
12     printf("Value = %u\n",data);
13
14     uint8_t *ptr = &data;
15
16     *ptr = 50;
17
18     printf("Value = %u\n",data);
19
20     getchar();
21 }
```

Figure 3.55: Changing a const variable via pointer

Also, there exist several cases, which are illustrated in Figure 3.56.

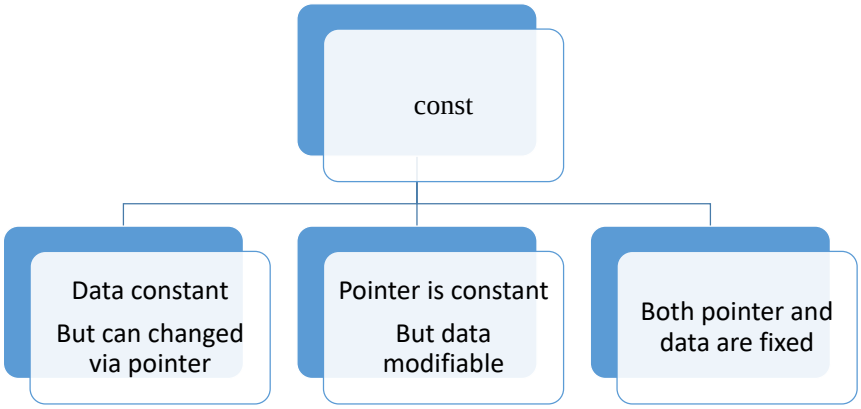


Figure 3.56: const usage case

For the 1st case, an example is shown Figure 3.57.

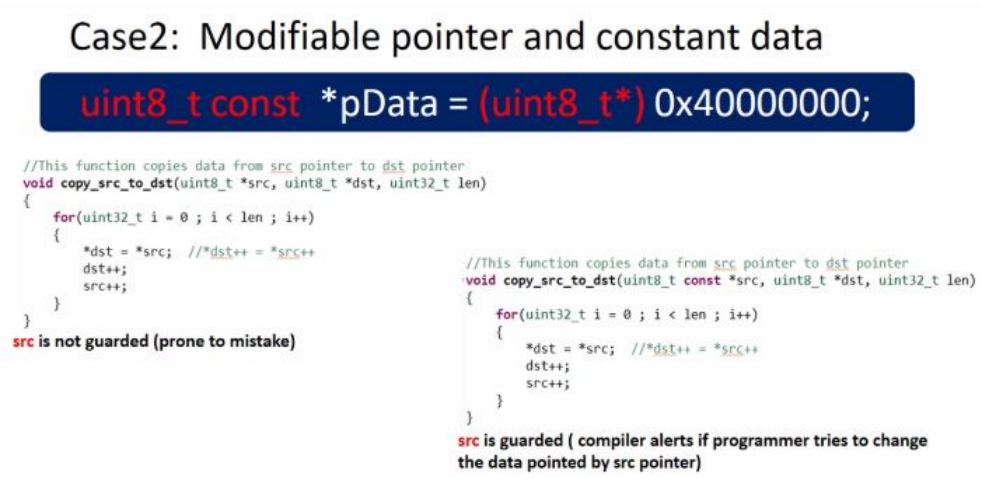


Figure 3.57: const case: modifiable pointer and constant data

The goal is to protect the data contained in the source from being modified accidentally.

As for the 2nd case, example shown in Figure 3.58.

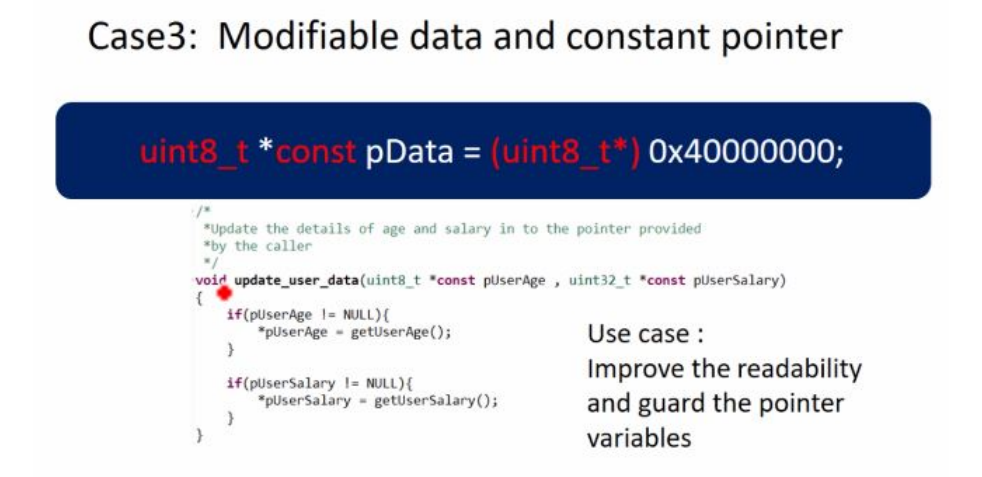


Figure 3.58: const case: modifiable data and constant pointer

Last case is shown in Figure 3.59

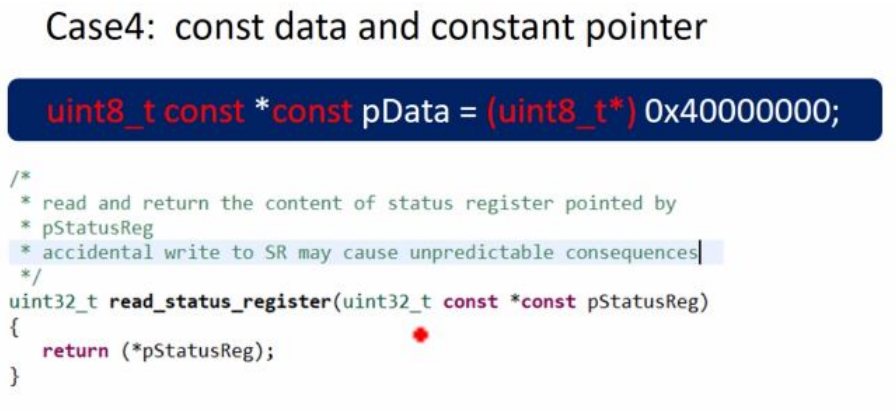


Figure 3.59: const case: all is fixed

- `const* src`: the programmer know that the content pointed by `src` should not be modified
- Several cases for the usage of `const` qualifier

3.23 Pin read

Now we will do another exercise to practice more.

3.23.1 Problem Statement

Write a program which read the status of pin PA0. If PA0 is high, then the LED pin PD12 (used in exercises before) is ON, and if PA0 is low, then PD12 is OFF.

Change the status of PA0 manually by connecting between GND and VDD points of the board.

3.23.2 Hints and Checking

When dealing with such program, the 1st thing we need to do is to check whether PA0 if a free IO pin or not. In smt32407g discovery board, usually it is.

To know that, we open the *user manual* at table 7 in chapter 6. We see at the row containing PA0, the column named free IO pin contain PA0, so we can use PA0 as free IO pin.
For other pins at port A, it is not necessary the case. For example, the PA4 row, the free IO column is empty, sow can't use PA4 as free IO pin.

3.23.3 Programming

Now in this exercise, we have 2 ports: A and D.
For port A, we should read from it so it is considered as input.

Also, we need to enable the clock for port A the same way we did for port D in the LED exercise. We use also the same register (RCC AHB1ENR) because all GPIO peripheral use AHB1 bus. For GPIO D, the pin we set was pin 3, and for GPIO A it could be a different pin, so we need to check the *reference manual* (section 6.3.10, it will be pin 0 for GPIO A).

To configure PA0 as input, we need to access 1st the GPIO-A base address (see section 2.3 memory map in the reference manual) the same way we did for GPIO-D, then setting the input mode via GPIO MODE register.
Once we configured PA0 as input, we need to read from it voltage values (GND or VDD). In the LED exercise, we used the GPIO ODR to write values to the LED, and here we will use GPIO IDR (input data register, see section 8.4.5 in the reference manual).

3.24 Compiler Optimization

Before continuing in the next features of embedded C, the `volatile` concept, we need 1st to understand optimization concepts done by the compiler.

Optimization is a series of actions taken by the compiler on our written embedded code to:

- number of instruction (code space optimization)
- Memory access time (time space optimization)
- Power consumption

By default, compiler doesn't do any optimizaiton, but we can enable optimization using *flags*. Different flags exist: `-O0`, `-O1 --> -O3`.

The `-O0` is the phase were no optimization is done. The benefits of it:

- debugging friendly (for each code, 1 assembly instruction is generated)
- fast at compilation time.

However, it is not recommended for application constrained by their space or RAM.

The other flags are quite the opposite: more efficient but less friendly debugging, and take longer time to compile. Each level has its own sophistication and features, and we need to use them depending on our application and needs (no specific guideline about them).

3.24.1 Pin Read Optimization

As a concrete example how compiler optimization can effect our code, we will retake the `Pin read` exercise, and set the IDE to 2 different compilation mode flag: `-O0` and `-O2`.

`-O2` Level:
In Figure 3.60, we have the disassembly equivalent code of line 37

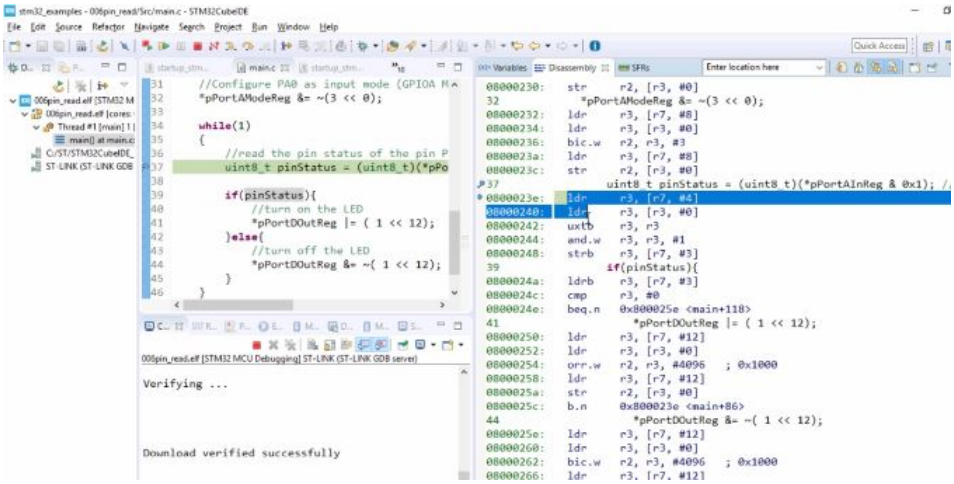


Figure 3.60: Pin read demo: -O0 optimization

The `pinStatus` variable value is loaded successfully at each seconds during the `while` loop. This essential to our pin reading application because we need to be able to check at every time moment the status of the pin, so we can turn ON or OFF the led `PD12`.

`-O2` Level:

Now when switching compilation mode to `-O2`, line 17 will not be break down in a 1-to-1 disassembly code as in Figure 3.60. That is because the compiler will add some optimization algorithm to reduce either time space or code space. The correspondent disassembly program to `-O2` is shown in Figure 3.61.

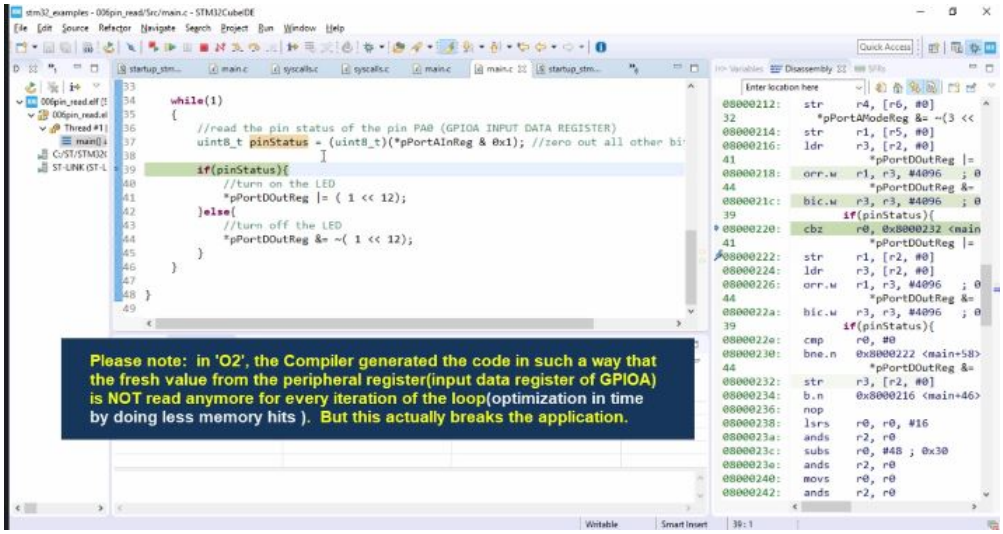


Figure 3.61: Pin read demo: -O2 optimization

- In debug mode, the execution of the code is random and not line by line as in `-O0`.
  This is because the compiler is generating some instruction to do optimization
- When reaching the `while` loop, the compiler will not execute line 17 anymore.
  The reason is so he can optimize time, by reducing memory hit and not reading the `pinStatus` value in the memory.

It is true it is maybe an optimization in time, but now the application is not working.

So the question to arise: how we can prevent compiler from doing unwanted optimization to our code? here it comes the role `volatile`.

3.25 Volatile Type Qualifier

`volatile` is type qualifier which tells the compiler to not do any optimization on the variable operation (read and write).
It also tells the compiler that the value of the variable may change at any time with or without programmer consent. So it turns off optimization on variable operation (read and write).

Let's take another example to see compiler optimization. In , we have some code along with the assembly code, at `-O0` optimization level.

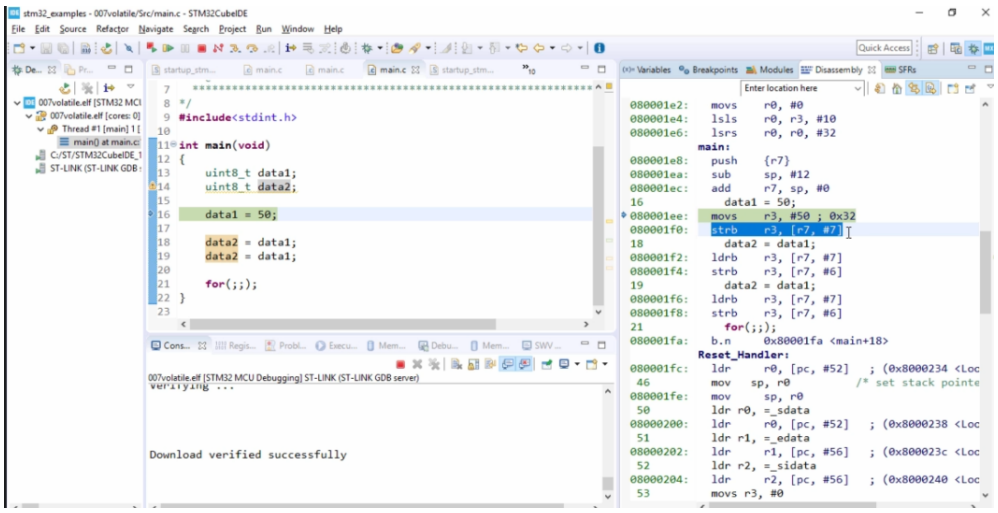


Figure 3.62: Demo -O0 optimization

Notice that although we have redundant instruction (so instruction optimization can be performed), compiler generate all the assembly code, even for the redundant instruction, that's because we are in `-O0` level.

Now suppose we use `-O1` level (shown in Figure 3.63), compiler won't generate anything except for the `for(;;)` statement

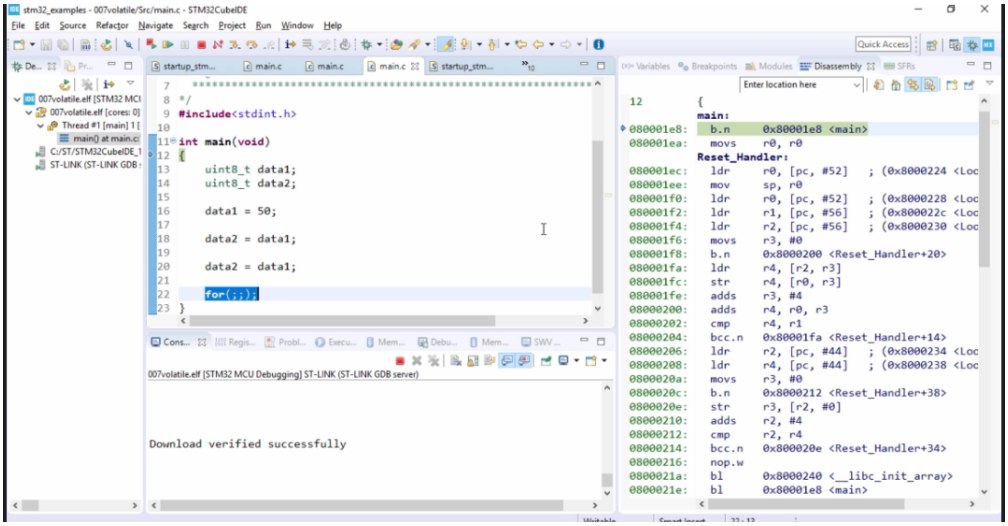


Figure 3.63: Demo -O1 optimization

The compiler did this because it is true we declare and initialize the variables, but we didn't use them at all (we didn't do any operation, pass them to some function,$\cdots$).

If we don't want any optimization, we use the `volatile` type qualifier as shown in Figure 3.64

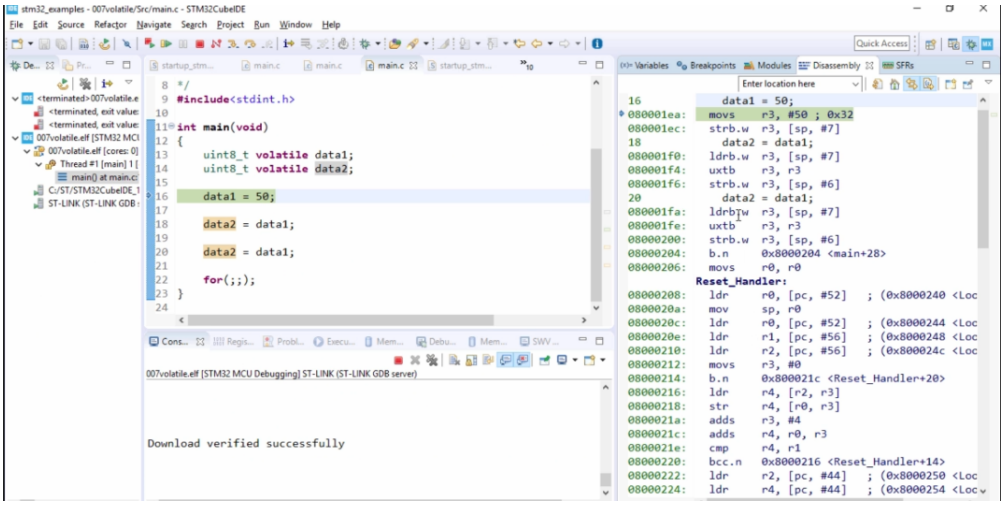


Figure 3.64: Demo -O1 optimization with volatile

Notice that now even if we are in `-O1` (or higher) levels, the compiler generate all assembly instructions.

Note: Type qualifier (such as `const` and `volatile`) comes always after type specifier (`uint8_t`).

3.25.1 Use case of volatile

The `volatile` is used whenever there is a possibility to a variable to be changed (change can be coming from software or Hardware).

In embedded system context, here the 3 cases where `volatile` should be used:

1. Memory mapped peripheral registers of the micrcocontroller (as in the pin read exercise in 3.24.1)
2. Multiple tasks accessing global variables (read/write) in an RTOS multithreaded application
3. When a global variable is used to share data between the main and an ISR code

3.25.2 volatile different syntax

As in the case `const` different syntax meaning (section 3.22), same thing exist for `volatile` type qualifier. Figure 3.65 shows the different use case for `volatile`

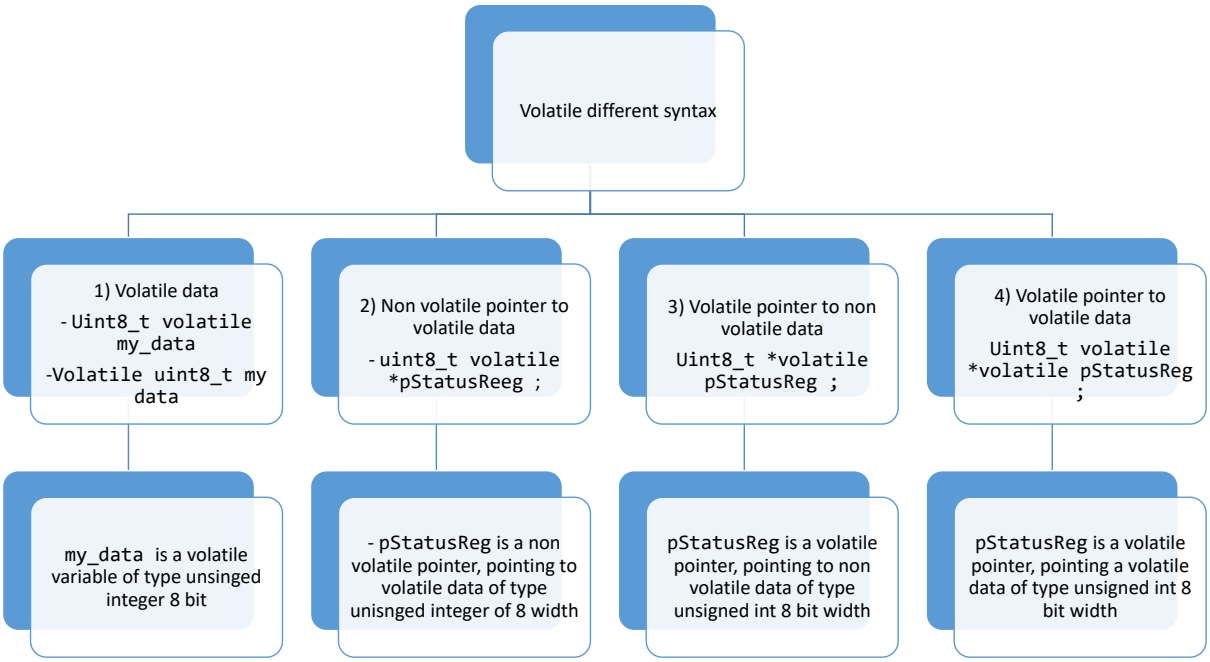


Figure 3.65: volatile syntax use cases

- Case 2 is the most used case in embedded programming: we use it whenever we access memory mapped register
- Case 3 and 4 are rarely used

3.25.3 volatile and ISR

In this section, we explore the usage of `volatile` with ISR (interrupt service routine).

Note: IDE usage: To see how the counter is increased every time we press the button, we go to debug mode, and then run the code using the green arrow, and set the console at `SWV ITM Data Console` as shown in Figure 3.66.

