Mastering Microcontroller with Embedded Driver Development +HAL - Using STM32F407 – Discovery Board

L&T Technology Services

Dr. Vivek Kaundal
Mr. Bhargav N.

LTTS GLOBAL ENGINEERING ACADEMY

Compilation approach

Program execution cycle

Source File
- - - - - - - - - - - - - - - - -
File1 ... File n .c

Preprocessor

Preprocessed file
- - - - - - - - - - - - - - - - -
File1 ... File n.i

Compiler

Assembly Code
- - - - - - - - - - - - - - - - -
File1 ..File n.s (.asm *.lst etc)

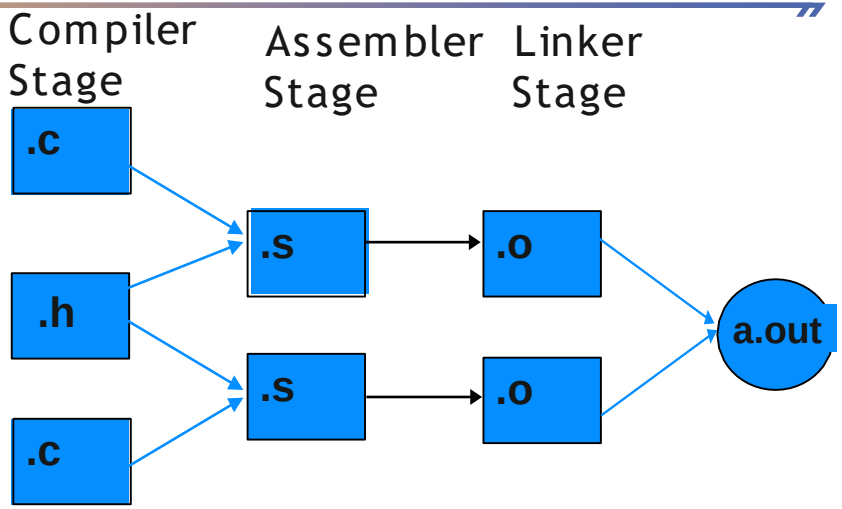
Assembler

Object Code
- - - - - - - - - - - - - - - - -
File1 ... File n.o

Linker

Linker file
(Target Code)
Ex: *.exe, *.hex,
*.srec, *.dll

Debug Info
*.elf, *.cof, *.sym etc

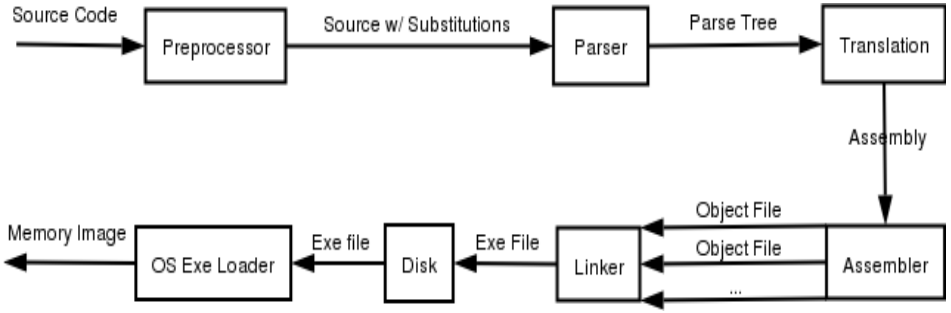


So the three different tasks required to produce a.out file are summarized below:

- gcc -c hello1.c — compiler stage
- gcc -c func1.c — compiler stage
- gcc hello1.o func1.o — linker stage

Compilation Stages

- Preprocessing is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.
- Compilation is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
- Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
- Linking is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file.



GCC compilation approach

source files	gcc command	Executable Name
function.c main.c	gcc –Wall function.c main.c	a.out
function.c main.c	gcc –Wall function.c main.c – O project1	project1

- When we compile the program the executable file is created in the current directory in which we are working.
- a.out is a file format used in older versions of Unix and Unix like operating systems for executable, object codes and shared libraries. In common a.out means assemblers output.
- The option -Wall turns on all the most commonly-used compiler warnings---it is recommended that we always use this option!
- -o option tells compiler to create executable file project1.o (it may be any other file name as given by user)

Compilation flags – commonly used

- Controlling compilation output
- For Debugging programs
- Optimization flags
- Linker options
- Target specific options

Commonly used GCC options

Options	Purpose
	Controlling Compilation options
-C	Compile or assemble the source files, but do not link
-S	Stop after the stage of compilation proper; do not assemble.
-E	Stop after the preprocessing stage; do not run the compiler proper.
-o file	Place output in the file file ;it can be a executable, an object file, an assembler output or pre-processed C code
	* Debuggingoptions
-g	Enable support for debugging
-ggdb	Produce debugging information used by GDB

*use level to specify how much debug information required (The default level is 2)

Level 0 produces no debug information at all. Thus, -g0 negates -g.
Level 1produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug.
Level 3 includes extra information, such as all the macro definitions present in the program.

GCC optimization options

Option	Optimization level	Execution time	Code size	Memory usage	Compile time
-O0	optimization for compilation time (default)	+	+	–	–
-O1	optimization for code size and execution time	–	–	+	+
-O2	optimization more for code size and execution time	– –		+	+ +
-O3	optimization more on execution time or speed	– – –		+	+ + +
-Os	optimization for code size		– –		+ +
-Ofast	O3 with fast none accurate math calculations	– – –		+	+ + +

Library search PATH

- Include PATH
  - By default, gcc searches the following directories for header files:
  - /usr/local/include/
  - /usr/include/
- Link PATH
  - and gcc searches the following directories for library files:
  - /usr/local/lib/
  - /usr/lib/

```
gcc -Wall square.c usr/lib/libm.a -o square
or
gcc -Wall square.c -lm -o square
```

Environment variables

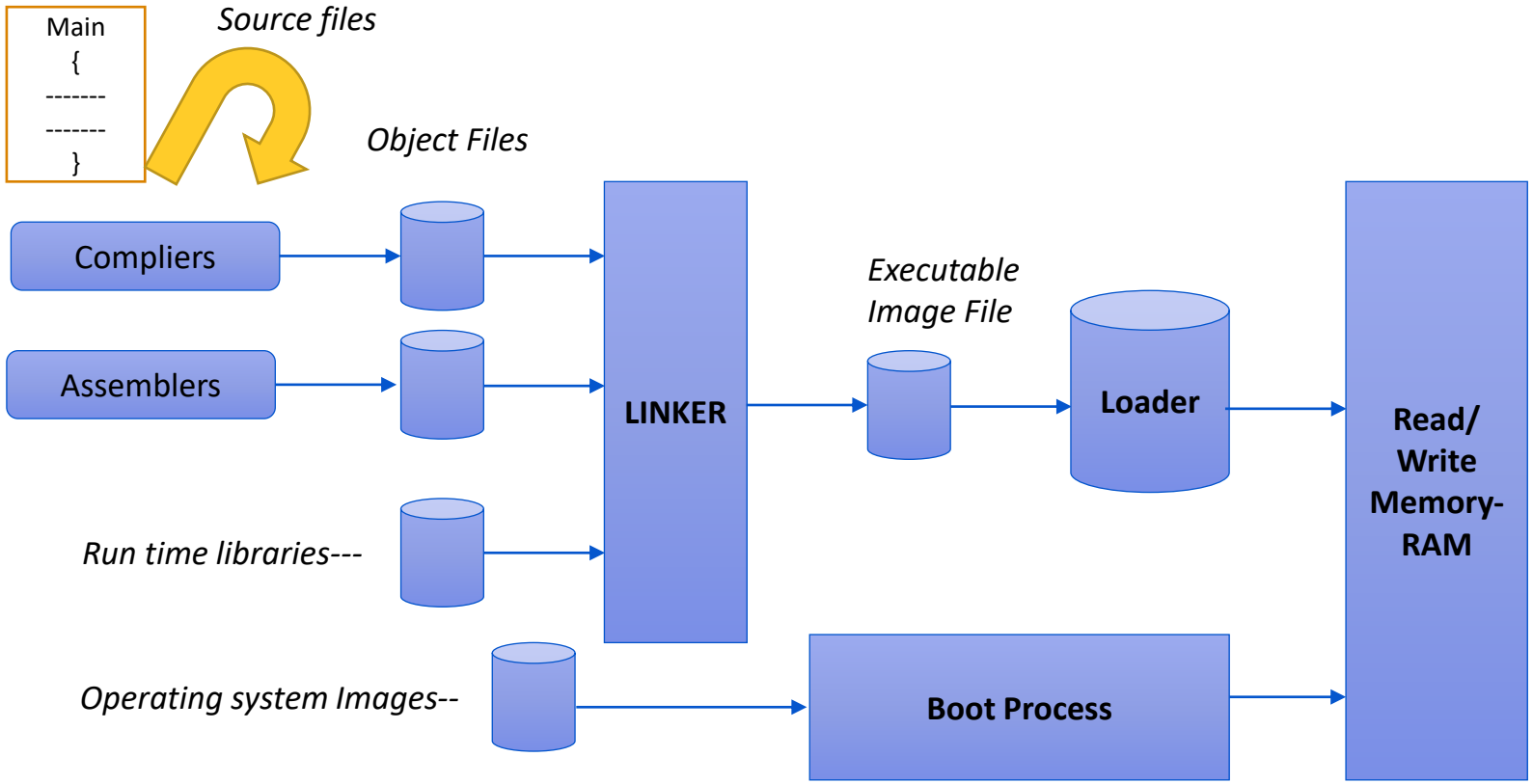
Environment variables	Purpose	Usage with GCC options	Usage by setting PATH environment variables
C_INCLUDE_PATH	C header file	-I	C_INCLUDE_PATH=/opt/mylibm-1.8.3/include export C_INCLUDE_PATH
LIBRARY_PATH	Library path	-L	LIBRARY_PATH=/opt/mylibm-1.8.3/lib export LIBRARY_PATH
LD_LIBRARY_PATH	Load path		LD_LIBRARY_PATH=/opt/mylibm-1.8.3/lib export LD_LIBRARY_PATH

By default gcc compiles programs against shared libraries. Whenever a static library 'libmylibm.a' would be used for linking with the option -lNAME the compiler first checks for an alternative shared library with the same name and a '.so' extension.

Static linking can be forced with the -static option to gcc to avoid the use of shared libraries

Build Process in Embedded System

Desktop based application development

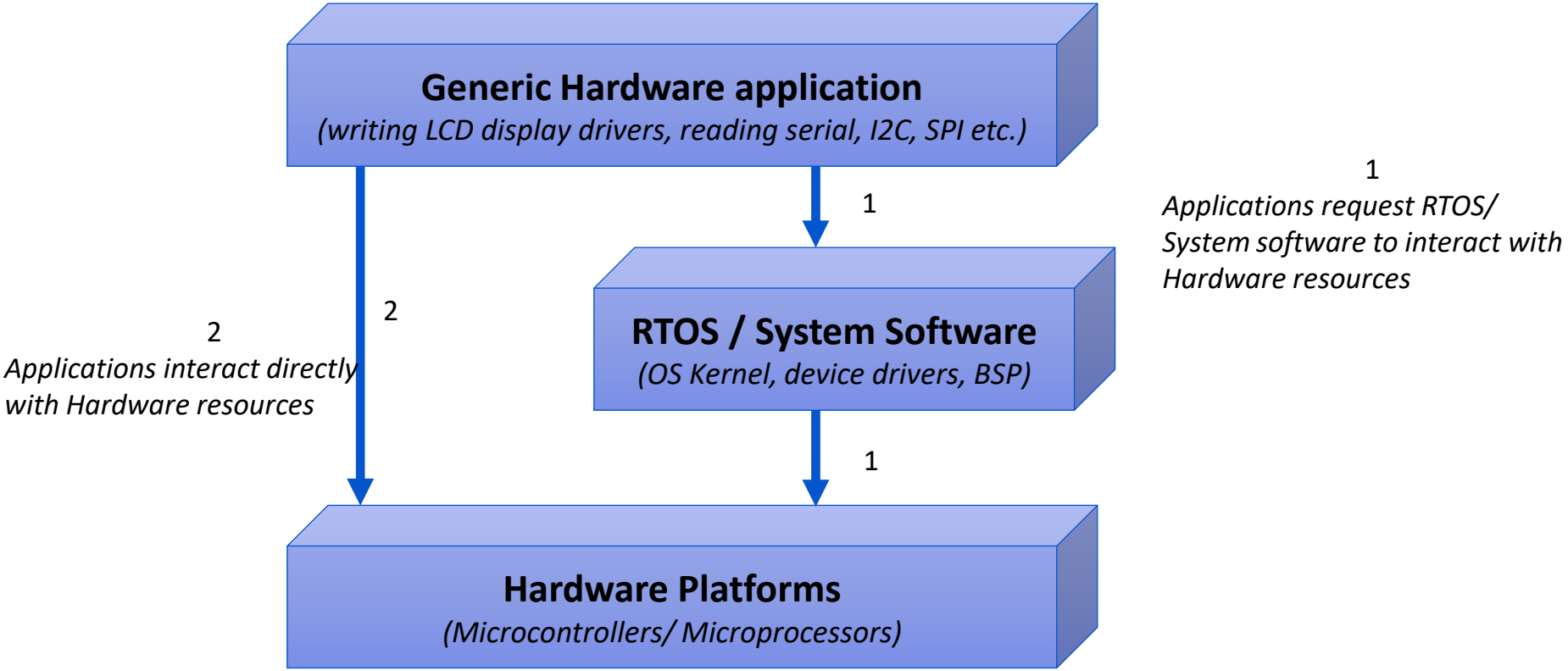


Desktop based application development

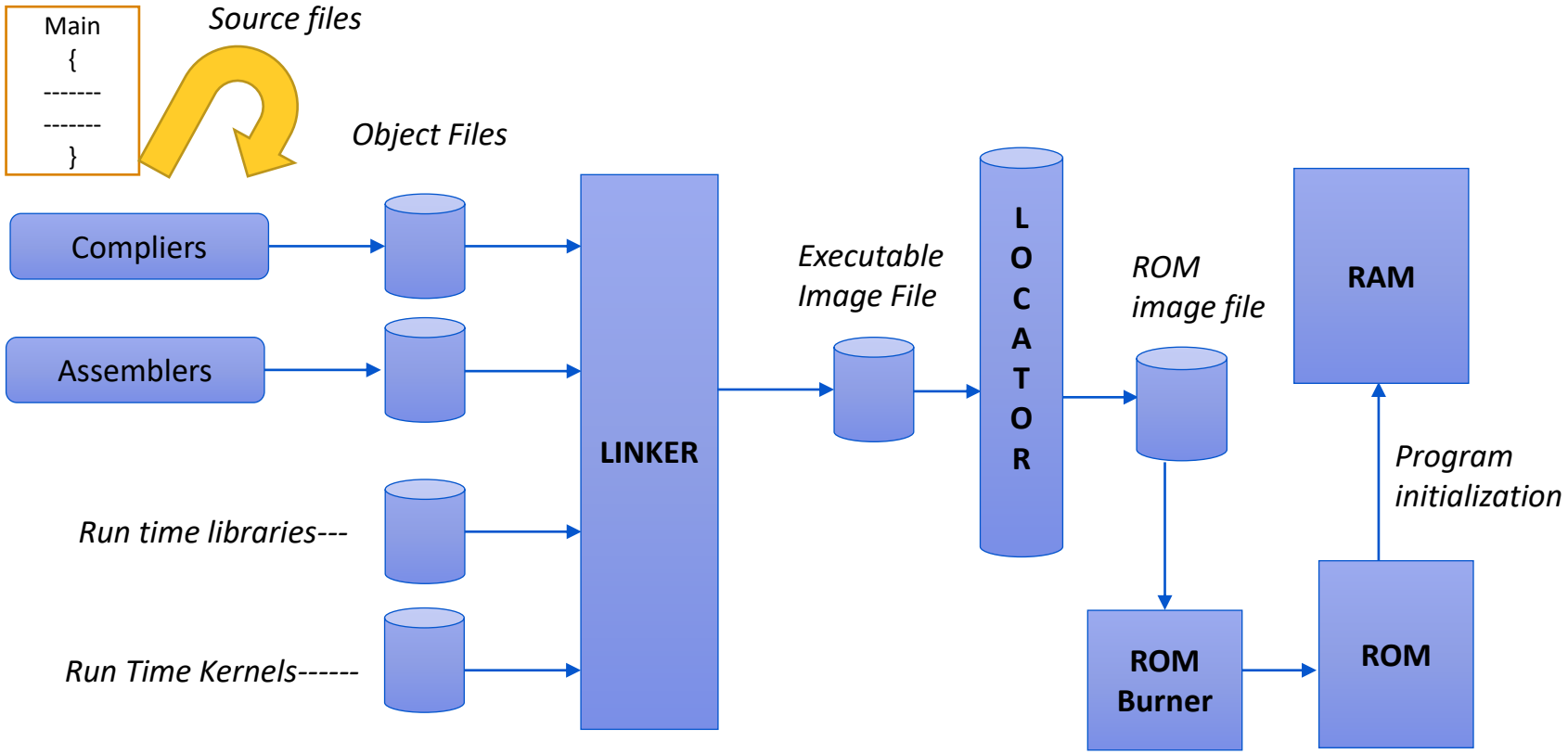
- Source code is seldom translated directly into loadable binary images.
- Sophisticated suites of tools translate the source into relocatable modules, sometime with and without debug and symbolic information.
- Complex, highly optimized linkers and loaders dynamically combine these modules and map them to the specific memory locations when the application is executed.

“Embedded system developers don’t enjoy this kind of luxury”

Embedded application development



Build and load process in Embedded application



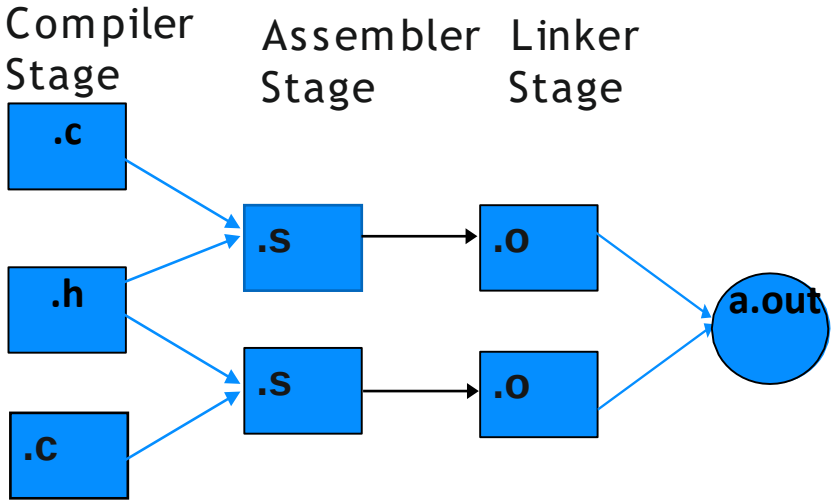
Various stages of build process

- **Pre-processing** is the first pass of any c compilation. It processes include- files, conditional compilation instructions and macros
- **Compilation** is the second pass. It takes the output of the pre-processor, and the source code, and generates assembler source code.
- **Assembly** is the third pass of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
- **Linking** is the final stage and pass of compilation. It takes one or more objects files or libraries as input and combines them to produce a single (usually executable) file.

Although there is clear distinction between compilation and assembly stage in build process, but developers collectively called these steps as compilation step.

Object files vs Executable file

- Object files is created after assembly stage of the build process. Source code is compiled and assembled by compiler and assembler respectively.
- All object files along with library object files are linked together by linker to create final executable.



Concept of headers and Linker

Interface file or header file structure

- Header file: Constant declarations
- Header file: Type declarations
- Header file: Global variables
- Header file: Function prototypes

Basic Rules – header files

- Each module with its .h and .c file should correspond to a clear piece of functionality.

- Always use “include guards” in a header file.
- Example adc_base.h header file
- #ifndef ADC_BASE_H
- #define ADC_BASE_H
- and end with:
- #endif
- All of the declarations needed to use a module must appear in its header file.

- Header file contains only declarations, and is included by the .c file for the module.
  - structure type declarations,
  - function prototypes, and
  - global variable extern declarations

- Put the function definitions and global variable definitions and initializations in the .c file.

Basic Rules – header files

- Global or public variables
- For global variables that will be known throughout the program, place an extern declaration in the .h file, as in
- extern int g_adc_presentvalue;
- The .c file for the module must include this same .h file, and near the beginning of the file, a defining declaration should appear –
- This declaration in C both defines and initializes the global variables, as in: int g_adc_presetvalue = 0;

adc_base.h

```
#ifndef ADC_BASE_H
#define ADC_BASE_H
extern int g_adc_presetvalue;
#endif
```

adc_base.c

```
#include<adc_base.h>
int g_adc_presetvalue=0;
---
---
---
```

Basic Rules – header files

- **Module's internal declarations out of the header file.**

- Put definitions or declarations near the top of the .c file and do not mention them in the .h file.
- Furthermore, declare globals and functions static in the .c file to give them internal linkage.

Various sections/ segments of executable file

- .text – used for program code.
- .bss – used for uninitialized objects (static variables)
- .data – used for initialized non- const objects (global variables)
- .const or .rodata – used for initialized const objects (strings constants or literals, variables declared const.)
- .reloc – used to store information required to be relocating image while loading.
- .stack – used for the function call stack.
- .sysmem – used for the dynamic memory allocation pool.

Depending upon the complier/ linker these segments may be named differently

Linking concepts

- Normally a compiler or assembler, has no visibility of the source code it is processing and has very limited information about the target system memory map.
- One of the linker's functions is to relocate sections into the target system's memory map; this function is called placement. Because most systems contain several types of memory, using sections can help you use target memory more efficiently.

Linker File Format

- The Linker file has to define all memory regions for Program memory and RAM. Also, it has to define start and end labels for each section and the LINK constants (LNK_AI) which are being used by the application code.

Linker script file allows the linker to map the text/data/bss regions in every object file into real physical addresses and also influences the merging of all the sections together into the final executable file.

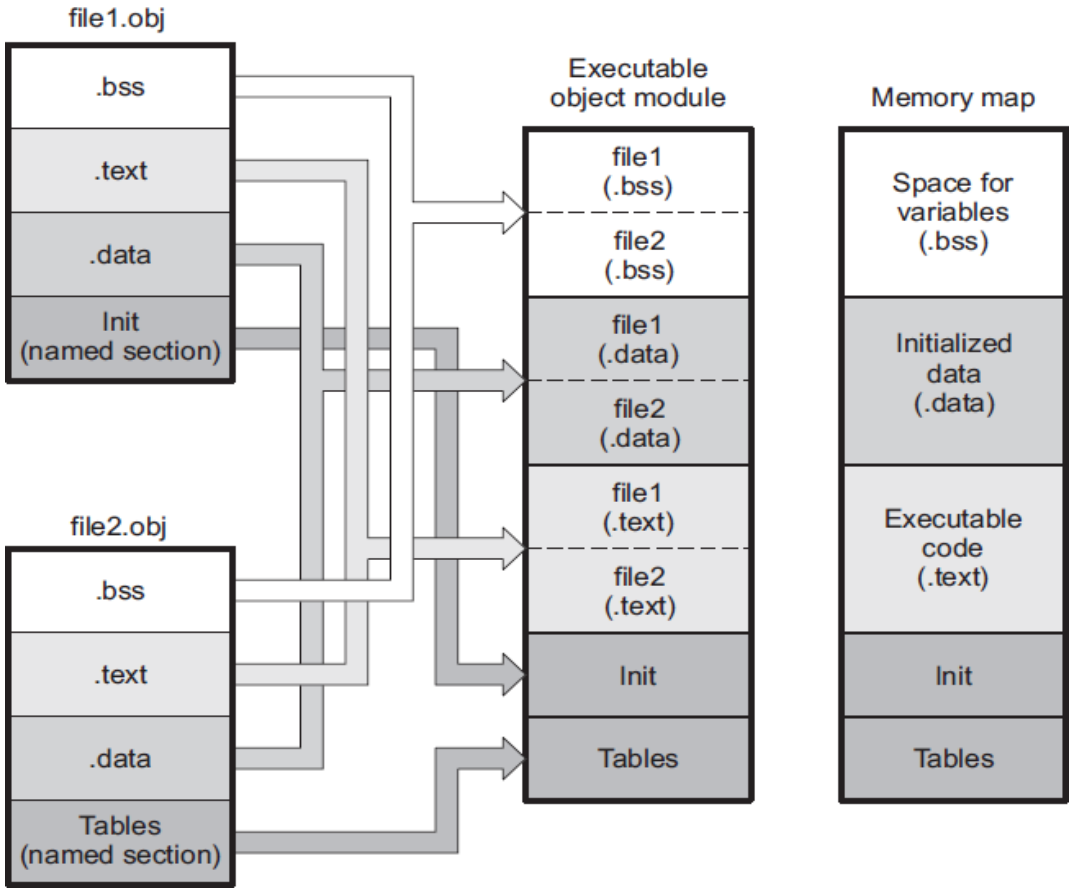
Linker role and object code

- The *MEMORY* directive allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.

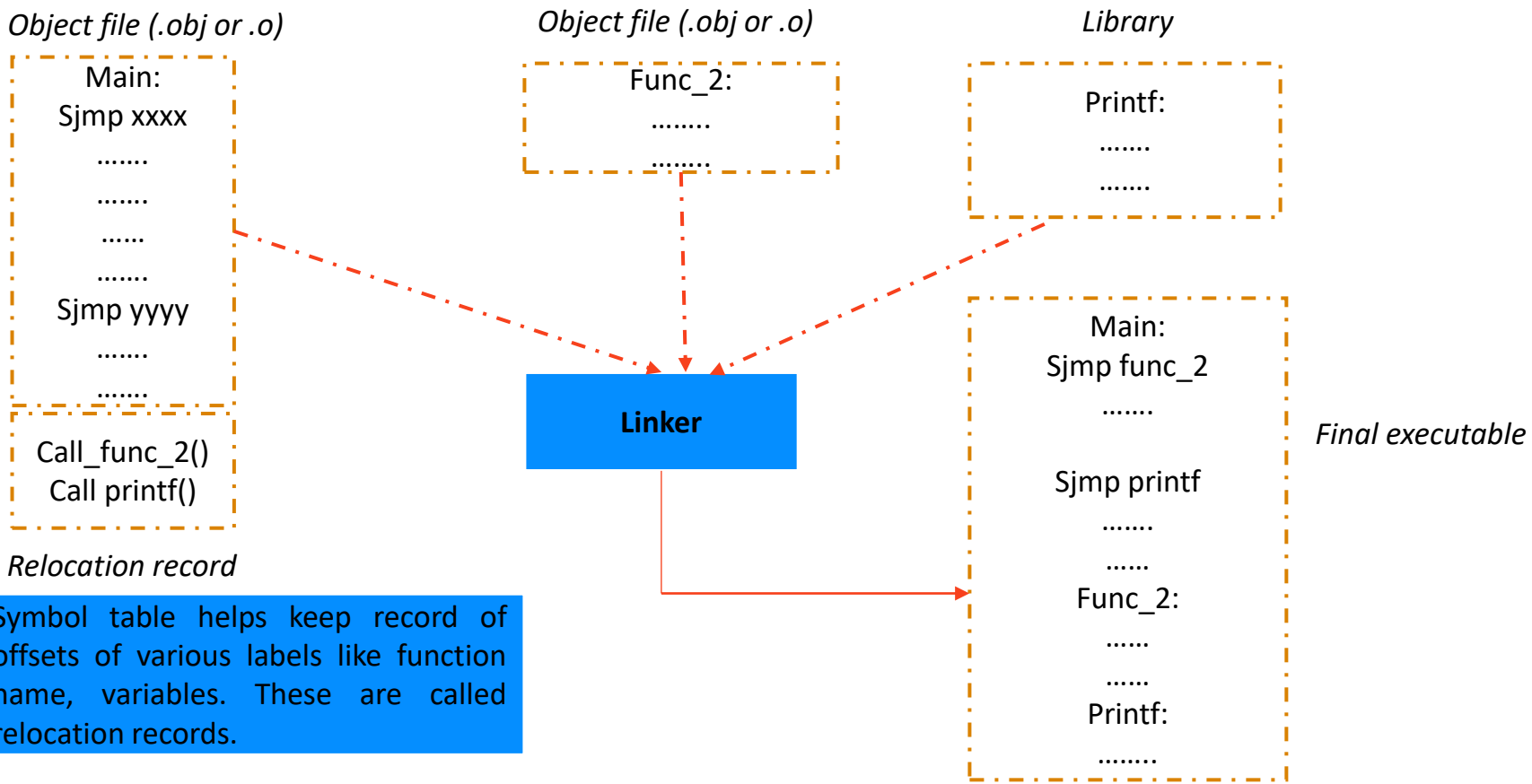
- The *SECTIONS* directive tells the linker how to combine input sections into output sections and where to place these output sections in memory.