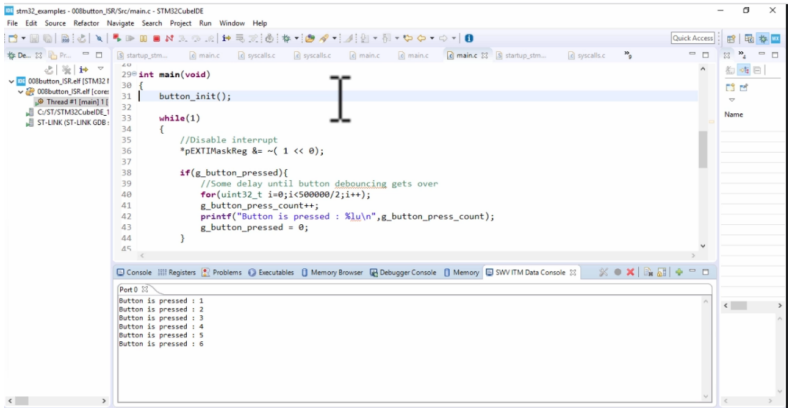


Figure 3.66: IDE usage: SWV console for counter using push button

Make sure that port0 is checked (as shown in Figure 3.67).

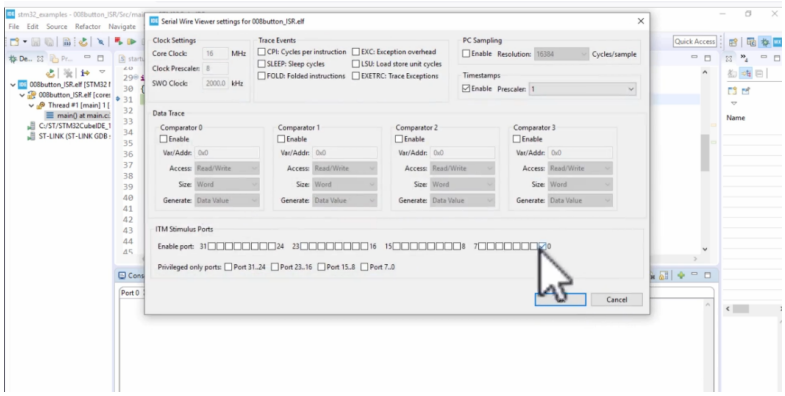


Figure 3.67: IDE usage: SWV console for counter using push button

- *To explore later why we need to check port0 is checked*
- *This technique can be used in debugging our application*

3.26 Structure and Bit Field

3.26.1 Some Key Concepts

- structures are usually defined in header file or outside the `main` function

  Review from C book how to create (declare) initialize and access structures.

  C book Reference: *To include C book reference*

  C book Reference:

- Padding concept in structure:

  - compiler doesn't store variable values at arbitrary addresses, rather it stores them in a way such that the end of each variable address correspond to natural boundary of the variable data type.
  - As example, consider Figure 3.68 where we have 3 data types: `char` (consume 1 byte), `int` (consume 4 byte) and `short`(consume 2 byte).

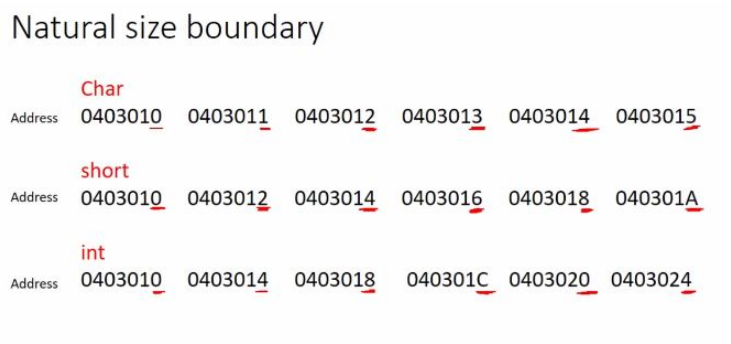


Figure 3.68: Structure padding example

Notice the end of each address. For `char`, the end of address can be 0,1,2,$\cdots$ because `char` consume 1 byte.

For `short`, it can be 0, then 2 (and not 1), then 4,$\cdots$, because `short`consume 2 byte.

- Packed Structure allow more efficient communication between the microcontroller and the memory (Use less assembly code for each member initialization of a structure member. Less assembly means less clock cycles)

  Packed code demo:

  Packed code demo

  - *See demo* `packed vs Unpacked` *and compare the assembly code.*
  - *to run the code and document it, along with structure demo.*

- When using a *pointer of type structure*, such as `struct Dataset *pdata` (where the initial structure is `struct Dataset data`, and `data` is an instance), we use the arrow operator `->` to access/set members (`pdata-> date = 12`).

  If we deal with normal structure variable, we use `.` operator (`data.date = 12`)

3.26.2 Packet Exercise

Concepts used:

- Bit extraction, mask and bit shift operation (Review section 3.19)

Problem Statement: Enter some 32 bit packet value in Hexadecimal. The composition of the packet is shown in Figure 3.69. Print the value of each section in this packet.

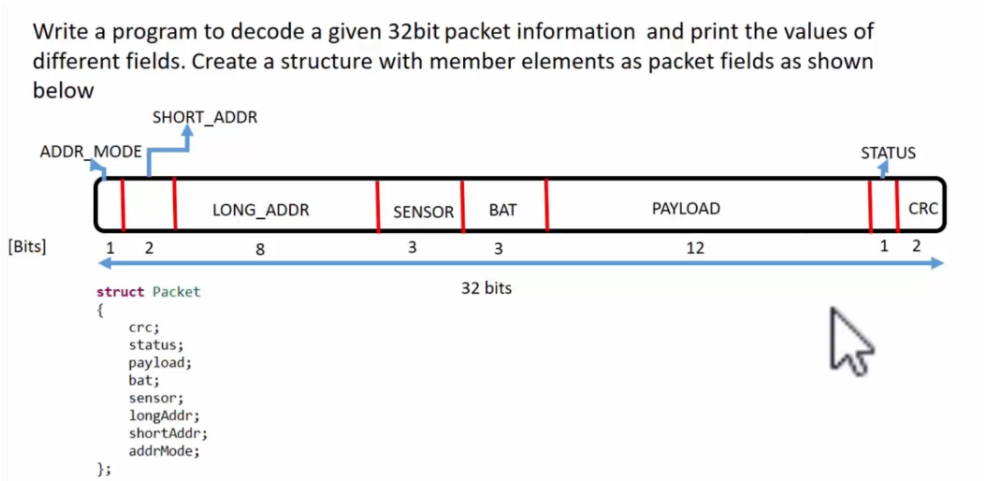


Figure 3.69: packet exercise: problem statement

,byte,...

Note: For hexadecimal with `scanf`, we use `%X`.
Insert a table of definition bit,byte, hexa and different base number format (from digital design course)

Project: See `Packets` and `Packets bit fields`.

3.27 Union

Unions are variables which share the same memory location. An example is shown in Figure 3.70.

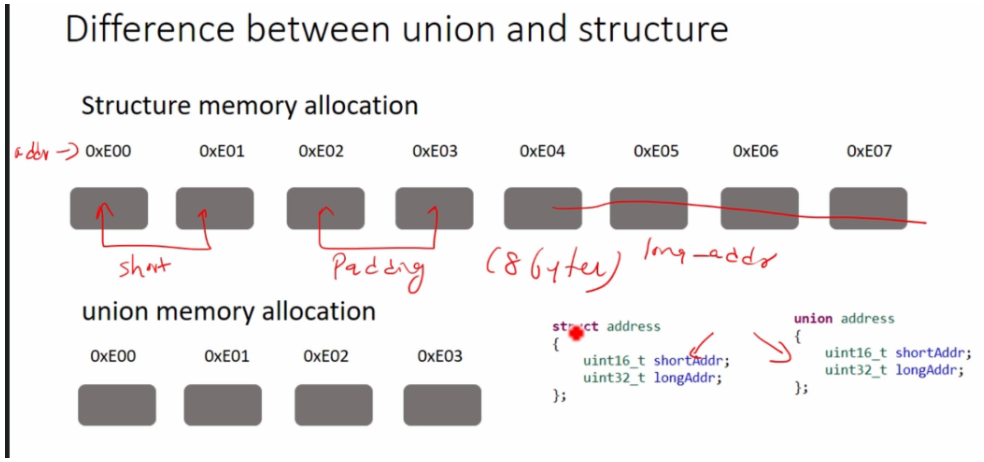


Figure 3.70: Union Example

- We define the union the same way we define a structure: we just replace the keyword `struct` by `union`
- The size of a union is equal to the size of its largest members
  - In the example of Figure 3.70, the size will be 4 byte

We usually use the union whenever the member usage is ***mutually exclusive***, that is we only use 1 of them at a time.

Union Demo

Union Demo:

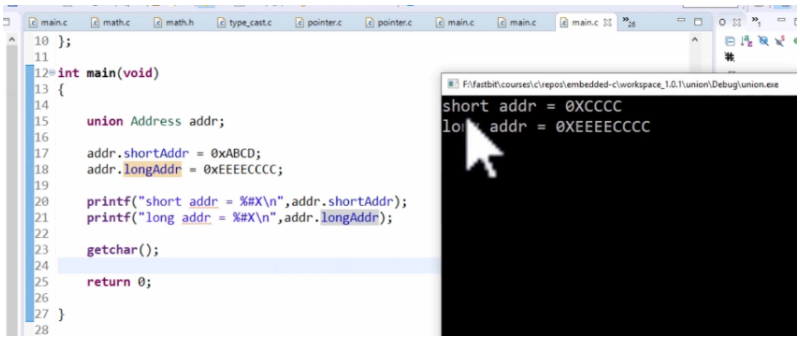


Figure 3.71: Union demo: overwritten values

- *Write the code of union demo*
- *In union, we will see that we have variable overwritten*
- *since both members share the same memory location, the longaddr will overwrite the short addr, that's why we don't see 0xABDC*

3.27.1 Applicability of Union in embedded programming

Now we will do the packet exercise (see 3.26.2) using a combination of `union` and `struct` implementation.

on Packet
rcise

Union Packet Exercise:

- *Create a project, and repeat the packet exercise*
- *Compare the output and write note and explanation*

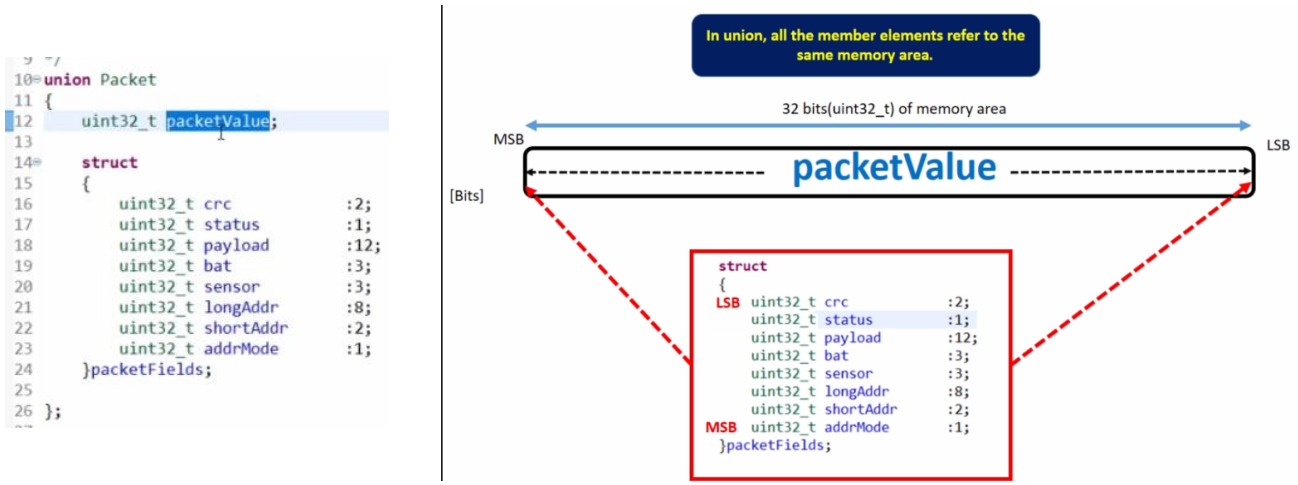


Figure 3.72: Union Packet Exercise

Project: See `Packets Union struct version` in `Host`

3.28 Bit fields in Embedded Code

Bit fields and structure are heavily used in embedded programming to present abstraction when dealing with code. We will use these concepts by refactoring the LED toggling exercise (see section 3.21).

For example, in the `LED Toggle` project, we use the `RCC-AHB1ENR` (shown in Figure 3.73) to enable the clock for the `GPIO-D` (since we are using LED on `port D`). This is implement it via the following `C` command: *clck-register—= 1¡¡3. The only way we can know that we need shift left via position 3 is if we have a reference manual at our side.

7.3.10 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Reserved		DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw			rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 3.73: RCC Peripheral with its corresponding AHB1 register

In setting this register, we used bit shift and masking operation. Now we will use bit fields and structure to abstract these low level operation, and make our code more usable by other users.

Prerequisite:

- section 3.21
  - Review the concept of bit shift, accessing register concept, because we will do abstraction implementation and eliminate all the details, and then *compare both implementation*
- Structures section 3.26 and Unions section 3.27

Project Demo: See `Led Toggles Bit Fields`.

Some Notes:

- In the `.h` file: don't forget to include `stdint.h`
- Compile the header file which contains all the defined structure (by doing right click on `.h` file then `Build project`) directly to see if there is an error in the `.h` file.
- Once we create the `typedef struct` which encode different register, Figure 3.74 shows the implementation and what happen inside the compiler

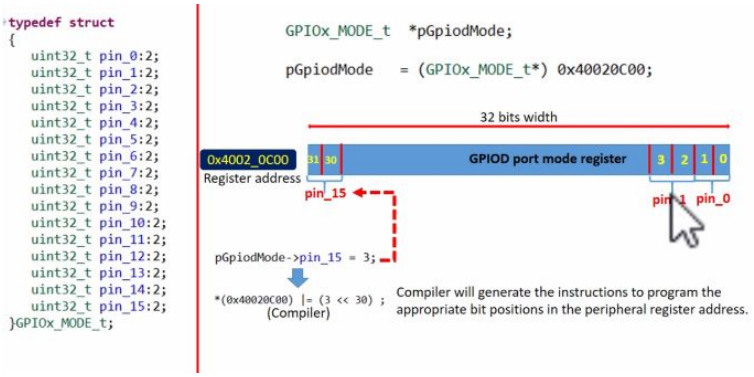


Figure 3.74: Structure implementation and encoding for register

3.29 Keypad Interfacing

In this section we will implement another project for practice: matrix keypad and interface it with the stm microcontroller.

In Figure 3.75, we have the keypad along with its connection.

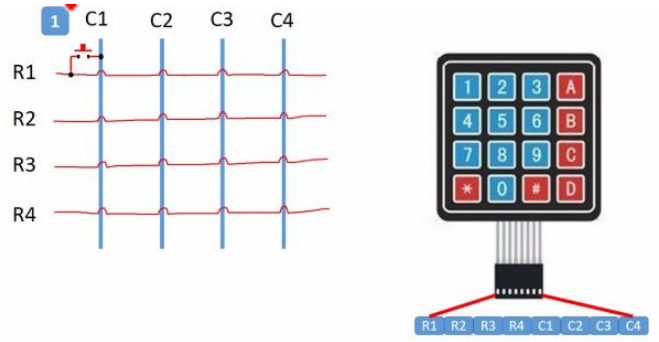


Figure 3.75: Keypad

- The R1 → R4 are called row lines, and C1 → C4 are column lines
- Inside the keypad, we have different keys which connect R and C.
- When the key is not pressed, there is no connection between R and C lines, and otherwise (key is pressed) there will be a contact between a row and a column
- : C are input to the microcontroller, and R are output from microcontroller

A more detailed schematic of the keypad with each keys is shown in Figure 3.76.

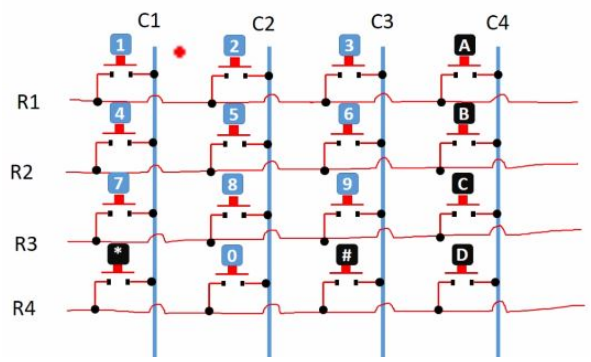


Figure 3.76: Keypad with each different key

In Figure 3.77, we have the connection of the keypad to the microcontroller.

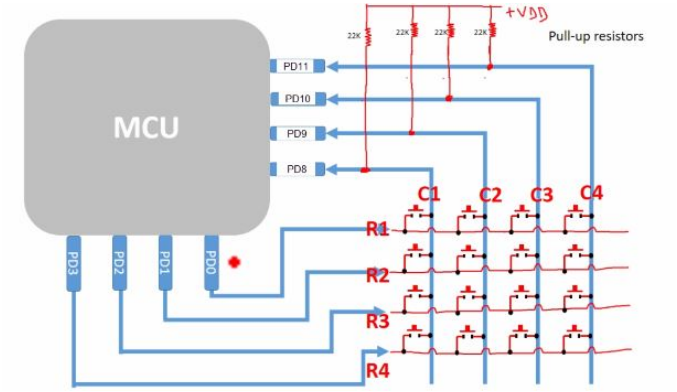


Figure 3.77: Keypad connection with the microcontroller

- We need 8 ***free IO pins*** from the microcontroller
  - We need to go to the reference manual to check whether the pin are free or not (same as we did in section 3.23 with the pin read exercise)
- Also we usually need some pull up resistors concerning the column part
  - We can use the internal pull up resistors that come with the microcontroller. No need for extra resistors

3.29.1 Why Pull up resistors

In Figure 3.78 we have a hand sketch of a key, connected to a MCU.

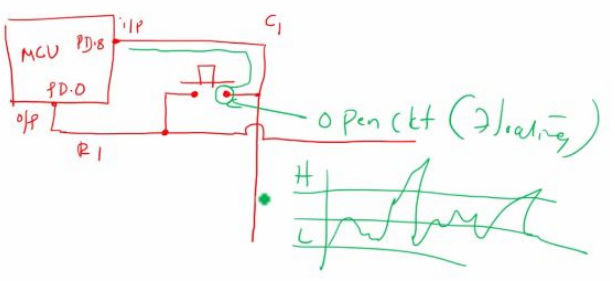


Figure 3.78: Hand sketch of MCU and a Key

When the key is not pressed, the circuit is in open state (open circuit), and hence the key (the pin at C1) can take several voltage values due to random noise coming from circuitry in the open state. This is called *floating state*.

en circuit:

Open circuit: To review later the concept of open circuit, and its relation to noise, $\cdots$.

To avoid such floating state, we connect pull up resistor (it should be of high value) between the pin C and the Vcc as shown in Figure 3.79. We can also use pull down resistor between the pin and ground.

After connecting the pull up resistor, the voltage at pin C is Vcc.

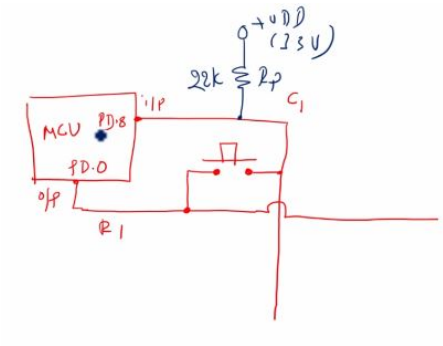


Figure 3.79: Hand sketch of MCU and a Key plus a pull up resistor

Voltage at p C

Voltage at pin C: To review later why this is the value. Need to review circuit analysis concepts.

So now, whenever we read pin `PD8` from the MCU, if the key is not pressed, the value will be high equal to Vcc.

So first rule: when the key is not pressed (and off course there is pull up resistors), C is high, and the input status of MCU is high.

Now the next logical state is when the key is pressed, the C should be low. So here we use the output of MCU (because we can control it) and connect it to ground as shown in Figure 3.80. By this way, when key is pressed, C is connected to ground and the input MCU is low.

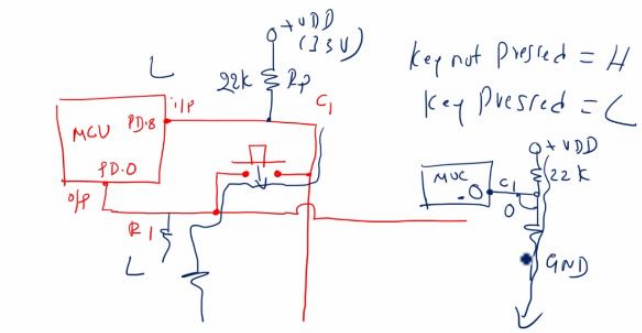


Figure 3.80: Hand sketch of MCU and a Key: output pin connected to ground

2nd Rule: when the key is pressed, the value is low.

3.30 Preprocessor Directive

Preprocessor are the `C` statements which begin with `#` (such as `#include`). They are used as textual replacement for numbers, and other things.
The important thing also is that these statements are taken care during the preprocessing stage of the compilation time, and instruct the compiler to do number of things.

In Figure 3.81, we have different type of preprocessing directive

Pre-processor Directives supported in 'C'

Objective	Syntax of pre-processor directive
Macros	#define <identifier> <value> Used for textual replacement
File inclusion	#include <std lib file name> #include "user defined file name" Used for file inclusion
Conditional compilation	#ifdef , #endif, #if , #else, #ifndef, #undef Used to direct the compiler about code compilation
Others	#error #pragma

Figure 3.81: Different type of preprocessing directives

Now we start with each type of operation in the following subsections.

3.30.1 Macros Preprocessor

In Figure 3.82, we have the syntax of a macros.

Macros in 'C' (#define)

- Macros are written in 'C' using **#define** pre-processor directive
- Macros are used for textual replacement in the code, commonly used to define constants.
- Syntax : #define <Identifier> <value>
- Example : #define MAX_RECORD 10

Figure 3.82: Macros Syntax Example: Replace some constant

Some important notes about macros:

- There is a space between each word
- No ; is required to terminate the statement
- The macros are not variables: they are just textual replacement where this replacement takes place during compilation time
- Use case in embedded programming: define pin number,pin values,crystal speed,peripheral register addresses,memory addresses,$\cdots$.

  Examples: `#define PIN-8 8`

`#define LED-STATE-ON 1`

We use capital case to distinguish between variables and macros text.

Another type of macro is the ***function like macros***, where an example is shown in Figure 3.83.

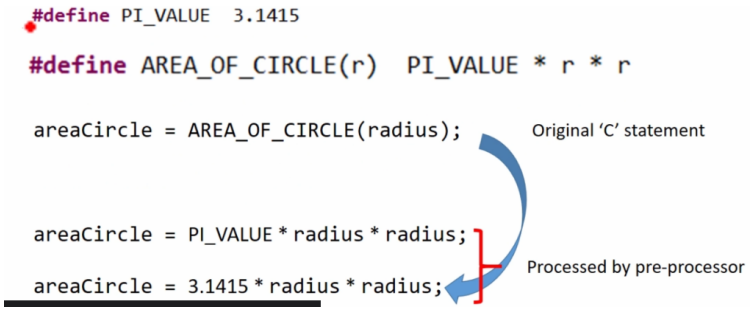


Figure 3.83: Macros function

However, it should be noted that the macros function shown in Figure 3.83 is poorly written, because a user can use the macros as this for example:`AREA-OF-CIRCLE(1+1)`. Because now `r = 1+1` and due arithmetic precedence, we will obtain a different area value then `AREA-OF-CIRCLE(2)`.