Line numbers	Object code	Section
19	403A	.text
19	1234	
20	521A	
20	0001!	
35	403C	
35	3456	
36	421D	
36	0000!	
37	1290	
37	FFEE!	
6	0011	.data
6	0022	
6	0033	
14	0123	
25	00AA	
25	00BB	
25	00CC	
10	No data - ten bytes reserved	.bss
29	No data - eight bytes reserved	newvars
30		

Linker role and object code

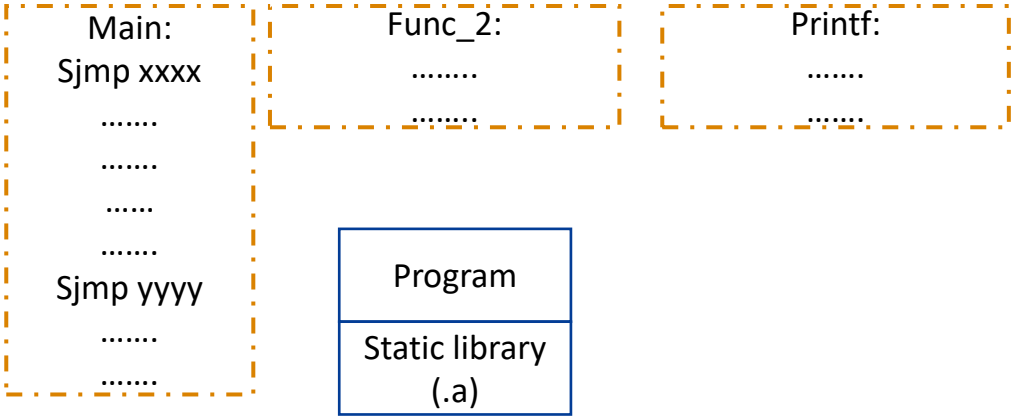


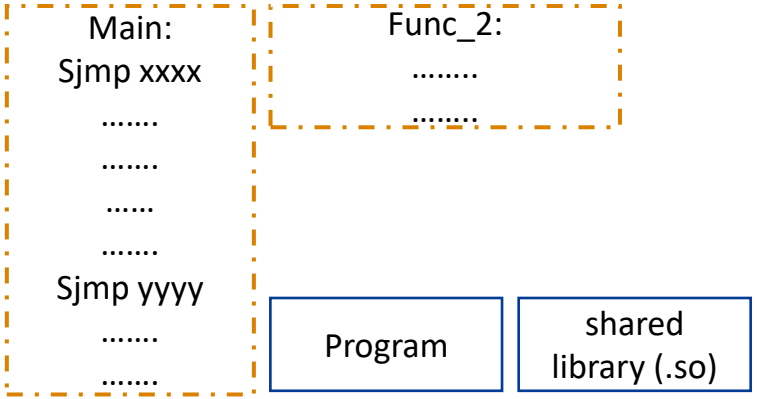
An example of linking



Symbol table helps keep record of offsets of various labels like function name, variables. These are called relocation records.

Type of linking : static and dynamic linking





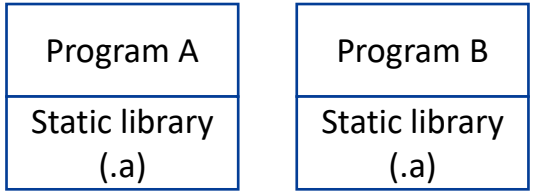
Static linking

Program includes the code of library function in final executable e.g. printf code is included in final executable

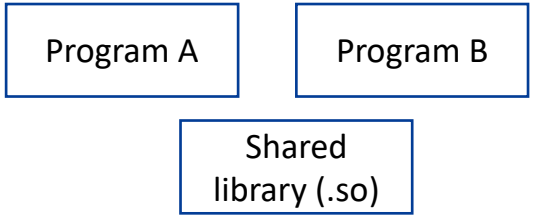
Dynamic Linking

Program doesn't include the code of library functions in final executable e.g. printf code is not included in final executable

Type of linking : static and dynamic linking



Static linking at link time

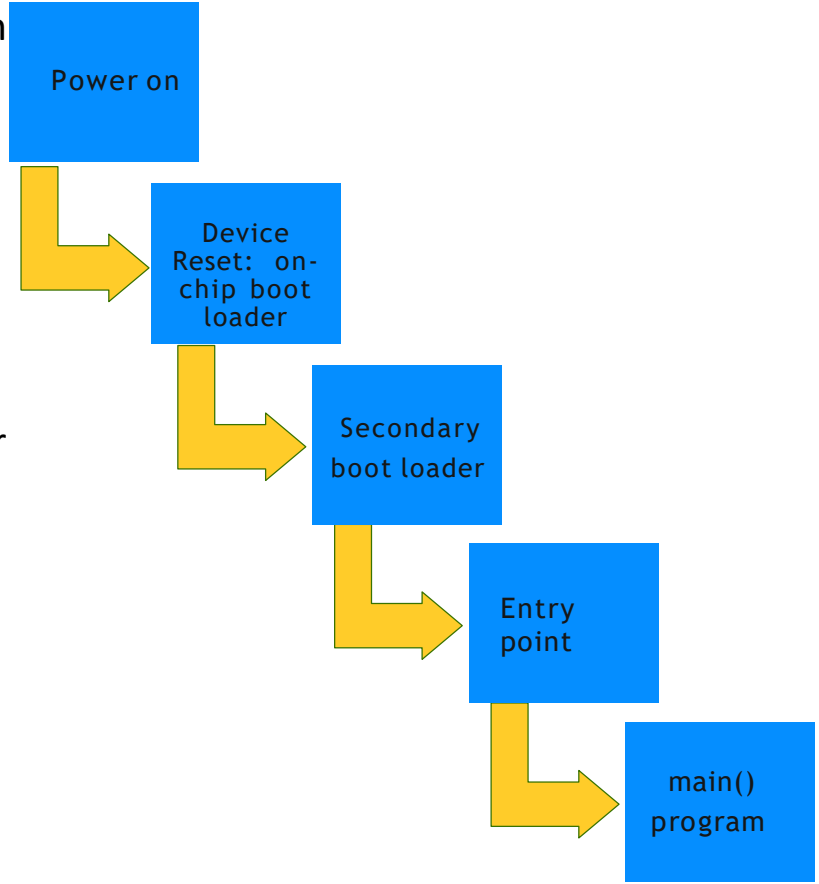


Dynamic linking at run time

Static linking leads to code duplication and memory wastage in multi-program applications, as the library code is a part of final executable for each program

Loading

- *Loading* is the process of preparing a program for execution by initializing device memory with the program's code and data. A *loader* might be
  - ✓ another program on the device,
  - ✓ an external agent (for example, a debugger), or
  - ✓ the device might initialize itself after power-on, which is known as *bootstrap loading*, or *boot loading*.
- Hex conversion utility automatically builds the boot table forthe secondary bootloader.
- specify the sections you want to initialize, the boot table location, and the name of the section containing the secondary bootloader routine and where it should be located.



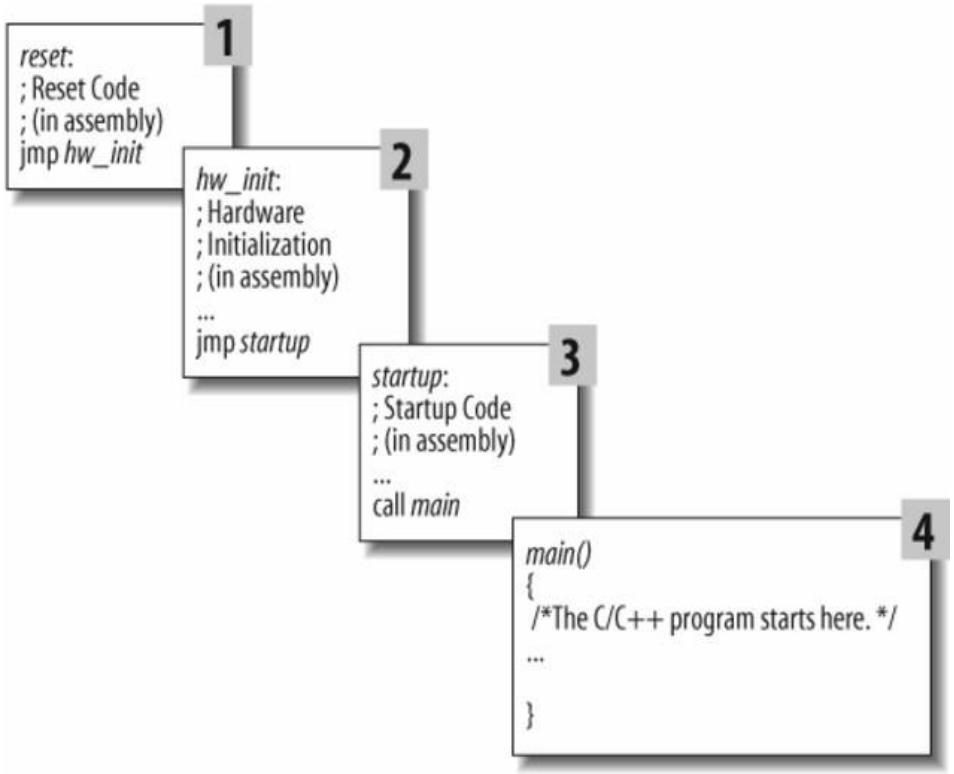
Entry point to main program

- The entry point is the address of the startup routine (address at which the execution of the program begins). Device itself cannot read the entry point field from the object file, so it has to be encoded in the program somewhere.

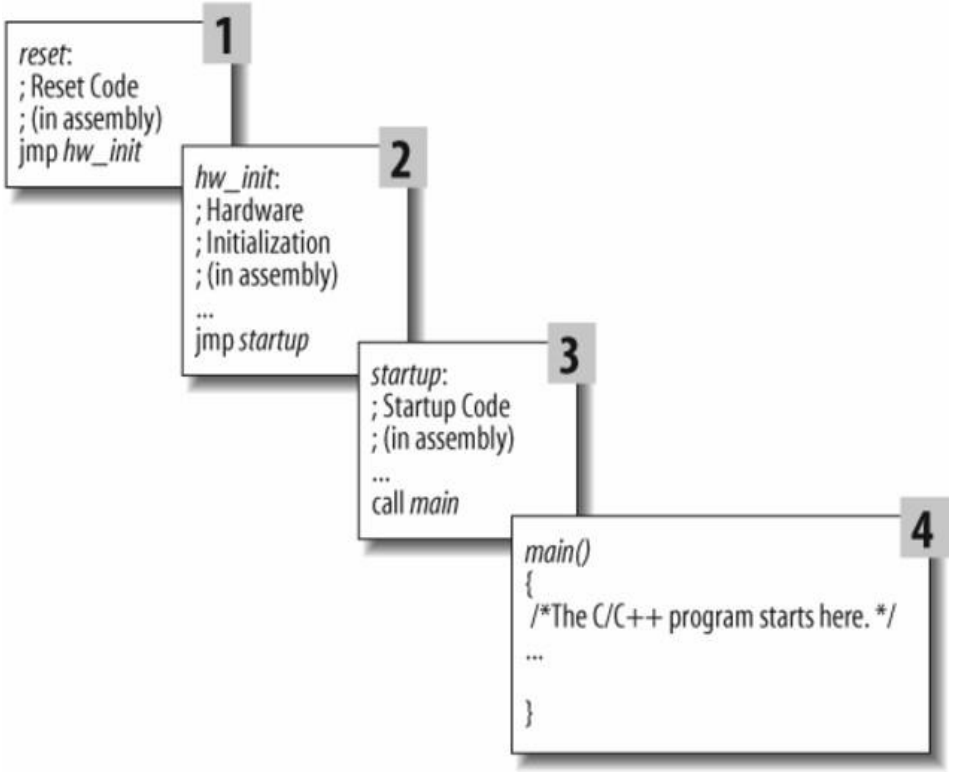
- If you are using a bootloader, the boot table includes an entry point field. When it finishes running, the bootloader branches to the entry point.
- If you are using an interrupt vector, the entry point is installed as the RESET interrupt handler. When RESET is applied, the startup routine will be invoked.
- If you are using a hosted debugger, the debugger may explicitly set the program counter (PC) to the value of the entry point.

Hardware – Software Initialization process

- The main () program only runs when CPU hardware is initialized
- The first stage of the initialization process is the reset code. This code is written in assembly language and is executed when CPU reset itself.
- The first instruction of the reset code must be placed at a specific location in memory, usually called the reset vector.
- Hardware initialization phase initialize the interrupt controller and other critical peripherals.
- Less critical hardware devices can be initialized when the associated device driver is started, usually from within main.



Hardware – Software Initialization process



- Startup code prepare the way for code written in a high-level language. One of the important tasks of this code is to set up the stack for the system.

- After initializing the stack, the startup code calls main

Important downloads

Link for Reference

- https://www.st.com/en/evaluation-tools/stm32f4discovery.html
- https://www.st.com/en/evaluation-tools/nucleo-f446re.html
- For Downloading IDE: https://www.st.com/en/development-tools/stm32cubeide.html

Hardware Required

For hands-on- Hardware and software required

HARDWARE			
No	Item	Nos	Note
1	STM32F407 DISCOVERY board	1	This is the main development board for which drivers are developed. Use either one. These boards are explained in this course
	LEDs	1	5mm LED
2	Push button	1	Mini Pushbutton Switch
3	Resistors	1	22KΩ, 4.7KΩ, 470Ω ; 1 each
4	jumper wires	15	M-M, M-F,F-F; 5 each
5	breadboard	1	full or medium size

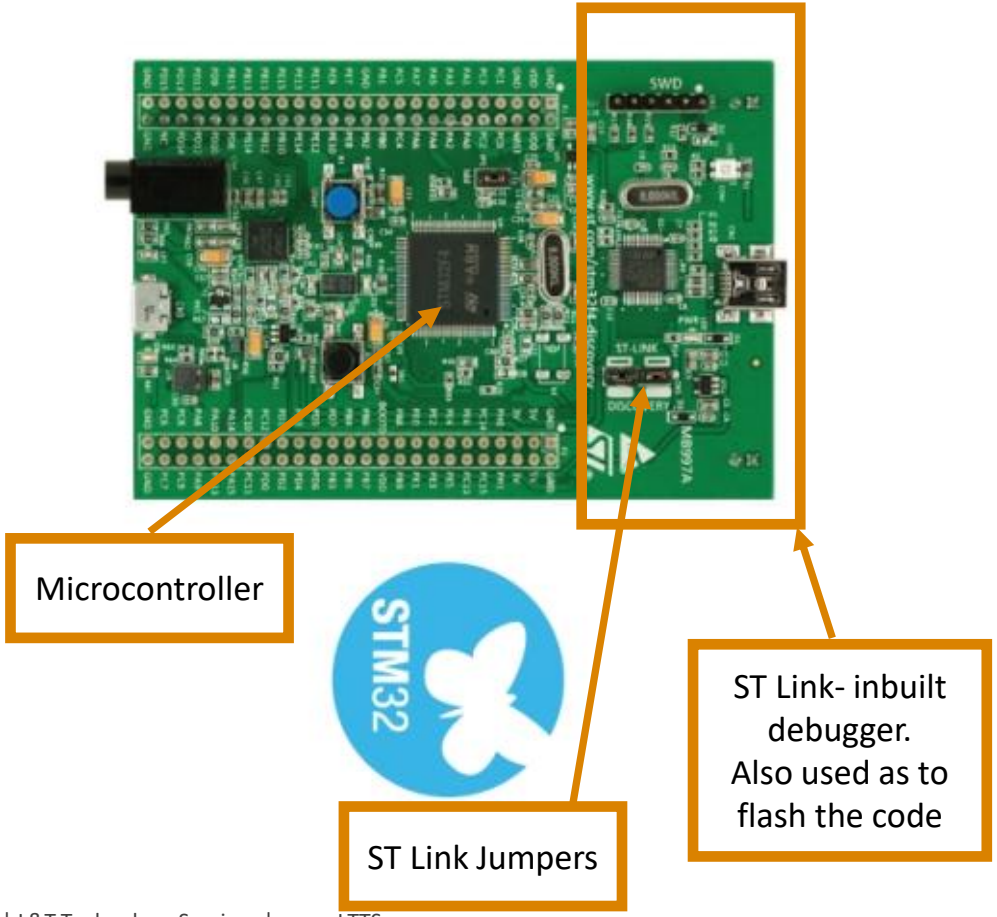
SOFTWARE				
No	Item		Note	version
2	STM32CUBEIDE		Eclipse based IDE . Provided by ST . Multi OS support (windows/linux/mac)	FREE

About the STM32 Discovery board

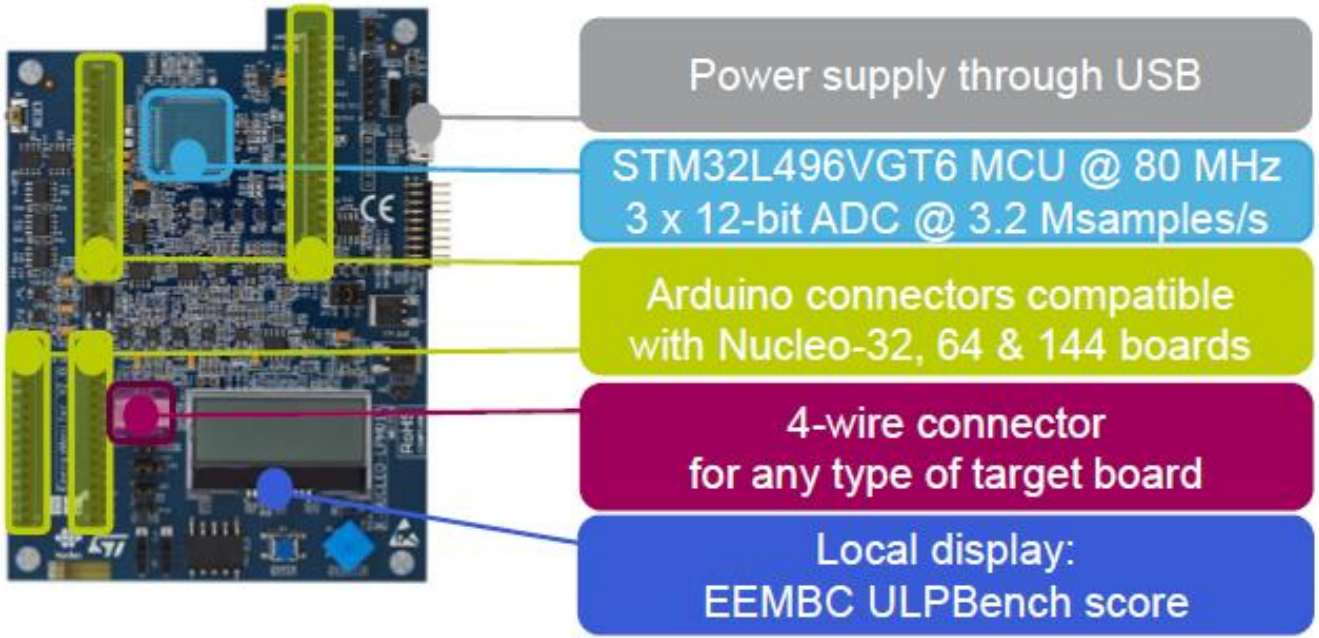
STM Discovery board

Features:

1. STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex -M4 with FPU core, 1-Mbyte Flash memory, 192-Kbyte RAM in an LQFP100 package
2. ST Link is inbuilt in the discovery board.
3. USB ST-LINK with re-enumeration capability and three different interfaces:
   1. Debug port
   2. Virtual Com port
   3. Mass storage
4. Board power supply: through USB bus or from an external 5 V supply voltage
5. LIS302DL or LIS3DSH ST MEMS 3-axis accelerometer
6. Eight LEDs
7. MP45DT02 ST-MEMS audio sensor omni-directional digital microphone
8. Two push-buttons (user and reset)
9. 8MHz of external clock but microcontroller by default uses internal RC based clock



STM32 Power Shield anatomy



IDE installation

- IDE is a software that contains all the essential tools to develop, compile, link, debug your code.
- In some cases, you may have to integrate compiler and debugger tool to the IDE manually
- We will be using Eclipse based STM32CubeIDE which is developed by ST microelectronics to write, compile, debug application on STM32 ARM based microcontroller.
- STM32CubeIDE is an eclipse IDE with STM32 related customization

Installation steps

- Refer slide No 3 to download the IDE. Once you go to the link mentioned you will get below screen.

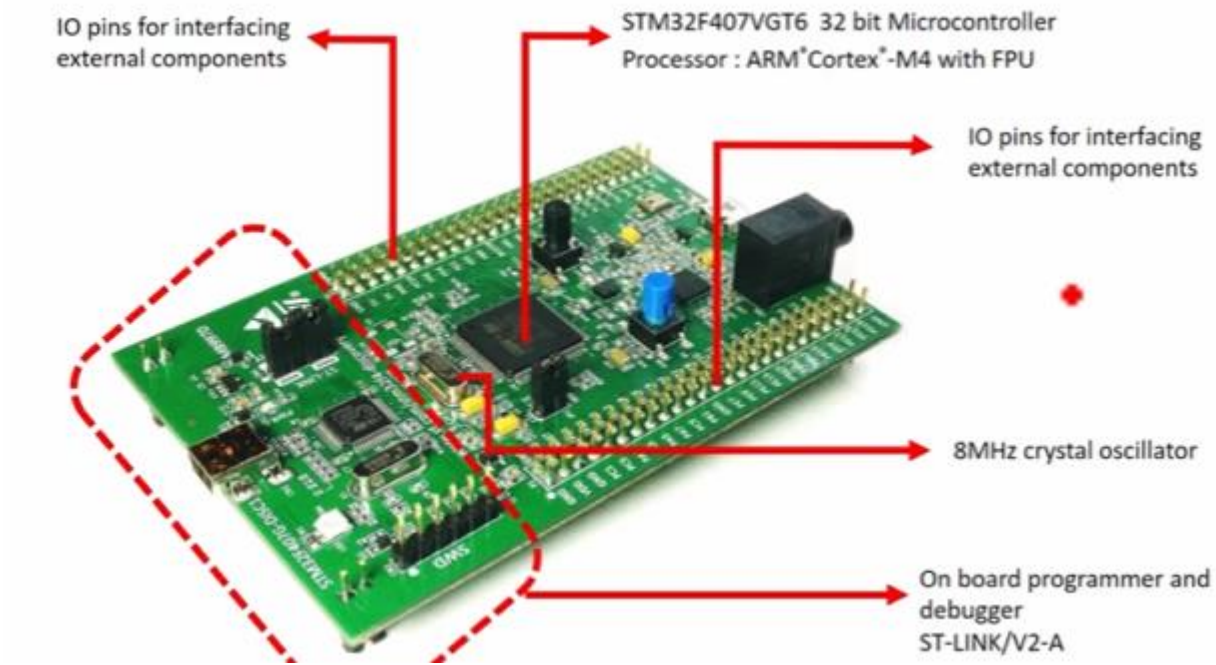
Get Software

	Part Number	General Description	Software Version	Download	Previous versions
+	STM32CubeIDE-DEB	STM32CubeIDE Debian Linux Installer	1.4.0 - 1.4.0	Get Software	Select version
+	STM32CubeIDE-Lnx	STM32CubeIDE Generic Linux Installer	1.4.0	Get Software	Select version
+	STM32CubeIDE-Mac	STM32CubeIDE macOS Installer	1.4.0	Get Software	Select version
+	STM32CubeIDE-RPM	STM32CubeIDE RPM Linux Installer	1.4.0 - 1.4.0	Get Software	Select version
+	STM32CubeIDE-Win	STM32CubeIDE Windows Installer	1.4.0	Get Software	Select version

- Select the latest version and it will ask for email id. Once you will fill the details along with your valid email id, it will send the link to your email id and from there you can download and install the IDE

Embedded Target

- STM32F407VGT6 MCU based DISCOVERY board is used as an embedded target board.



Board connection to PC

Documents required

- Two set of documents required

1. Document related to microcontroller-

https://www.st.com/en/microcontrollers-microprocessors/stm32f407vg.html

Step to be followed : After clicking above link go to resources and download datasheet and reference manual by clicking on DS8626 and reference manual option.

2. Document related to discovery board-

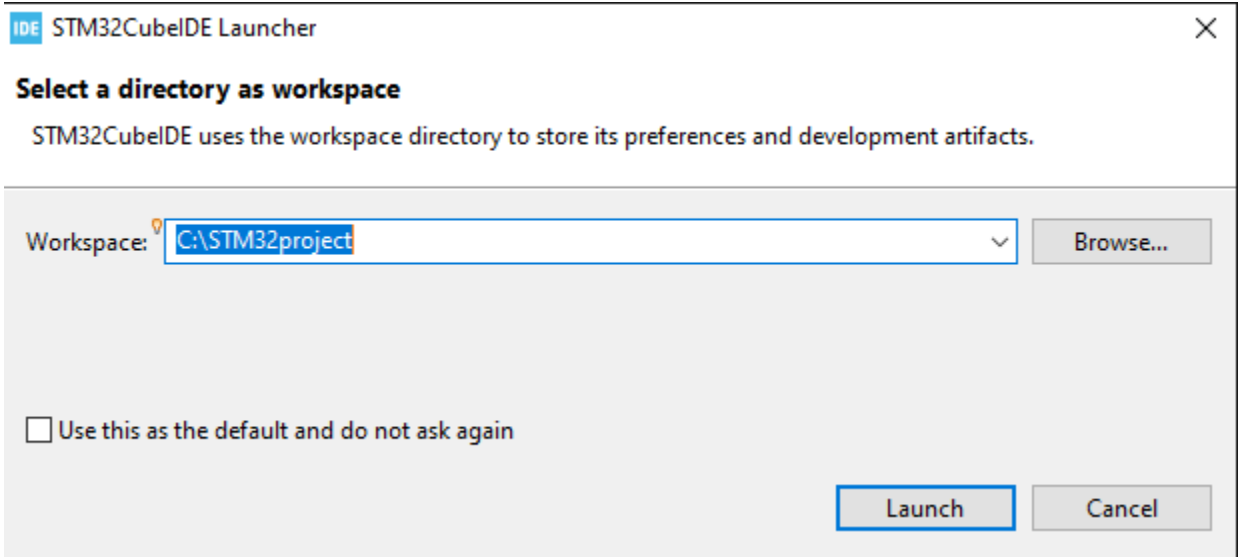
https://www.st.com/en/evaluation-tools/stm32f4discovery.html

Steps to be followed: After clicking above link go to the landing page of board and click on resources. In recourse page click on these options and download:

a) DB1421
b) User manual
c) Schematics

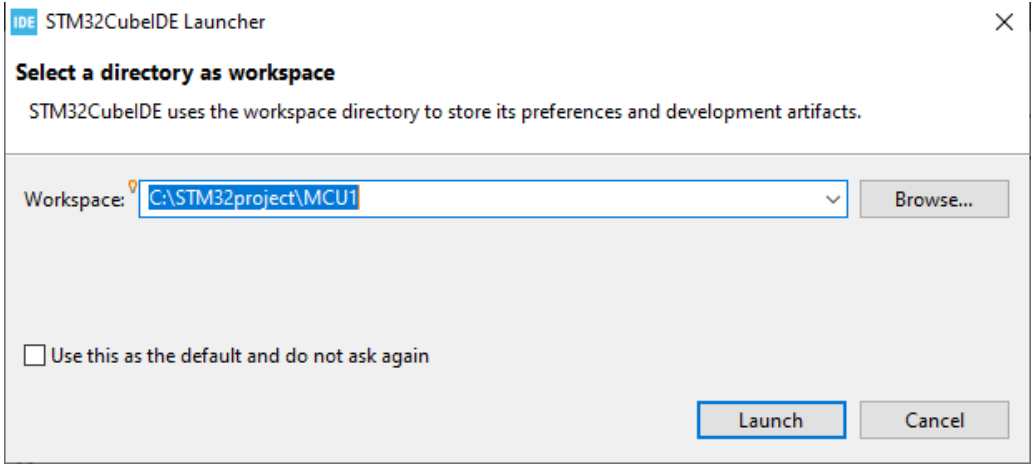
Creating project in STM32CUBEIde – Step 1

- Launch the STM32CubeIde and it will ask for workspace folder.
- Create a workspace folder in your local machine where you will keep all your projects and paste it in the path. See the image below.

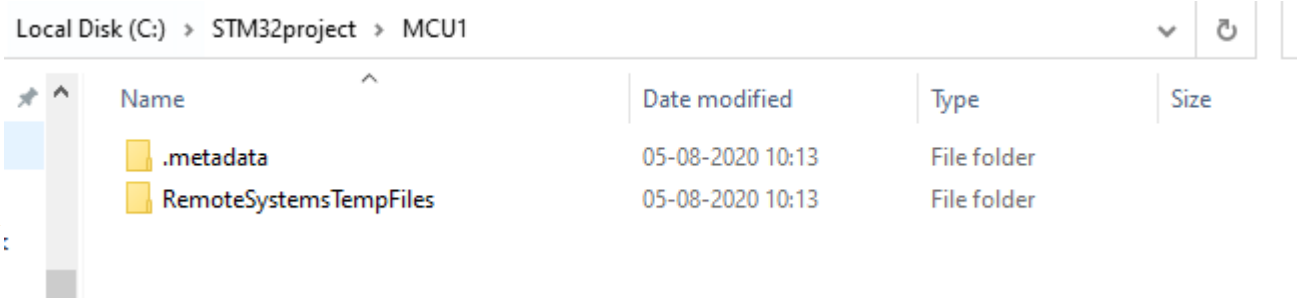


Step 2

- Give one back slash and give the workspace name. In my case I am giving **MCU1.** See the image below and launch the IDE.

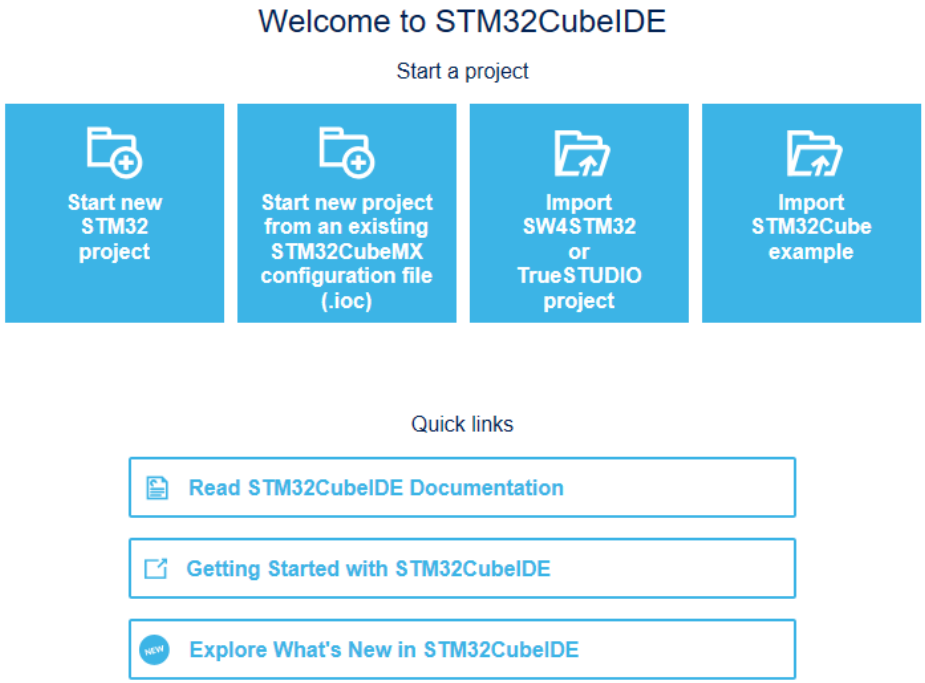


- Workspace got created



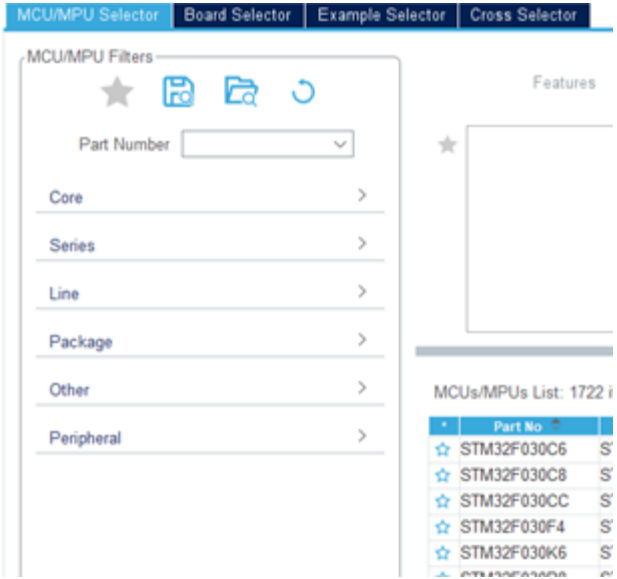
Step 3

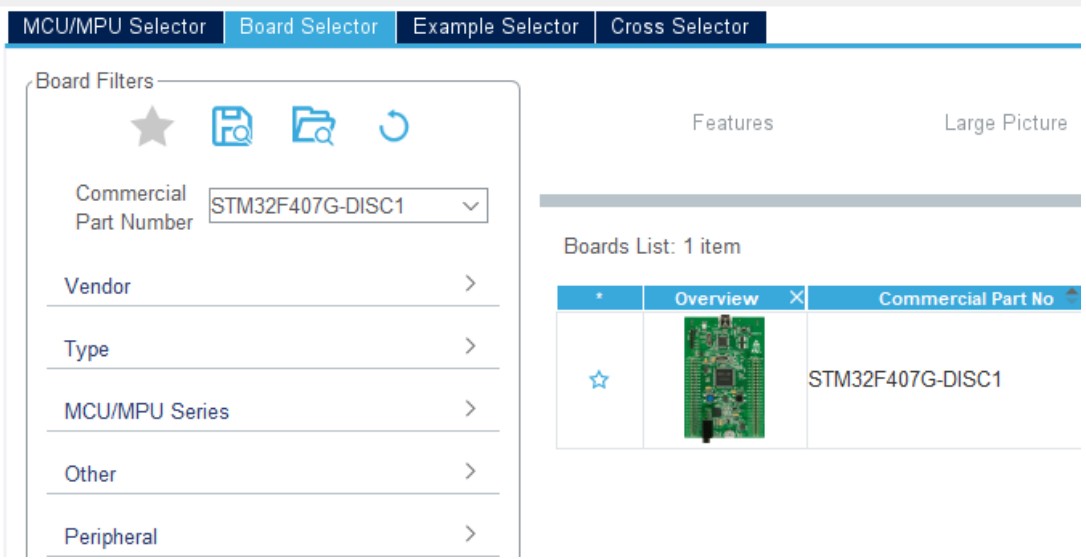
- Starting page of IDE looks like below image. Click on start new STM32 Project and it will redirect you to the target selection window.



Step 4

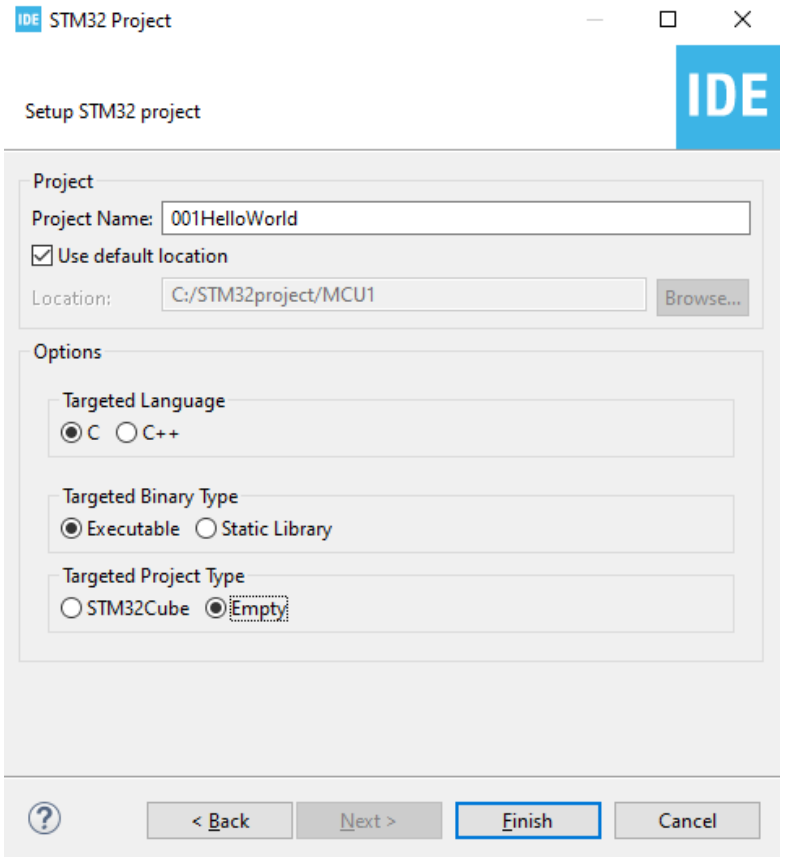
- In target selection window, select the target board. Below is the image of window
- Select the board selector tab as we are using standard STM board.
- In part no. tab write the board name – “STM32F407G-DISC1” and hit next tab.





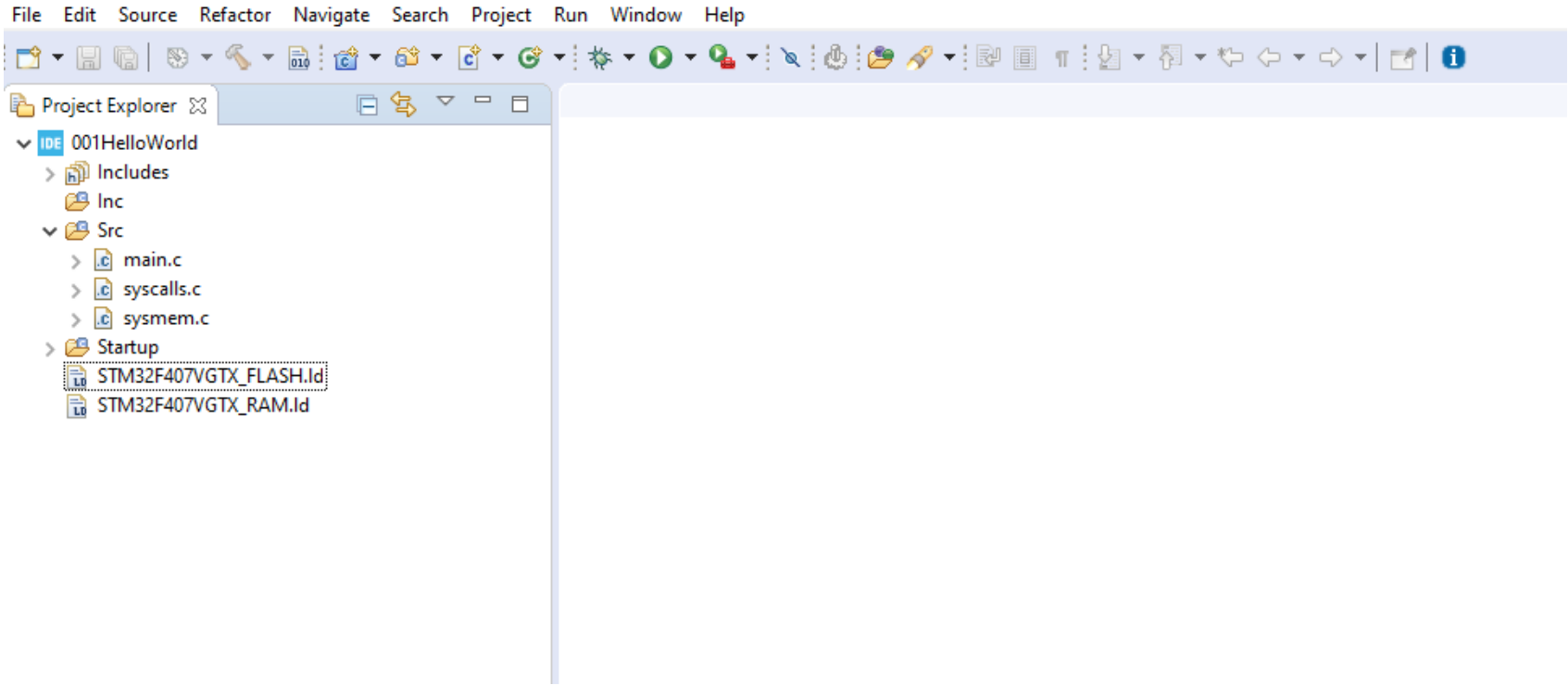
Step 5

- After pressing next, you have to give the name of the project.
- In my case the project name I have kept is 001HelloWorld.
- Please select empty targeted project type as you are going to write the code from the scratch.



Step 6

- After clicking finish button your first hello world program is ready.
- Open the main.c file and build this empty code to check that your compiler is working.



Step 7

- The compile the code- right click on the project and click build project and observe the console window

Problems | Tasks | Console | Properties

CDT Build Console [001HelloWorld]

```
10:37:43 **** Build of configuration Debug for project 001HelloWorld ****
make -j8 all
arm-none-eabi-gcc -mcpu=cortex-m4 -g3 -c -x assembler-with-cpp -MMD -MP -MF"Startup/s
arm-none-eabi-gcc "../Src/main.c" -mcpu=cortex-m4 -std=gnu11 -g3 -DSTM32 -DSTM32F407C
arm-none-eabi-gcc "../Src/syscalls.c" -mcpu=cortex-m4 -std=gnu11 -g3 -DSTM32 -DSTM32F
arm-none-eabi-gcc "../Src/sysmem.c" -mcpu=cortex-m4 -std=gnu11 -g3 -DSTM32 -DSTM32F40
../Core/Inc/stm32f4xx_hal_msp.h:27:1: warning: #warning "FPU is not initialized, but the project is com
   #warning "FPU is not initialized, but the project is compiling for an FPU. Please
    ^~~~~~~
arm-none-eabi-gcc -o "001HelloWorld.elf" @"objects.list"   -mcpu=cortex-m4 -T"C:\STM3
Finished building target: 001HelloWorld.elf

arm-none-eabi-size   001HelloWorld.elf
arm-none-eabi-objdump -h -S  001HelloWorld.elf  > "001HelloWorld.list"
arm-none-eabi-objcopy  -O binary  001HelloWorld.elf  "001HelloWorld.bin"
   text    data     bss     dec     hex filename
    676       8    1568    2252     8cc 001HelloWorld.elf
Finished building: default.size.stdout

Finished building: 001HelloWorld.bin

Finished building: 001HelloWorld.list
```

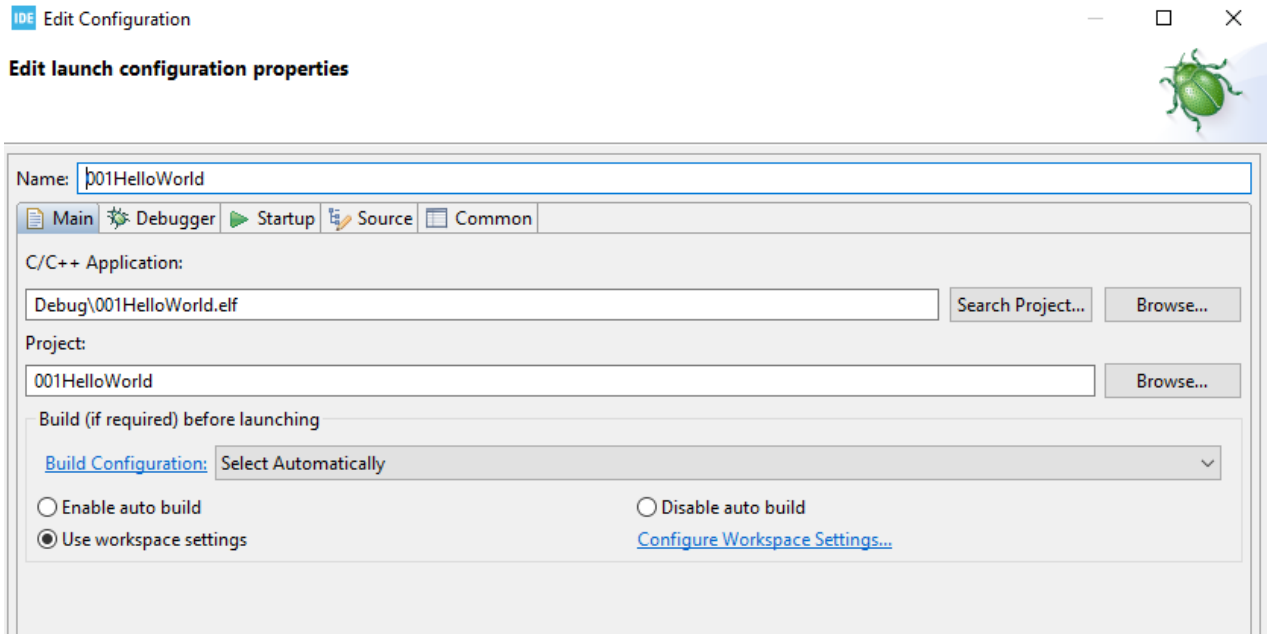
Cross compiler invoked

Warning of FPU- Floating point unit

Note :
To remove the warning of FPU, disable the FPU. Right click on project and go to properties → go to C/C++build → settings → tool settings → MCU Setting In MCU setting drop down the floating point unit and select none and in floating point ABI select software implementation (-mfloat-abi=soft) and then click apply and again build the project and the warning will not come again.

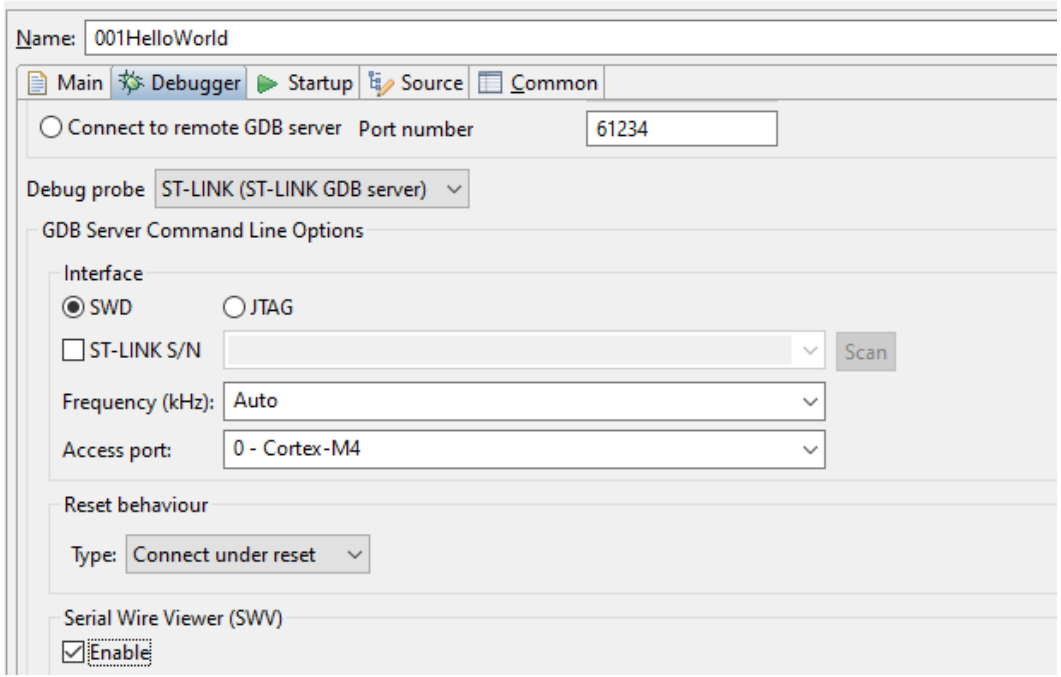
Step 8- Flashing the code on target board

- Connect your board to PC and flash the code.
- To Flash- right click on the project and select debug as and select STM32 cortex M C/C++ Application. Below image will appear. But before downloading we have to create debug configuration



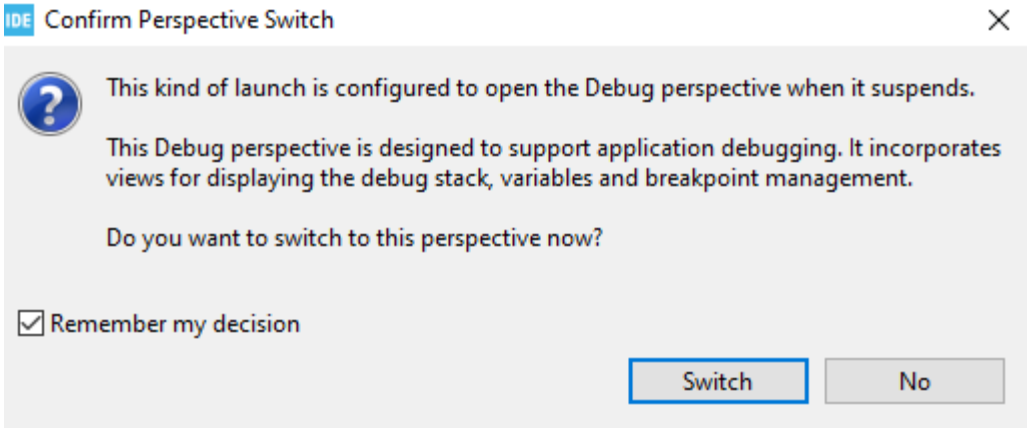
Step 9- debugging configuration

- Select the debug option and make sure the debugger named ST-Link (ST-LINK GDB server) must be select.
- Also enable SWV (serial wire viewer)



Step 10- debugging configuration

- In debugging option, system will ask to switch to debugging mode. Please check in the box “Remember my decision” and select switch tab.



- To terminate the debugging mode just select terminate button in the debugging selection tab.

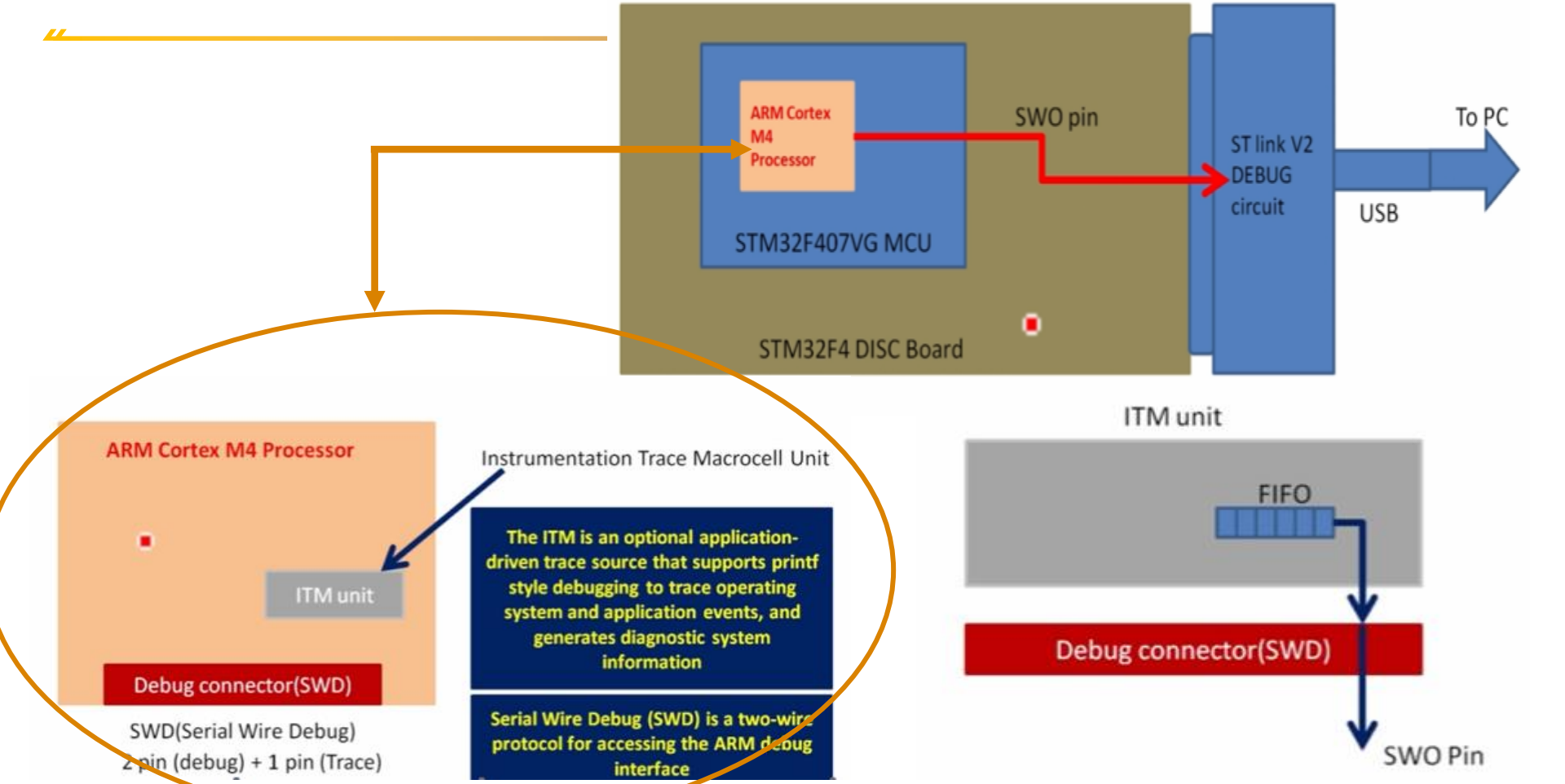
Step 11- Writing a code

- Include the library stdio.h and write printf statement as shown below

```
9
10 #if !defined(__SOFT_FP__) && defined(__ARM_FP)
11   #warning "FPU is not initialized, but the project is compiling for an FPU. Please initialize the FPU before use.
12 #endif
13
14 #include<stdio.h>
15
16 int main(void)
17 {
18     printf("Hello World\n");
19     for(;;);
20 }
```

SWV working principle

- We can use printf statement on ARM cortex M3/M4/M7 based MCUs
- Printf statement works over SWO pin (Serial wire output) of SWD (Serial wire debug) interface


ARM Cortex M4 Processor
STM32F407VG MCU
SWO pin
ST link V2 DEBUG circuit
To PC
USB
STM32F4 DISC Board
ARM Cortex M4 Processor
ITM unit
Debug connector(SWD)
SWD(Serial Wire Debug)
2 pin (debug) + 1 pin (Trace)
Instrumentation Trace Macrocell Unit
The ITM is an optional application-driven trace source that supports printf style debugging to trace operating system and application events, and generates diagnostic system information
Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface
ITM unit
FIFO
Debug connector(SWD)
SWO Pin

SWD

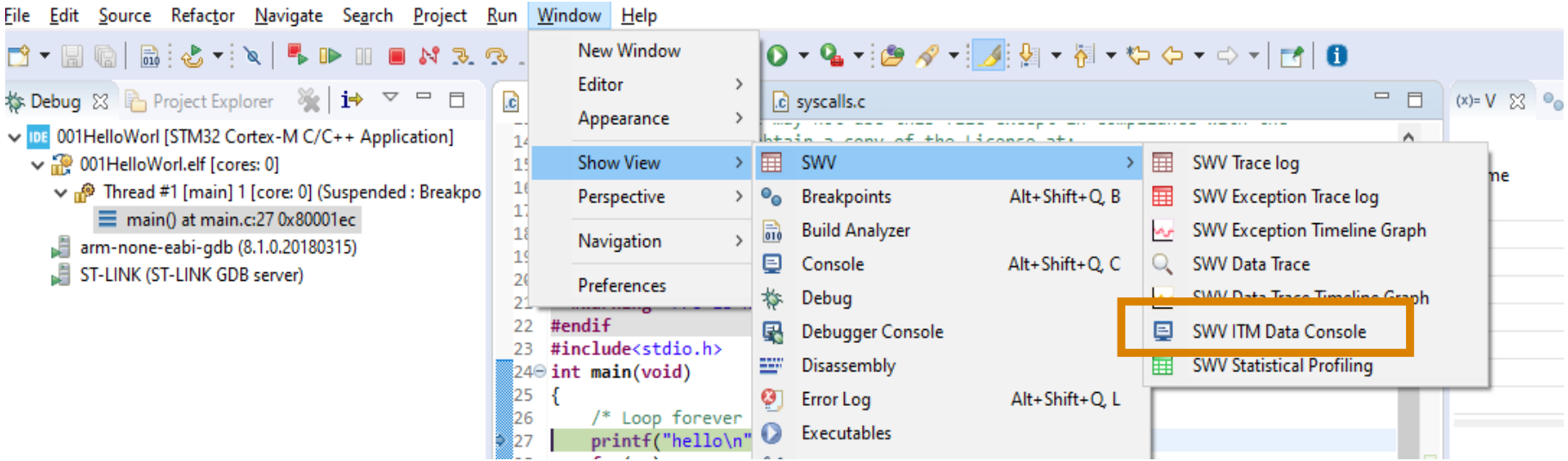
- Serial wire debug is a two wire protocol for accessing the ARM debug interface
- It is a part of the ARM debug interface specification v5 and is an alternative to JTAG
- The physical layer of SWD consists of two lines-
  - SWDIO: bidirectional data line
  - SWCLK: a clock driven by the host
- By using SWD interface should be able to program MCUs internal flash, you can access memory regions, add breakpoints, stop/run CPU.
- The other good thing about SWD is you can use the serial wire viewer for the printf statements for debugging.

Testing SWV using hello project

- AS ITM unit we are going to use so copy the ITM code (copy from the GIThub repo: https://github.com/niekiran/Embedded-C/blob/master/All_source_codes/target/itm_send_data.c)
- Copy and paste into the syscalls.c file after line no 32 and under write function comment __io_putchar(*ptr++); and add ITM_SendChar(*ptr++);
- Connect your board with PC and build the project and verify the debug configuration. Make sure SWV option is enabled.