To prevent this, we always use `()` when we have function like macros with multiple operands and operations, as shown in Figure 3.84.

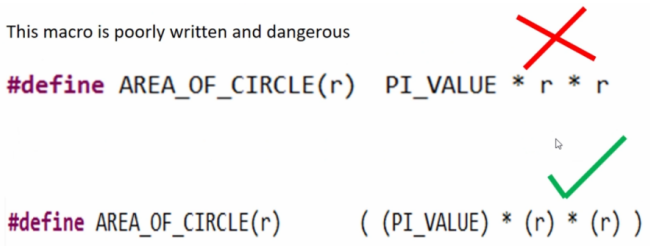


Figure 3.84: Macros function with ()

3.30.2 Conditional Directive

There exist several conditional directive. These are the follows:

- `#if`
- `#ifdef`
- `#endif`
- `#else`
- `#undef`
- `#ifndef`

The goal of these directive is to ***include or exclude block of codes***. Let's start now with each directive.

#if and #endif:
Explanation is shown in Figure 3.85

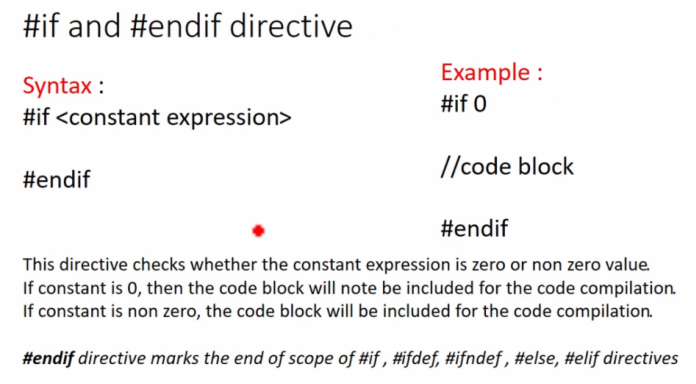


Figure 3.85: #if and #endif

- We have to use constant
- When constant = 0 → code will not be included during build time

Also, we can use `#else` inside the `#if`, as shown in Figure 3.86.

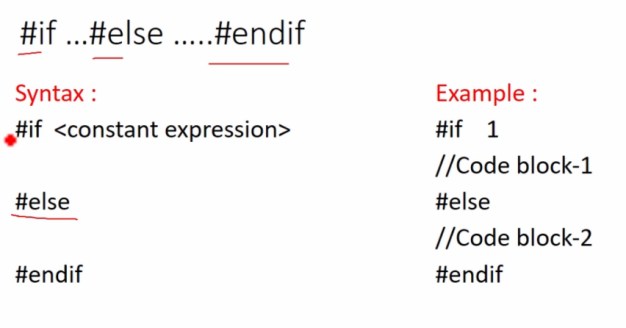
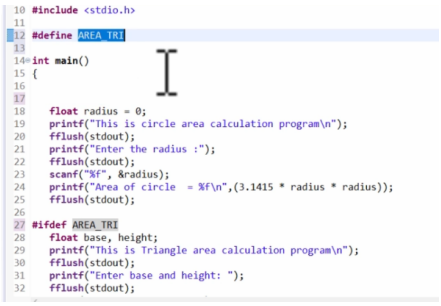


Figure 3.86: #if and #endif with #else

- When constant = 1 → code block 1 will be build, and code block 2 will be excluded
- When constant = 0 → code block 2 will be build (inside the `#else` block will be built), and code block 1 will be excluded

#ifdef:
The `#ifdef` is different from `#if` because we need to use a macros before the `main` function, using the `#define` operator. An example is shown in Figure 3.87.



```
#include <stdio.h>

#define AREA_TRI

int main()
{

   float radius = 0;
   printf("This is circle area calculation program\n");
   fflush(stdout);
   printf("Enter the radius :");
   fflush(stdout);
   scanf("%f", &radius);
   printf("Area of circle  = %f\n",(3.1415 * radius * radius));
   fflush(stdout);

#ifdef AREA_TRI
   float base, height;
   printf("This is Triangle area calculation program\n");
   fflush(stdout);
   printf("Enter base and height: ");
   fflush(stdout);
```

Figure 3.87: #ifdef preprocessor and #define macros

- There is a `#end` at the last to terminate the block (not shown in Figure 3.87)

So in other words, what `#ifdef` is doing is if a certain feature (defined by a macros) is defined, then the block code will be executed.

3.31 Summary: Introduction and Embedded C

1. Each microcontroller contains several peripherals
   - Each peripheral have different registers
   - Each register has its own start and end address
   - Addresses can be found in reference manual, section memory map
2. Ports in stm32 usually have 16 pins
   - Used in connecting external hardware (display,button,Bluetooth transceiver,$\cdots$)
3. Creating project: see section 3.5.

   Rules to remember regarding stm cube IDE:
   - 2 types of project we can create: 1 for host (3.5.3) and 1 for target (3.5.4)
   - Before creating a project, we need to create a workspace (see 3.5.1)
   - Note that when finishing creating a workspace for some directory, we will a `.metadata` folder associated with this workspace.
4. 3.17.6: nice for debugging and monitoring registers in real time
5. Structures (section 3.26):
   - When to use `->` and when to use `.` operator (1 with pointer of type structure and the other normal structure instance)
6. Unions and structure: From the packet exercise (see subsection 3.26.2)
   - In `Packet` project: learn how to compute mask value, and compute bit shift values. But this implementation consume more memory (Instead of 4 byte, it consumes 10 bytes due to structure padding)
   - In `Packet bit fields`: we use bit fields to do proper allocation (*consume less memory*)
   - In `Packets union struct version`: The we use union and struct implementation combined (nested implementation) to avoid bit shift operations and do direct assignment
7. Preprocessor Directives: `C` statements which begin with `#`, instruct the compiler to do certain things during the preprocessing stage in compilation time.

   Rule to remember: with function like macros: we always use `()` when we have function like macros with multiple operands and operations.

   To see more details review section 3.30.

3.31.1 Code Order

Describing order of projects in `stm32f4G-Discovery_Essentials` directory

1. `LED`: turning ON LED
   - Use mask computation in traditional way
2. `LED-bitshift`: turning ON LED but ***using bit shifting concepts***
   - No mask computation
   - See 3.18.4 for complete example.
3. `LED Toggling`: Illustrate loop and delay concepts via software
   - Not very important for now
4. `Packets`, `Packets bit fields` and `Packets union struct version`, in `Host` workspace. Exercise given in 3.26.2
   - `Packets` demo: use normal structure definition (no allocation of bit fields)
     - This cause in structure definition to allocate more bit then needed
   - `Packets bit fields` demo: use structure definition with bit fields
     - All structure members will have 32 bit allocation, and bit field definition takes the necessary amount of bits $\leftrightarrow$ memory reduction
   - `Packets union struct version` demo: use union data type
     - since we are using `union` (variables have same memory location), no need for mask values

Chapter 4

Embedded System on Arm Cortex M4/M3

4.1 Introduction

In this chapter, we study the ARM cortex M family processor. The reason behind studying this family of processors is because it is used in most microcontrollers. An example is shown in Figure 4.1

Figure 4.1: Example of ARM cortex used in industry

The other usage of this processor can be find in section 4.16.

The major competitor for the ARM cortex architecture, is the AVR processors, widely used in arduino board. The AVR processor is manufactured by Microship. Also MSP 430 microcontroller (manufactured by Texas instrument) uses its own processor architecture.

4.1.1 Processor Core Vs Processor

The best way to see the difference is via the ARM technical reference manual.

The cortex M4 block diagram is shown in Figure 4.2.

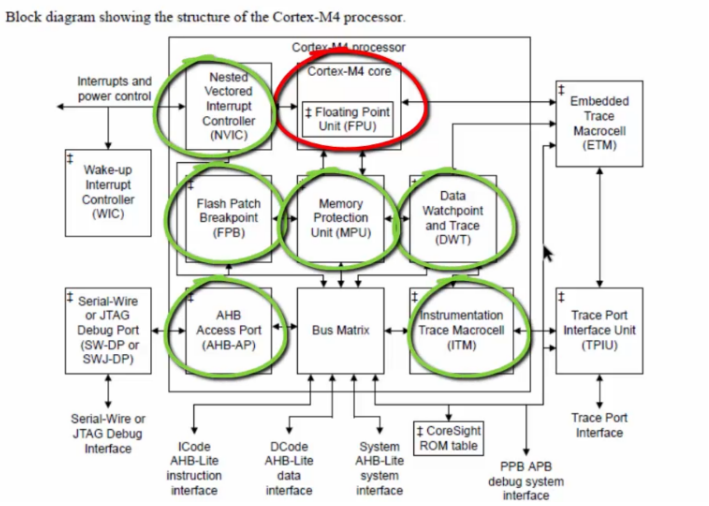


Figure 4.2: AMR cortex M4 block diagram: Core + peripheral

Note: Figure 4.2 can be found in the Arm cortex M4 ***technical reference manual***, section 2.1.

A processor is nothing but a processor core + surrounding peripheral.
Also a processor is called a CPU. Since we have a single core, the M4 processor is known as *single core processor*.

Now the fundamental question: what is inside the components of M4 core?

- ALU to perform computation
- Instruction decoder to decode instructions and fetches them from memory to verify if they are valid instructions or not
- Registers (and special function register to manipulate data)

Later in the chapter will see the details of some of the surrounding peripherals of the core (such as NVIC and the bus matrix).
Note that the processors uses several interfaces (IC code,DC code,$\cdots$) to communicate with the outside world

4.1.2 Processor vs Microcontroller

Now this is a more easy part. Simply, microcontroller is composed from peripheral (timers,GPIO,$\cdots$) and a processor.

The processor (such as Arm M4 cortex) comes in a software IP package, and it is given to the microcontroller vendor (such as stm microeletronics,NXP,$\cdots$), and they implement the interface between their architecture and the processor.

Note: some microcontroller vendors such as Texas instrument implement their own processor architecture, and integrate it in its microcontroller architecture.

4.2 Operations Modes of Cortex Processor

Now starting from this section and after, we will study several features of the cortex M0/M3/M4 processor, which are shown in Figure 4.3.

```
Features of the Cortex M0/M3/M4/ processor
==========================================

1) Operational mode of the processor
2) the different access levels of the processor
3) the register set of the processor
4) the banked stack design of the processor
5) exceptions and exception handling
6) interrupt handling
7) bus interfaces and bus matrix
8) memory architecture of the processor , bit banding , memory
9) endianness
10) aligned and unaligned data transfer
11) bootloader and IAP
```

Figure 4.3: AMR cortex different features

We start with the 1st point: operational mode of the processor.

Note: this discussion will handle cortex family M0/M3/M4, but the general concepts can be applied to other families such as M8 family with some technical differences. In such case, we need to refer to the reference manual as always to cover the different details.

Now we start with general points concerning the operational mode, and then in the next section we implement some code to see these concepts. These general points are as follows:

- The processor usually give us 2 mode: Thread mode (also known as user mode), and handler mode
- The processor always starts with thread mode, and all the application we code run under thread mode
- Whenever the processor meets some exception or interrupts coming from some peripheral, the core will change its mode to handler mode, and interrupt will run under the handler mode

4.2.1 Operation Mode Demo

Note: See project `operation-mode`
First we explain the general flow of the code

- The code starts with thread mode by entering the `main` function
  - Note: Before the `main()`, there is function called `reset-handler()` which will be called, then the `main()` will be called later. But since we don't know the handler yet, we will skip it for now and takes the assumption that `main()` function is executed first.
- When the code enter `generate-interrupt()`, it triggers an interrupt service using software, and the command `*pSTIR = (3&0x1FF)` trigger the interrupt, implemented by the function `void RTC-WKUP-IRQHandler(void)`
- The function `void RTC-WKUP-IRQHandler(void)` will switch the processor into handler mode
- Advantage of handler mode: we can access all resources of our microcontroller

Now how to know if the processor has switched from thread mode to handler mode ? This is done by status register called *interrupt program status register* (found in Table 2-5 from Arm cortex M4 generic user guide), shown in Figure 4.4.

Interrupt Program Status Register

The IPSR contains the exception type number of the current *Interrupt Service Routine* (ISR). See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-5 IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. . . . n+15 = IRQ(n-1)[a] see *Exception types* on page 2-21 for more information.

a. The number of interrupts, n, is implementation-defined, in the range 1-240.

Figure 4.4: Interrupt program status register bit assignments

By the running the code in a debug mode, the `ispr` will have a value different from 0.

4.3 Access Levels

Now we move to other technical term which is called *access levels*. In a processor, we have 2 different type of access levels:

1. Privilege access Level (PAL)
2. Non-Privilege access Level (NPAL)

In Figure 4.5, we have a summary block diagram which summarize the access levels along with operation mode (thread mode and handler mode)

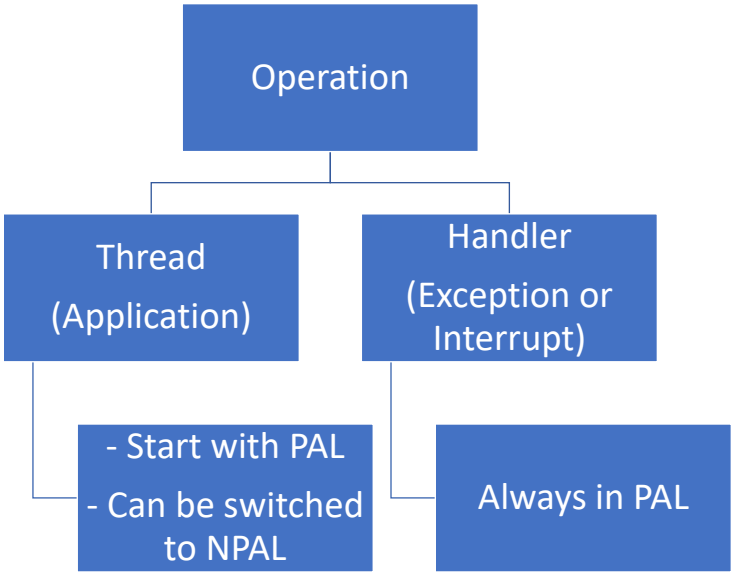


Figure 4.5: Summary: Operation Mode and Access Level Mode

To switch between access levels of the processor, we need to configure the control register.

4.4 Core Registers

Core registers can be found in section 2.1.3 from the generic user guide of the cortex M4 processor. Its decomposition is also shown in Figure 4.6.

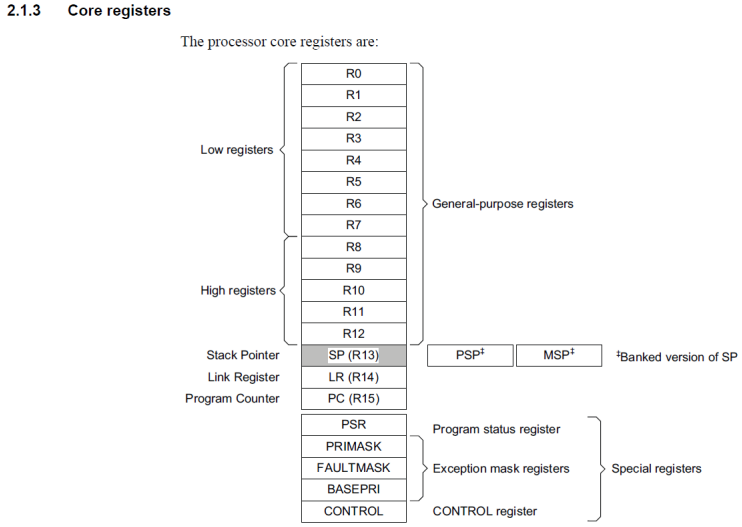


Figure 4.6: Summary: Operation Mode and Access Level Mode

- `R0` $\rightarrow$ `R12` (13 total registers) are general registers, used for example for data manipulation, data storage
- `SP`: stack pointer register, which handle memory, and it can be used in 2 modes: PSP and MSP
- `LR`: it stores the return address value from a function call, in order to return to the caller and continue the flow of the code. An example is shown in Figure 4.7.

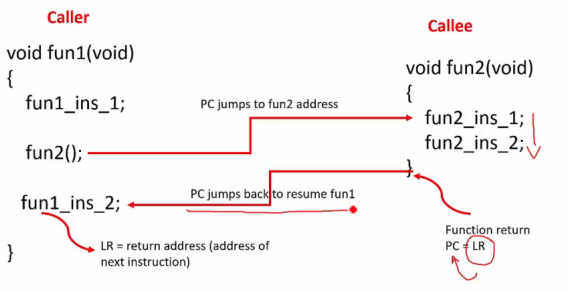


Figure 4.7: LR register flow

LR and assembly debugging : *video 23 contains nice debugging in assembly mode. To redo it later*

LR and asse bly debuggi

- R15, the `PC` register: holds the addressee of the next instruction to be executed

- `PSR`: is the combination of the 3 different register as shown in Figure 4.8

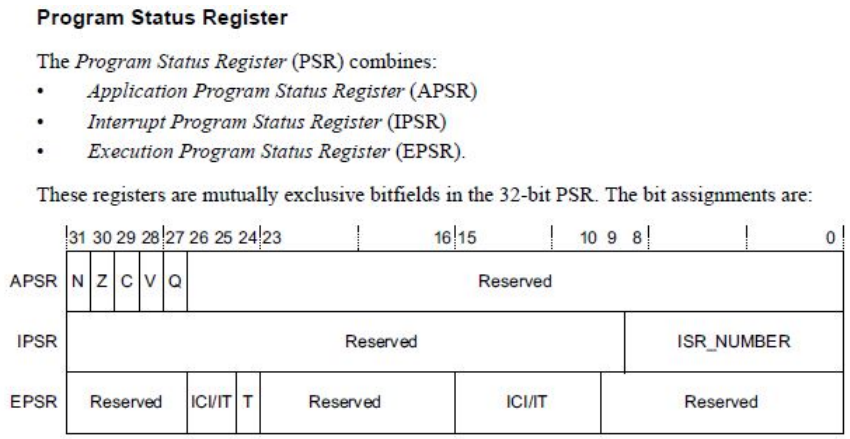


Figure 4.8: PSR registers

  - `ASPR` for conditional flags (detect carry,overflow,$\cdots$)
  - Other will be seen later during coding experiments

4.5 Memory Mapped vs Non-Memory Mapped Registers

In section 4.4, we saw briefly different register relative to the core cortex M4. These registers are ***non memory mapped***. Likewise, there exist some registers (relative to cortex processor, or for the MCU such as timers,GPIO,$\cdots$), which they are ***memory mapped***, and we can access them using our `C` code. Figure 4.9 illustrates the differences between memory and non-memory mapped registers.

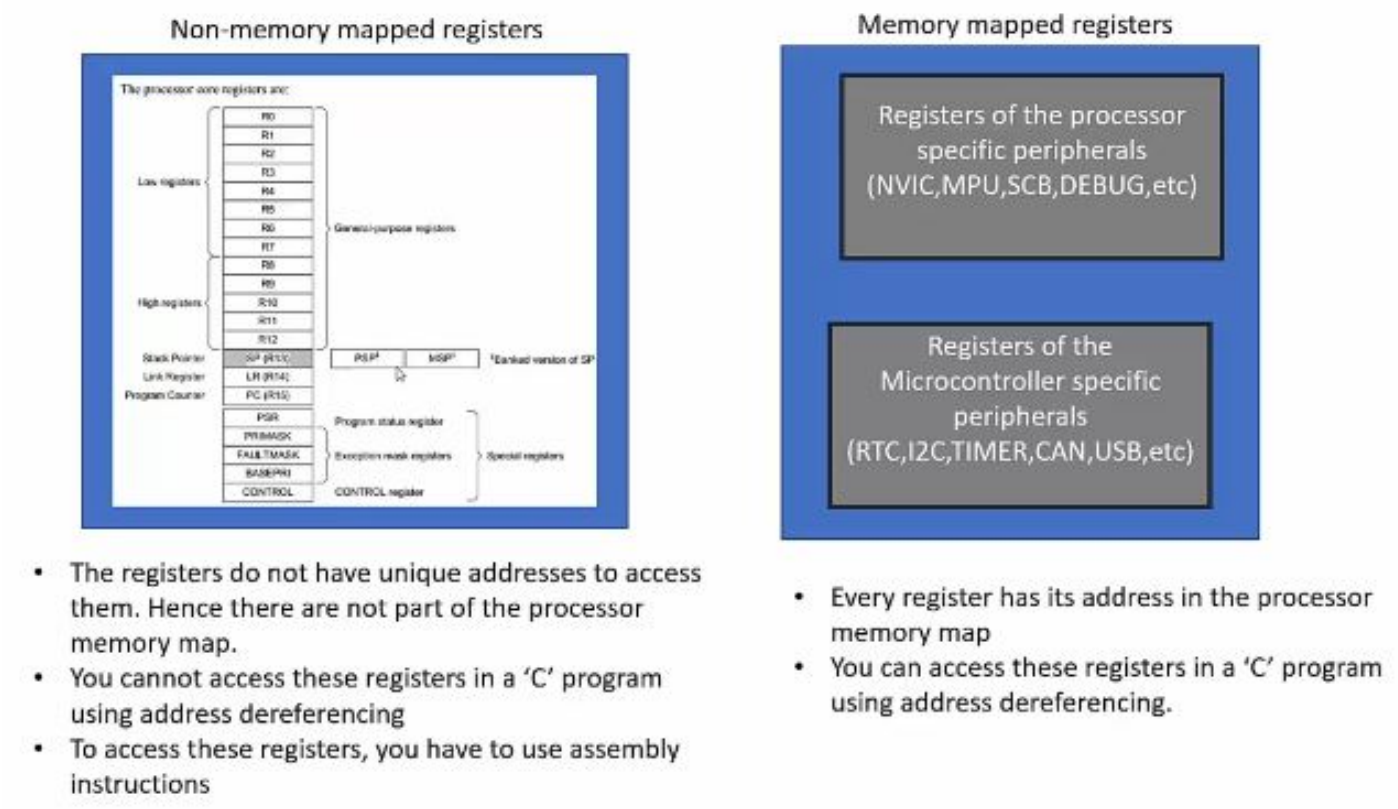


Figure 4.9: Memory vs Non-Memory mapped registers

4.6 ARM GCC Inline Assembly

In this section we learn how to write inline assembly language. By inline we mean that the assembly language will be inside our `C` program. The purpose behind this if we want to control some registers which are non memory mapped, and do some manipulation between the `C` code and the arm cortex registers (example: move some result coming from a `C` variable to some arm registers).

Note: the syntax described in this section is only specific to ARM GCC compiler. Other compiler have different syntax.

Figure 4.10 contains example and the general syntax of arm assembly relative to ARM GCC compiler.