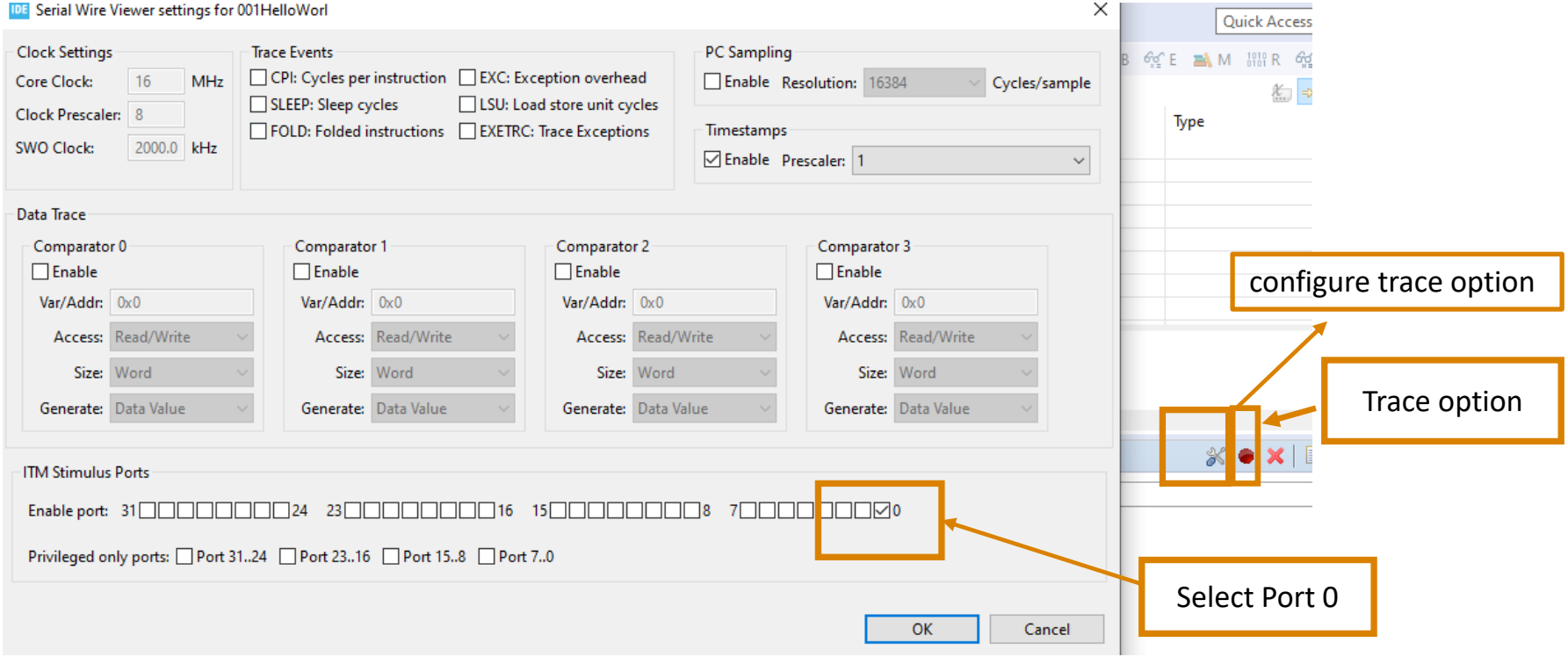
Testing SWV using hello project

- Now go to windows-→show view → SWV→ and select SWV ITM data console.
- In console window select configure trace option shown in next slide



Trace configuration and usage of SWV

- Click Configuration trace option and select port 0 as mentioned below and run trace option



- After running trace just click the resume button in IDE as mentioned below and you will see the output in the console window

File Edit Source Refactor Navigate Search Project Run Window Help

Debug | Project Explorer

001HelloWorl [STM32 Cortex-M C/C++ Application]
001HelloWorl.elf [cores: 0]
Thread #1 [main] 1 [core: 0] (Running)
arm-none-eabi-gdb (8.1.0.20180315)
ST-LINK (ST-LINK GDB server)

Resume tab

main.c | sysmem.c | syscalls.c | startup_stm32f407vgtx.s

```
14  * License. You may obtain a copy of the License at:
15  *                        opensource.org/licenses/BSD-3-Clause
16  *
17  ******************************************************************************
18  */
19
20  #if !defined(__SOFT_FP__) && defined(__ARM_FP)
21    #warning "FPU is not initialized, but the project is compiling for an FPU. Please
22  #endif
23  #include<stdio.h>
24  int main(void)
25  {
26      /* Loop forever */
27      printf("hello\n");
28      for(;;);
29  }
30
```

Console | Problems | Executables | Debugger Console | Memory | SWV ITM Data Console

Port 0

```
hello
```

Console output

OpenOCD (Open On-Chip Debugger) debugger and Semi- hosting

- Set the linker arguments

```
-specs=rdimon.specs -lc -lrdimon
```

- Add semi-hosting run command

```
monitor arm semihosting enable
```

- Add the below function call to main.c

```
extern void initialise_monitor_handles(void);
initialise_monitor_handles();
```

Steps involved

- Go to debug configuration option and select new debugging configuration and select ST-Link OpenOCD debug probe
- After that click on startup tab and add run command - *monitor arm semihosting enable*
- Then close it and right click on the project and select C/C++ build option and go to settings and then to tool setting and select miscellaneous in MCU GCC linker and add flags- *-specs=rdimon.specs -lc –lrdimon*
- And now in code add *initialise_monitor_handles(void);* before any print command and add its prototype - *extern void initialise_monitor_handles(void);*
- And also disable syscall file
- Now build the project and go to the debug the mode after connecting the target board. The board enabled semi hosting.
- Press step over in the debug mode and hello will get printed in the console window

Snapshot for your reference

- 001HelloWorldSemiHosting
  - Binaries
  - Includes
  - Inc
  - Src
    - main.c
    - sysmem.c
    - syscalls.c
  - Startup
  - Debug
  - 001HelloWorldSemiHosting Debug.cfg
  - 001HelloWorldSemiHosting Debug.launch
  - STM32F407VGTX_FLASH.ld
  - STM32F407VGTX_RAM.ld

```
14  * License. You may obtain a copy of the License at:
15  *                        opensource.org/licenses/BSD-3-Clause
16  *
17  ******************************************************************************
18  */
19
20  #if !defined(__SOFT_FP__) && defined(__ARM_FP)
21    #warning "FPU is not initialized, but the project is compiling for an FPU. Please
22  #endif
23
24  #include<stdio.h>
25  extern void initialise_monitor_handles(void);
26  int main(void)
27  {
28      /* Loop forever */
29      initialise_monitor_handles();
30      printf("hello\n");
31      for(;;);
32  }
33
```

Problems | Tasks | Console | Properties

```
<terminated> 001HelloWorldSemiHosting Debug [STM32 Cortex-M C/C++ Application] ST-LINK (OpenOCD)
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : device id = 0x10076413
Info : flash size = 1024 kbytes
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000384 msp: 0x20020000
configuring PLL
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
Info : Unable to match requested speed 8000 kHz, using 4000 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000278 msp: 0x20020000
semihosting is enabled

hello
```

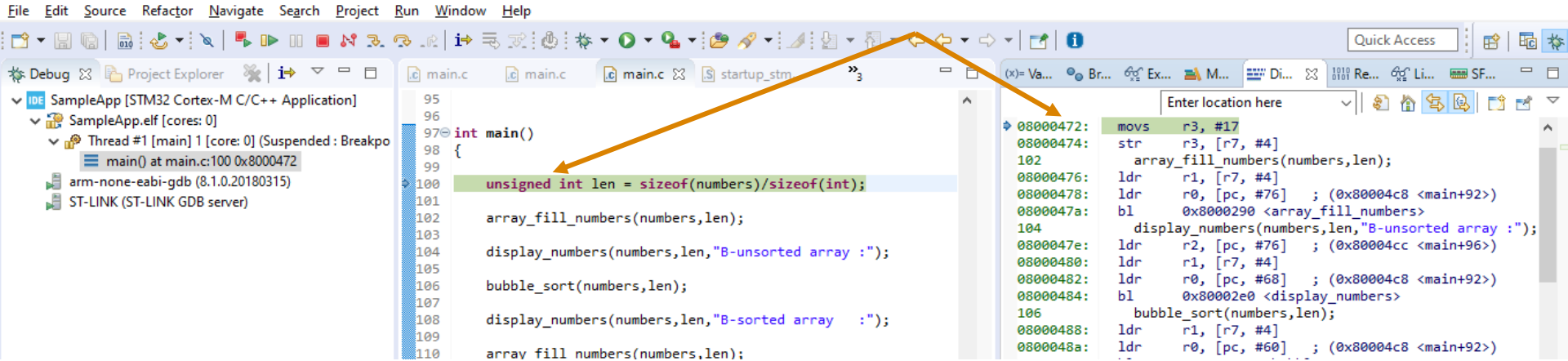
Embedded Code debugging- Tips and Tricks

- Serial wire viewer and data tracing (printf style debugging)
- Single stepping, stepping over and stepping out.
- Breakpoints (inserting, deleting and skipping breakpoints).
- Disassembly
- Call stack
- Expression and variable window
- Memory browser
- Data watch points

SampleApp Code

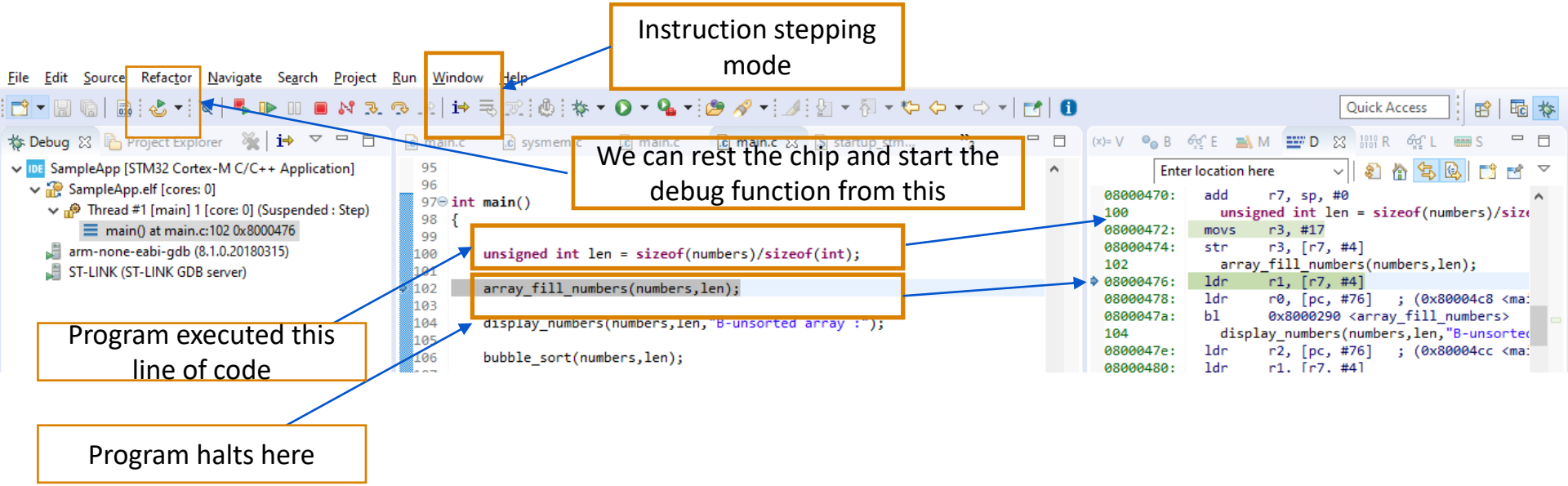
Single stepping, stepping over and stepping out

- Go to the debugging mode. You all know the process. The processor execution halted at the very first instruction of main(). IDE always halts the processor at the very first instruction or code statement of main(). Remember at this step code is not running on target as it is halted.
- If you want to know that what is running in the target window which is equivalent to the ARM instruction then go to windows tab and show the disassembly view. It looks like the below picture



Single stepping, stepping over and stepping out

- If you press step into then the IDE behaves as below. It will execute the program step wise.
- Now after one more step in it will go into the function and keep on executing till the statement become false.
- To come out of the loop we can use step return option.
- We can also use debugging on instruction level by activating the instruction stepping mode.



Disassembly and Register window

- Now you already know that how to enter into assembly window. Now you can see the registers also in the view panel

Location of flash memory

Opcode

```
(x)= Varia... Break... Expre... Modul... Disass... Regis... Live ... SFRs
Enter location here

                          main:
0800046c:  0x000080b5     push    {r7, lr}
0800046e:  0x000082b0     sub     sp, #8
08000470:  0x000000af     add     r7, sp, #0
100                         unsigned int len = sizeof(numbers)/sizeof(int);
08000472:  0x00001123     movs    r3, #17
08000474:  0x00007b60     str     r3, [r7, #4]
102                         array_fill_numbers(numbers,len);
                                [r7, #4]
                                [pc, #76]   ; (0x80004c8 <main+92>)
                          000290 <array_fill_numbers>
                          umbers(numbers,len,"B-unsorted array :");
                                [pc, #76]   ; (0x80004cc <main+96>)
                                [r7, #4]
                                [pc, #68]   ; (0x80004c8 <main+92>)
                          0002e0 <display_numbers>
                          rt(numbers,len);
```

```
Console  Problems  Executables  Debugger Console  Memory  SWV ITM Data Console  Registers
```

Name	Value	Description
General Registers		General Purpose and FPU Register G...
r0	536870912	
r1	536871084	
r2	536871128	
r3	17 (Decimal)	
r4	536871128	
r5	0	
r6	0	
r7	537001068	

```
Name : r3
    Hex:0x11
    Decimal:17
    Octal:021
```

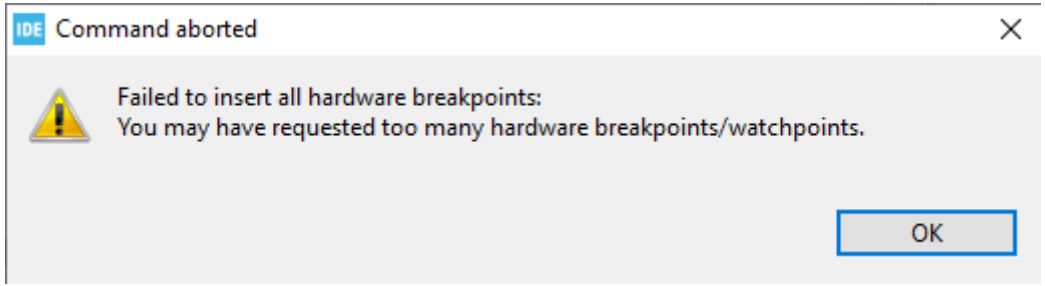
The value of r3 in assembly and register window

Breakpoints

- Breakpoints are the way to tell processor to halt or stop execution at certain instruction address
- Breakpoints are realized through hardware address comparators that are there inside the processor debug unit.
- The underlying processor must support breakpoints.
- By using IDE's we can set the breakpoints, we can delete it, and even skip it.
- Breakpoints are also known as hardware breakpoints because these are realized through the hardware unit of the processor
- There are also software breakpoints which are realized through breakpoint instructions

Breakpoint

- Right click on window tab and then show and then click on breakpoint.
- If we want to stop our execution at some line of code say line no. 62 then place the cursor in line no 62 and go to the run tab and select the toggle breakpoint option.
- Other short cut is double click on the blue line in the IDE code writing area.
- Now when we resume the code then code execute till line no. 62 and stops with the help of comparator in the hardware. So when the address get matched with the address of line no. 62 program stops executed.
- Always remember the no. of breakpoint depends upon the no. of comparator available in the microcontroller otherwise IDE will throw error like below pic



Expression and variable window

- To use this functionality, go to the debug mode and place a breakpoint and go to window tab and then show the variable window. Its looks like below
- Variable window shows only the main stack frame variables and not the global variables

Run Window Help

Quick Access

main.c | main.c | main.c | startup_stm...

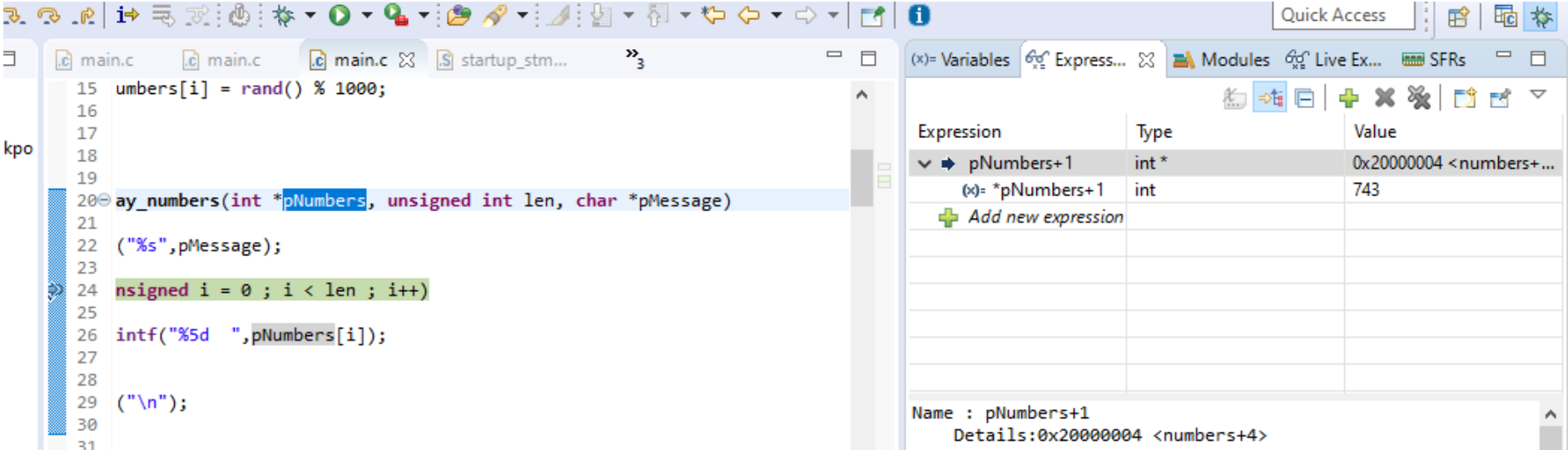
```
15          pNumbers[i] = rand() % 1000;
16      }
17
18  }
19
20  void display_numbers(int *pNumbers, unsigned int len, char *pMessage)
21  {
22      printf("%s",pMessage);
23
24      for (unsigned i = 0 ; i < len ; i++)
25      {
26          printf("%5d  ",pNumbers[i]);
27      }
28
```

(x)= Variables | Express... | Modules | Live Ex... | SFRs

Name	Type	Value
pNumbers	int *	0x20000000 <num...
len	unsigned int	17
pMessage	char *	0x80015d4 "B-uns...
i	unsigned int	17

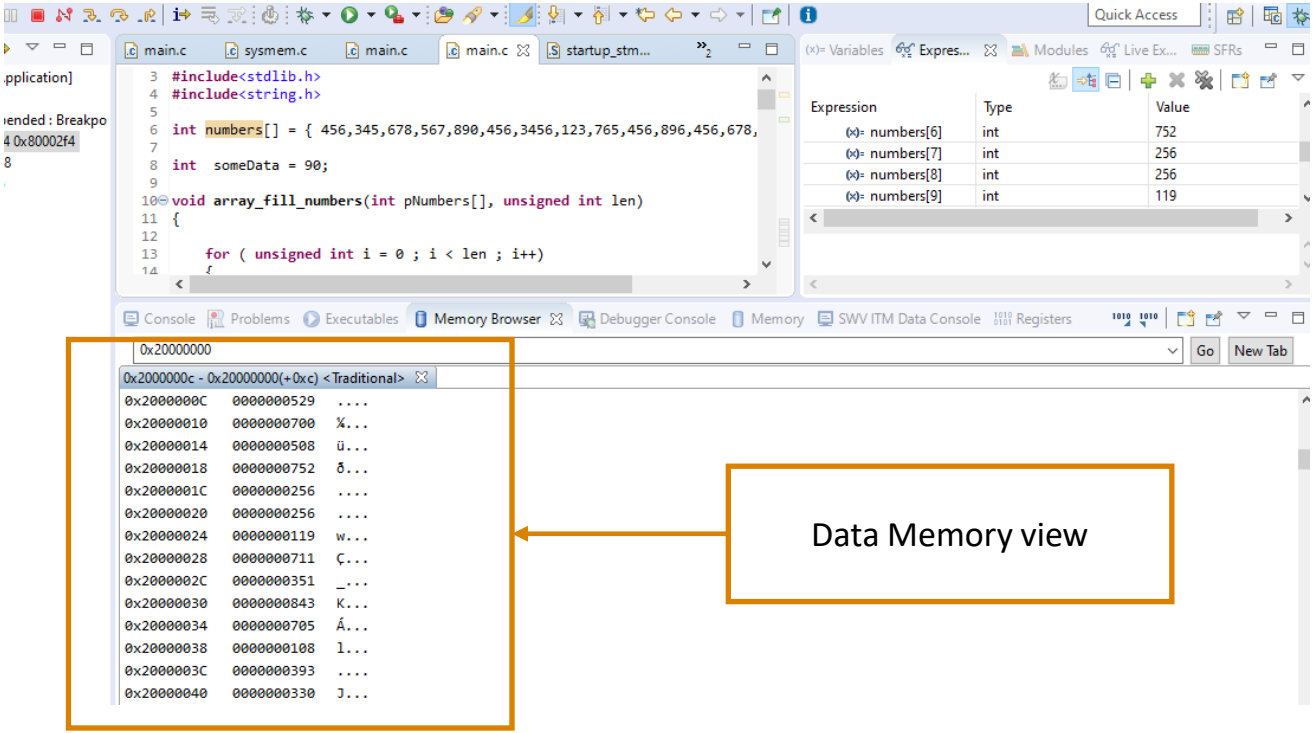
Expression and variable window

- If on same pointer you want to do some operation then expression window will help you out. See example below.
- Just drag and drop the pnumber and it will show the address.
- Now try modifying the code in expression window. Expression window give you satisfying execution level on single line.
- Micro level instruction we can write and execute in the expression window



Memory browser window

- Lets us see the SRAM view of the variables in the code which is storing the array for that just type the address of the variable in the memory window. And you all know how to get the address (use expression window for the same)



Different view of Disassembly and memory window

- See the location and map how the code sits in the flash memory

```
#include<stdlib.h>
#include<string.h>

int numbers[] = { 456,345,678,567,890,456,3456,123

int  someData = 90;

void array_fill_numbers(int pNumbers[], unsigned i
{

    for ( unsigned int i = 0 ; i < len ; i++)
```

```
080002f4: 0x00000023    movs    r3, #0
080002f6: 0x00007b61    str     r3, [r7, #20]
080002f8: 0x00000be0    b.n     0x8000312 <display_num
 26                     printf("%5d  ",pNumbers[
080002fa: 0x00007b69    ldr     r3, [r7, #20]
080002fc: 0x00009b00    lsls    r3, r3, #2
080002fe: 0x0000fa68    ldr     r2, [r7, #12]
08000300: 0x00001344    add     r3, r2
08000302: 0x00001b68    ldr     r3, [r3, #0]
08000304: 0x00001946    mov     r1, r3
```

```
0x080002F4  00  .
0x080002F5  23  #
0x080002F6  7B  {
0x080002F7  61  a
0x080002F8  0B  .
0x080002F9  E0  à
0x080002FA  7B  {
0x080002FB  69  i
0x080002FC  9B  .
0x080002FD  00  .
0x080002FE  FA  ú
0x080002FF  68  h
0x08000300  13  .
0x08000301  44  D
0x08000302  1B  .
0x08000303  68  h
```

Flash Memory view

Call stack and fault analyzer

- Let us make some exception in a code and try to find the exception in call stack window.
- We have written a code
  - **void (*jump_addr) (void); // function pointer**
  - jump_addr = **(void*)0x20000009; //**
  - jump_addr();

Debug | Project Explorer

- SampleApp [STM32 Cortex-M C/C++ Application]
  - SampleApp.elf [cores: 0]
    - Thread #1 [main] 1 [core: 0] (Suspended : Signal : S
      - WWDG_IRQHandler() at startup_stm32f407vgtx
      - <signal handler called>() at 0xfffffff1
      - _end() at 0x200000dc
      - swap_numbers() at main.c:42 0x8000356
  - arm-none-eabi-gdb (8.1.0.20180315)
  - ST-LINK (ST-LINK GDB server)

main.c | sysmem.c | syscalls.c | startup_stm32f407... | main.c | main.c | startup_stm32f407...

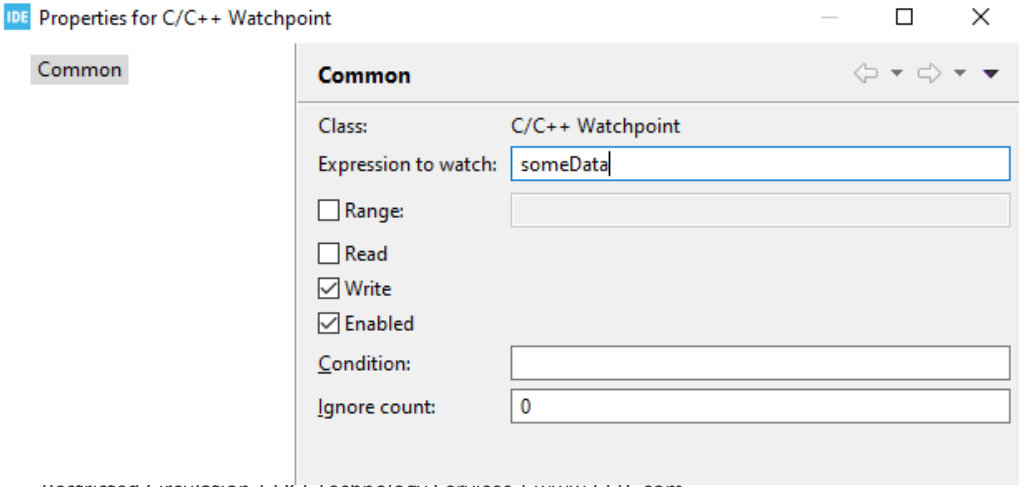
```
107 *         unexpected interrupt.  This simply enters an infinite loop, preserving
108 *         the system state for examination by a debugger.
109 *
110 * @param  None
111 * @retval : None
112 */
113   .section .text.Default_Handler,"ax",%progbits
114 Default_Handler:
115 Infinite_Loop:
116   b Infinite_Loop
117   .size Default_Handler, .-Default_Handler
118
119 /*******************************************************************************
120 *
121 * The STM32F407VGTx vector table.  Note that the proper constructs
122 * must be placed on this to ensure that it ends up at physical address
123 * 0x0000.0000.
124 *
125 *******************************************************************************/
126   .section .isr_vector,"a",%progbits
127   .type g_pfnVectors, %object
```

Program get trapped in infinite loop handler

Call stack window will give us information that where is the problem

Data Watch points

- Let us assume that we have global variable <someData =90;> and in the function somewhere the value is changed to 10.
- How do you find out which instruction in the project changing the global variable someData?
- Data watch points will help in finding the corrupt values.
- Go to the debug mode and open breakpoint window and add watch point. This window will appear
- Add the variable name (you can also write address) <someData> in our case and press apply and close. Watch point got added. Now if someone attempt to change the value of this variable. System will halt the program



SFR (Special function Register) Window

- View of SFR window is like below.
- Very useful to write drivers.

SFR Window

```
28
29      printf("\n");
30  }
31
32
33  void swap_numbers(int *x,int *y)
34  {
35      int temp=*x;
36      *x=*y;
37      *y=temp;
38
39  #if 0
40      void (*jump_addr) (void); // function
41      jump_addr = (void*)0x20000009; //
42      jump_addr();
43  #endif
44
45      someData = 10;
46  }
47
48
49  void bubble_sort(int *pNumbers , unsigned i
50  {
```

Register	Address	Value
SPI5		
SPI6		
SDIO		
ADC1		
SR	0x40012000	0x0
CR1	0x40012004	0x0
CR2	0x40012008	0x0
SMPR1	0x4001200c	0x0
SMPR2	0x40012010	0x0
JOFR1	0x40012014	0x0

Device: STM32F407
Version: 1.4

Description:

SFR (Special function Register) Window

- Without writing code we can control the led of the board also.
- Go to RCC Section→AHB1ENR→GPIODEN and change bit 4 from 0 to 1.

RD | x_{16} x_{10} x_2

type filter text

Register	Address	Value
BKPSRAMEN	[18:1]	0x0
CRCEN	[12:1]	0x0
GPIOIEN	[8:1]	0x0
GPIOHEN	[7:1]	0x0
GPIOGEN	[6:1]	0x0
GPIOFEN	[5:1]	0x0
GPIOEEN	[4:1]	0x0
GPIODEN	[3:1]	0x1
GPIOCEN	[2:1]	0x0
GPIOBEN	[1:1]	0x0

MSB 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 LSB

Bit field:	GPIODEN
LSB:	3
MSB:	4
Size:	1

SFR (Special function Register) Window

- Then go to GPIOD, expand it and change MODER12 from 00 to 01.
- Then in GPIOD, expand ODR (Output data register) and set ODR12 to 1 and LED turns ON

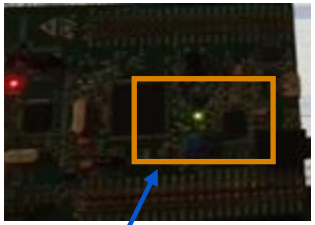
RD | x_{16} x_{10} x_{2}

type filter text

Register	Address	Value
ODR	0x40020c14	0x1000
ODR15	[15:1]	0x0
ODR14	[14:1]	0x0
ODR13	[13:1]	0x0
ODR12	[12:1]	0x1
ODR11	[11:1]	0x0
ODR10	[10:1]	0x0
ODR9	[9:1]	0x0
ODR8	[8:1]	0x0
ODR7	[7:1]	0x0

MSB 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 LSB

Bit field:	ODR12
LSB:	12
MSB:	13
Size:	1



Led Turns ON

Understanding MCU (STM32F407X) Memory map

- In ARM cortex M series, width of the system bus is 32 bits. i.e. the processor can produce 2^32 of different addresses.
- The memory address starts from 0x0000_0000 to 0xffff_ffff.
- For memory map refer reference document of the microcontroller. And go to the memory map section. See image below.
- E.g. if the processor produces 0x4002 0000 on the system bus then it is actually referring to the GPIOA register.

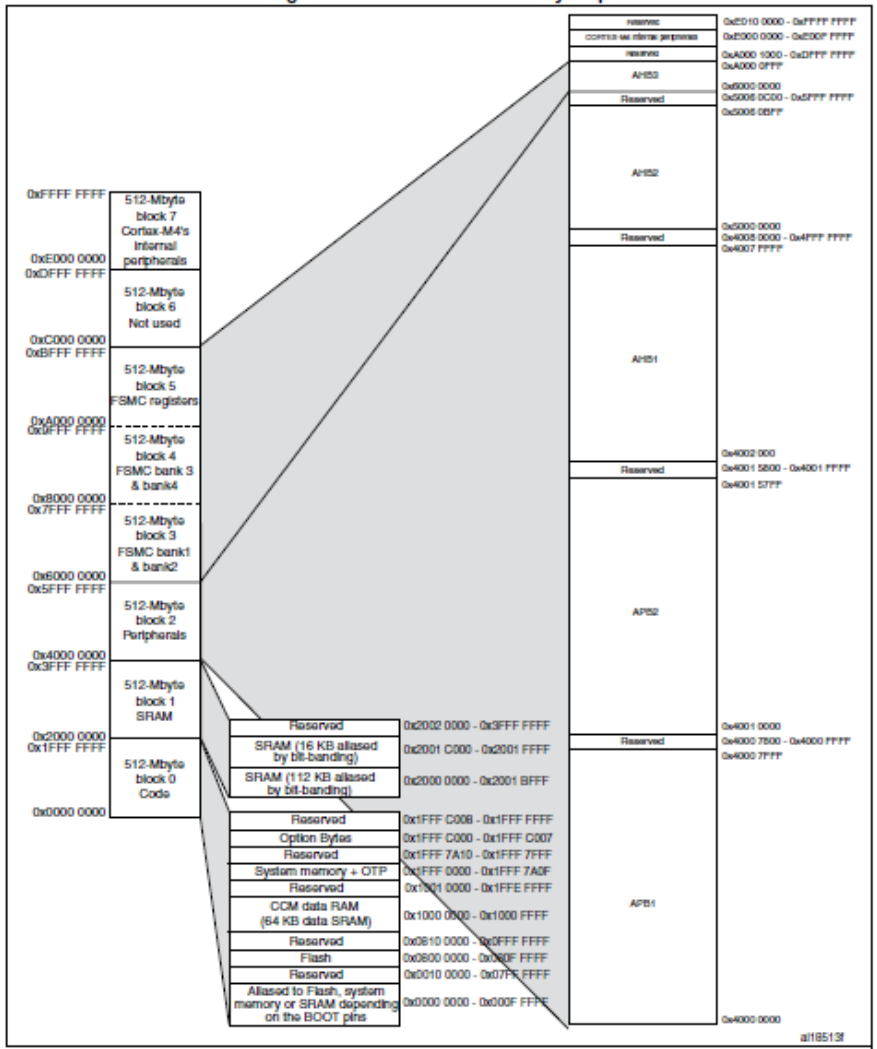
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		
0x4001 6800 - 0x4001 6BFF	LCD-TFT	APB2	*Section 16.7.26: LTDC register map on page 512*
0x4001 5800 - 0x4001 5BFF	SAI1		*Section 29.17.9: SAI register map on page 963*
0x4001 5400 - 0x4001 57FF	SPI6	APB2	*Section 28.5.10: SPI register map on page 925*
0x4001 5000 - 0x4001 53FF	SPI5		

Memory Map Chart

EXERCISE:

1. What is the base address of the AHB1 BUS peripheral register?
2. What is the base address of the GPIOA register
3. What is the base address of the RCC engine
4. What is the base address of the APB1 peripheral resister
5. What is the base address of the Flash Memory
6. What is the base address of the SRAM 2
7. What is the base address of the ADC register

Figure 18. STM32F40xxx memory map

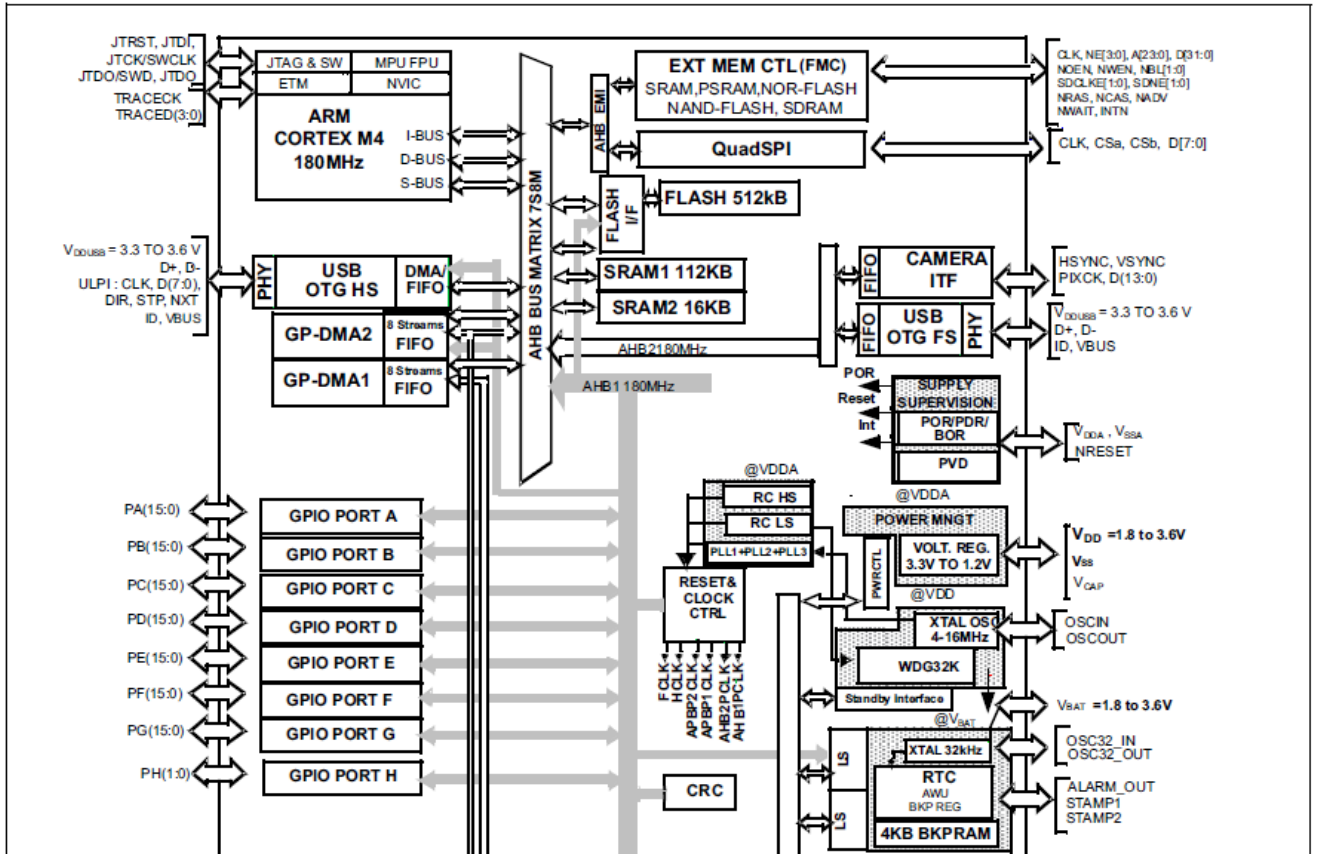


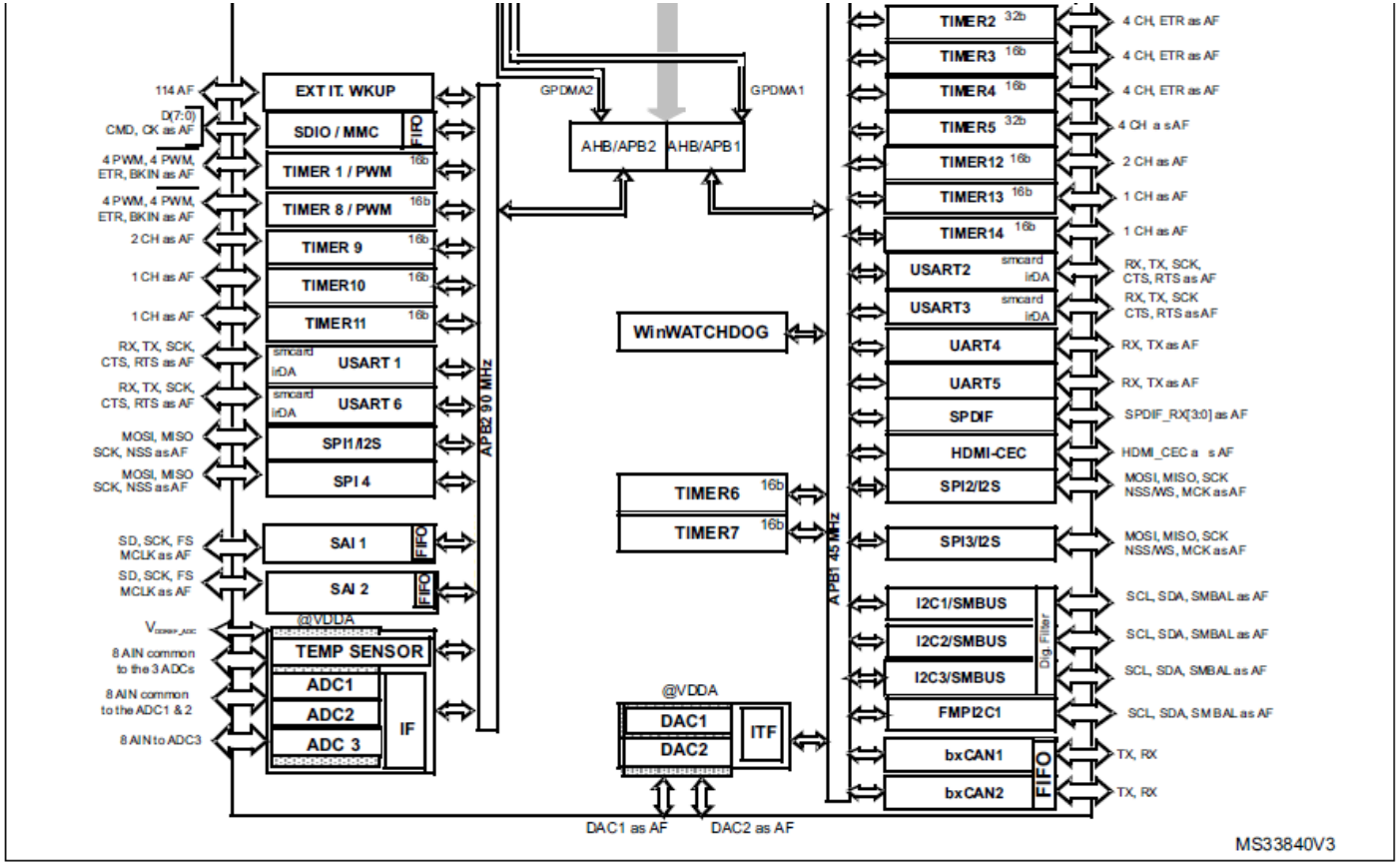
MCU Bus interface

- To understand Bus interface let us take two microcontrollers for better understanding
- **STM32F446xC/E and Tiva™ TM4C123GH6PM**

STM32F446xC/E

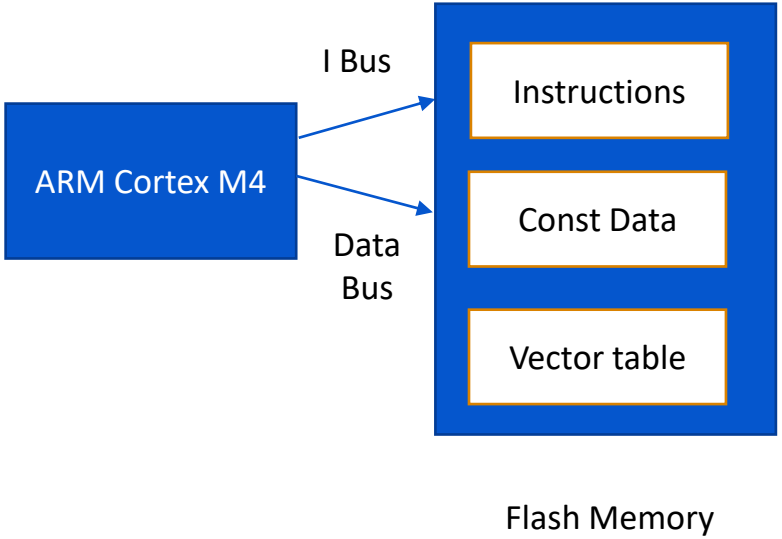
Figure 3. STM32F446xC/E block diagram




114 AF
D(7:0)
CMD, CK as AF
4 PWM, 4 PWM,
ETR, BKIN as AF
4 PWM, 4 PWM,
ETR, BKIN as AF
2 CH as AF
1 CH as AF
1 CH as AF
RX, TX, SCK,
CTS, RTS as AF
RX, TX, SCK,
CTS, RTS as AF
MOSI, MISO
SCK, NSS as AF
MOSI, MISO
SCK, NSS as AF
SD, SCK, FS
MCLK as AF
SD, SCK, FS
MCLK as AF
8 AIN common
to the 3 ADCs
8 AIN common
to the ADC1 & 2
8 AIN to ADC3
EXT IT. WKUP
SDIO / MMC
FIFO
TIMER 1 / PWM
16b
TIMER 8 / PWM
16b
TIMER 9
16b
TIMER10
16b
TIMER11
16b
smcard
irDA
USART 1
smcard
irDA
USART 6
SPI1/I2S
SPI 4
SAI 1
FIFO
SAI 2
FIFO
@VDDA
TEMP SENSOR
ADC1
ADC2
ADC 3
IF
APB2 90 MHz
GPDMA2
GPDMA1
AHB/APB2
AHB/APB1
WinWATCHDOG
TIMER6
16b
TIMER7
16b
@VDDA
DAC1
DAC2
ITF
DAC1 as AF
DAC2 as AF
APB1 45 MHz
TIMER2
32b
TIMER3
16b
TIMER4
16b
TIMER5
32b
TIMER12
16b
TIMER13
16b
TIMER14
16b
USART2
smcard
irDA
USART3
smcard
irDA
UART4
UART5
SPDIF
HDMI-CEC
SPI2/I2S
SPI3/I2S
I2C1/SMBUS
I2C2/SMBUS
I2C3/SMBUS
Dig. Filter
FMPI2C1
bxCAN1
bxCAN2
FIFO
4 CH, ETR as AF
4 CH, ETR as AF
4 CH, ETR as AF
4 CH as AF
2 CH as AF
1 CH as AF
1 CH as AF
RX, TX, SCK,
CTS, RTS as AF
RX, TX, SCK
CTS, RTS as AF
RX, TX as AF
RX, TX as AF
SPDIF_RX[3:0] as AF
HDMI_CEC as AF
MOSI, MISO, SCK
NSS/WS, MCK as AF
MOSI, MISO, SCK
NSS/WS, MCK as AF
SCL, SDA, SMBAL as AF
SCL, SDA, SMBAL as AF
SCL, SDA, SMBAL as AF
SCL, SDA, SMBAL as AF
TX, RX
TX, RX
MS33840V3

STM32F446xC/E

- I Bus (Instruction)
- D Bus (Data)
- S Bus (System)

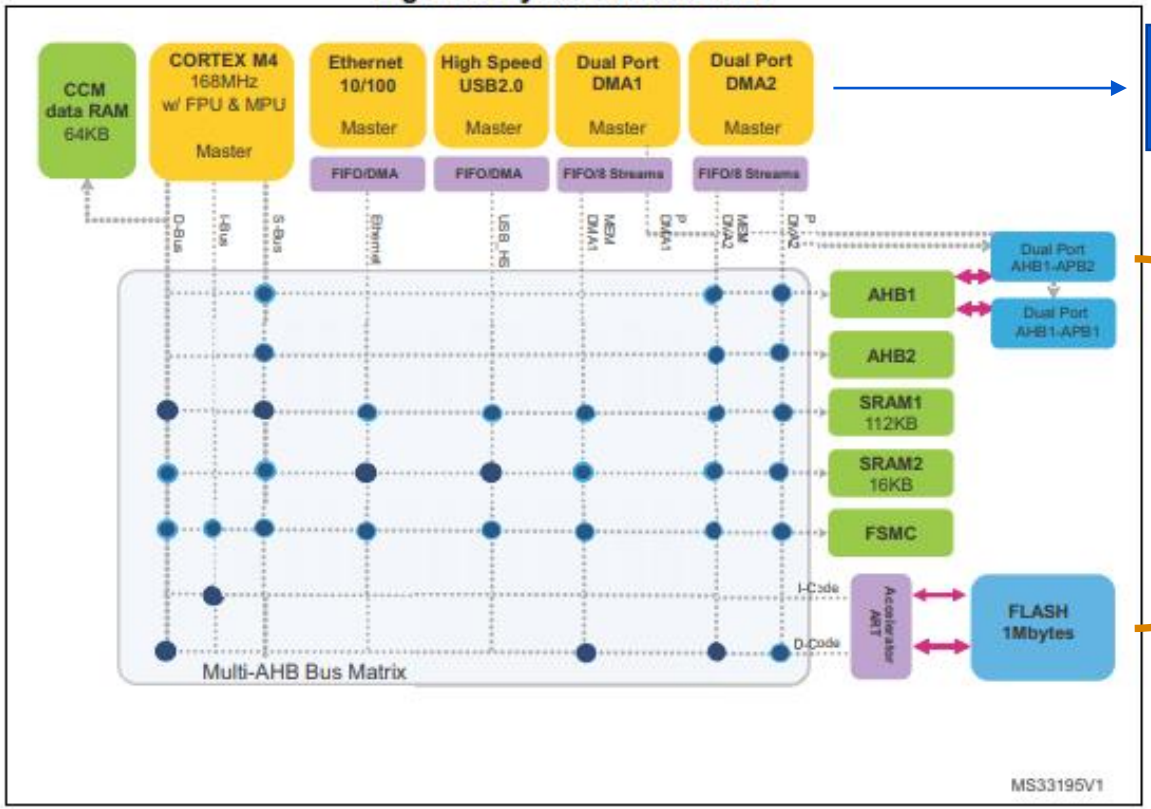


Demo Code

```
#include <string.h>
#include <stdint.h>

// This is a string literal (COnstant data). This will store in the ROM (Flash Memory)
const char *pMessage = "Hi!! i am Vivek" ;
// This is a COnstant data. This will store in the ROM (Flash Memory)
const int value = 200;

// This is a COnstant data. This will store in the SRAM (Static RAM)
char myData[50];

int main(void)
{
    for (uint32_t i=0; i<strlen(pMessage); i++);
    //This is a data copy from Flash to SRAM
    myData[i]=*(pMessage+i);
}
for(;;);
}
```

MCU Bus Matrix

Figure 7. System architecture



Masters

Slaves

MCU Clocks

Three different clock sources can be used to drive the system clock (SYSCLK):

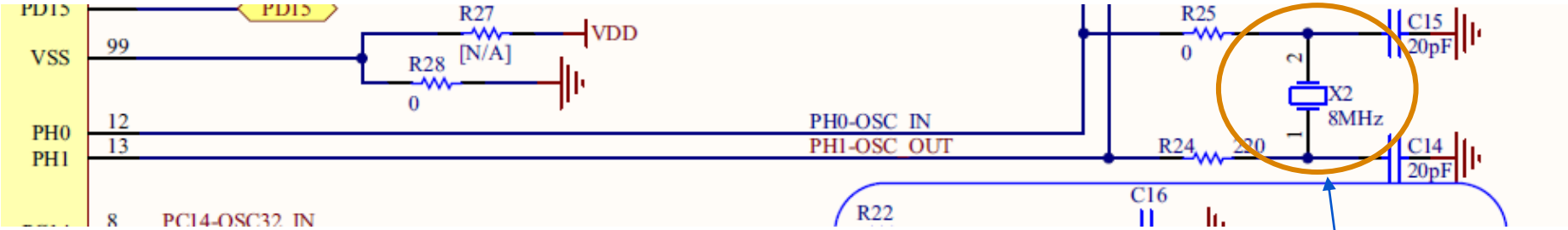
- HSI (High Speed Internal) oscillator clock or it is known as RC oscillator.
- HSE (High Speed external) oscillator clock or crystal oscillator.
- Main PLL (PLL) clock

The devices have the two following secondary clock sources:

- 32 kHz low-speed internal RC (LSI RC) which drives the independent watchdog and, optionally, the RTC used for Auto-wakeup from the Stop/Standby mode.
- 32.768 kHz low-speed external crystal (LSE crystal) which optionally drives the RTC clock (RTCCLK)

• HSE (High Speed external) oscillator clock or crystal oscillator

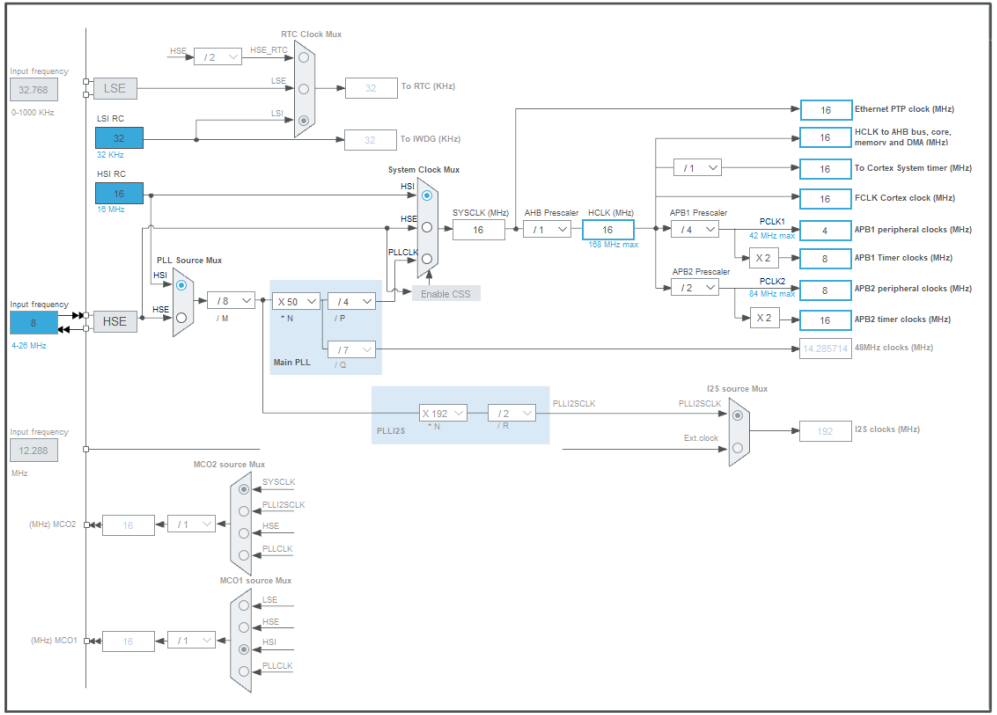
- It is in-built in the STM disc board. See the figure below
- In Nucleo board its not available and the external clock is derived from ST-Link circuitry.



Picture taken from reference manual of microcontroller

HSI (High Speed Internal) oscillator clock or it is known as RC oscillator

- The HSI clock signal is generated from an internal 16 MHz RC oscillator and can be used directly as a system clock, or used as PLL input.
- Also note, that when MCU goes on reset then it uses HSI as its default clock source.



Screenshot from STMCubeIde

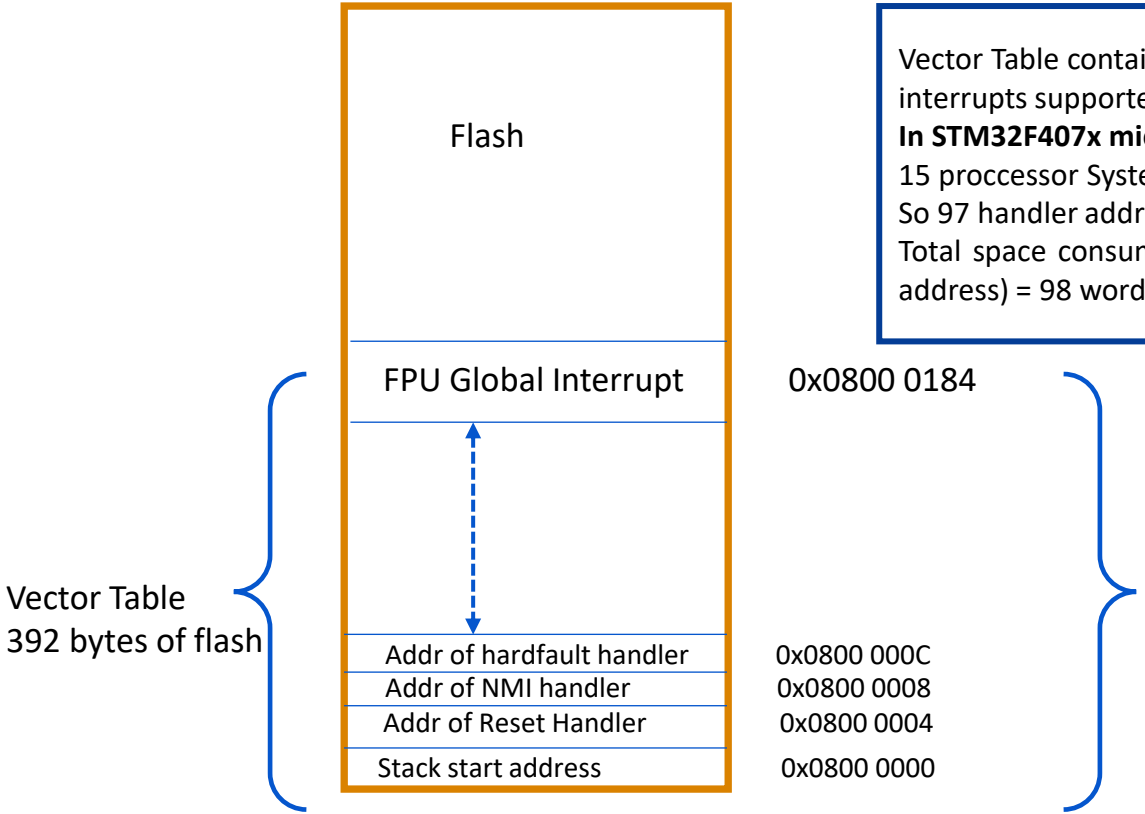
Peripheral clock configuration

- In Modern MCUs, before using any peripheral, you must enable its peripheral clock using peripheral clock registers
- By Default, peripheral clocks of all most all peripheral will be disabled to save power
- A peripheral won’t take or respond to your configuration values until you enable its peripheral clock
- In STM32 microcontrollers, peripheral clocks are managed through RCC registers.

MCU Vector table

- Vector table is a table of addresses or pointers (addresses of exception handlers)
- Exceptions are system exceptions and interrupts.
- We have 15 system exceptions and 240 interrupts.
- For more details lets explore the vector table in reference manual

Vector Table Placement in Flash Memory



Vector Table contains ISR addresses of all system exceptions and interrupts supported by the microcontroller.

In STM32F407x microcontroller:

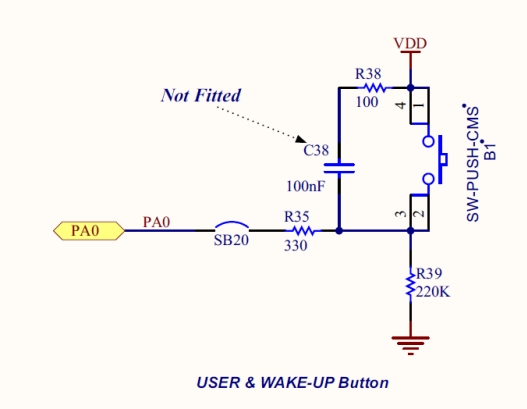
15 proccessor System exception +82 interrupts(IRQs)= 97

So 97 handler addresses must be stored in VT.

Total space consumed by the VT = 97 +1 (to store initial stack address) = 98 words = 392 bytes

Understanding MCU interrupt design, NVIC, Interrupt handling

- Let us consider the reference manual of ST and STM board datasheet
- Important points: Some peripherals deliver their interrupt to NVIC directly and some through EXTI line.