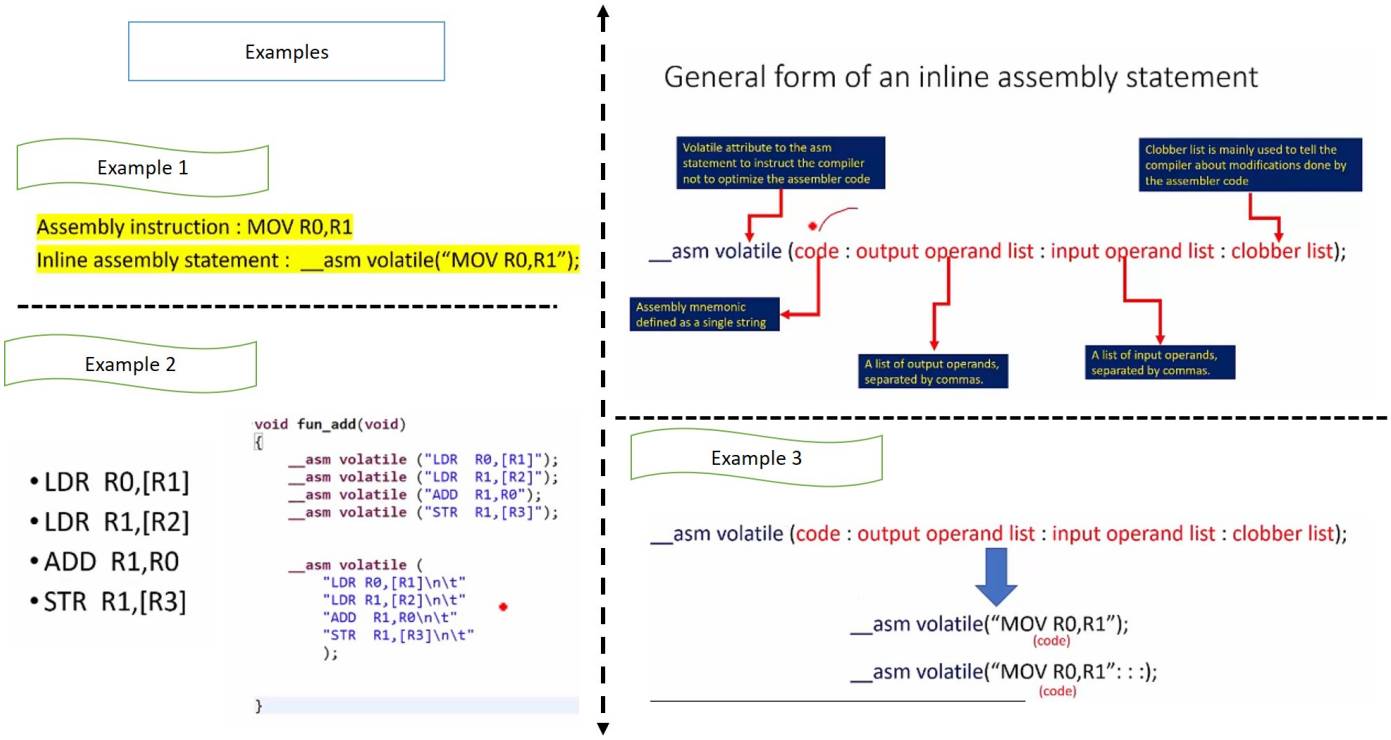


Figure 4.10: Inline Assembly: Examples and General Syntax

- We always begin with a leading double underscore
- In example 1: we embrace the code by " " and terminate it by a ;
- Example 2 shows 2 different syntax: the 2nd one is more compact
  - But notice no ; as in the 1st
- In all the examples, there is only the code par: no operands and no clobber

4.6.1 Code Ex 1

Given: Load 2 values from memory, add them and store the result back into the memory using inline assembly language.

Code Demo

Code Demo: There is next some simple code demo illustrating the language, nothing important, maybe to do it later. This is done in video 29.

4.6.2 Code Ex 2: C to ARM Reg

Given: Move the content of `C` variable to ARM reg.

In this type of exercise, we will use input operand, and not just code section. The code is shown in Figure 4.11.

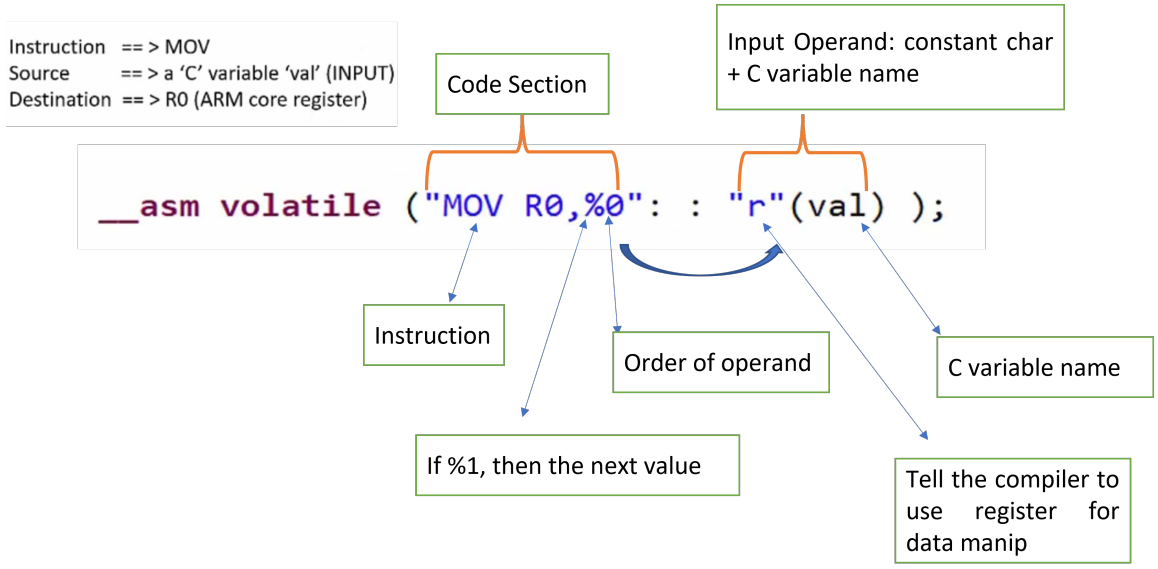
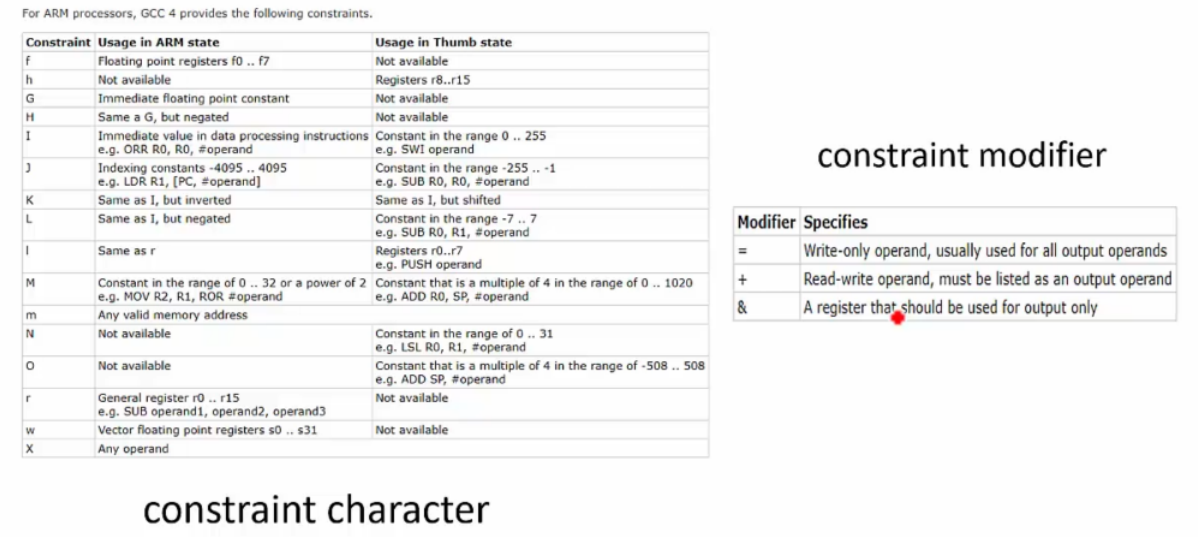


Figure 4.11: C to ARM Register

In order to see different constant character other then `r`, contains all possibilities.



For ARM processors, GCC 4 provides the following constraints.

Constraint	Usage in ARM state	Usage in Thumb state
f	Floating point registers f0 .. f7	Not available
h	Not available	Registers r8..r15
G	Immediate floating point constant	Not available
H	Same a G, but negated	Not available
I	Immediate value in data processing instructions e.g. ORR R0, R0, #operand	Constant in the range 0 .. 255 e.g. SWI operand
J	Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand
l	Same as r	Registers r0..r7 e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand
O	Not available	Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand
r	General register r0 .. r15 e.g. SUB operand1, operand2, operand3	Not available
w	Vector floating point registers s0 .. s31	Not available
X	Any operand	

constraint character

constraint modifier

Modifier	Specifies
=	Write-only operand, usually used for all output operands
+	Read-write operand, must be listed as an output operand
&	A register that should be used for output only

Figure 4.12: Different constraint character

4.6.3 ARM to C variable

Given: Move the content of `CONTROL` reg to `C` variable `control-reg`.

In this exercise, we can see that `C` variable is in the output operand, as shown in Figure 4.13

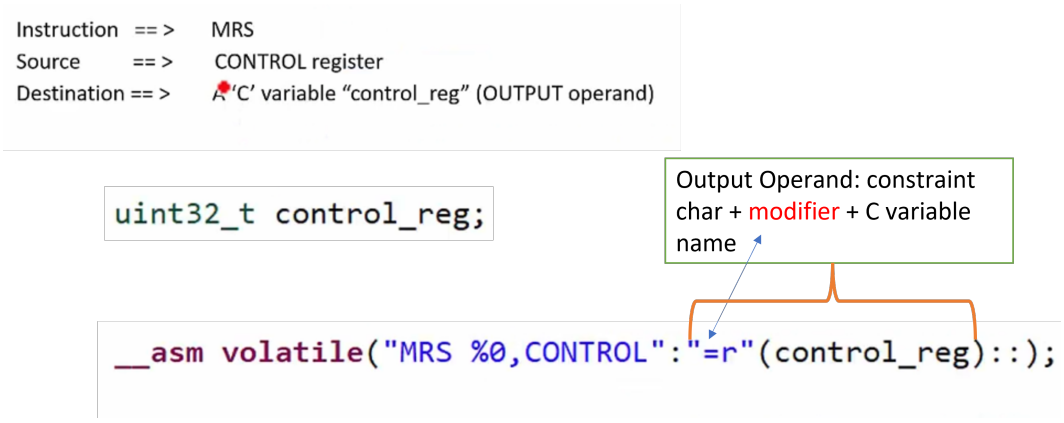


Figure 4.13: C to ARM Register

- We use the constraint modifier `=` to indicate a write operation

ra Ass Examples

Extra Ass Examples: *There are some extra examples also, maybe see them later.*

4.7 Reset Sequence

Now in this section, we discuss what happens when we press the reset button in our board: its mechanism, and what happen behind the scenes.

The reset handler function:

1. doing some initialization
2. Call the `main` function

Figure 4.14 shows this concept.

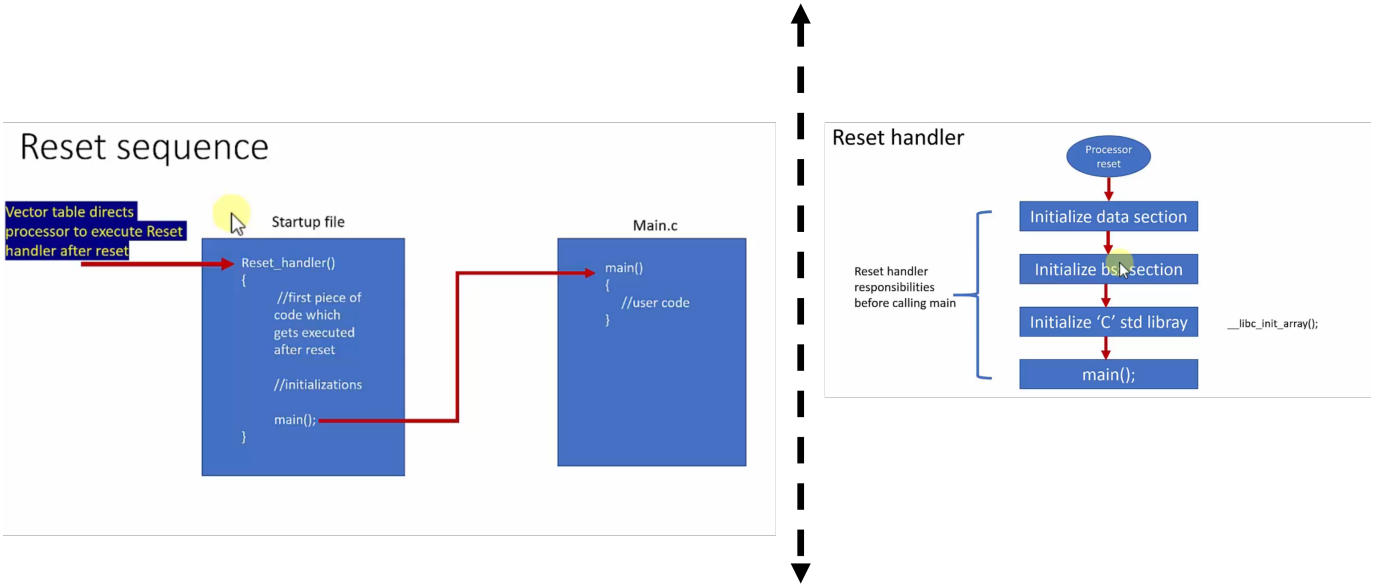


Figure 4.14: Reset Handler Task

Now as regarding the steps, Figure 4.15 illustrate this steps from register side:

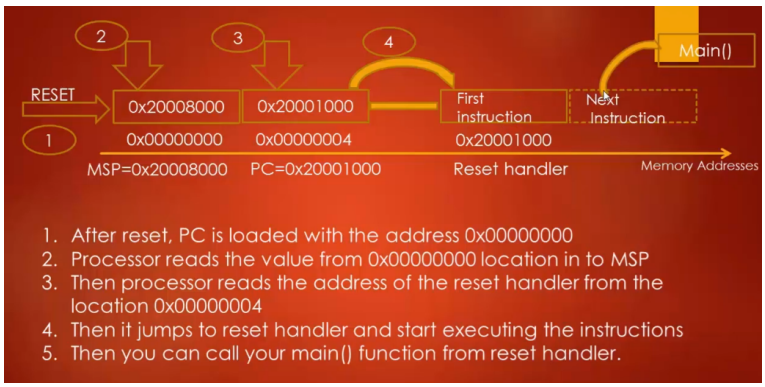


Figure 4.15: Reset Handler steps with registers

- Registers: `MSP`: memory stack pointer , `PC`: program counter
- Notice that `PC` stores the address of `Reset` function which is 0x20001000 at location 0x0000 0004.

4.8 Demo for Access Level

Now in this section, we see how to change access level.
Review:

- We know from section 4.2 that the code starts in thread mode, with PAL access.
- In the demo code of 4.2.1, we switch the code from thread mode to handler mode using an interrupt

Now the big idea about privilege and unprivileged mode:

- Suppose we have some real time OS and some user task
- The user task shouldn't modify the kernel system (like switching off or ON a ISR)
- That's why sometimes we switch from thread mode PAL to thread model NPAL
- Because in NPAL we can't access the resources of the kernel and modify it

T bit in `ESPR`: T bit should always set to 1.

it

T bit: To watch the video later about the T bit, thumb state and so on, and why ARM is in thumb state always. This is video 33 and 34 from section 8.

4.9 Memory Map

In this section, we discuss memory map of the processor and processor communicates with different peripheral of the MCU. Figure 4.16 shows block diagram about communication concepts: (a) using the bus interface), and the memory map part (b)

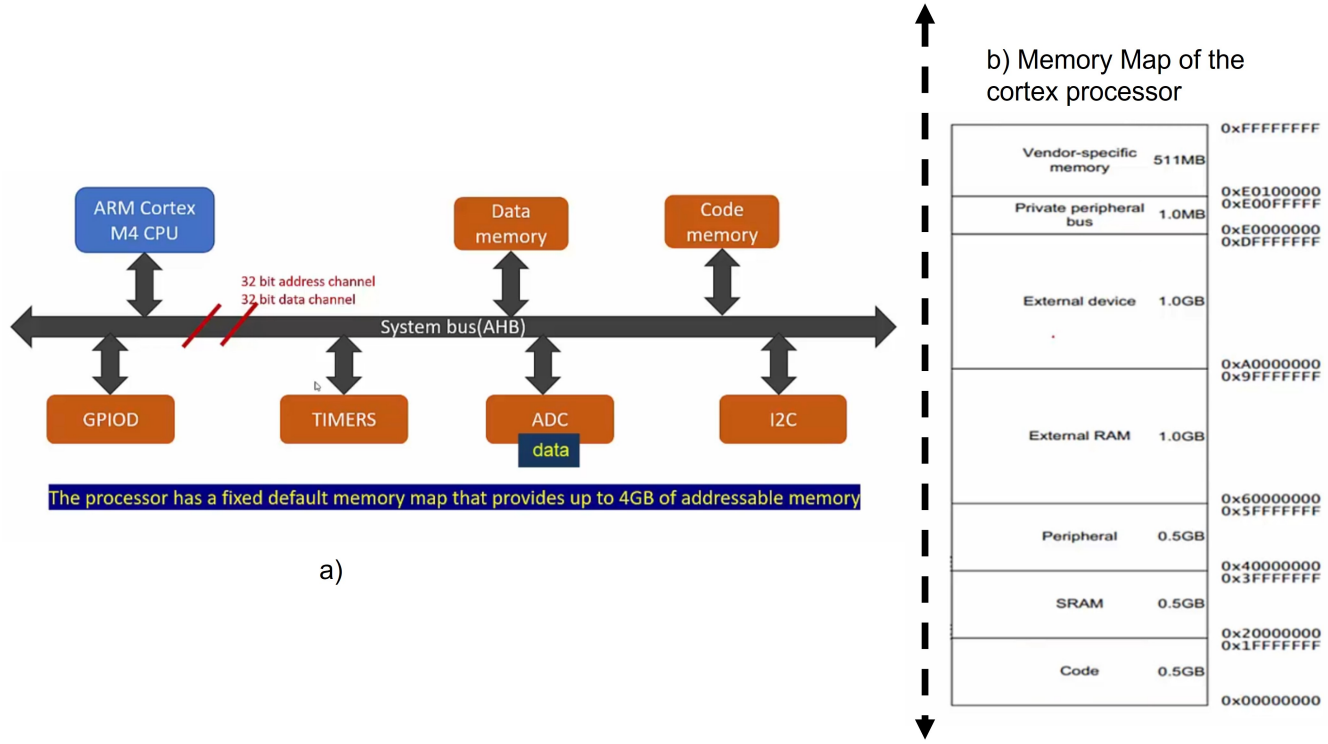


Figure 4.16: Communication between Processor and different peripheral

Suppose that the ADC have some data in it, and we want to use this data in our software, so we need to store these data in the *data memory*. The steps are:

1. The processor generate addresses, and compare them to the address in which the data is hold in the ADC register (call it `RegADC`)
    - Here comes the role of the memory map. For example, when the processor wants to communicate with the ADC, the generated address should be in the peripheral range (0x4000 0000 $\rightarrow$ 0x5FFF FFFF)
2. Once the address of `RegADC` is found, `RegADC` will be unlocked and its content (the data we need) will be sent using the data bus to the processor, and stored in some register of the processor
3. The move it to the Data peripheral part

These steps can be summarized using 2 instructions:

1. `load` instruction from the ADC to the processor
2. `store` instruction to store the data in the data memory

One thing to be noted also: the peripheral space map range is used in the design when specifying the needed addresses of the peripheral of the MCU (such as timers,ADC,$\cdots$).

4.9.1 Bus Interface

From previous explanation, we can see that the communication is done using buses. IN ARM architecture, there are 2 bus:

1. AHB bus:
   - High speed communication with peripheral that demands high speed operation
   -
2. APB bus:
   - Low speed communication
   - For PPB (private peripheral bus region) such as NVIC, system timer,$\cdots$

Design Concept: we have 2 bus (AHB and APB) because by this way, MCU vendors can put the peripheral which don't require high speed on low performance bus (the APB in our case) and reduce power consumption.

4.9.2 Bit Banding Feature

Now we explore a feature called *bit banding*

- Definition: capability to address a single bit of a memory address.
- Not all memory map are bit banded. As shown in Figure 4.17, only SRAM and the peripheral sections are bit banded.

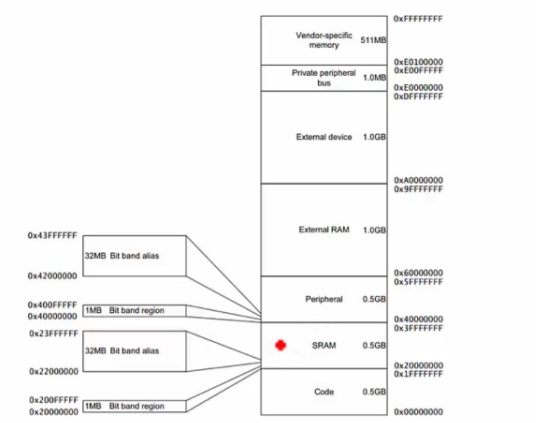


Figure 4.17: Bit banding: allowed section of the memory map

- How to address the bits: via the equivalent alias address as shown in Figure 4.18.

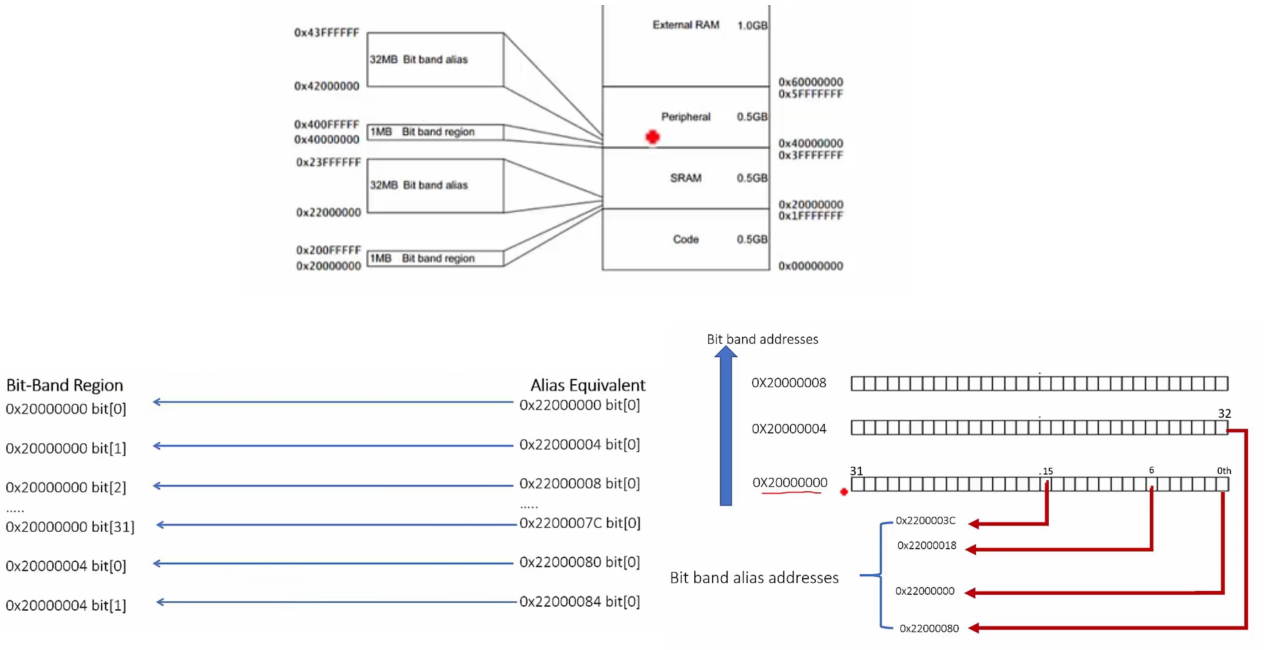


Figure 4.18: Bit banding mechanism

For each bit, correspond an alias address.