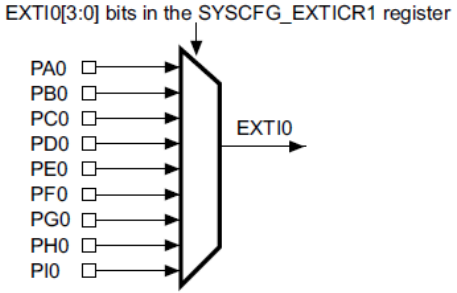
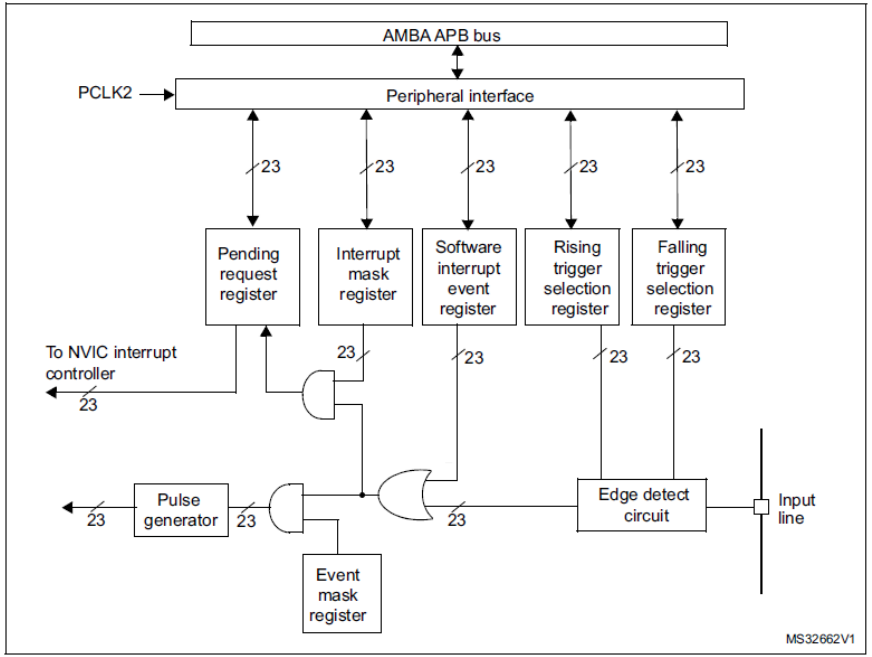


Figure 41. External interrupt/event controller block diagram

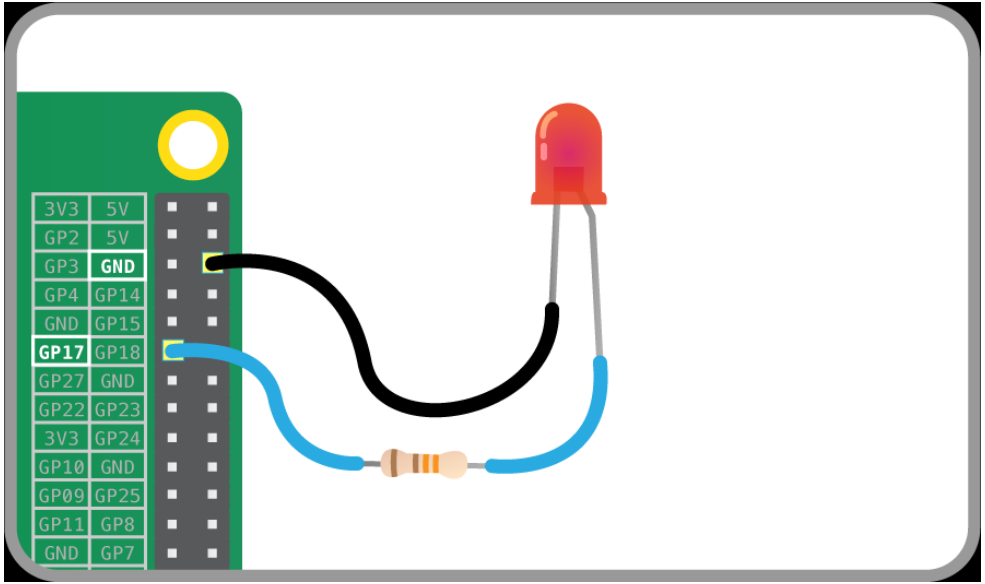


Button Interrupt

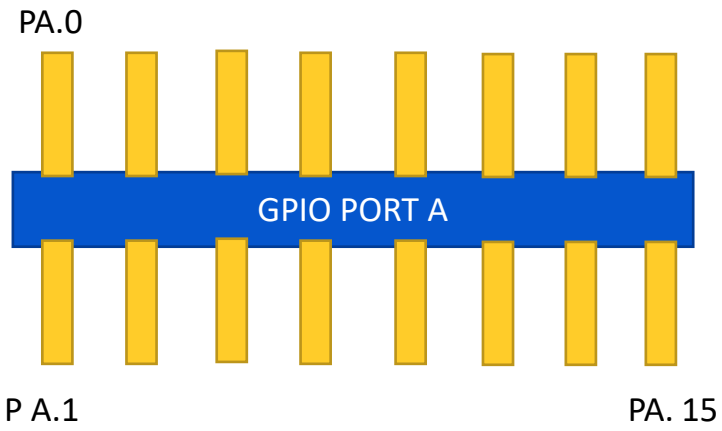
- How does a button issue interrupt to the processor in STM 32?
  - The Button is connected to a GPIO pin of the microcontroller
  - The GPIO pin should be configured to input mode
  - The link between a GPIO port and the relevant EXTI line must be established using the SYSCFG_EXTICRx register.
  - Configure the trigger detection (falling/rising/both) for relevant EXTI line (This is done via EXTI controller register)
  - Implement the handler to service the interrupt

GPIO Pin and GPIO port

- General purpose input output
- Used for reading digital signal
- Generating triggers for external components
- Issuing interrupts
- Waking up of processors



Must know things about GPIOs

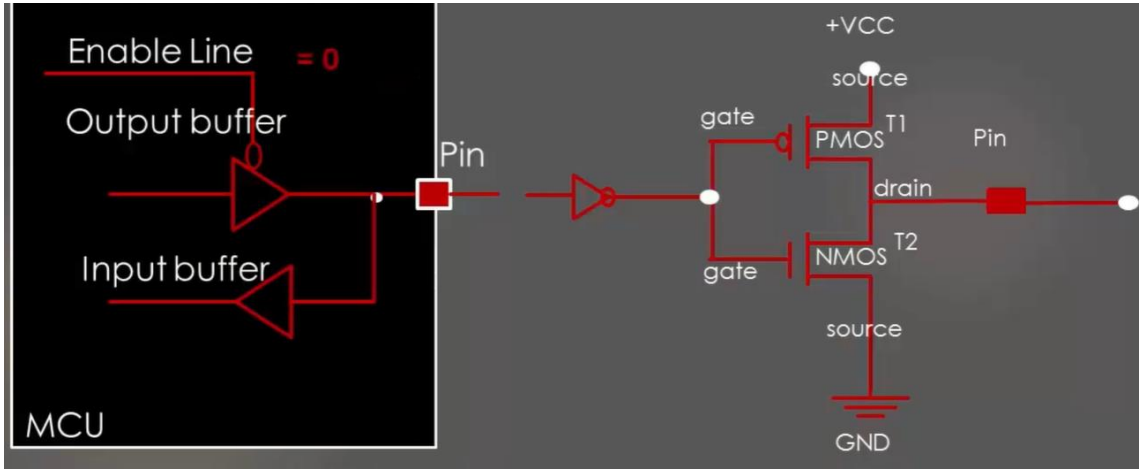


Port ‘A’ having many pins

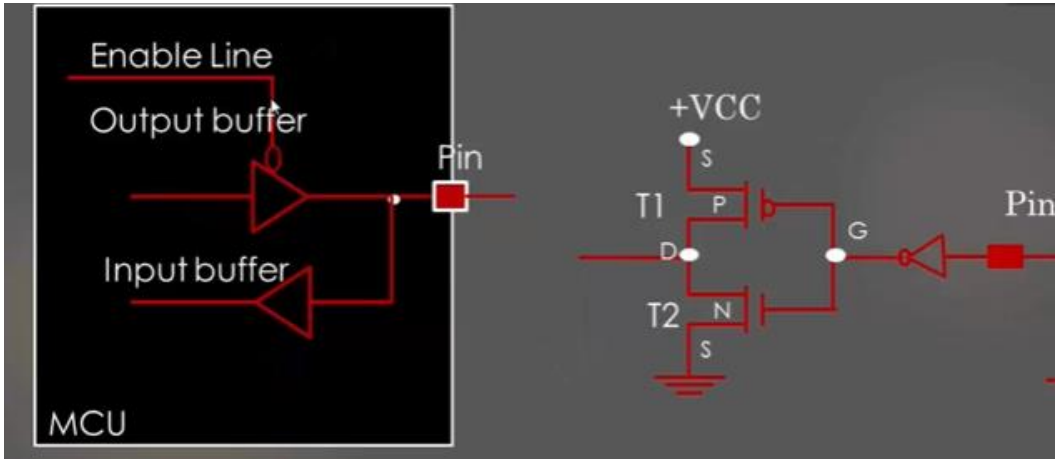
Boat parking in port Venice, Italy

Must know things about GPIOs

- How does a GPIO pin is actually implemented inside the MCU?
- If we give enable pin = 0, then output buffer will get activated and if enable pin =1 then Input buffer get activated.
- See the image below, If we give 1 to enable pin the transistors are fed with logic 0 and the PMOS activated and NMOS deactivated and pin will be pulled to logic 1 and vice versa.



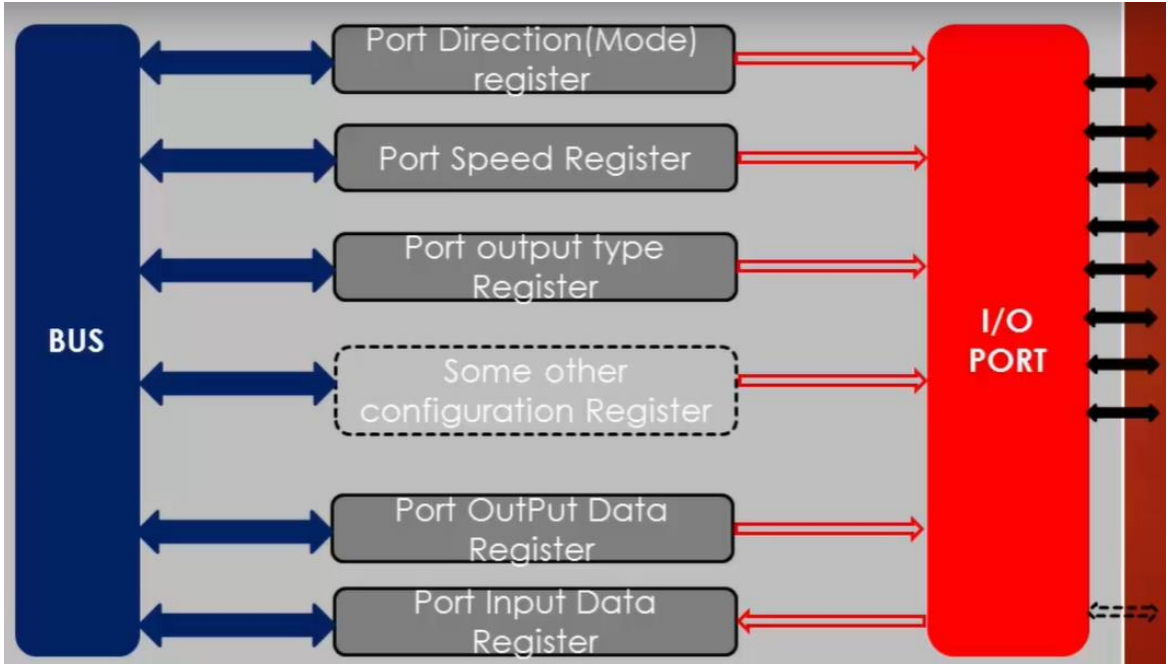
- If we give enable pin = 1, then input buffer get activated.
- See the image below, If we pin is driven to high voltage then T1 is on and T2 is off and we get high at point D and vice versa.



Note: By default all the pins are in floating state (with High Impedance (Hi-Z) when the MCU got power.

GPIO Register and programming structure

- Minimum set of registers available in microcontrollers.
- Lets explore reference manual
- In STM32F407VG chip we have GPIOA,B,C,D,E,F,G,H,I having total of 144 GPIO pins.
- Whereas in Discovery board, we have GPIOA,B,C,D,E having 80 GPIO pins.



GPIO Port Mode register

- GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)
- Each GPIOA to I is of 32 bit.
- Before using any GPIO, you must decide its MODE. Whether, you want to use it as an input, output or Analog purpose
- Refer reference manual

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

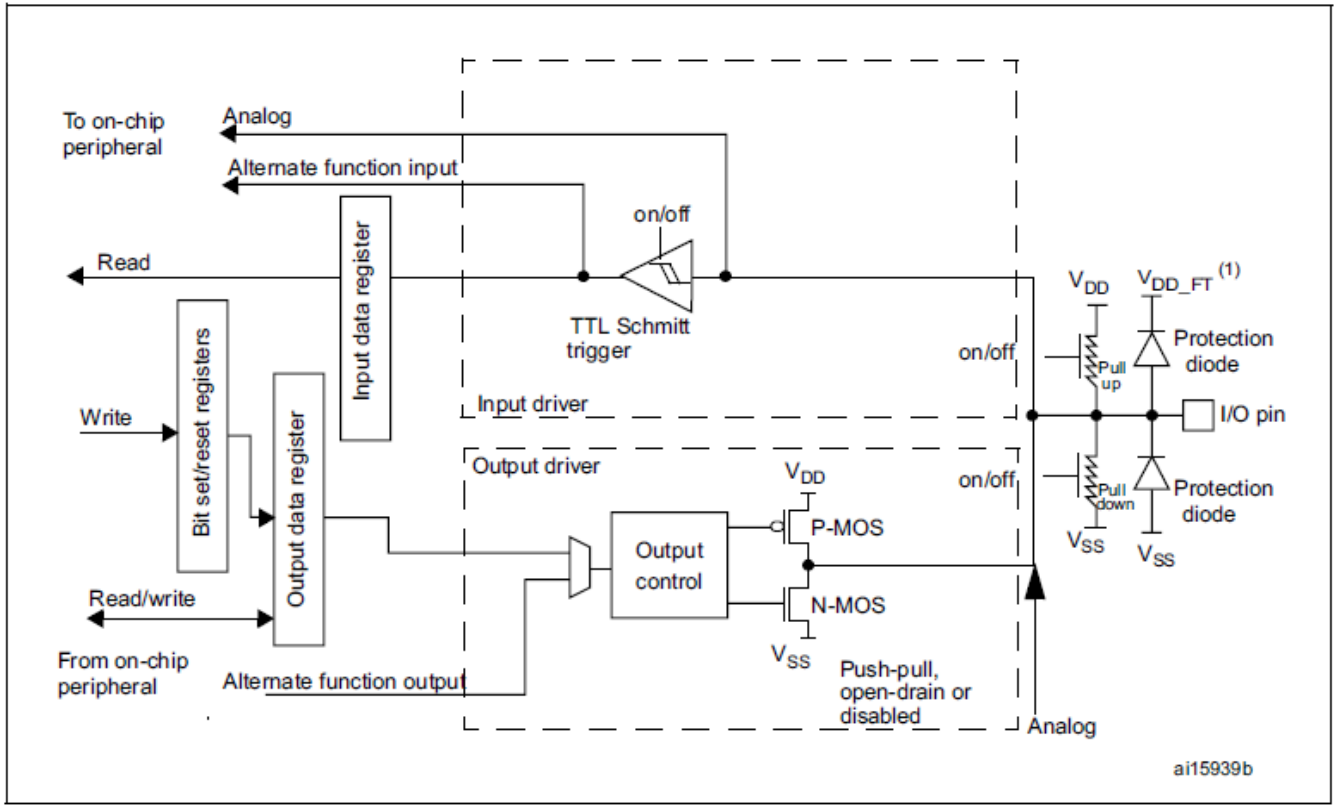
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Different modes of Pins

Input / Output configuration

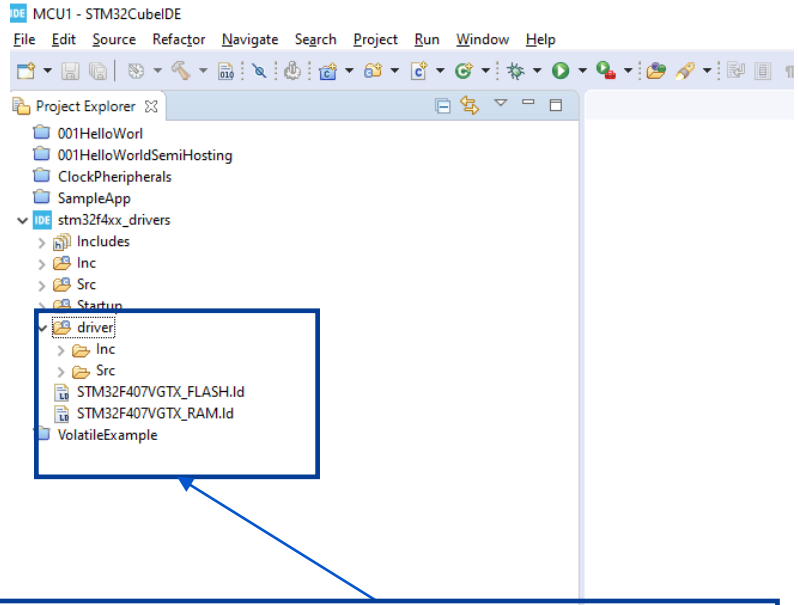


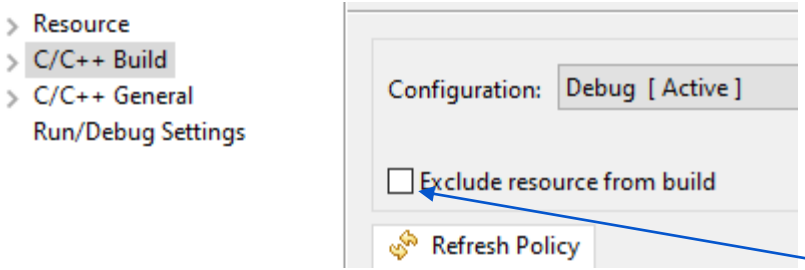
Device header file contents

- This is a header file (‘C’ in our case) which contain microcontroller specific details such as:
  1. The base addresses of various memories present in the microcontroller such as (Flash, SRAM 1, SRAM 2, ROM, etc.)
  2. The base addresses of various bus domains such as AHBx domain, APBx domain)
  3. Base addresses of various peripherals present in different bus domains of the microcontroller
  4. Clock management macros (i.e. clock enable and clock disable macros)
  5. IRQ definitions
  6. Peripheral register definition structures
  7. Peripheral register bit definitions
  8. Other useful microcontroller configuration macros.

Creating Project for driver development

After Creating project create header file STM32F407XX under Inc folder.



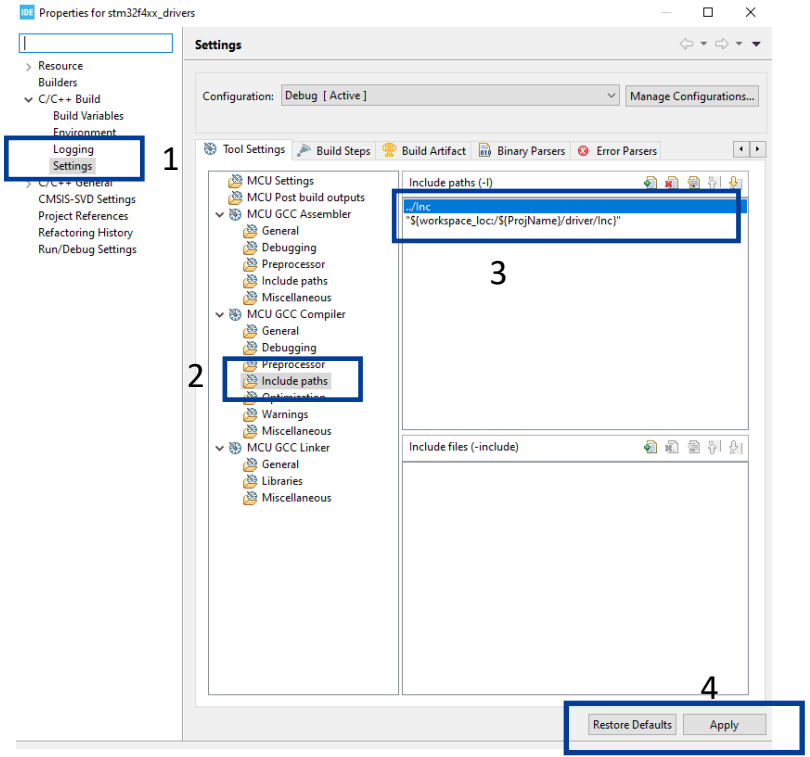


Create a project and designed a driver folder with Inc and Src sub folder. Remember to uncheck executable by going to the properties of the driver folder and then go to C/C++ build and uncheck " exclude resource from build

Include paths in the application layer

- Go the main.c of application layer and add the syntax **#include "stm32f407xx.h“**
- Right click on the project and go to properties.
- Click on C/C++ build.
- Click on settings > include paths >

>add workspace> apply and close.

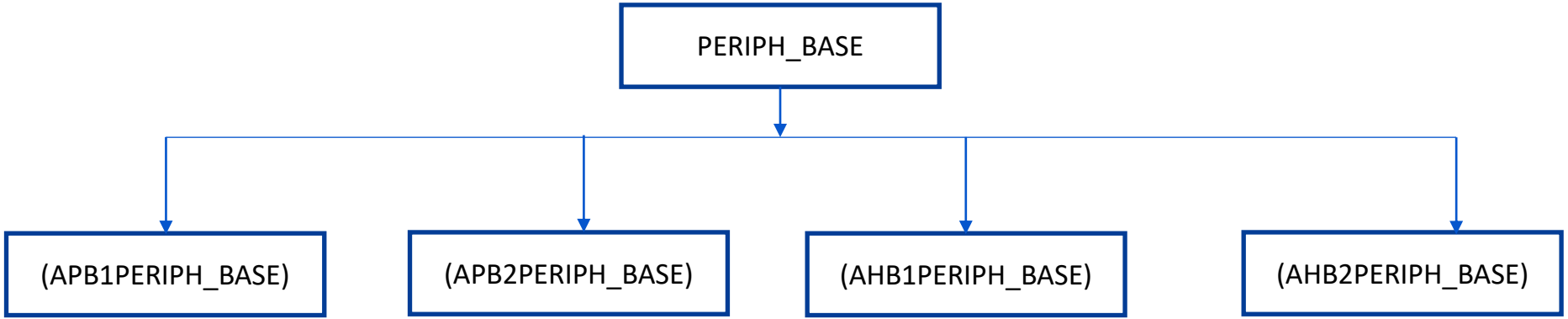


Updating MCU specific header file with Bus domain and Peripheral

- Adding base addresses of FLASH, SRAM, ROM etc.

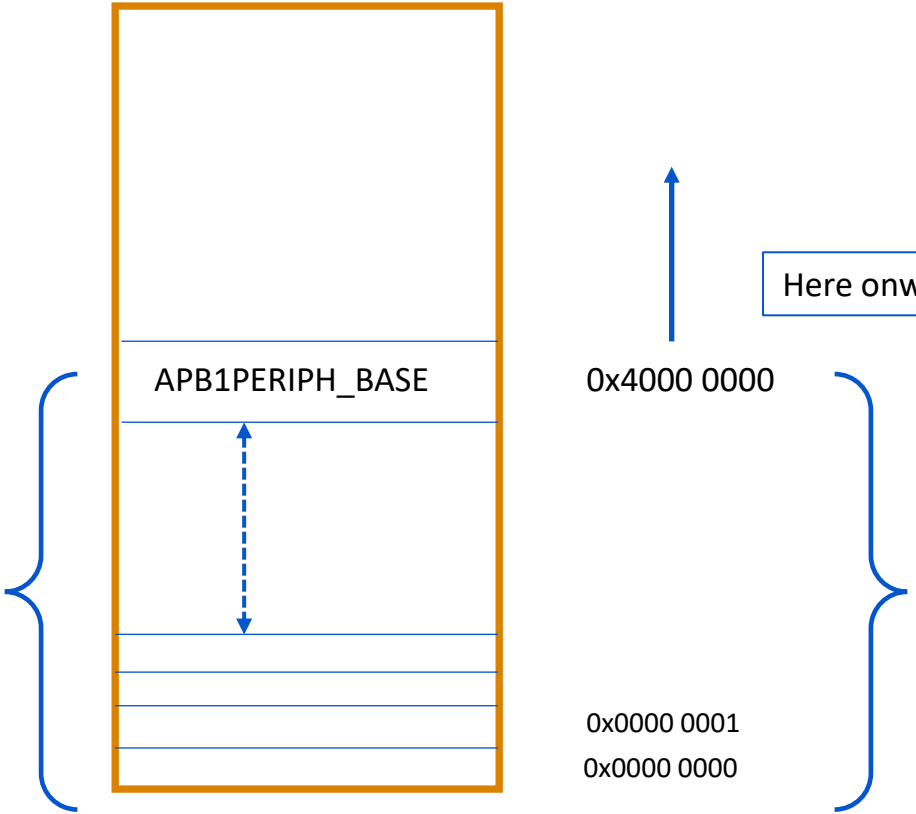
```
7
8  #ifndef INC_STM32F407XX_H_
9  #define INC_STM32F407XX_H_
10
11
12 #define FLASH_BASEADDR        0x08000000U
13 #define SRAM1_BASEADDR        0x20000000U   //112KB
14 #define SRAM2_BASEADDR        0x2001C000U
15 #define ROM_BASEADDR          0x1FFF0000U
16 #define SRAM                  SRAM1_BASEADDR
17
18
19
20
21
22 #endif /* INC_STM32F407XX_H_ */
23
```

Peripheral base addresses of different bus domains



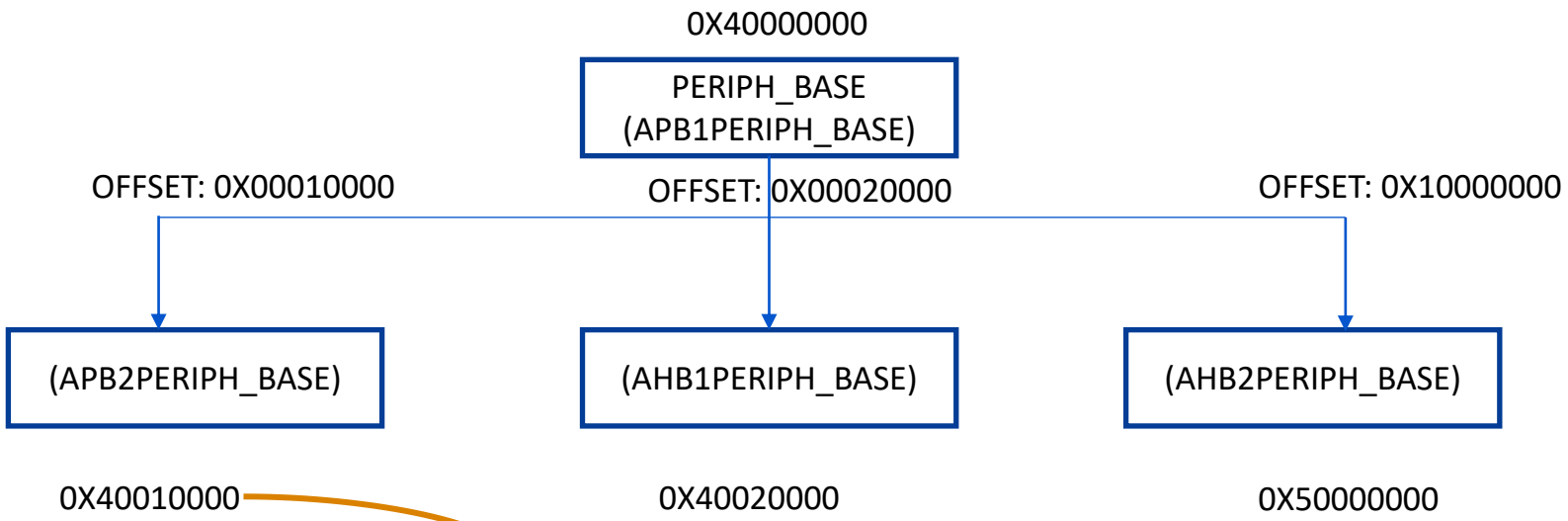
1. Different Peripherals are hanging on different busses
2. AHB bus is used for those peripherals which need high speed data communication (ex. Camera, GPIOs)
3. APB bus is used for those peripherals which need low speed data communication.

Memory Map of MCU



Here onwards peripheral register addresses start

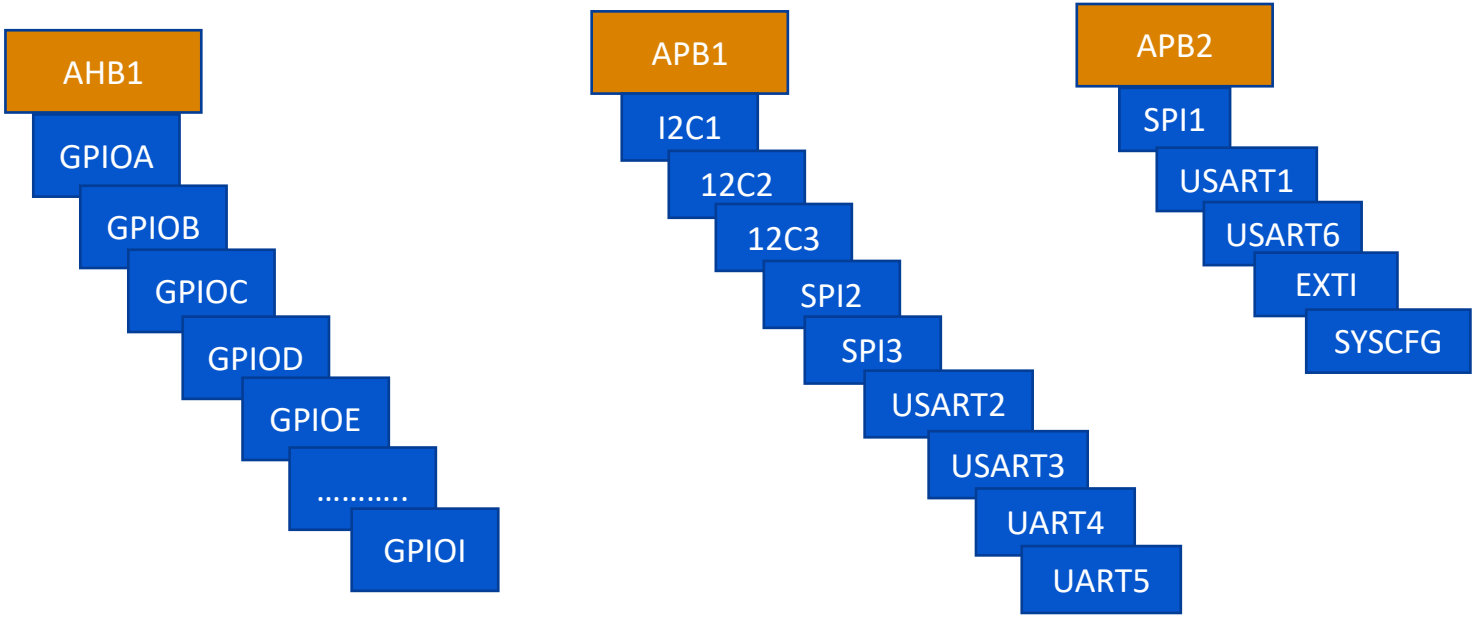
Peripheral base addresses of different bus domains in stm32f407xx MCU



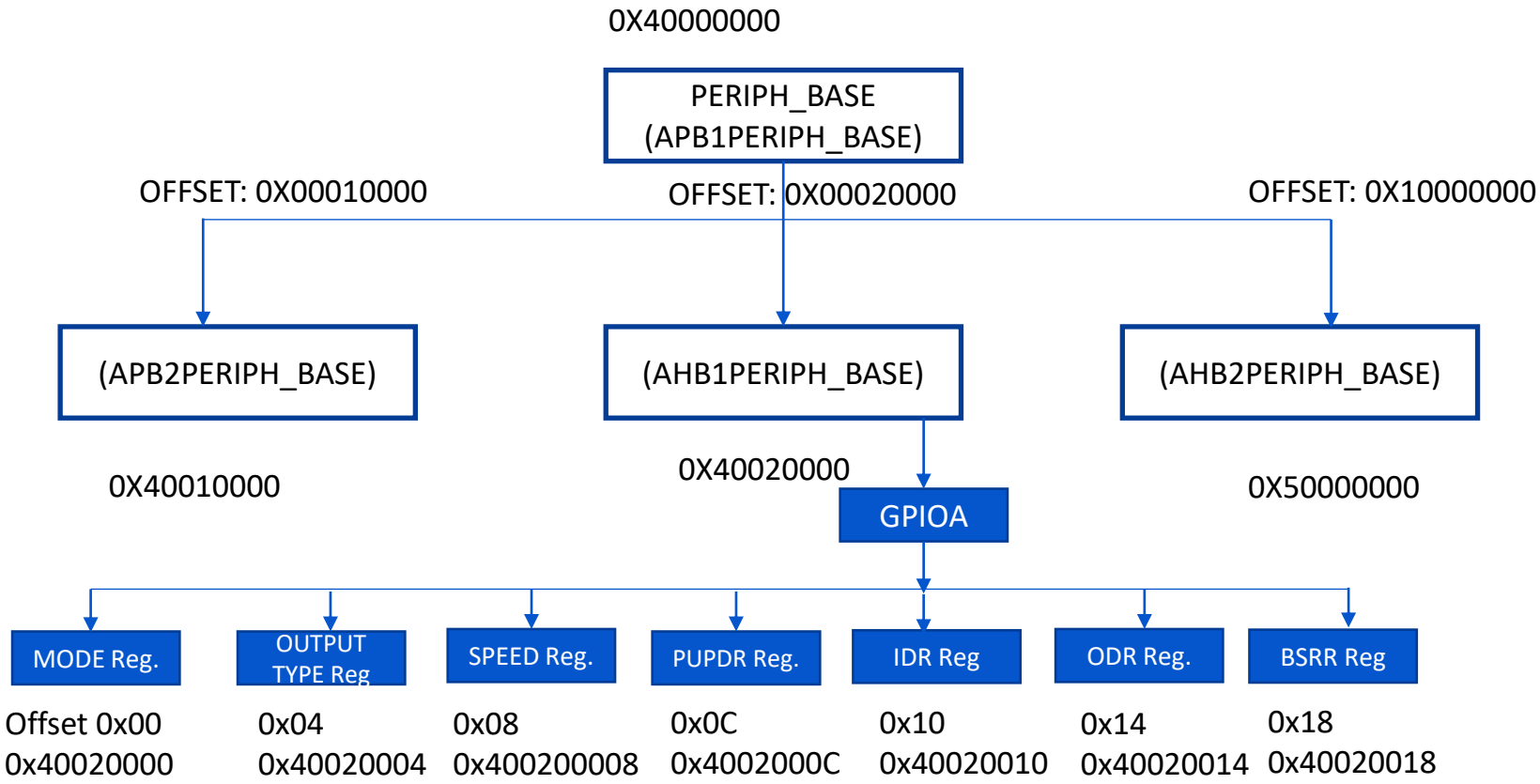
This Means that registers of peripherals hanging on APB2 bus will appear on address 0x40010000 onwards in the memory map of the MCU

```
/********* AHBx and APBx Bus peripherals base address**********/

#define PERIPH_BASE          0x40000000U
#define APB1PERIPH_BASE      PERIPH_BASE
#define APB2PERIPH_BASE      0X40010000U
#define AHB1PERIPH_BASE      0X40020000U
#define AHB2PERIPH_BASE      0X50000000U
```

Defining Base Addresses of peripherals in AHB x and APBx



Structural Peripheral register



Structural Peripheral register

Address of MODER variable(&MODER) will be (0x40020000+0x00)

Address of OTYPER variable(&OTYPER) will be (0x40020000+0x04) and so on

```
 */
typedef struct
{
   uint32_t MODER;      /*!< Give a short description,          Address offset: 0x00      */
   uint32_t OTYPER;     /*!< TODO,                              Address offset: 0x04      */
   uint32_t OSPEEDR;    /*!< TODO,                              Address offset: 0x08      */
   uint32_t PUPDR;      /*!< TODO,                              Address offset: 0x0C      */
   uint32_t IDR;        /*!< TODO,                              Address offset: 0x10      */
   uint32_t ODR;        /*!< TODO,                              Address offset: 0x14      */
   uint32_t BSRRL;      /*!< TODO,                              Address offset: 0x18      */
   uint32_t BSRRH;      /*!< TODO,                              Address offset: 0x1A      */
   uint32_t LCKR;       /*!< TODO,                              Address offset: 0x1C      */
   uint32_t AFR[2];     /*!< TODO,                              Address offset: 0x20-0x24 */
} GPIO_RegDef_t;

GPIO_RegDef_t *pGPIOA = (GPIO_RegDef_t*)0x40020000;
```

Base address of GPIOA

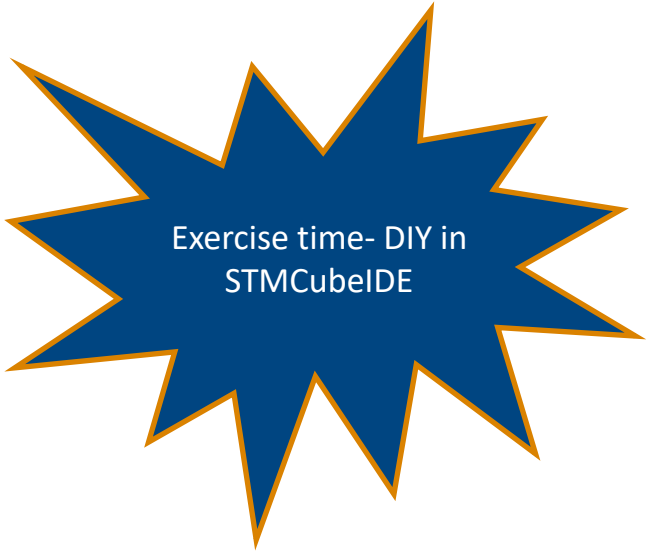
```
//Helps programmers for easy access to various registers of the peripherals
pGPIOA->MODER = 25; //storing value 25 in to MODER register
*(0x40020000+0x00) = 25; //this is how compiler does
pGPIOA->ODR = 44;   //storing value 44 in to OD register
*(0x40020000+0x14) = 44; //this is how compiler does
```

GPIO_RegDef_t *pGPIOA = (GPIO_RegDef_t*)GPIOA_BASEADDR

Can be replaced by a macros like #define GPIOA ((GPIO_RegDef_t*)GPIOA_BASEADDR)

Peripheral Clock enable and disable Macros

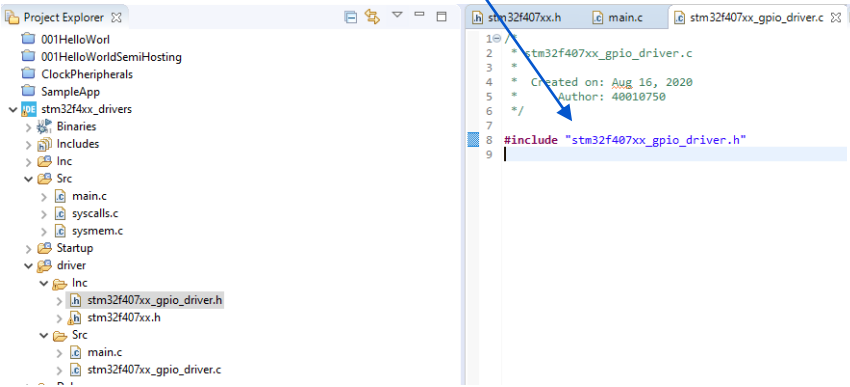
- Create the structure definition of RCC too as a same manner you have done for GPIO structure definition.
- Define Base Address if RCC
- Now define the clock enable Macros for GPIOx, I2Cx, SPIx, USARTx as mentioned in the slide no 80

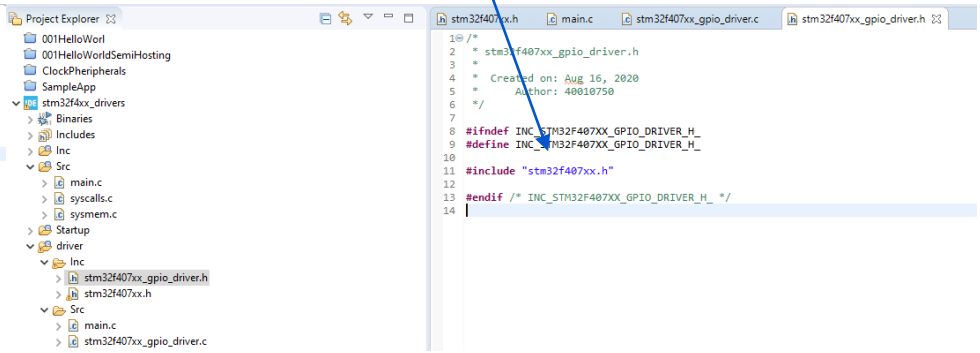


Creating GPIO driver header and source file

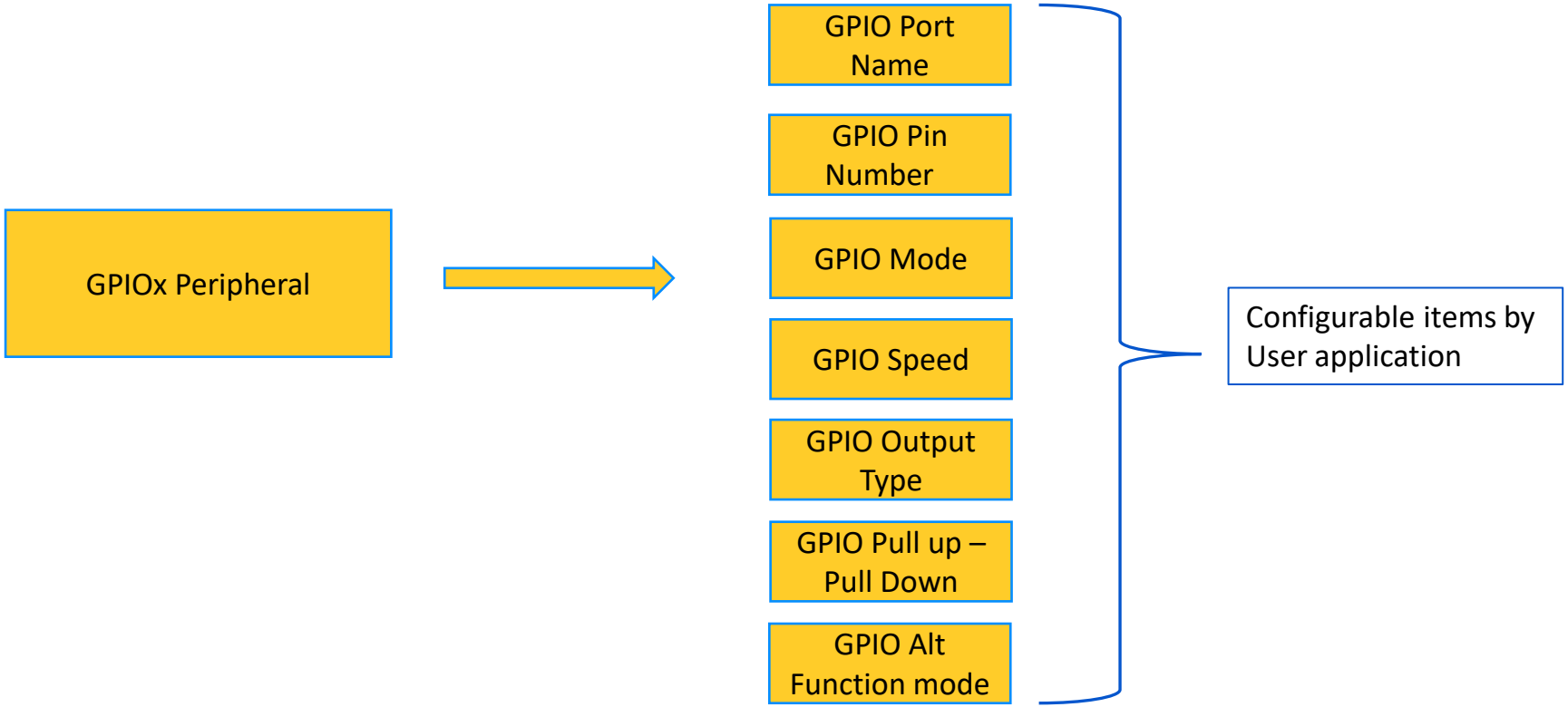
Source file should contain the driver header file

Header file should contain the system header file





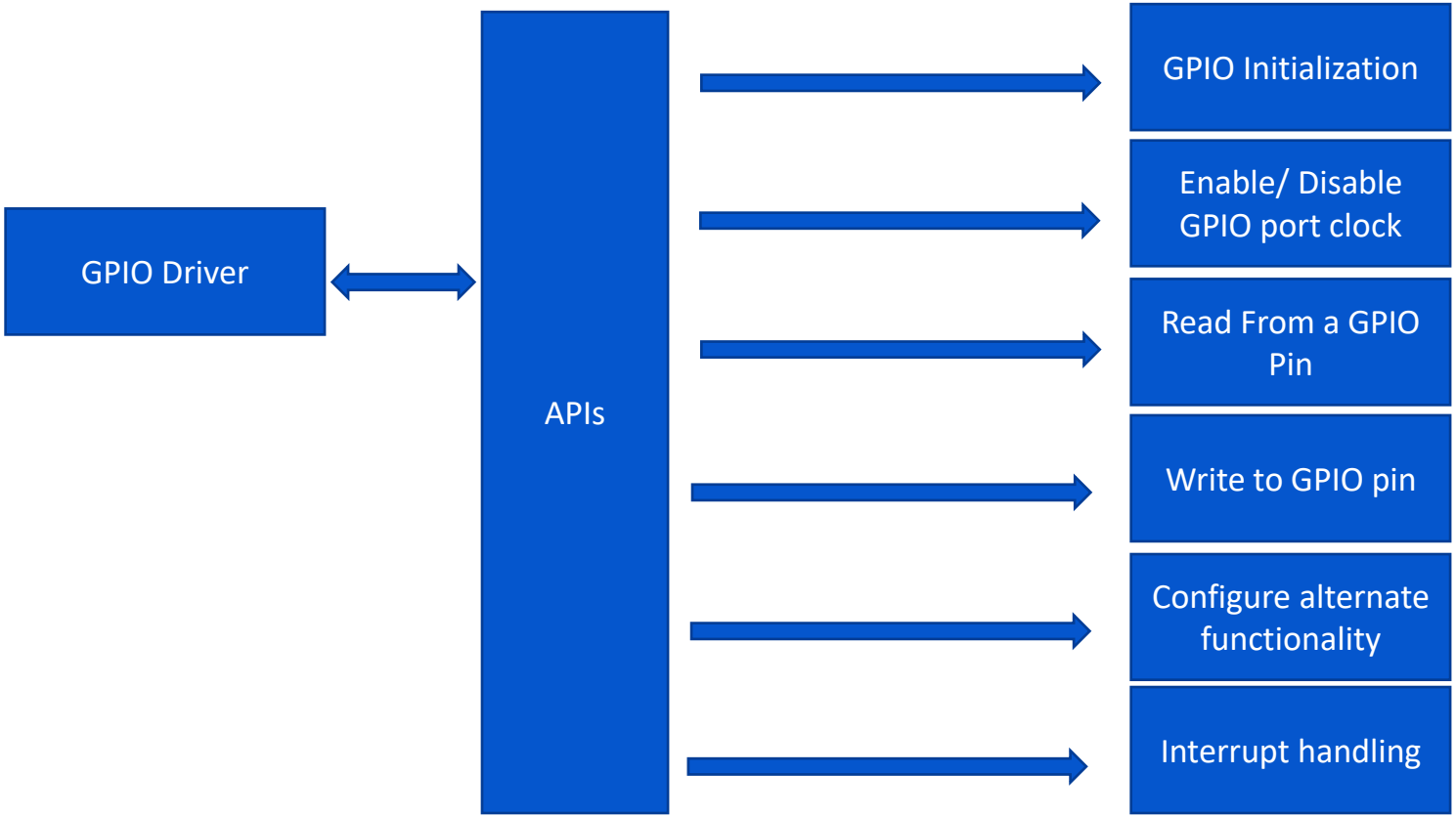
GPIO Handle and GPIO configuration structure



GPIO Handle and GPIO configuration structure

```
7
8  #ifndef INC_STM32F407XX_GPIO_DRIVER_H_
9  #define INC_STM32F407XX_GPIO_DRIVER_H_
10
11 #include "stm32f407xx.h"
12
13 typedef struct
14 {
15 uint8_t GPIO_PinNumber;
16 uint8_t GPIO_PinMode;
17 uint8_t GPIO_PinSpeed;
18 uint8_t GPIO_PinPuPdControl;
19 uint8_t GPIO_PinOPType;
20 uint8_t GPIO_PinAltFunMode;
21 }GPIO_PinConfig_t;
22
23 /**********Handle structure for a GPIO pin**************/
24
25 typedef struct
26 {
27
28     GPIO_RegDef_t*pGPIOx;      //This hold the base address of the GPIO port to which the pin belongs
29     GPIO_PinConfig_t GPIO_PinConfig;   //this holds GPIO pin configuration setting
30
31 }GPIO_Handle_t;
32
33 #endif /* INC_STM32F407XX_GPIO_DRIVER_H_ */
34
```

GPIO Driver APIs requirement



GPIO Driver APIs requirement

```c
 * Peripheral clock setup
 */
void GPIO_PeriClockControl(void);

/*
 * Init and DeInit
 */
void GPIO_Init (void);
void GPIO_DeInit (void);

/*
 * Data read and write
 */
void GPIO_ReadFromInputPin(void);
void GPIO_ReadFromInputPort(void);
void GPIO_WriteToOutputPin(void);
void PIO_WriteToOutputPort(void);
void GPIO_ToggleOutputPin(void);


/*
 * IRQ configuration and ISR handling
 */
void GPIO_IRQConfig(void);
void GPIO_IRQHandling(void);
```

```c
 * Peripheral clock setup
 */
void GPIO_PeriClockControl(GPIO_RegDef_t *pGPIOx, uint8_t EnorDi );

/*
 * Init and DeInit
 */
void GPIO_Init (GPIO_Handle_t *pGPIOHandle);
void GPIO_DeInit (GPIO_RegDef_t *pGPIOx);

/*
 * Data read and write
 */
uint8_t GPIO_ReadFromInputPin(GPIO_RegDef_t *pGPIOx, uint8_t PinNumber);
uint16_t GPIO_ReadFromInputPort(GPIO_RegDef_t *pGPIOx);
void GPIO_WriteToOutputPin(GPIO_RegDef_t *pGPIOx, uint8_t PinNumber, uint8_t Value);
void PIO_WriteToOutputPort(GPIO_RegDef_t *pGPIOx, uint16_t Value);
void GPIO_ToggleOutputPin(GPIO_RegDef_t *pGPIOx, uint8_t PinNumber);


/*
 * IRQ configuration and ISR handling
 */
void GPIO_IRQConfig(uint8_t IRQNumber, uint8_t IRQPriority, uint8_t EnorDi);
void GPIO_IRQHandling(uint8_t PinNumber);
```

APIs function with and without return types

Implementation of the APIs- some Foundation rules

```
 * Peripheral clock setup
 */
/*****************************************************************
 * @fn                  - GPIO_PeriClockControl
 * -
 * @brief- This function enables or disables peripheral clock for the given GPIO port
 *
 * @param[in]- Base address of the GPIO peripheral
 * @param[in]- Enable or Disable macros
 * @param[in]-
 *
 * @return              - none
 *
 * @Note- none
 */
void GPIO_PeriClockControl(GPIO_RegDef_t *pGPIOx, uint8_t EnorDi )
{


}

/*
```

Configuring Alternate function register

PIN 7 PIN 6 PIN 5 PIN 4

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	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
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

PIN 3 PIN 2 PIN 1 PIN 0

PIN 15 PIN 14 PIN 13 PIN 12

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw	rw	rw	rw	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
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

PIN 11 PIN 10 PIN 9 PIN 8

Configuring Alternate function register

- Assuming user wants to configure Pin no 9.

Divide 9/8 = 1 (integer division) and store it in temporary register.

This temp register will point that which register we have to use either AFR [0] or AFR[1]. In our case we have to use AFR[1].

Now calculate to which bit position we have to store the value. For that assume one more variable temp2 (may be) and take a mode i.e. Mode 9%8 = 1

Then do like this : AFR [temp 1] = value << (4 * temp 2)

Now try it with Pin no .6

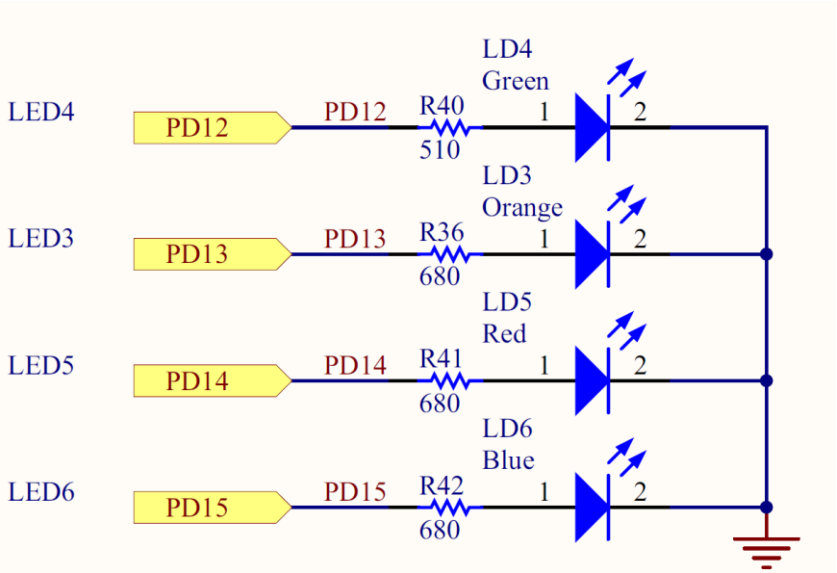
Temp 1 = 6/8 = 0

Temp 2 = 6%8 = 6

So AFR [temp 1] = value << (4 * temp 2) yields 24 bit position.