Bit banding code

Bit banding code: to redo the coding exercise later. See video 38 section 9.

4.10 Stack

Main Concepts:

- Stack is a memory used when executing functions,or interrupts/exception, and also to store variables
- Stack operation mode: they are general 4 operation mode as seen in Figure 4.19

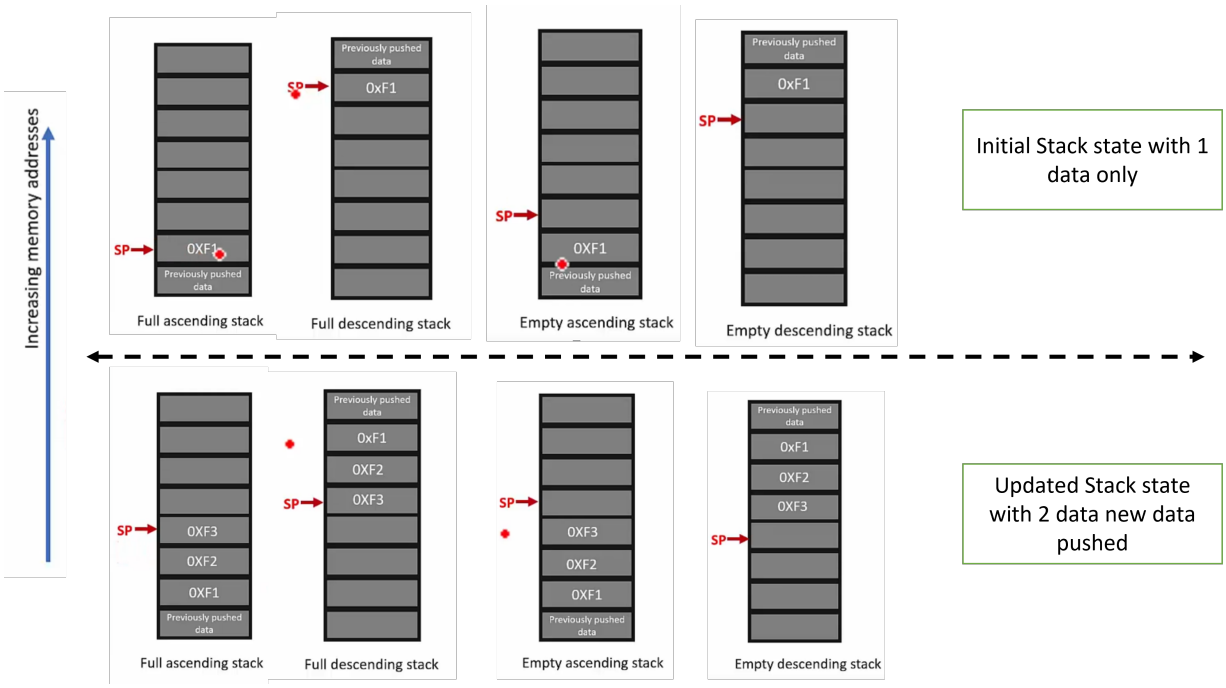


Figure 4.19: Stack Operation Mode

  - Ascending mode: `SP` is increasing
  - Empty mode: `SP` points toward empty memory address (ascending and descending mode)

In arm cortex, we use full descending

- Stack placement: we can place the stack in many ways as shown in Figure 4.20. Usually we use the 2nd one.

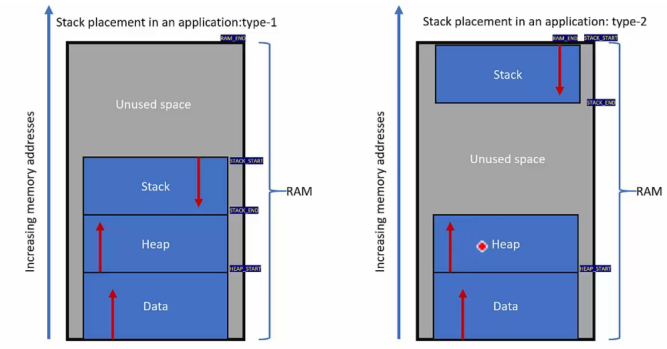


Figure 4.20: Stack Operation Mode

- Bank stack pointer: if we go to the user guide of the cortex M4 processor, we see that at `R13` we have a bank of pointers as shown in Figure 4.21.

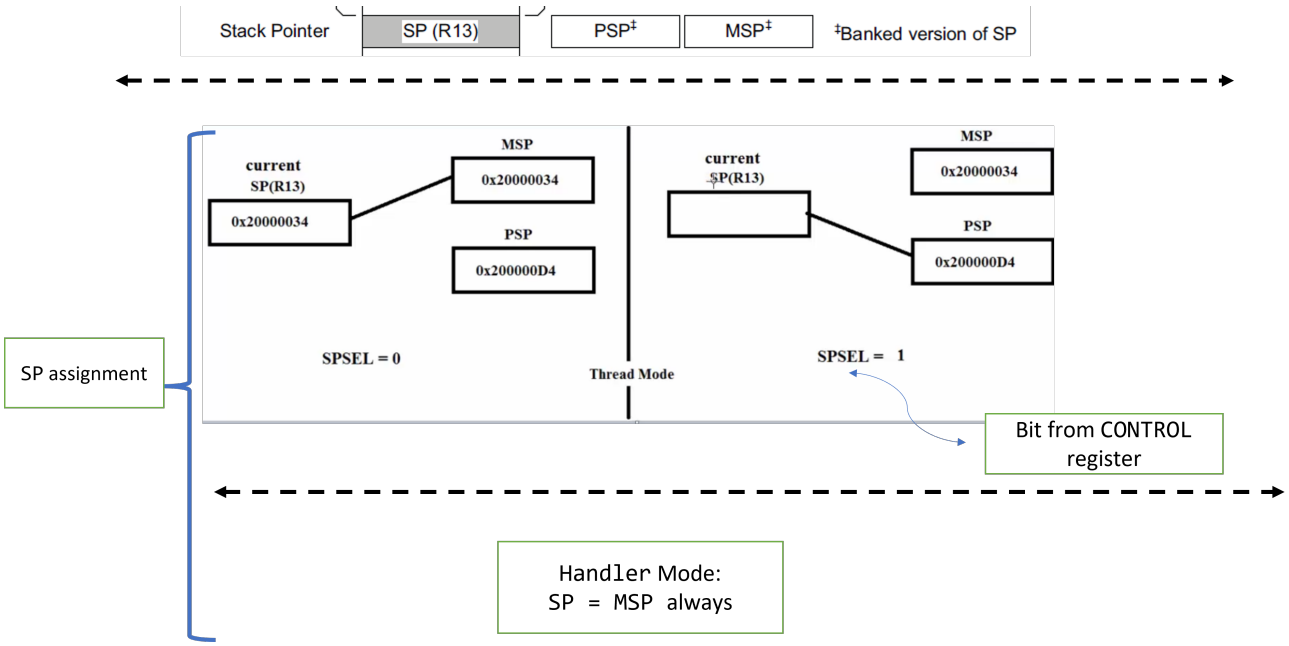


Figure 4.21: Stack Pointer assignment in thread and handler mode

  - Initially after reset, `SP` read the content of `MSP`. We can change it to `PSP` by changing the bit value of `SPSEL` from the `CONTROL` register.
  - In handler mode, `SP = MSP` always

4.10.1 Stack Exercise

Stack Ex

Stack Ex: Implement a code where we change the SP to MSP. To do it maybe later. See section 10 video 43 and 44.

4.10.2 Function Call for Arm Architecture

When calling a function, Arm uses a procedure called AAPCS. This procedure is mainly used when writing routines in assembly. As an example, Figure 4.22 shows an example of this standard, when we have some function calling.

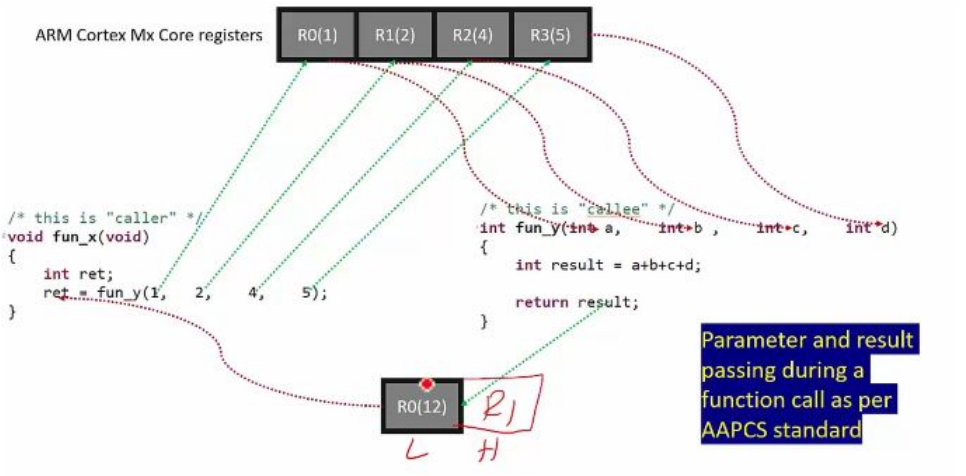


Figure 4.22: Stack Pointer assignment in thread and handler mode

- The caller responsibility is to copy the data to the registers (`R0,R1`,till `R3`)
  - If more the 4 arguments is used, then we use the stack to store these arguments
- Then the callee copy from the those registers the value to their local variable
- The result is always stored in `R0`, and if we need more then 32 bits, we use `R1` where higher bits are stored.

4.10.3 Stack during Exception and Interrupt

When calling some function, the AAPCS protocol will be followed. In a big picture, each of caller and the callee has some responsibilities. However, when we have some exception, ***there is no caller*** (in other words no AAPCS protocol to apply), because exception and interrupts can be called any time (they are asynchronous in nature), and they are called the processor hardware. At this phase, all the correspondent registers will be saved in the stack frame by the processor.

4.11 Exceptions

Now we present the concepts of exceptions in a processor. Figure 4.30 contains definitions and their types.

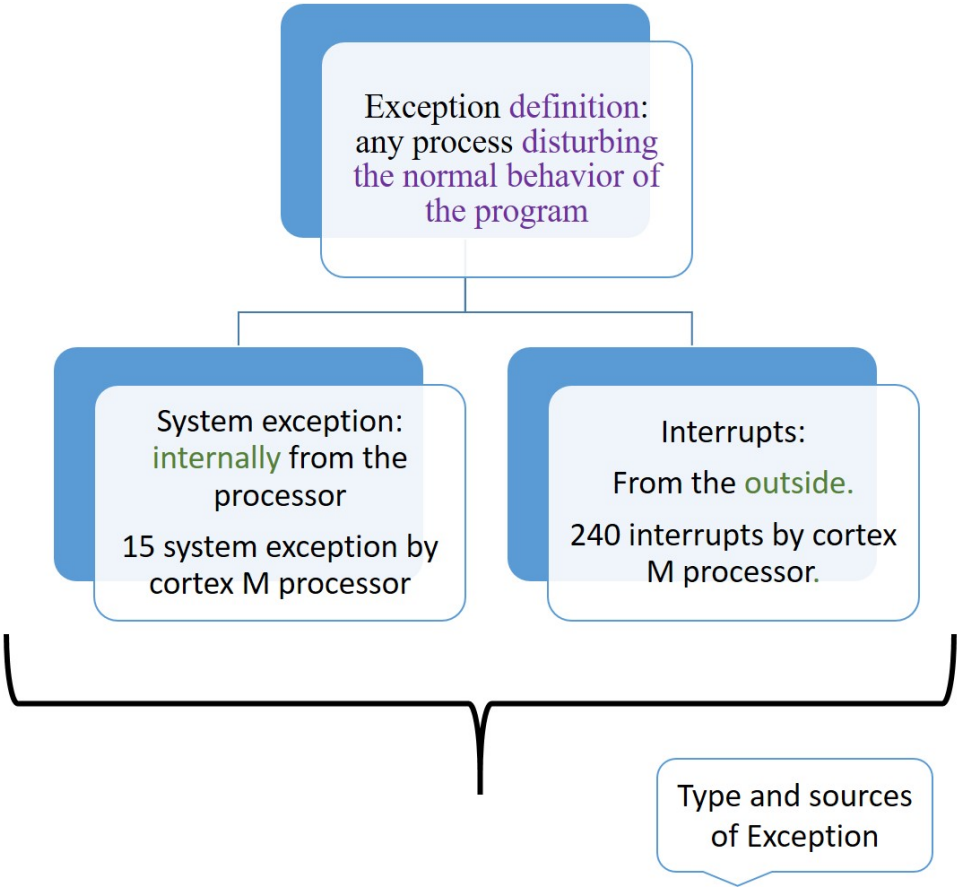


Figure 4.23: Exceptions: definition and type

- Exception is any type of process which break the normal flow of a code
- It can comes from inside the processor or outside the processor

4.11.1 System Exception

For arm cortex m4, we can see the different system exception in generic user guide, section 2.3 named Exception model.

The m4 cortex has as we said earlier (in Figure 4.30) has 15 system exception, all illustrated in table 2-16
In general, what we can retain from reading the table and the description, that each exception is triggered by some specific event. Example:

- UsageFault if we try to divide by 0

Note: read todo note later, to see why there is not enough content or info

tem Excep-
ns

System Exceptions:

- *System Exception is in video 48 of the course*
- *There were allot of theory and no practice about system exceptions, so for now I didn't write any, as infor as simply stated in the manual*
- *hardFault exception is encountered in the T bit banding excercice*
  - *So maybe to do this later*
- *Maybe after completing the course, to re-write this section later*

4.12 System Exception Vector Address

System Exception can be found in the reference manual, in what is called a ***vector table*** (table 62 page 357).

Vector Tabl

Vector Table:

- *To see later what is that vector table, and the difference between memory map and the vector table*
- *In the lectures, I don't remember explaining such kind of tables*

4.13 System Exceptions Control Registers

In this section we explore processor registors responsible for the system exceptions (exceptions that come from inside the processor).

The arm cortex processor comes with different peripherals, as show in Figure 4.24, where each peripheral comes with its own register set to control a peripheral.

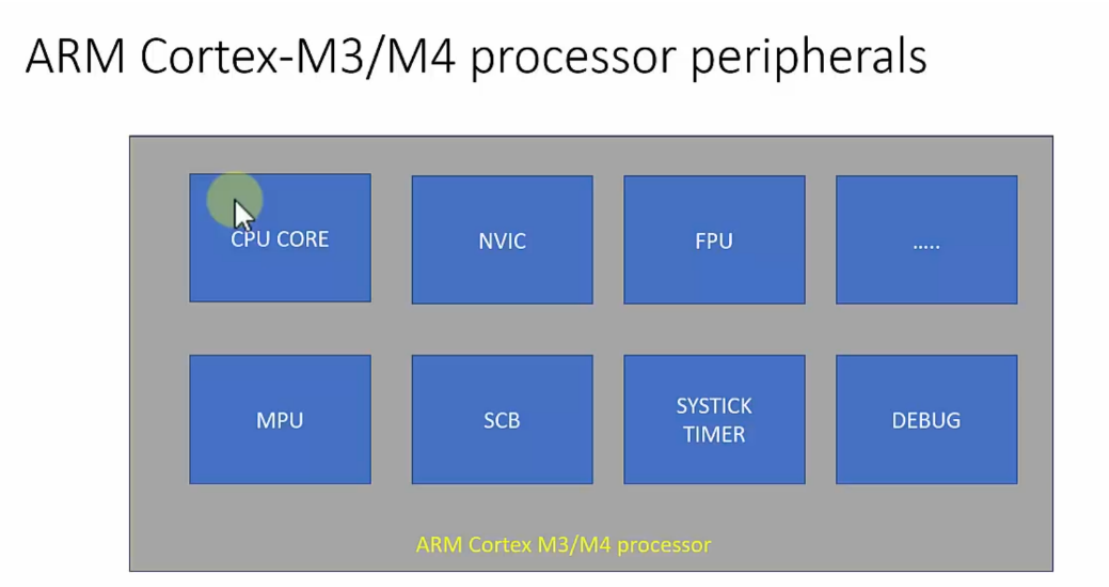


Figure 4.24: Arm cortex Peripherals

- See section 1.1.14 in the generic user guide for peripheral descriptions.

These peripherals can be accessed via the PPB (private peripheral bus), where its addresse map is shown in Figure 4.25.

4.1 About the Cortex-M4 peripherals

The address map of the *Private Peripheral Bus* (PPB) is:

Table 4-1 Core peripheral register regions

Address	Core peripheral	Description
0xE000E008-0xE000E00F	SyStem Control Block	Table 4-12 on page 4-11
0xE000E010-0xE000E01F	System timer	Table 4-32 on page 4-33
0xE000E100-0xE000E4EF	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
0xE000ED00-0xE000ED3F	System Control Block	Table 4-12 on page 4-11
0xE000ED90-0xE000ED93	MPU Type Register	Reads as zero, indicating MPU is not implemented[a]
0xE000ED90-0xE000EDB8	Memory Protection Unit	Table 4-38 on page 4-38
0xE000EF00-0xE000EF03	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
0xE000EF30-0xE000EF44	Floating Point Unit	Table 4-49 on page 4-48

Figure 4.25: PPB Adresse map

- The table in Figure 4.25 can be found in the generic user guide of the arm cortex, page 218.

4.14 NVIC

NVIC stands for nested vector interrupt controller, and its one of the arm-cortex peripherals.

The NVIC is used to control the 240 interrupts external to the processor. In other words, it controls the traffic of the interrupts coming to the arm processor.

Note:

- recall that external means that these interupts are not generated by the processor.
- for interrupts, the NVIC hardware is responsible to control the traffic of the interrupt
- for the system exceptions (interrupts generated from inside the processor), the system control block (see Figure 4.25, where its mentioned its addresse space) is the entity responsible

4.14.1 NVIC doc

The NVIC detailed description can be found in the arm generic user guide, section 4.2.

4.14.2 Interrupt sources

Now a question arise: what are exactly these 240 interrupts ?

Well the answer to that question can't be found in the arm generic user guide, but rather in the reference manual of the microcontroller (table 62 page 357).

In stm32f407 discovery board, among the 240 interrupts provided by arm, stm (the microcontroller vendor) has implemented 82 interrupts.
We can see them in table 62, spanning from $0 \rightarrow 81$.

So in other words, the microntroller vendor is responsible of the external interrupt. These interrupts are coming from the micrcontroller peripherals, such as GPIO, SPI, timers,$\cdots$.

4.14.3 Generic design

To recap, figure Figure 4.26 present a generic design.

- In the right hand side, we have the arm processor
  - The NVIC peripheral is shown along with its 240 lines
- The lines of the NVIC takes input form the peripheral of the micrcontroller
- Each micrcontrtoller vendor has the choice of design to select which peripheral uses whihc line to send an interrupt signal
- Example: st for example, uses line 0 for the watchdog pereipheral

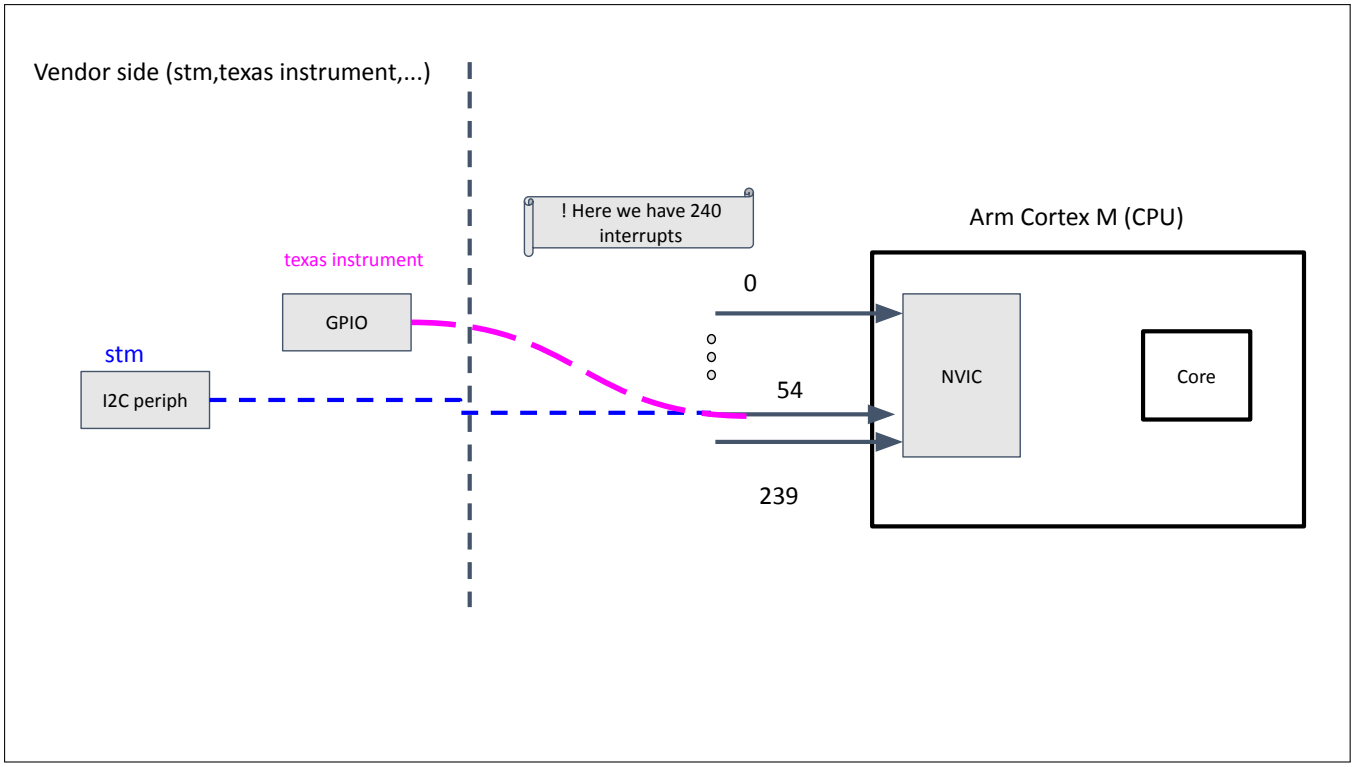


Figure 4.26: PPB Adresse map

  - since watchdog uses line 0, its number is called IRQ0

- 2 vendors (such as stm and texas instrument for example), can use the same IRQ number, but to take interrupts from 2 different peripherals

  - We have an example for IRQ54 in Figure 4.26, where the line is used by 2 different peripheral for each vendor

4.14.4 NVIC Registers

To explore the NVIC registers, we open the arm-generic user guide, section 4.2. This section describe all registers, and also there is table Table 4-2, which contains a register summary.

Now let's begin the sruvey.

Interrupt Set-enable Registers:

- there are 8 registers (from $0 \rightarrow 7$) to handle the 240 interrupt
- If we take for example `NVIC_ISER0`, we can set 32 interrupts, since this register contains 32 bit.
- `NVIC_ISER0` can be ***used only to enable interrupt*** $\leftrightarrow$ we can't disable an interrupt via this register

Interrupt Clear-enable Registers:

- also we have 8 registers to cover the 240 interrupts
- this register is used to disable an interrupt, but setting some bit to 1 (again 0 has no effect)

Interrupt Active Bit Registers:

- Also 8 registers we have
- Used to see which interrupt is active (executed by the processor)

ding *Pending*:

- *There is what is called Interrupt Set-pending Registers and Interrupt Clear-pending Registers*
- *The concept of pending I didn't understood it yet, since we didn't work with some example or code*
- *See video 52 when he describe the pending, and the priority concept related to it*
  - *Again, the concept of priority wasn't explain also*

4.15 Peripheral Interrupt Exercice

Now we will do some coding exercice, about the interrupt generated by some peripheral of the micrcontroller. The description is shown in Figure 4.27.

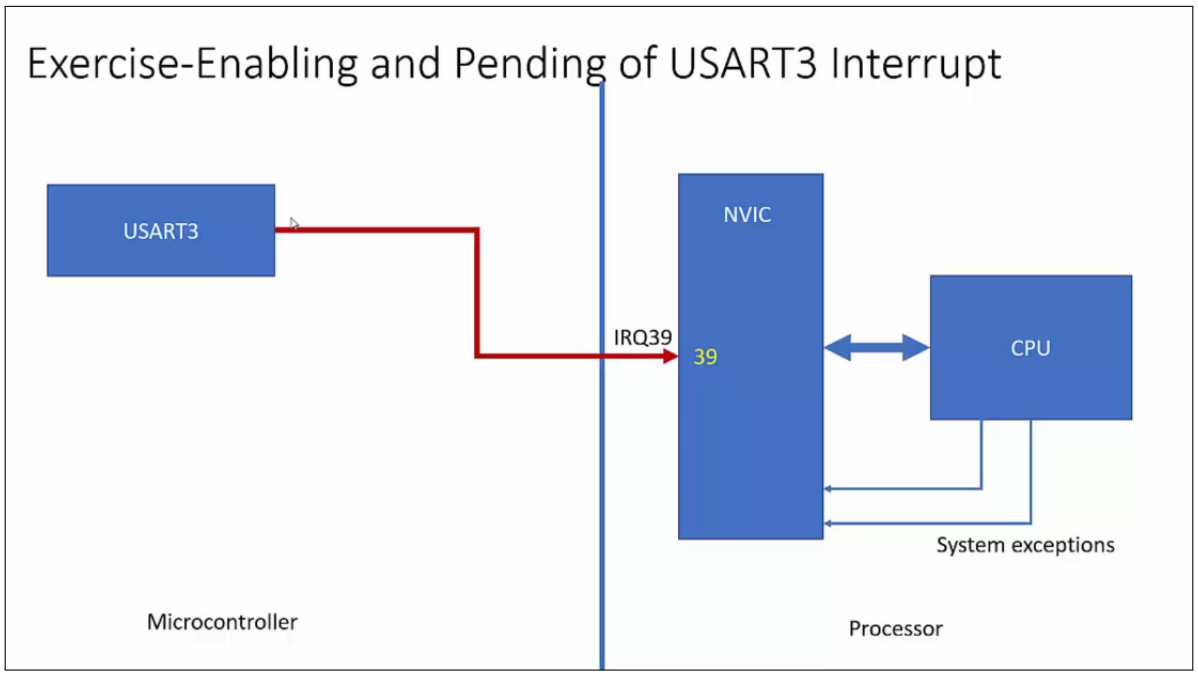


Figure 4.27: Interrupt by USART Periphral

- The peripheral is USART3
- If we look to the reference manual (table 62), the USART3 peripheral has and IRQ number equal to 39

4.15.1 Interrupt programing steps

Before diving into the code, let's state some general steps for interrupt programming

1. Identify what is the IRQ number for the particular peripheral we are working with
   - Recall that IRQ number are vendor specific ↔ stm has different attribution for IRQ number the texas instrument for example
2. Enabling the IRQ via registers of the processor (that is the Arm cortex registers, document in the generic user guide)
   (a) we can also the configure the ***priority***, which is optional
   (b) By default, and interrupt has the higher priority, equal to 0
   (c) We will see later interrupt prorities in another sections
3.

- *To review later the paragraph above*
- *See video 53 at 5:31*

In Figure 4.28, we have a diagrame of how an interrupt will be executed

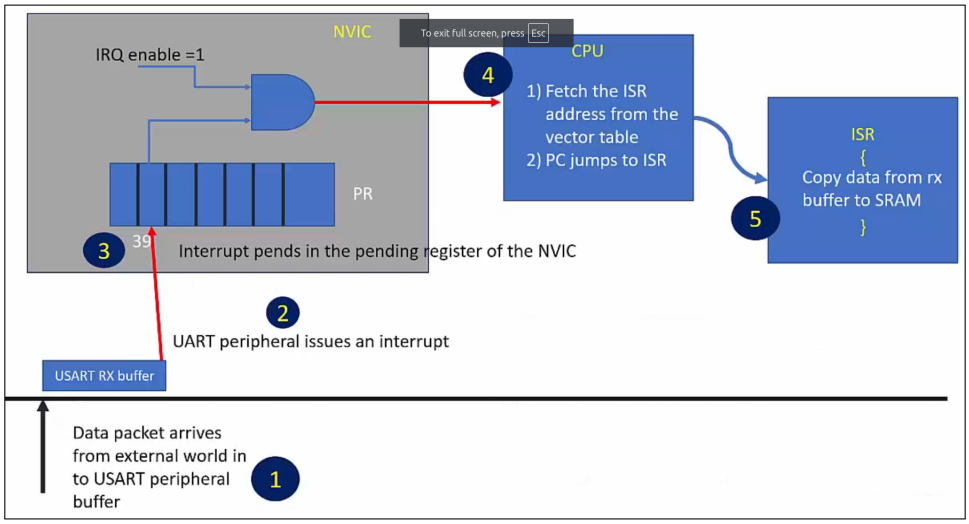


Figure 4.28: Interrupt by USART Periphral

- Interrupt are ***triggered by some events always***
  - In the case of USART, is the arrival of a packet in the Rx buffer (step 1)
- Once the USART3 Rx buffer is full, it triggers an interrupt on the IRQ line, and make the line high or low (step 2)

  Until now, step 1 and 2 are from the peripheral side
- When the interrupt is issued, it will set the pending state in the pending register of the NVIC (step 3)
  - now we start processor side sinc we are in the NVIC
- If the IRQ is enabled, the NVIC will allow the interrupt to be executed and send it to the processor (the CPU) $\leftrightarrow$ the NVIC interrupt the CPU (step 4)
- When the interrupt is accepted by the CPU, the CPU will
  - fetches the ISR address for that IRQ number in the vector table
  - the processor (program counter) jumps to the ISR
- In the ISR, we can copy the data from the RX buffer to the memory (the SRAM)

4.16 Summary: Embedded System on Arm Cortex M3/M4

- Motivation behind ARM cortex
  - ARM cortex processor are used in most microcontoller manufacturing because of its minimal cost (save money),minimal power,minimal silicon area
  - It is based on 32 bit architecture which can boost computational power, and has same price as traditional 8 and 16 bit processors
  - It can be customizable to include different units (DSP units, memory protection units,floating point units,$\cdots$), depending on the target application
  - RTOS friendly
  - Instruction set is rich and memory efficient Instruction

    Instruction set: *to be reviewed later why it is efficient.*

    The instruction set concept maybe I saw it in previous chapter, to be seen later.

- Key concepts
  - A processor is nothing but a processor core + surrounding peripheral.
  - A microcontroller = a processor + some other peripheral, connected by many *buses*
  - Keep mind that not all vendors uses ARM cortex architecture (some of them use their own CPU architecture in their microcontroller)

- See the assembly of the `C` file:
  1. Build the project
  2. Right click on the project
  3. `Properties`, a pop window will appear as shown in Figure 4.29.

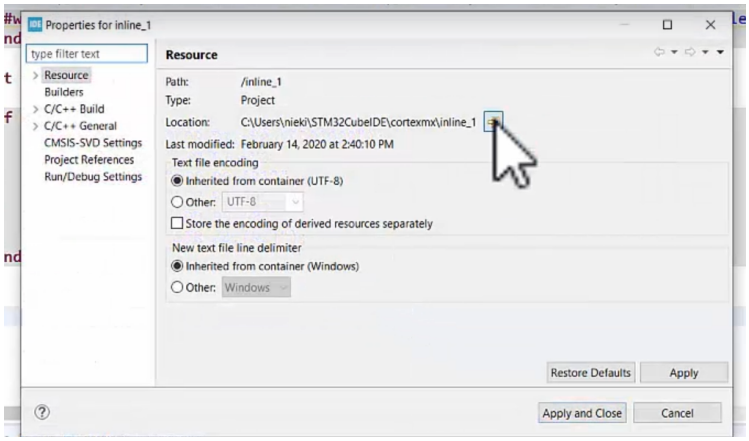


Figure 4.29: Go to file location

It will take us to the project

4. We go inside the project and click on the `Debug` folder
5. We open the file with `.list` extension.

- Variable created in the `C` file are in the stack (so in the RAM)
- Inline assembly language: used whenever we need to control some registers presented in the core , and can't be accessed by standard `C` code.

  For details see section 4.6.
- The reset handler function:

  1. doing some initialization
  2. Call the `main` function
- Design Concept: we have 2 bus (AHB and APB) because by this way, MCU vendors can put the peripheral which don't require high speed on low performance bus (the APB in our case) and reduce power consumption.

- Exceptions (section 4.11):

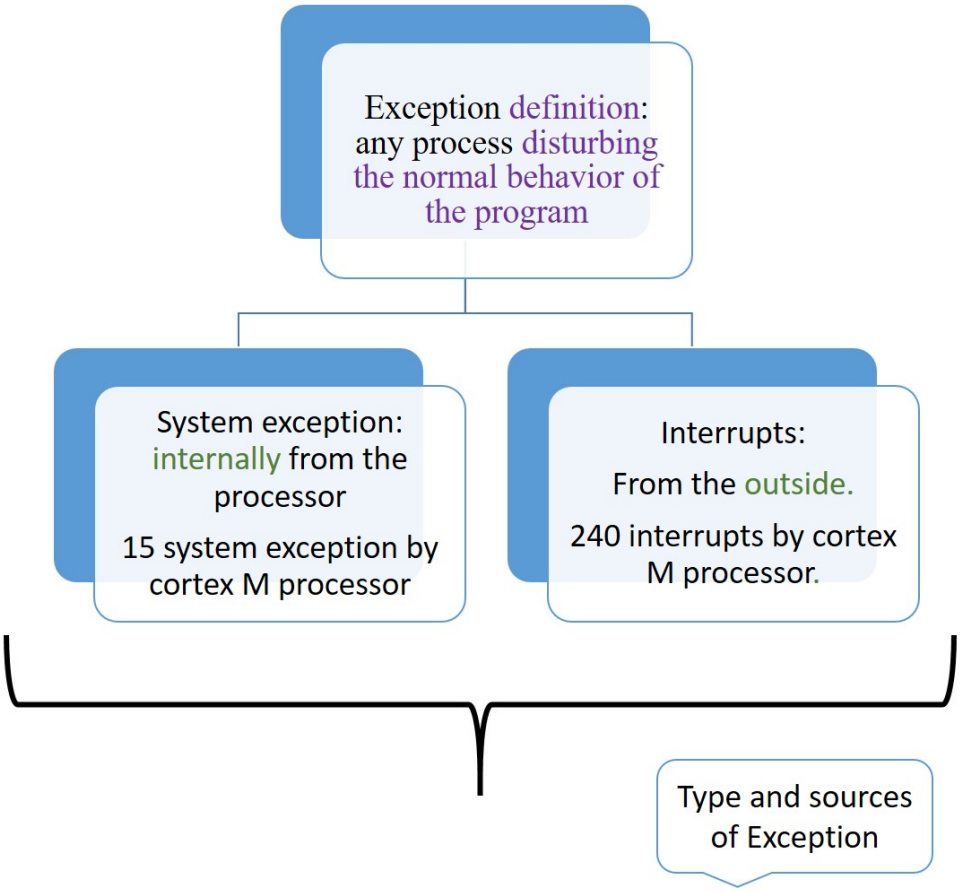


Figure 4.30: Exceptions: definition and type

Keep in mind that exceptions:

  - break the normal flow of the code
  - their are 2 types:
    1. ***internal*** ↔ from inside the microcontroller ↔ called system exceptions
    2. ***external*** ↔ from outside the micrcontroller ↔ ***interupts***

- Doc for exceptions:
  - Generic user guide:
    * section 2.3: Exceptions Model page 2-21 (34)
    * Arm cortex peripheral chapter:
    * the peripherals side of the arm cortex: system control block page 4-11 (227) responsible for system exceptions
    * NVIC page 219 section 4.2, responsible for the interuupt side
  - Reference Manual:
    * Vector Table table 62 page 357

- Sources of interrupt: see 4.14.2 and 4.14.3

4.17 TODO Arm Cortex

- An idea: I might split up this chapter based on Zhu book
  - But 1st I need to review later sections talking about the arm cortex in general in my report
  - Combine this with reading in Zhu book
  - Chapter 3 is a good starting point
- Search about different type of Arm cortex family
  - difference between M0,M4,M7,...
- To redo later all the sections related to processor registers
  - It was too much theory
  - I will search for some applications, or a goal later about these ideas
- To see later what is vector table, and its relation with exception addresses in the memory map of the microcontroller
  - I don't recall or note in my report a definition of this kind of tables

Chapter 5

GPIO Programming

5.1 Introduction

The structure of chapter 5 is illustrated in Figure 5.1.

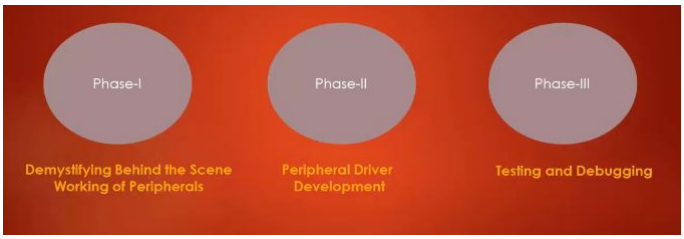


Figure 5.1: Chapter Structure

- Part 1 focuses on understanding the internal workings of key peripherals, such as GPIO and communication protocols like SPI.
- Part 2 covers the development of driver header files.
- Part 3 presents sample applications for each driver, including testing and debugging with a logic analyzer.

Bonus: Guidance on installing and working with development tools, such as KEIL Microvision.

5.2 Debugging Techniques

Before starting peripheral development, we will review several debugging techniques, as illustrated in Figure 5.2.

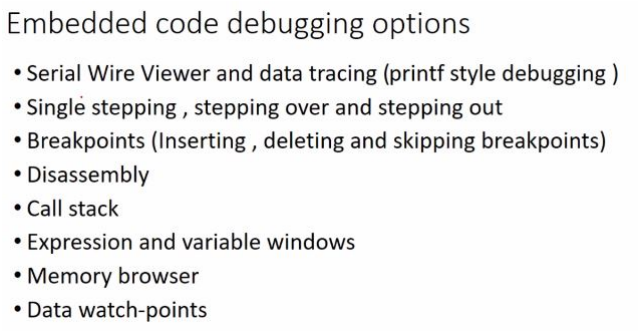


Figure 5.2: Different Debugging Techniques

Each technique will be discussed in the following sections.

5.2.1 Debugging General Steps

General Rule: To use any debugging technique shown in Figure 5.2, we first need to ***switch to debugging mode***. This is done by:

- Building the project. Right-click, then select `Debug as` → `Debug as stm32 Cortex M applicati`
- The window will switch to what is known as the ***debug perspective***, as shown in Figure 5.3.

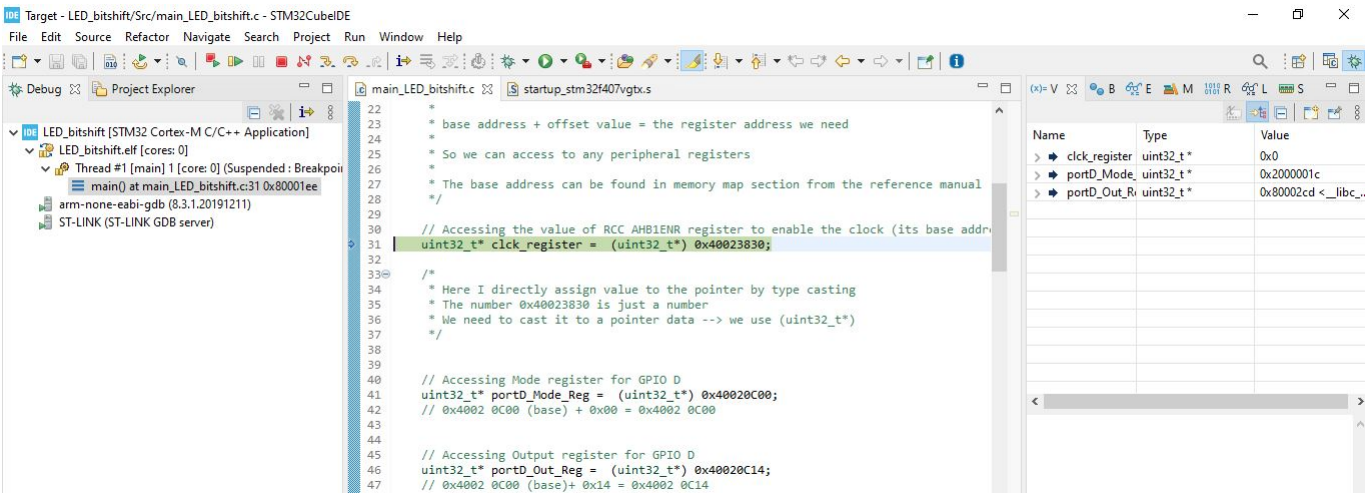


Figure 5.3: Editor Switched to Debug Mode

  - Notice the highlighting at line 31: this indicates that the code execution is currently at this line.

- In Figure 5.3, we can switch between debugger mode and editor mode. To return to editor mode, click on `C/C++` as shown in Figure 5.4.

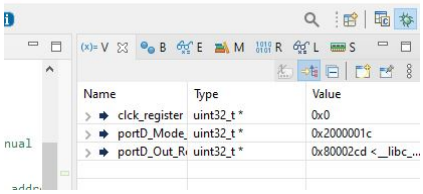


Figure 5.4: Switching Back to Editor Mode

- Assembly Code: To view the equivalent assembly code of the `C` code (for example, line 31 in Figure 5.3), go to the `Window` tab, then `Show View` → `Disassembly`.

  A new mini window will appear (shown in Figure 5.5) on the right-hand side, displaying the Arm assembly instructions for the `C` code.

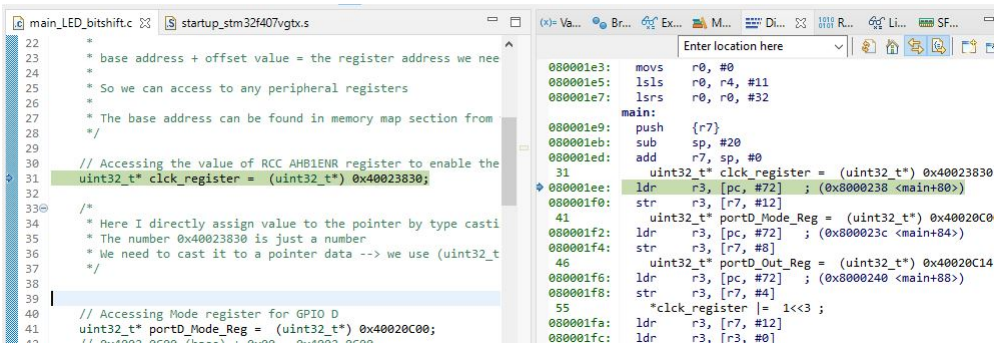


Figure 5.5: Debug Mode: Displaying Assembly Instructions

  – Notice that in the disassembly window, we first see the `C` code, followed by the equivalent Arm assembly code.

- Alternating between Debug and Editing Mode: On the right-hand side of Figure 5.3, we have what is called the ***debug view***, where all the functions called by our code are listed.

5.2.2 Stepping Options

Stepping refers to line-by-line execution of the `C` code (and not the assembly code, since one `C` code line is generally equivalent to many assembly instructions). We have three options for stepping, as shown in Figure 5.6.

Figure 5.6: Debug Mode: Stepping Options

- Step Into: Enter the statement (e.g., go inside the function).
- Step Over: Execute one `C` statement (e.g., execute a function without stepping into it).
- Step Return: Return from the function.

5.2.3 Assembly Code Debugging

To execute assembly-level code instead of `C` code, follow these steps:

1. Display the assembly window as described earlier:
   - Go to the `Window` tab, then `Show View` → `Disassembly`.
2. Activate instruction-level code by clicking the 'i' icon, as shown in Figure 5.7.

Figure 5.7: Debug Assembly Mode: Activating Instruction Level Code

3. Then use the same stepping options explained in 5.2.2.

Analyzing Assembly Window: Let's take, for example, the assembly code shown in Figure 5.5.

- On the right side, we see the *addresses*: this indicates that this instruction is stored at this memory address. The memory type here is flash memory (used to download our code).
- These instructions are not stored in a textual way; they are stored as *opcode*. To see the opcode, right-click on the assembly instruction, then click on `Show Opcode`. For example, the opcode for Figure 5.5 is shown in Figure 5.8.

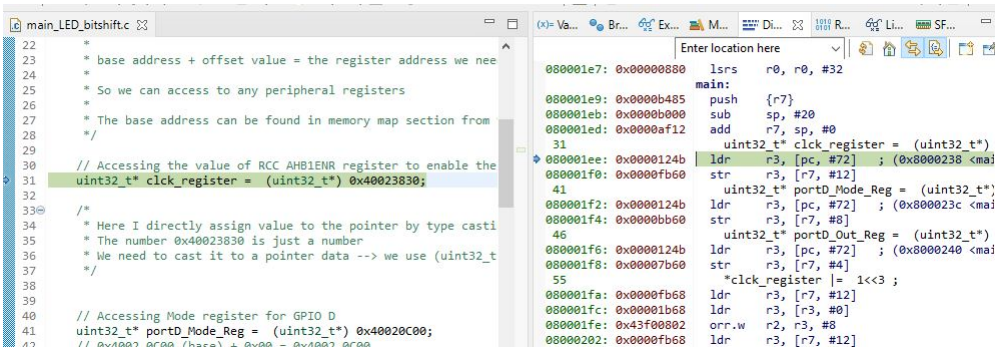


Figure 5.8: Debug Assembly Mode: Showing the Opcode of Figure 5.5

- If we want to see the content of the registers, we can go to the `Window` tab, → `Show View` → `Registers`.
  - We can also change the number format to hexadecimal, since the default value is in decimal.

5.2.4 Breakpoints

Breakpoints are used to make the processor stop at a certain instruction. In embedded programming, these breakpoints are ***hardware breakpoints***, as the processor uses hardware units (comparators inside the processor) to compare the instruction address to the breakpoint address.
Since these breakpoints depend on hardware, there are limitations on setting a maximum number of breakpoints.

To set a breakpoint:

- We have to be in debug mode.
- Hover the mouse over the intended line.
- Double-click the blue strip as shown in Figure 5.9.

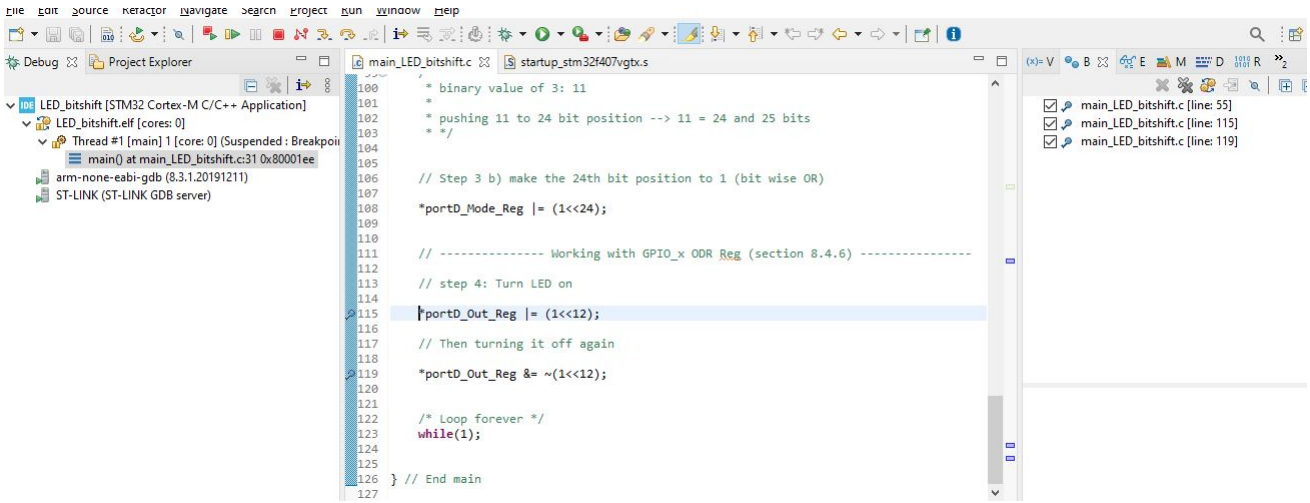


Figure 5.9: Setting a Breakpoint by Double-Clicking the Blue Bar Strip

  - In the debug view, the right side will show a ***breakpoints window***, where all the breakpoints are listed with their lines. We can deselect or remove some breakpoints from here.

- To go directly to the breakpoint, click on resume, as shown in Figure 5.10.

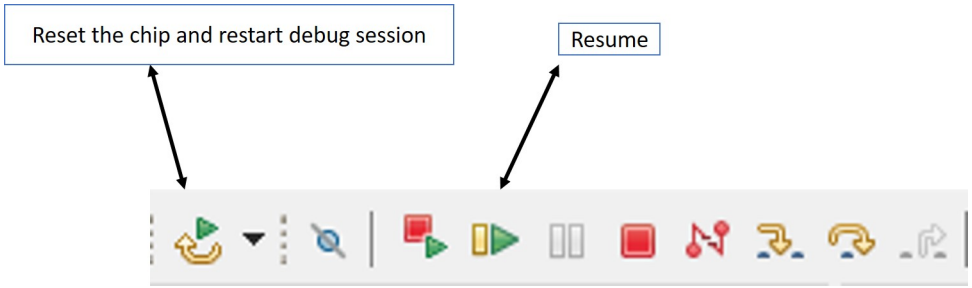


Figure 5.10: Going to a Breakpoint

  - To restart debugging or reset the chip, click on the reset button shown in Figure 5.10.

5.2.5 Expression Window

The expression window is used to make real-time changes (like incrementing a variable or pointer) during a debug session.
To use the expression window:

- Start a debug session.
- Open the expression window by going to the `Window` tab, → `Show View` → `Expression`. The expression window will open on the right-hand side.
- Drag and drop the variable from the editor: select the variable name, hold, and drag it. After this step, the variable name will appear in the expression window as shown in Figure 5.11.

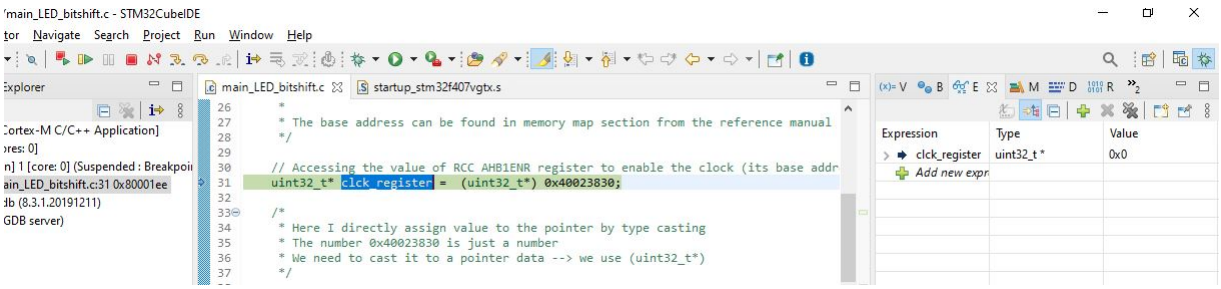


Figure 5.11: Expression Window: Dragging and Dropping a Variable

5.2.6 Memory Window

The memory window is used to track different memory locations in the microcontroller, such as RAM, ROM, FLASH, etc.

Data variables are stored in the RAM of the microcontroller.

Memory window

Memory window: *Nothing much important for now concerning addresses, maybe for later when I advance for the course material.*

5.2.7 Call Stack

The call stack can be used to check for mistakes in our code when we don't know their source. If there is a mistake, the call stack will raise an exception, helping us identify the source of the mistake.

An example of the call stack in STM32Cube IDE is shown in Figure 5.12.

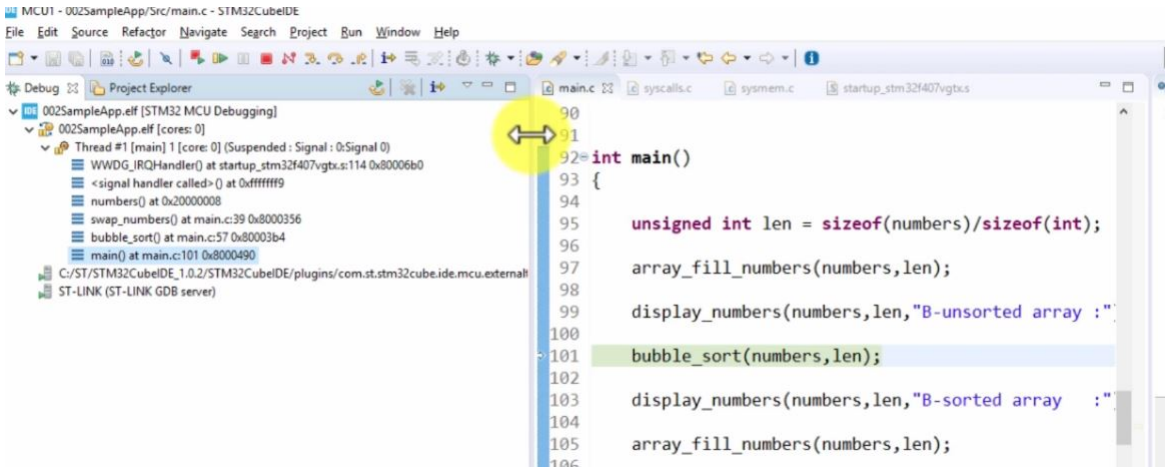


Figure 5.12: Call Stack Example

First, we see the `main` function, then the `bubble sort` function, and so on.

l Stack

Call Stack:

- *To redo it later*
- *To insert some mistake, and see what will happen*

5.2.8 Data Watchpoint

Data watchpoints are similar to breakpoints. They detect whether the value of a variable has changed, using a debug event for detection.

To define a watchpoint:

- Enter debug mode.
- Open the breakpoints window.
- Click on `Add Watchpoints` as shown in Figure 5.13.

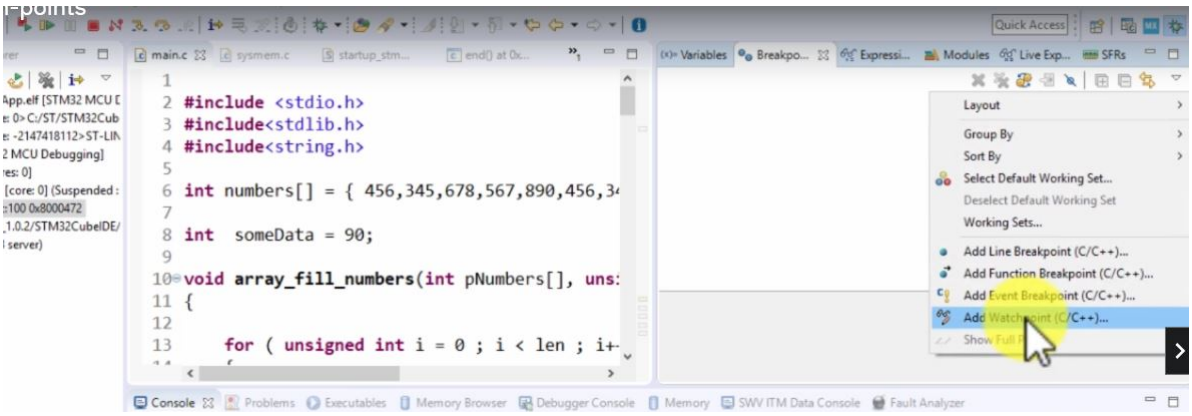


Figure 5.13: Adding Watchpoints

- Add an expression (like the variable name we are trying to monitor), as shown in Figure 5.14. In this example, the variable is `someData`.

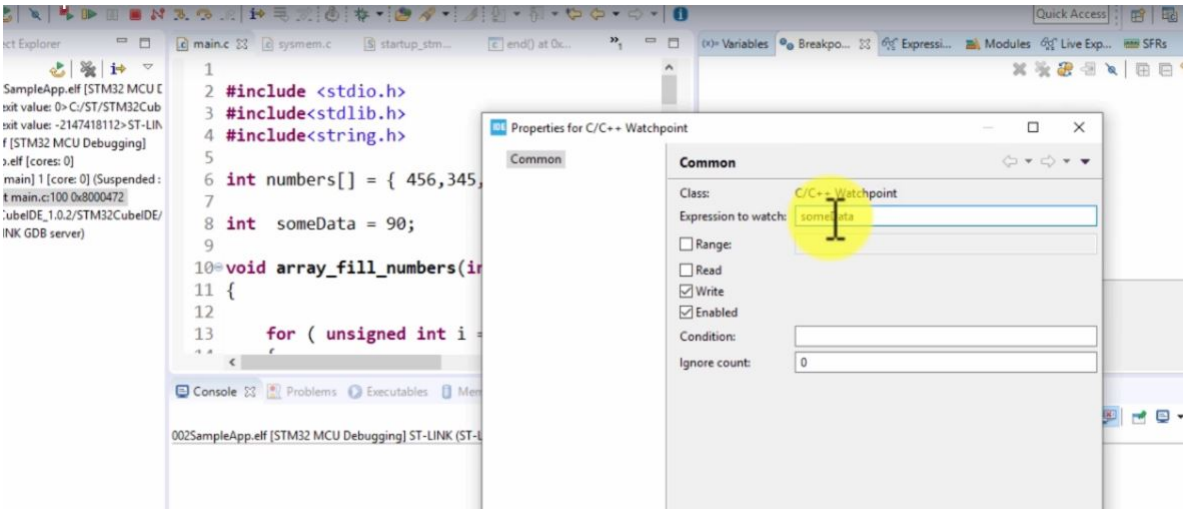


Figure 5.14: Adding Expression Name to Watchpoints

Data Watch points

Data Watchpoints: To repeat this part and try to see the difference between watchpoint in Read and Write mode.

5.2.9 SFR

SFR stands for special function registers. These registers contain the configurations for the different peripherals of the microcontroller (such as timer peripherals, power, communication, etc.).

To view the different SFRs:

- Enter debug mode.
- Go to the `Window` tab, → `Show View` → `SFR`.

A screenshot of the SFR is shown in Figure 5.15.

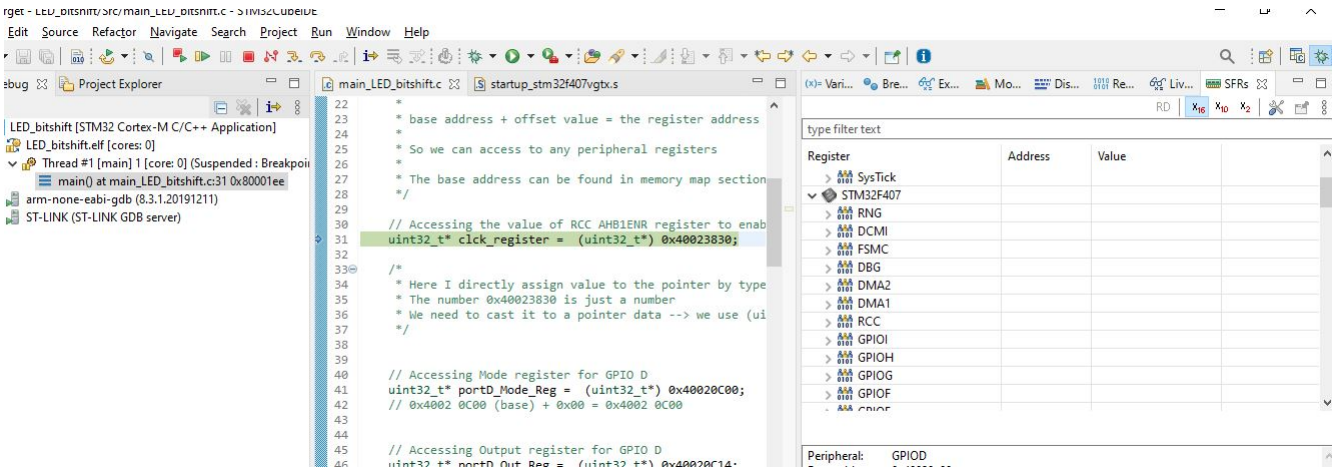


Figure 5.15: Different SFRs of the Microcontroller

- Different SFRs for the STM32F407.

Notice that SFRs are the registers of the microcontroller. There are also some registers relative to the processor (to the ARM Cortex). These can be shown using the `Window` tab, → `Show View` → `Registers` (and they are also shown at the top of the STM32F407 microcontroller).

Note: We can also directly modify the registers using the SFR window shown in Figure 5.15, for example to turn on some LED as we did in section 3.17, but here by directly modifying the bits without writing code.

5.2.10 Extra Features

- List all functions implemented in the `main` (to avoid scrolling too much): `Ctrl + o`.
- In debug mode, we can run the code on the MCU using the run button as shown in Figure 5.10.
  - Once the code is running, the resume button will not be highlighted anymore.
  - If we want to stop the execution, we click on the suspend button (next to the resume button).

- To enable autocomplete and code proposals, press left `Ctrl`+ space.
- To edit some code in a debug session without closing the debug and rebuild (to automate tasks):

  1. Switch from debug view to editing mode.
     Add the necessary code.
  2. Click on terminate and relaunch (icon shown in the IDE).
     This will rebuild, download the code, and switch the debug perspective.

5.3 MCU Memory Map

This section describes how to fetch the corresponding addresses of each peripheral (GPIOx, timer, etc.).

Note: Most information in this section is described back in 3.17.1 and 3.17.2. Review them if you need more information.

5.4 MCU Bus Interface

When fetching the address of some peripheral, we need to know to which bus it is connected. Also, buses play an important role in communication between several peripherals of the MCU.
In this section, we present several buses used in the MCU.

Note: The bus information is relevant to the processor vendor (ARM in our case), and we need to refer to the ARM documentation (such as the ARM technical reference manual).

In Figure 5.16, we have the block diagram where we have 3 buses between the Cortex M4 processor and the FLASH.
Note: Figure 5.16 is from the data sheet, section 2.2.

Figure 5. STM32F40xxx block diagram

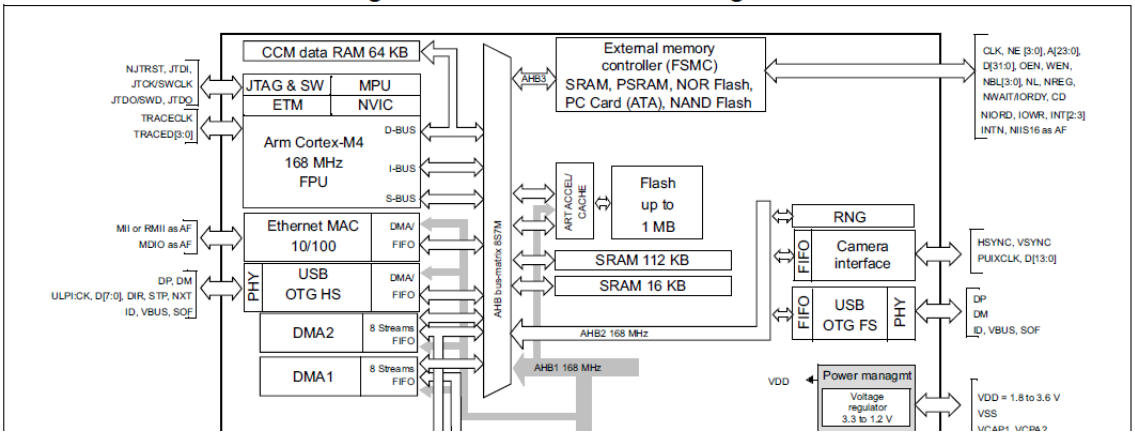


Figure 5.16: Bus between Cortex M Processor and FLASH

- I bus: for instruction.
- D bus: for data.
- S bus: called system bus.

Suppose we have some code as shown in Figure 5.17.

```
*main.c
1
2 #include<string.h>
3 #include<stdint.h>
4
5 //This is string literal(constant data) . This will be stored in ROM (Flash memory )
6 const char *pMessage = "Hi there, Good Morning !";
7
8 //This is constant data. This data also will be stored in ROM (Flash memory )
9 const int value = 200;
10
11 //This is not a constant data . This will be stored in SRAM
12 char myData[50];
13
14 int main(void)
15 {
16     for(uint32_t i = 0 ;  i < strlen(pMessage); i++){
17
18         /*This is data copy from FLASH to SRAM */
19         myData[i] = *(pMessage+i);
20     }
21     for(;;);
22 }
```

Figure 5.17: Some Code Example

- Constant data is stored in FLASH.
- The variable data is stored in SRAM.

When we compile and download the code, the processor reads the instruction from the FLASH using the I bus, and the constant data using the D bus from FLASH.

Note: We know that the I and D buses are connected to the FLASH from the reference manual, as shown in Figure 5.18.

Figure 3. Flash memory interface connection inside system architecture (STM32F405xx/07xx and STM32F415xx/17xx)

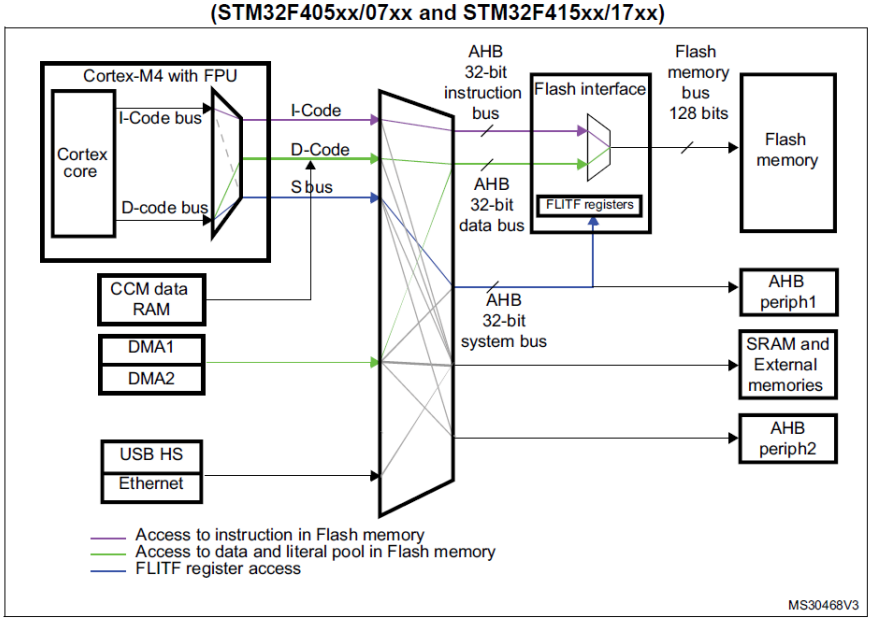


Figure 5.18: I and D Bus Connected to FLASH

To know more about these buses, we use the *ARM Cortex M4 Technical Reference Manual r0p1*, and refer to section 2.3: Interfaces.

From this manual, we can see that each bus has a specific operating address range. For example, the ARM Cortex uses the I bus to fetch instructions that are within the range 0x00000000 to 0x1FFFFFFC. The same concept applies to the D bus and system bus.

5.4.1 Bus Matrix

One last thing to explain is the bus matrix. In Figure 5.19, we have a diagram illustrating the concept of the bus matrix.

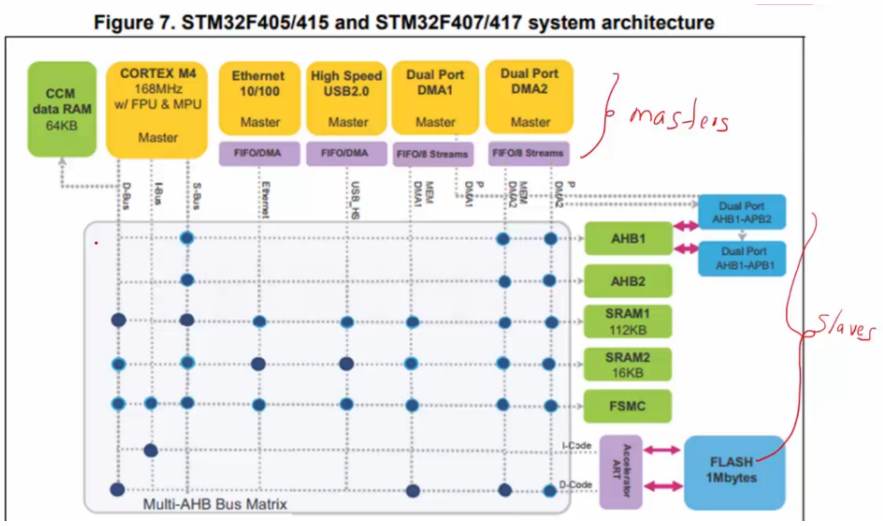


Figure 5.19: Illustrating Bus Matrix

- In the MCU, the communication between the processor and other peripherals is based on a Master-Slave model.
- The dots indicate possible communication paths.
- For example, the Cortex M4 can communicate with four peripherals using the D bus (as we have four dots): SRAM 1 and 2, FSMC, and the FLASH.
- Notice that we can't communicate between the S bus of the Cortex and FLASH since we don't have a dot.

Also, concerning the bus matrix, by observing Figure 5.17, we can see that the bus matrix is extended to the `AHB1` bus, then continues until we have a conversion from `AHB` $\rightarrow$ `APB`, but with lower speed.

5.5 Clocks in MCU

Now we begin with clocks. MCUs are essentially ***digital synchronous circuits***. By synchronous, we mean they operate in sync with a clock: we can't do anything if we don't have some clock source.

Usually, we have three types of clock sources in any MCU:

1. Crystal oscillator: an external circuit that provides the clock.
    - Also called HSE (high-speed external).
2. RC oscillator: internal circuitry inside the MCU.
    - Called HSI: stands for high-speed internal.
3. PLL: provides a high-frequency clock from lower frequencies (also an HSI).

Design Criteria: Choosing the clock frequency is important for low-power applications, as there is usually a relation between frequency and power consumption.
Frequency and Power Relation: *To be explored later.*

frequency a power relati

For the MCU discovery board, we have an 8 MHz clock, coming from an installed crystal oscillator (which can be viewed in the schematic, called `X2`). In the Nucleo board, we don't have an installed crystal oscillator like the discovery family board, but we can *extend one* from the ST-Link circuitry.
Notes:

- See the hand notes I wrote in the reference manual, section 6.2, Figure 16: clock tree.
- The PLL, in general, boosts the clock speed beyond the limits of HSE and HSI.
- Some peripherals, such as Ethernet, won't work properly if we underclock them, so we need to use a PLL.

To configure the clock, we need to use the `RCC` register (description of this full register is in section 6.3 of the reference manual). Using `RCC`, we can configure various domains such as `AHB`, `APB`, memory domain, etc.

Some Rules Regarding Clocks:

- In programming, and before using any peripheral, we need to configure the clock associated with the peripheral.
- By default, the clocks are in sleep mode to save power.
- To know the corresponding clock for a peripheral, we need to know the bus associated with this peripheral.
    - For that, we can use the memory map table in the reference manual (section 2.3, Table 1), or use the functional overview (the block diagram) from the data sheet (section 2.2, figure 5).

ADC Config ration Ex

ADC Configuration Ex: *To see the code later, take a screenshot. Nice way of using MACROS.*

5.5.1 HSI Measurement Exercise

In this part, we will try to code an exercise to measure an HSI clock.

Given:

- Write a program to output the HSI clock on an MCU pin and measure it using an oscilloscope or logic analyzer.

Steps to output a clock to an MCU pin:

1. Select the desired clock for the MCOx signal (stands for MC Clock Output).
   - We need to use the `RCC-CFGR` register (section 6.3.3 from the reference manual).
2. Output the signal to the MCU pin.
   - For the pin, we need to refer to the data sheet.
   - Go to chapter 3, table 1 for alternate function.
   - By alternate function, we mean that a port can be used for several different functions.
   - In our case, `MCO1` will be used by GPIO `PA8` such that `PA8` is in `AF0` mode.

Note: If we want to know `PA8` as a pin number, we can look at the board schematic: `PA8` $\leftrightarrow$ pin 67.

Logic Analyzer and Board Connection: See video 39.

In Figure 5.20, we have a block diagram to make the connection.

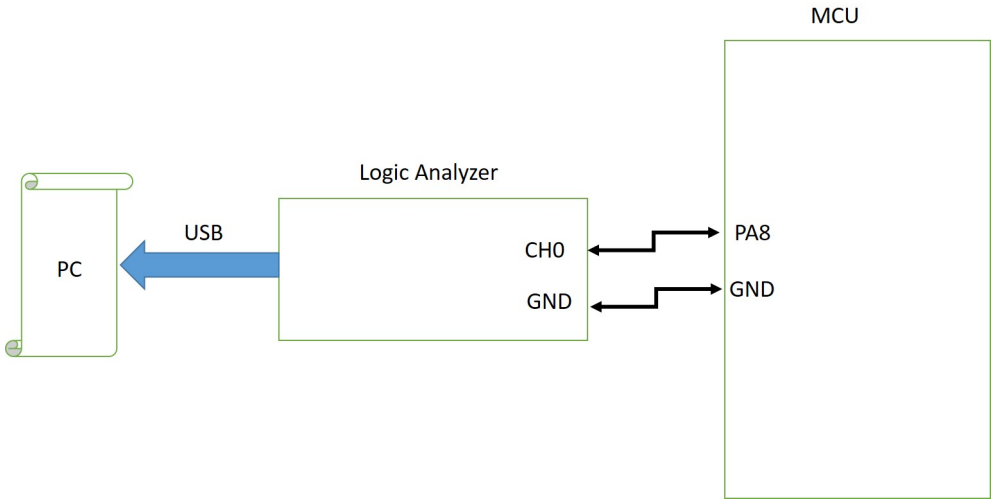


Figure 5.20: Connection between Logic Analyzer and MCU

de Clock
asurement

Code Clock Measurement: To be done later, when buying the logic analyzer (See video 40 and 41).

5.6 Vector Table

Resources: Refer to chapter 12 in [1] , and table 61 in the reference manual.

Naming and Definition:

The vector table is a table that contains the addresses of interrupts and system exceptions (review chapter 4 to get more details about interrupts and system exceptions).

The name vector usually stands for direction in mathematical terms. Here, in embedded programming, the direction is equivalent to pointers, which hold addresses; hence the name vector table since it is a table of addresses.

We try to explain all its columns:

- Position: it is also the IRQ number with respect to NVIC position
  - Note that for system exceptions, we don't have numbers since the MCU vendor don't have a choice in designing them
  - It is the processor manufacturer (ARM in this case) which has the choice, since the system exceptions are internal (from inside the processor)
- Priority: priority which comes first from the handler. By default, the less the number, the more prior it is.
- Type of Priority:
  - Fixed: we can't change it using code.
  - Settable: we can change it using code. Here, we can change the priority: make some handler which is by default less prior, more prior.
- Addresses: place to store the function which implements the interrupt. In other words, if we implement a function NMI, this function accepts the address `0x00000008`

  Example:
  - 0x 0000 0008 holds the function implementing the NMI handler.

interrupt and function pointer

Interrupt and Function Pointer: To review the concept of function pointer later and document it all.

As another example: take for example `I2C1` event interrupt. This will be a normal function contained in the startup file (having a `.s` extension).

This function is stored in a certain address (say for example 0x8000264). By referencing the table, we see that `I2C1` has 0x0000 00BC. This means when the processor activates the `I2C1`, it goes to 0x0000 00BC which contains 0x8000264.

Vector Tabl

Vector Table: This is more explained in chapter 2: embedded programming using Cortex M. To redo it then get back to the video of this section (video 42).

5.6.1 Startup and Linker Script

- In the MCU IDE, the interrupt and system exceptions are handled in the startup file, inside the startup folder.
- The vector table, for example, is coded as a big array of constants named `isr_vector`.
- The `isr_vector` is stored in the linker script (with `.ld` extension, inside the ROM, see line 54 in `STM32F407VGTX_FLASH.ld` file).

er and
tup:

Linker and Startup: To be explored later in the ARM processor course.

5.7 Programming: User Button Interrupt

In this section, we apply the concept of interrupt by using the button attached to our MCU.

5.7.1 Button Location in MCU

1st, we open the user *user manual* in order to know how the button interacts with our MCU.
We go to figure 16 (peripherals) in the figure labeled user and wake button (it is also shown in **??**)

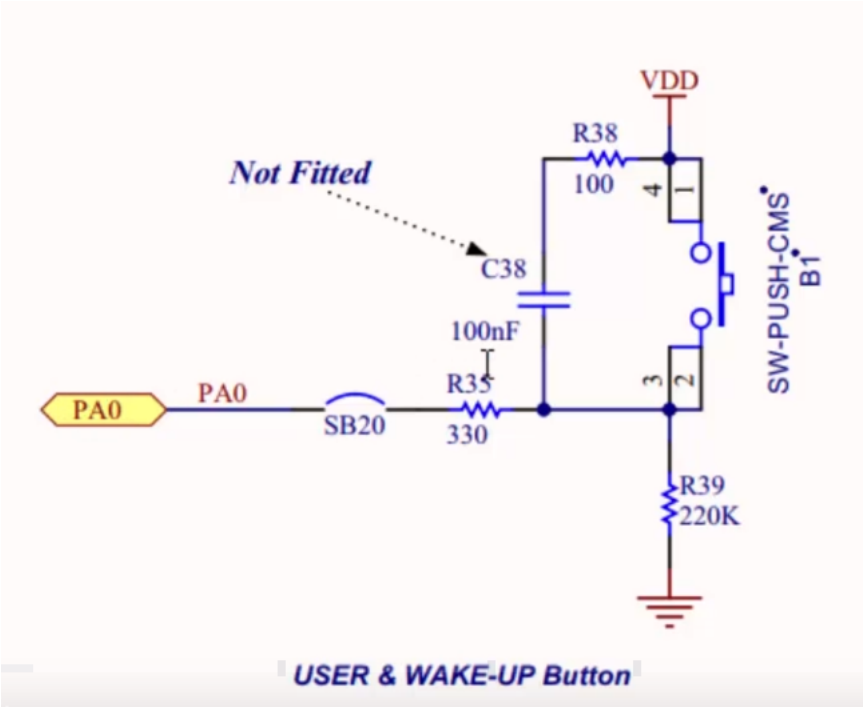


Figure 5.21: User button (push button) on the MCU

- The capacitor C38 marked *not fitted*: meaning if it is like it doesn't exist.
- When we push the button, we have a big resistance R39 compared to R35 → PA0 will be pulled to VDD ↔ a high state
- When we release the button, the easiest path is to ground → it will be in a low state.

Before implementing, we need to understand many concepts related to this task.

5.7.2 GPIO Interrupts

Now the next step is to see how GPIO interrupts are delivered to the processor. This task is ***vendor specific***.
For that, we go to the *reference manual*, section 12.2 (external interrupt/event controller) EXTI.
In Figure 5.22, we have the mapping for each GPIO pin and EXTI engine.

Figure 42. External interrupt/event GPIO mapping (STM32F405xx/07xx and STM32F415xx/17xx)

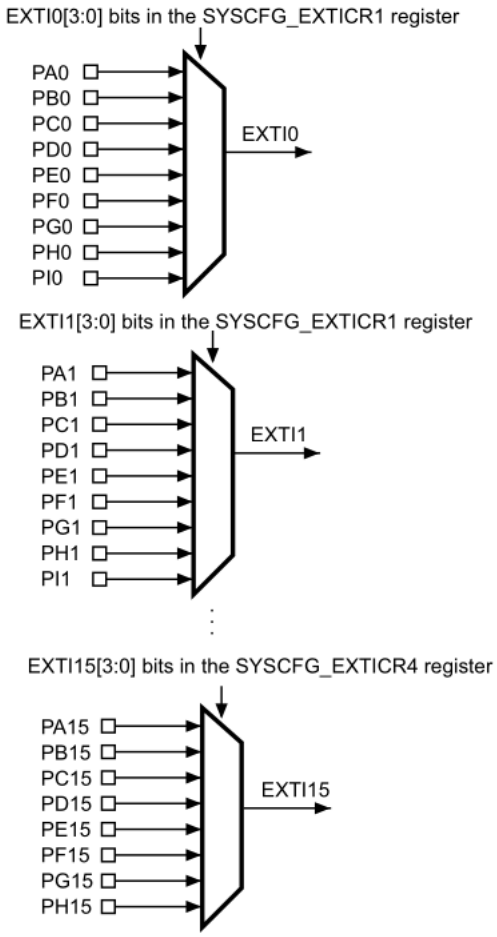


Figure 5.22: EXTI and GPIO Mapping

5.8 Summary: Driver Part 1

5.8.1 Board Info

- User Button (type push button): `PA0`
- Logic of the button (mainly on stm32f407V, for other board we need to see the circuitry)
  - Refer to Figure 5.21
  - When we press the button, `PA0` is pulled to high
  - When releasing, `PA0` is pulled to ground
- Design Criteria: Choosing the clock frequency is important if we want to work in low-power applications, since usually there is a relation between frequency and power consumption.
- Some Rules Regarding Clocks:
  - In programming, and before using any peripheral, we need to configure the clock associated with the peripheral.
  - By default, the clocks are in sleep mode to save power.
  - To know the corresponding clock for a peripheral, we need to know the bus associated with this peripheral.
    - For that, we can use the memory map table in the reference manual (section 2.3, Table 1), or use the functional overview (the block diagram) from the data sheet (section 2.2, figure 5).
- Interrupt resources: see

5.9 TODO

Some of the videos I didn't do it ,to do them later (in periph dirver 1 course)

- Section 27: 105,106 (mainly about external LED and push button that we need to connect)

Chapter 6

GPIO and Interrupts

6.1 Interrupts

GPIO and Interrutps

GPIO and Interrutps

- *In this section I will add the interrupt part concerning the GPIO from my overleaf writing*

6.2 Peripheral interrupt interaction: EXTI engine

Before proceeding to programming, we need to understand how GPIO or any peripheral delivers their interrupt to the processor.

Note: this concept of peripheral interrupt processor interaction is ***vendor specific***.

We know from chapter arm cortex, that when dealing with interrupts, we have peripheral side in the MCU, and the processor side. Since we are working with interrupts, we need to see how ***GPIO interrupts are passed to NVIC***.

To understand this concept, we open the *reference manual* at section 12.2: external interrupt/event controller (EXTI).

To handle peripheral interrupts, stm has what is called `EXTI`, that is external interrupt controller. It's block diagram is shown in Figure 6.1.

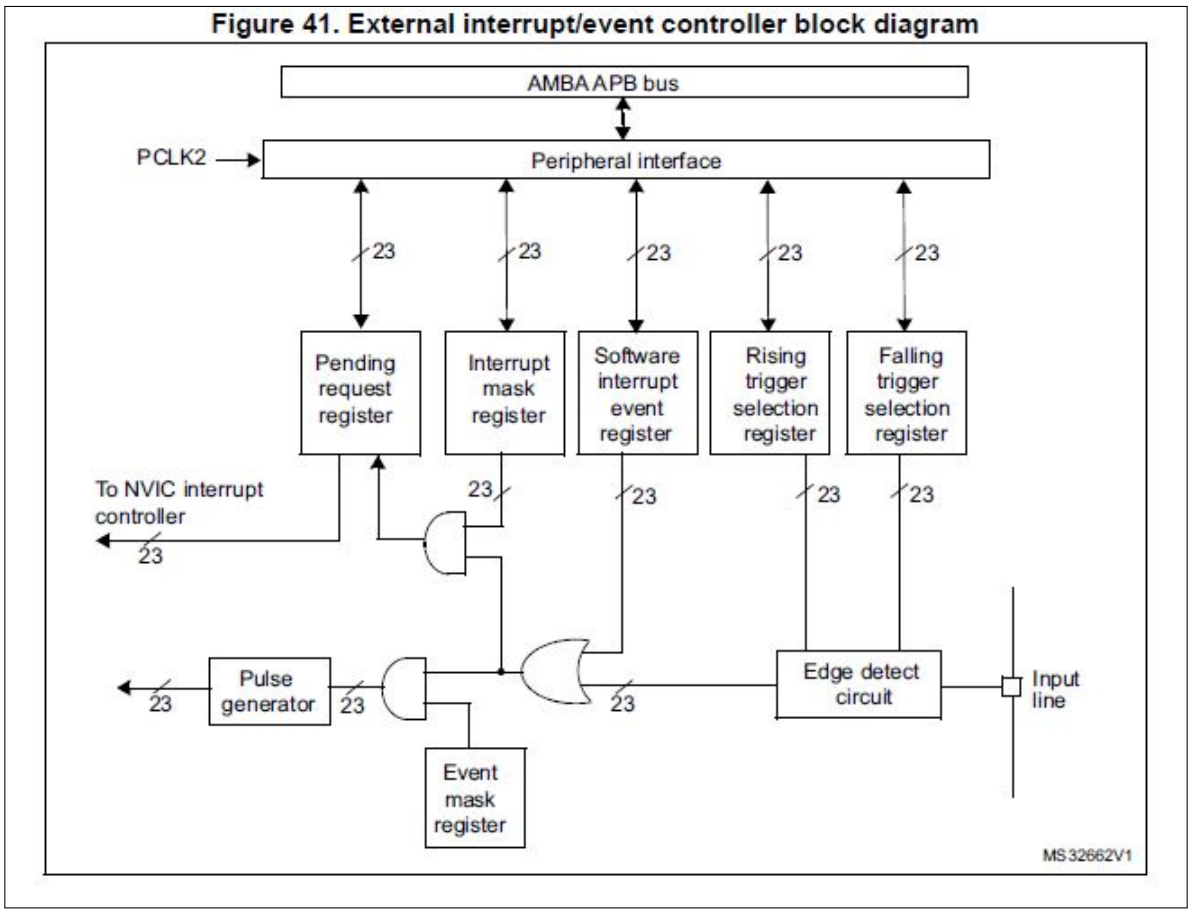


Figure 6.1: EXTI Peripheral from stm to handle some peripheral interrupts

We can see that this engine is connected to NVIC (which is a peripheral of the processor) using 23 connection. Which means, it captures 23 interrupt positions, or we can say it has 23 different IRQ numbers.
This means if we go to the vector table (table 61 in the reference manual), we should see 23 lines related to EXTI engine.
As example, we can see in IRQ 6 $\rightarrow$ 10, we have EXTI0 till EXTI4.

If we go also to IRQ 42, where we have the USB peripheral, we can see that the USB interrupt will go through EXTI, and not directly through NVIC.

6.3 GPIO Interrupts

As for GPIO, one thing we can notice that if we search the entire table 61 of interrupts, we don't found GPIO peripherals.

To understand the GPIO interrupts deliver, we go 12.2.5 in the reference manual, figure 42 (shown in Figure 6.2)

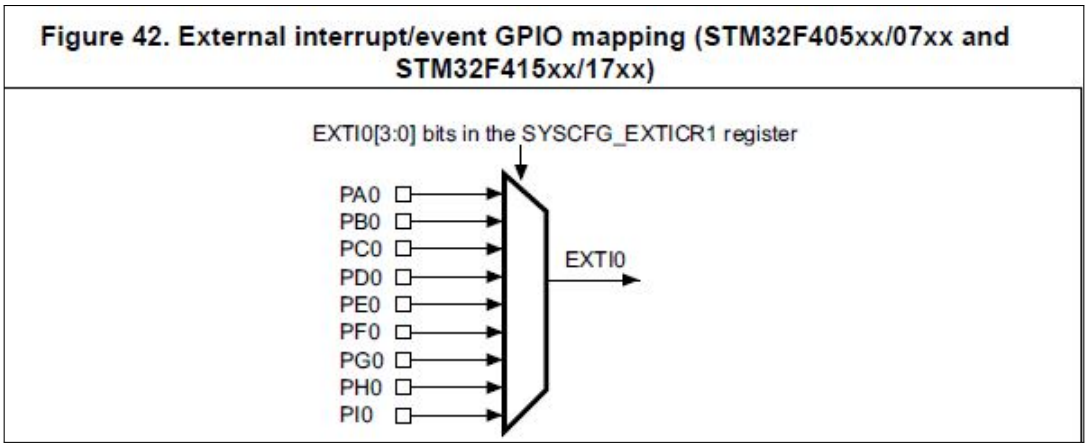


Figure 6.2: EXTI engine of stm to handle GPIO interrupts

- We can see that all 0th port (1st pin of each port) of the MCU deliver their interrupt through EXTI0 through a multiplexer, and by configuring the `SYSCFG_EXTICR1` register.
- We see later that `SYSCFG_EXTICR1` register select the link between a certain port and the EXTI0
  - for example, if we connect our button to PD0, we need to configure the link between PD0 line and EXTI0

Summary of Button Interrupt:

1. The button is connected to a GPIO pin of the MCU
2. Configure the GPIO pin to input mode
3. Link between GPIO pin and EXTI should established through `SYSCFG_EXTICRx` register
4. Configure the trigger detection (rising, falling or both) for relevant EXTI line
   - In Figure 6.1, we have several register such as Rising trigger selection, falling trigger selection
5. Implement the handler (`C` function) to serve the interrupt

6.3.1 EXTI Register

EXTI register can be found in section 12.3 in the reference manual.

- `EXTI_IMR` (Interrupt mask register):
  - Since EXTI desing support for 23 line, only 23 bit of the `EXTI_IMR` can be manipulated
  - By default, EXTI doesn't poke the NVIC, we need to enable it through `EXTI_IMR` register, by setting some bit to 1

All of the above can be summarized in Figure 6.3

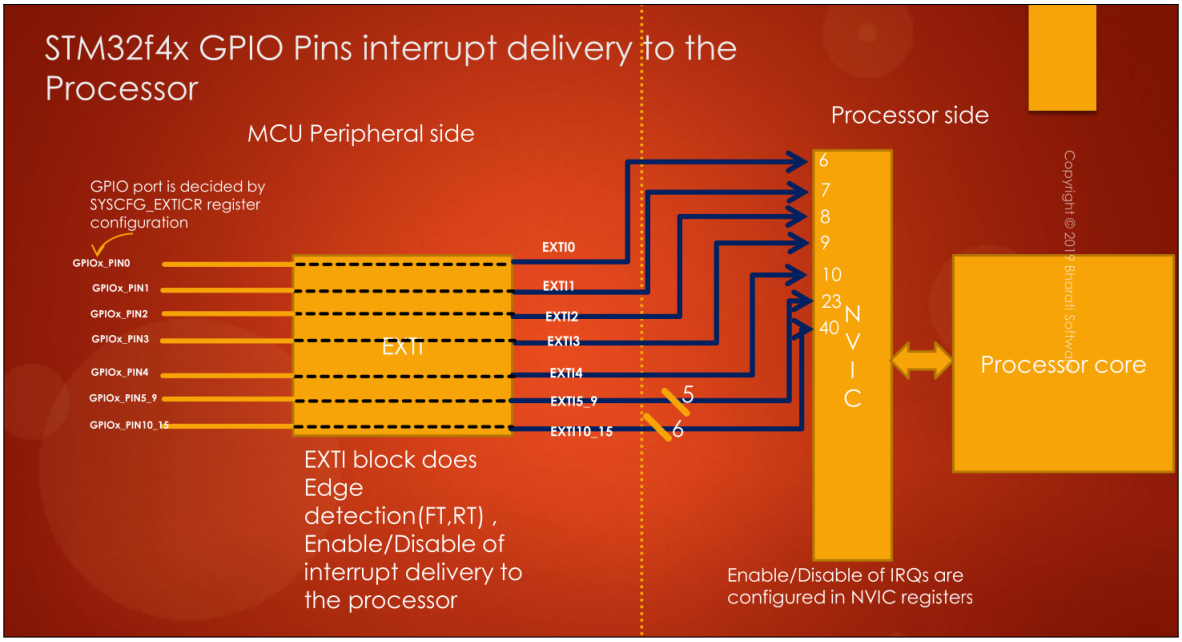


Figure 6.3: GPIO Interrupts flow

- Pin 0 from GPIOx is connected to `EXTI_0`
  - This can be viewed also in the reference manual in section 12.2.5, Figure 42
- The `EXTI_0` engine is connected to `NVIC` (which is again a *processor peripheral*) through IRQ number 6
  - The `EXTI_0` connection throuhgh IRQ 6 can be viewed in the vector table, table 62 in section 12.1 in the reference manual
- Using `NVIC` registers, we could enable or disable the interrupts, becaue by default the interrupts are disabled
- Remember also that the mapping between pin number from a `GPIOx` and `EXTI` is done via `SYSCFG` (system configuration controller). This is shown also in Figure 6.2
  - For `SYSCFG`: refer to chapter 9 from the reference manual

6.4 Steps for desiging interrupts

1. Pins must be in input mode configuration
    - That is because we are receiving interrupts, so we are in input mode
2. Configure the edge detection (rising,falling or both)
    - That is done through the `EXTI` engine
3. Enable interrupt delivery from peirpheral of the MCU → processor
    - Done using `EXTI`, so in the MCU side
4. Identiry the IRQ number on which the processor accept the interrupt from that pin
    - Done using vector table
5. Configure the IRQ priority for the indentified IRQ number
    - Also done using `NVIC`
6. Enable interrupt reception on that IRQ number
7. Implement the IRQ handler

Bibliography

[1] Y. Zhu, *Embedded systems with ARM Cortex-M microcontrollers in assembly language and C*, ser. Oxford Series in Electrical and computer engineering. E-Man Press LLC, 2009.