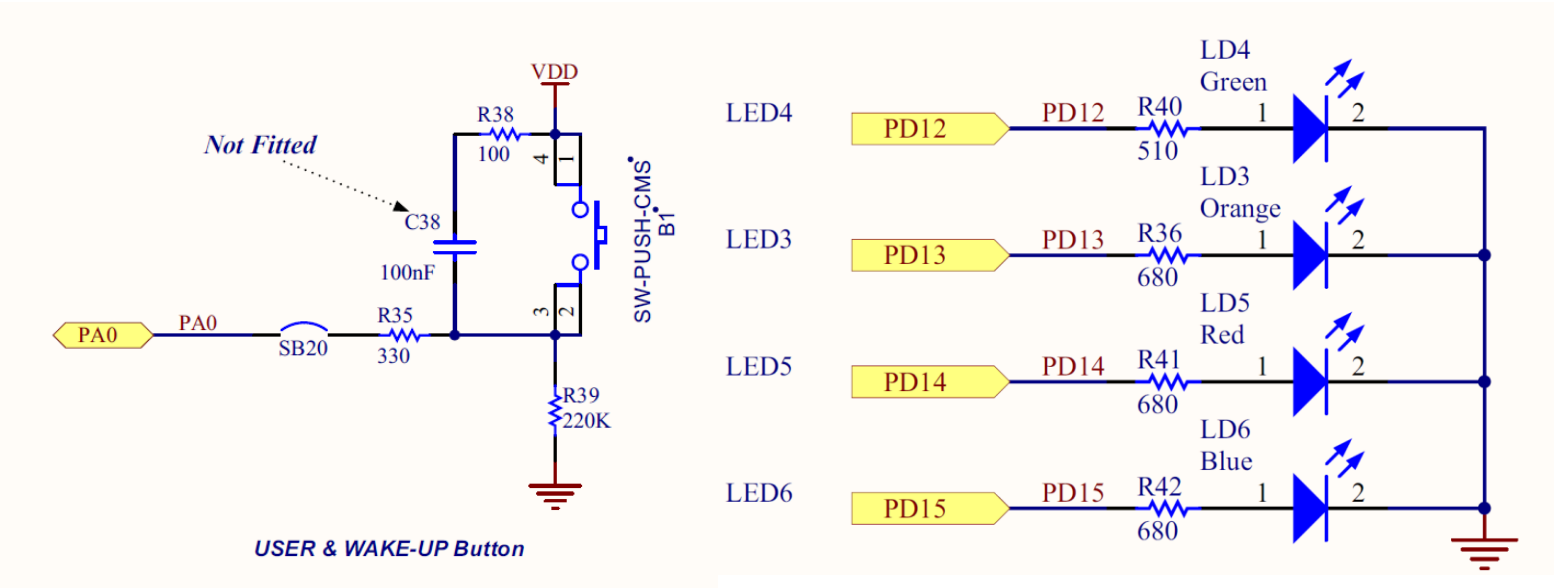
Exercise

- Write a program to toggle the on board LED with some delay
  - Case 1: Use push pull configuration for the output pin
  - Case 2: Use open drain configuration for the output pin



ON board 4 Led are present. To check connections please refer schematic

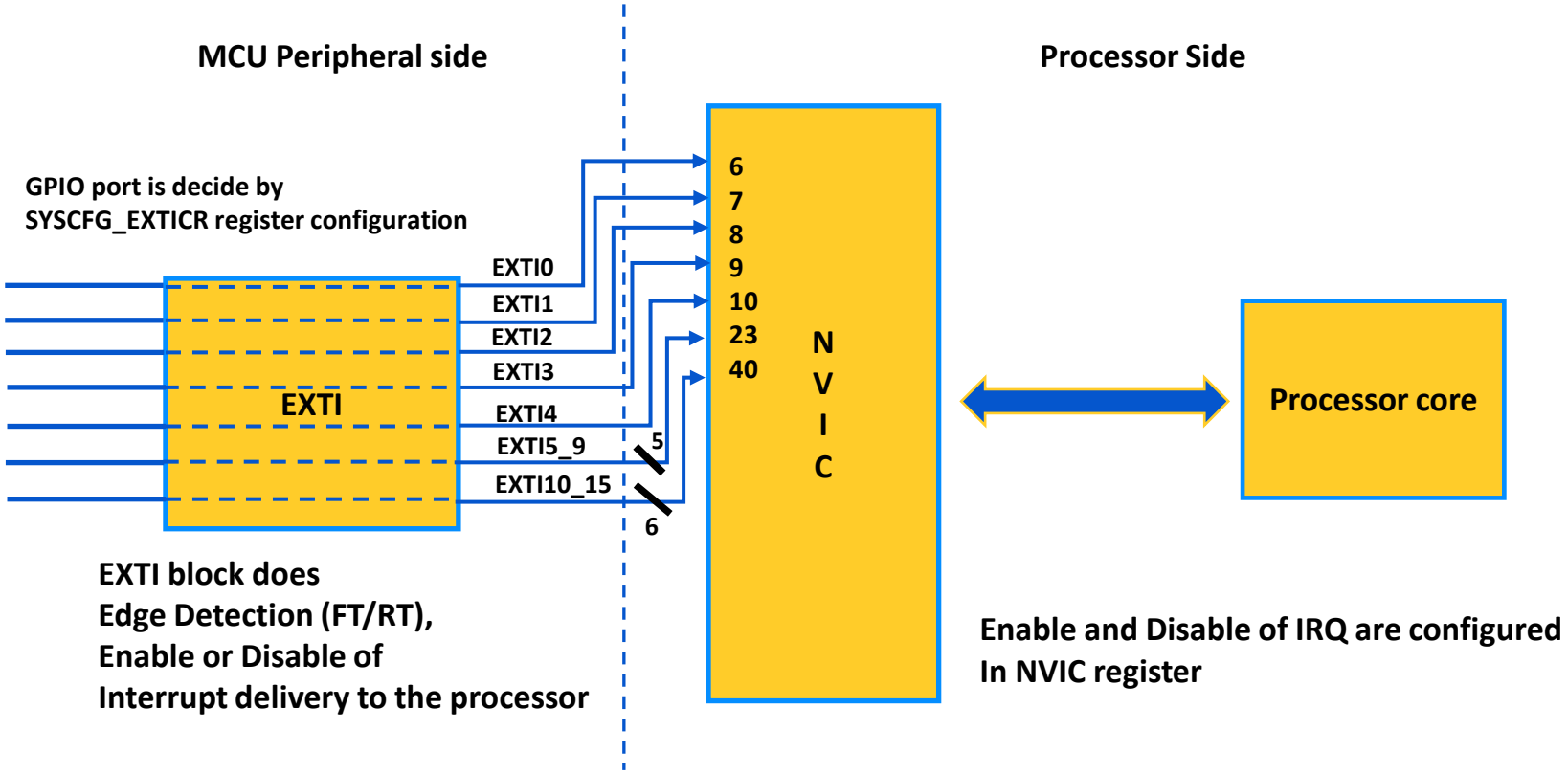
Handling on board led and Button



Exercise

- Connect and external button to PD5 pin and toggle the led whenever interrupt is triggered by the button press.
- Interrupt should be triggered during the falling edge of the button press

Stm32f4xx GPIO pins interrupt delivery to the processor



GPIO Pin Interrupt Configuration

- Pin must be in input configuration
- Configure the edge trigger (RT, FT, RFT)
- Enable interrupt delivery from peripheral to the processor (on peripheral side)
- Identify the IRQ number on which the processor accepts the interrupt from that pin.
- Configure the IRQ priority for the identified IRQ number (processor side)
- Enable interrupt reception on that IRQ number (Processor side)
- Implement IRQ handler

IRQ priority register

	31 … 24	23 … 16	15 … 8	7 … 0
IPR59	PRI_239	PRI_238	PRI_237	PRI_236
⋮			⋮	
IPR*n*	PRI_*4n+3*	PRI_*4n+2*	PRI_*4n+1*	PRI_*4n*
⋮			⋮	
IPR0	PRI_3	PRI_2	PRI_1	PRI_0

Exercise

- Connect an external button to PD5 pin and toggle the led whenever interrupt is triggered by the button press.
- Interrupt should be triggered during falling edge of the button press.

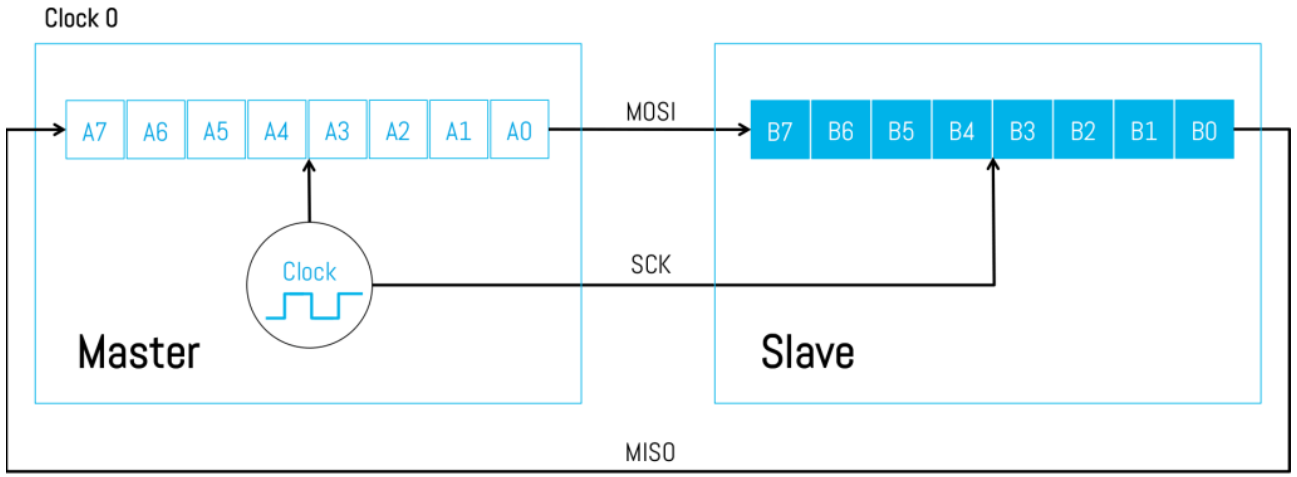
SPI Protocol

Serial Peripheral Interface (SPI)

- It is a synchronous serial data transfer protocol named by Motorola
- Here two or more serial devices are connected to each other in full-duplex mode. The devices connected to each other are either *Master* or *Slave*.
- The *Master* device is the one which initiates the connection and controls it. Once the connection is initiated, then the *Master* and one or more *Slave(s)* can transmit and/or receive data.
- As its full duplex mode of communication so *Master* can send data to *Slave(s)* and the *Slave(s)* can also send the data to the *Master* at the same time.

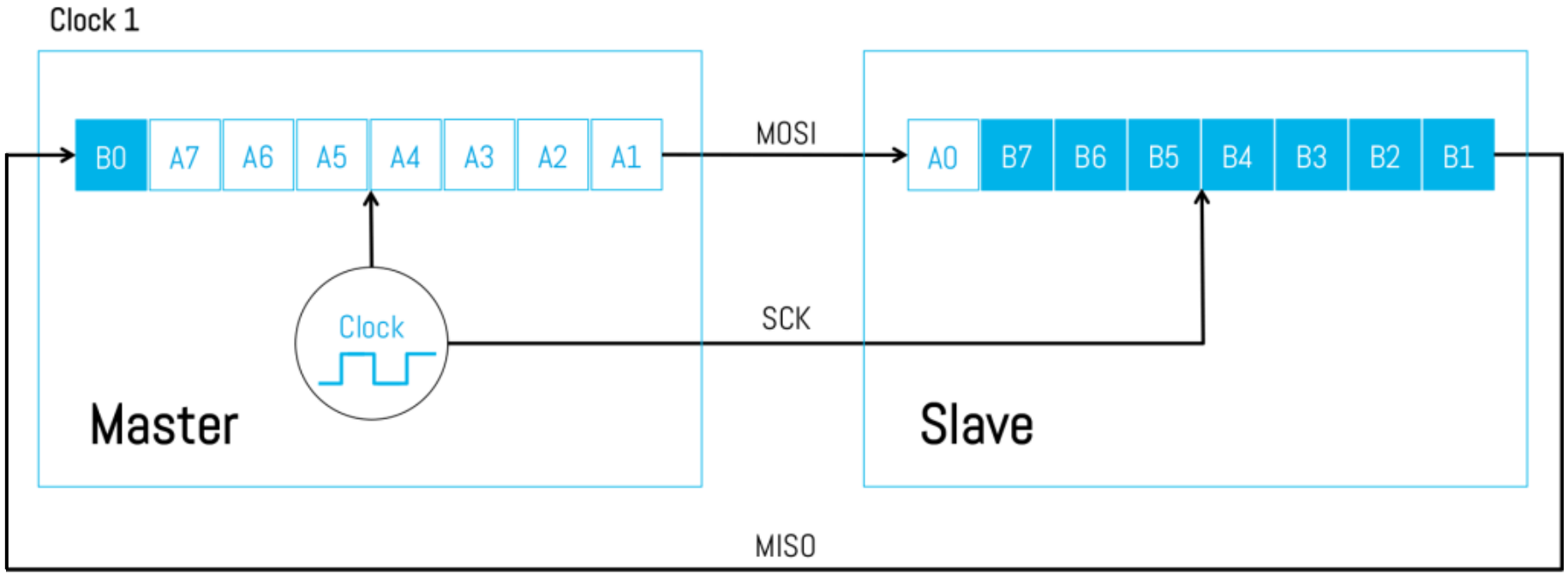
SPI operations

- The SPI operation is based upon shift registers. *Master* or *Slave* has an 8-bit shift register inside it.
- The size of the shift register could be more than 8-bit as well (like 10-bit, 12-bit, etc), but it should be the same for both *Master* and *Slave*, and the protocol should support it.
- The *Master* and *Slave* are connected in such a way that the two shift registers form an inter-device circular buffer.



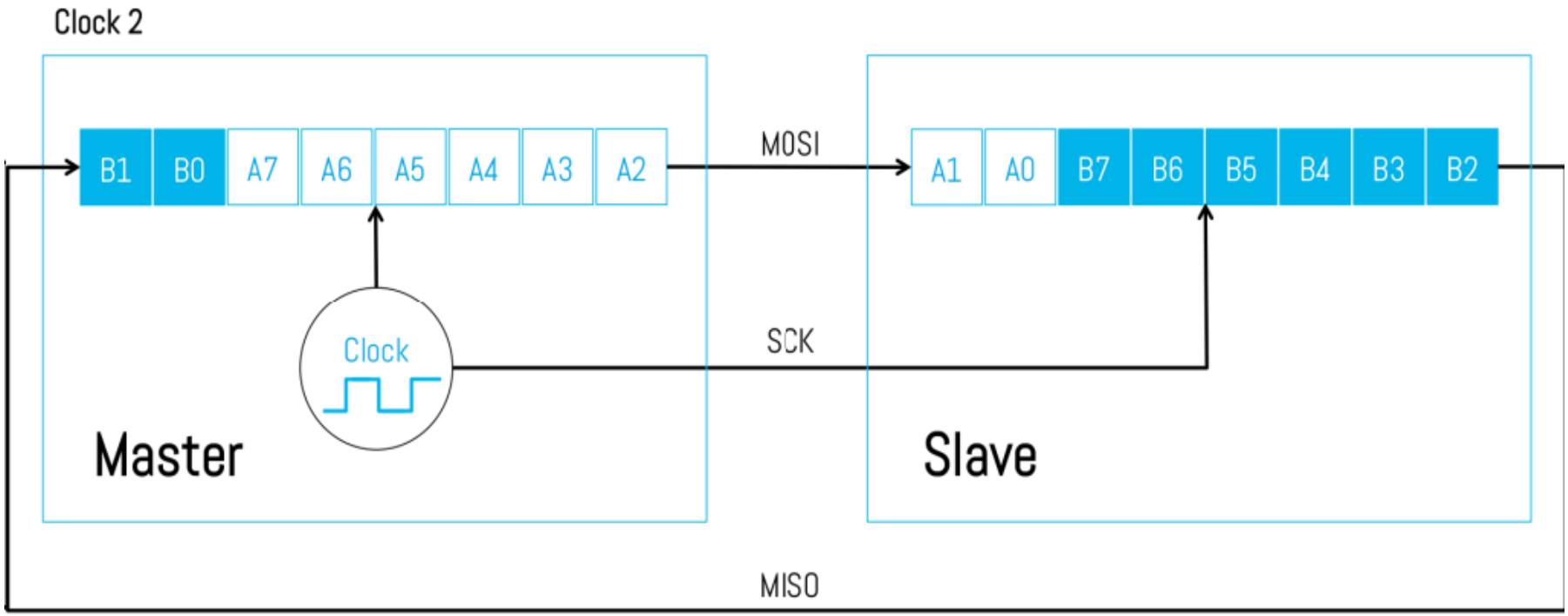
Data Transfer operation

- Transfer of 1st Bit

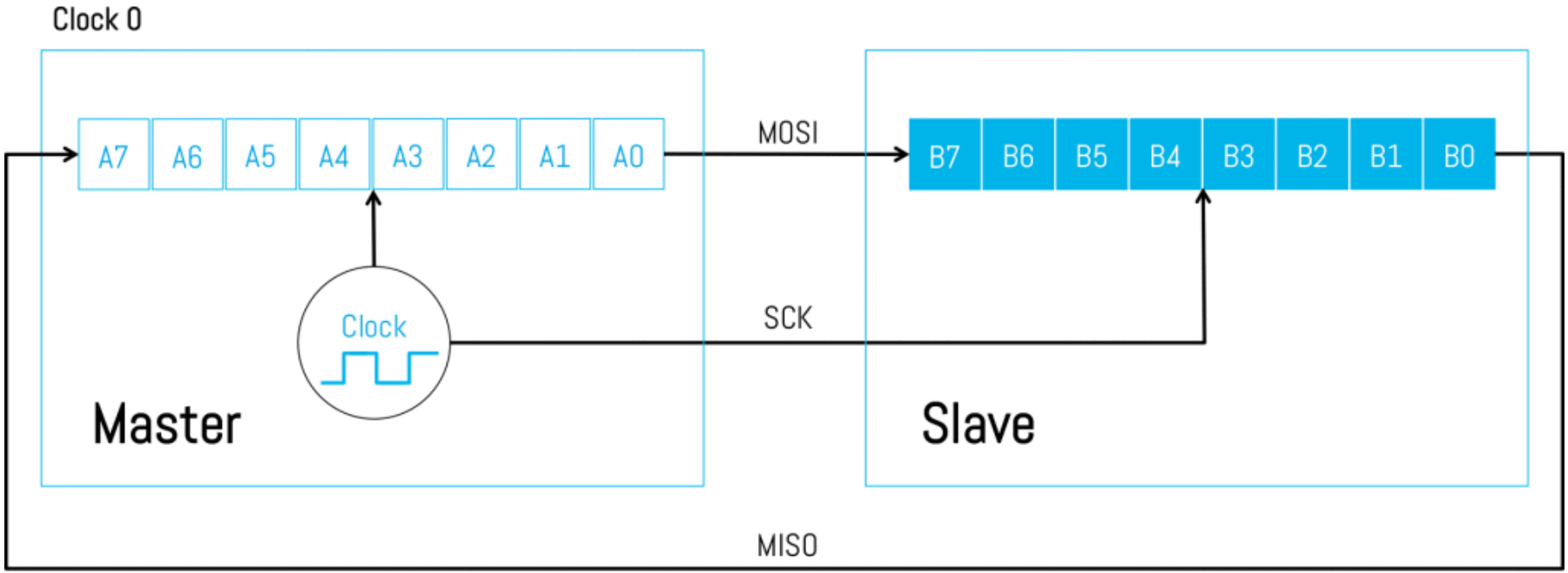


Data Transfer operation

- Transfer of 2nd bit



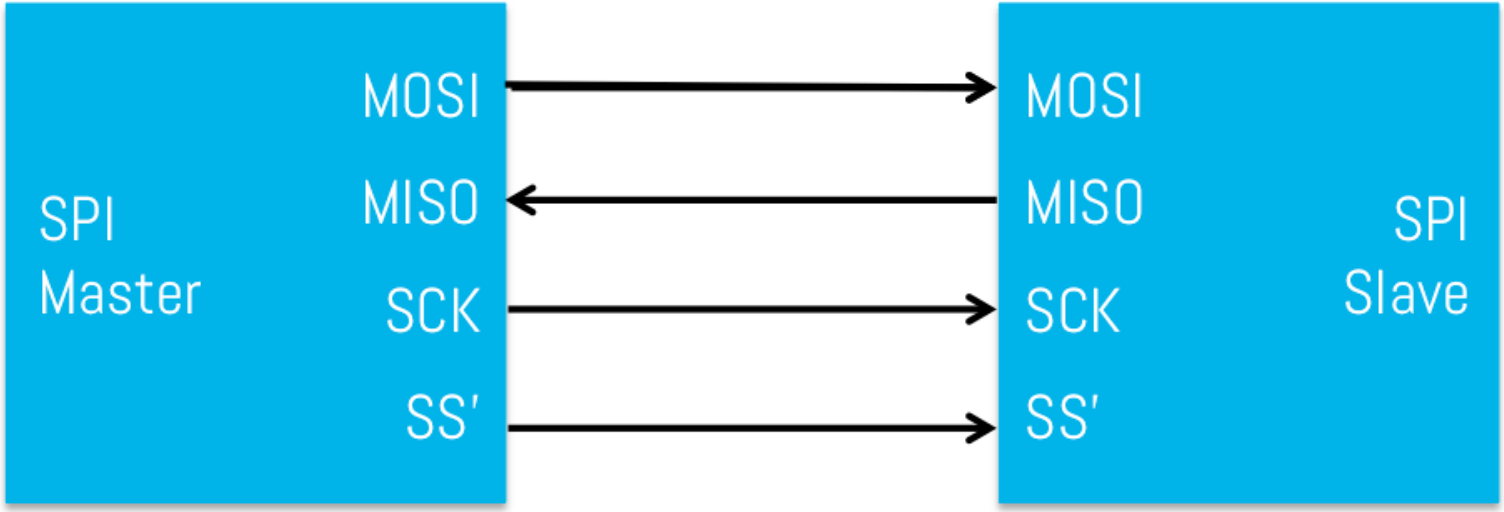
Data Transfer operation



SPI Bus Interface

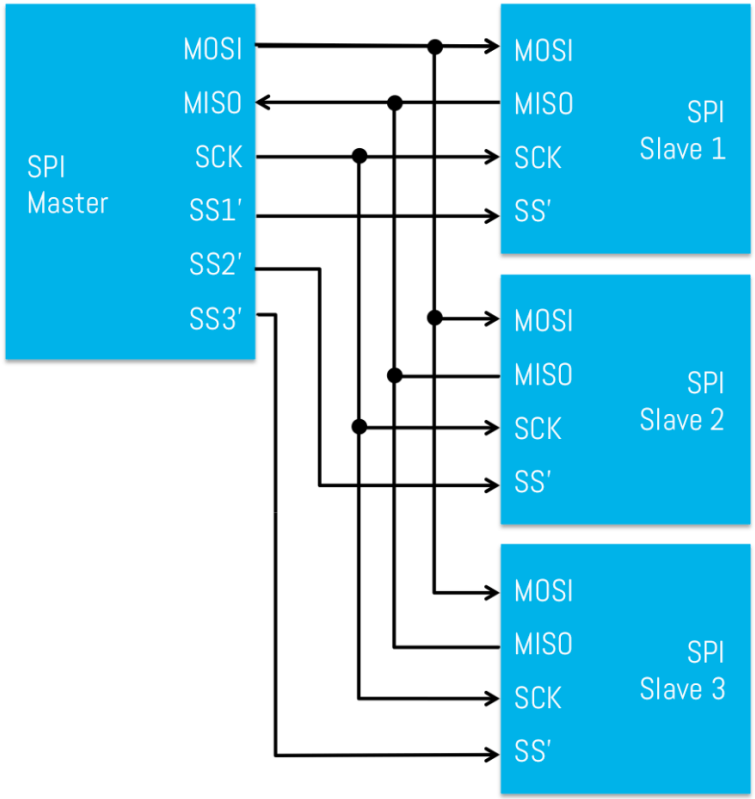
- The *Master* and *Slave* are connected by means of four wires. Each of these wires carries a particular signal defined by the SPI bus protocol. These four signals/wires are–
  - **MOSI – Master Out Slave In:** This is the wire/signal which goes from the output of *Master's* shift register to the input of the *Slave's* shift register.
  - **MISO – Master In Slave Out:** This is the wire/signal which goes from the output of *Slave's* shift register to the input of the *Master's* shift register.
  - **SCK/SCLK – Serial Clock:** This is the output of the clock generator for *Master* and clock input for *Slave*.
  - **SS' – Slave Select**

SPI Bus Interface



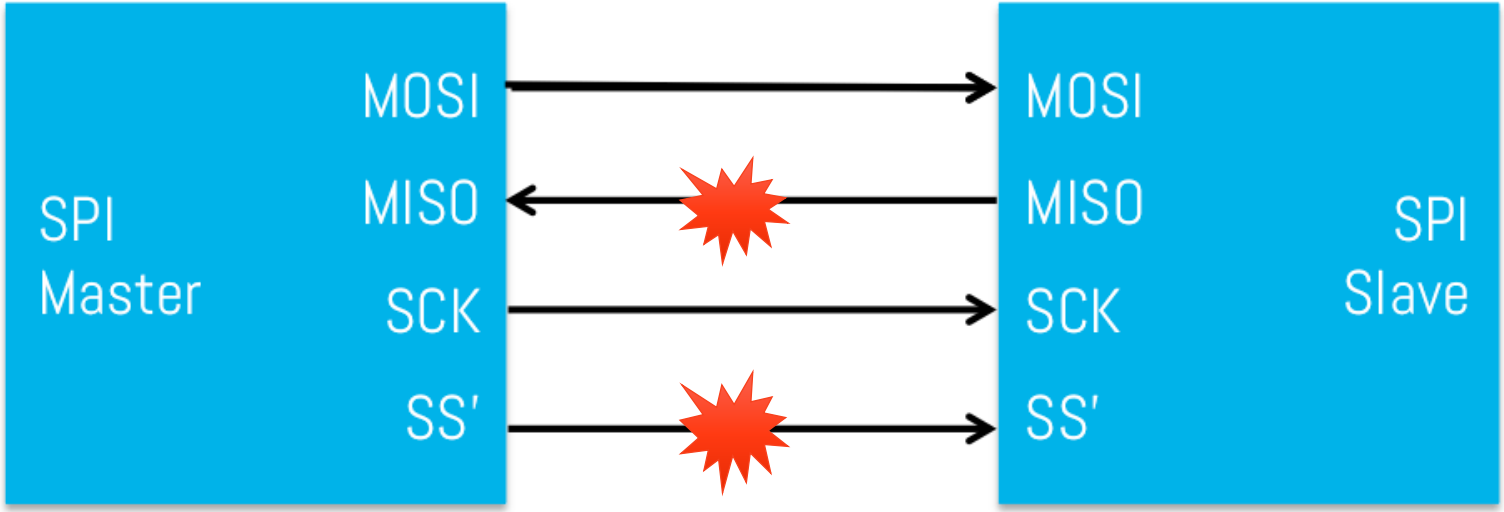
SPI Bus: Single Master
Single Slave

Multiple Slaves – Slave Select (SS') Signal



SPI Bus: Single Master
Multiple Independent Slaves

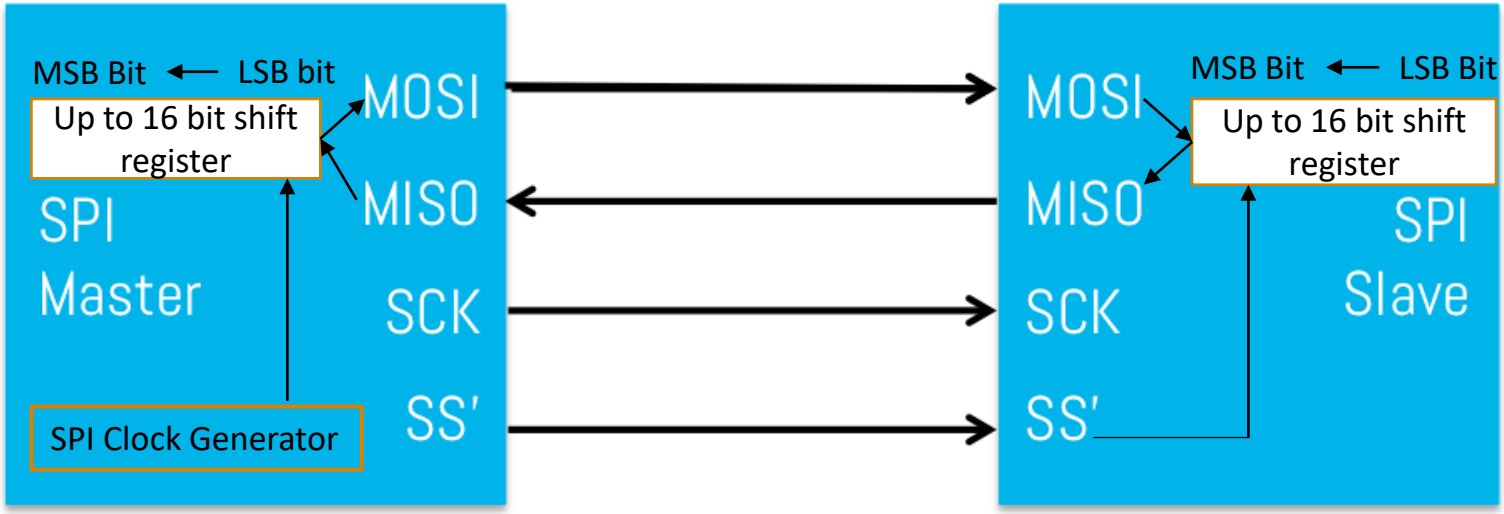
Minimal SPI Bus Interface



SPI Bus: Single Master
Single Slave

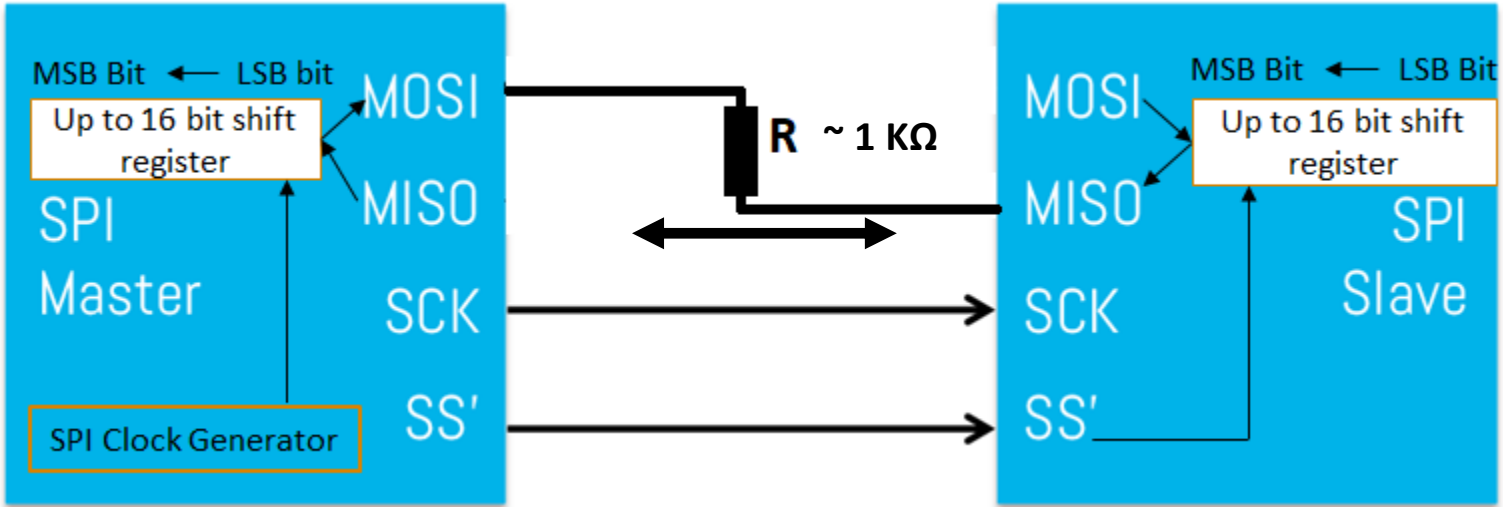
In some applications SPI bus may consist of just two wires- one for clock signal and the other for synchronous data transfer. Other signals can be added depending upon the data exchanges between SPI nodes and their slave select signal management.

SPI bus configuration: Full duplex, Half duplex and simplex



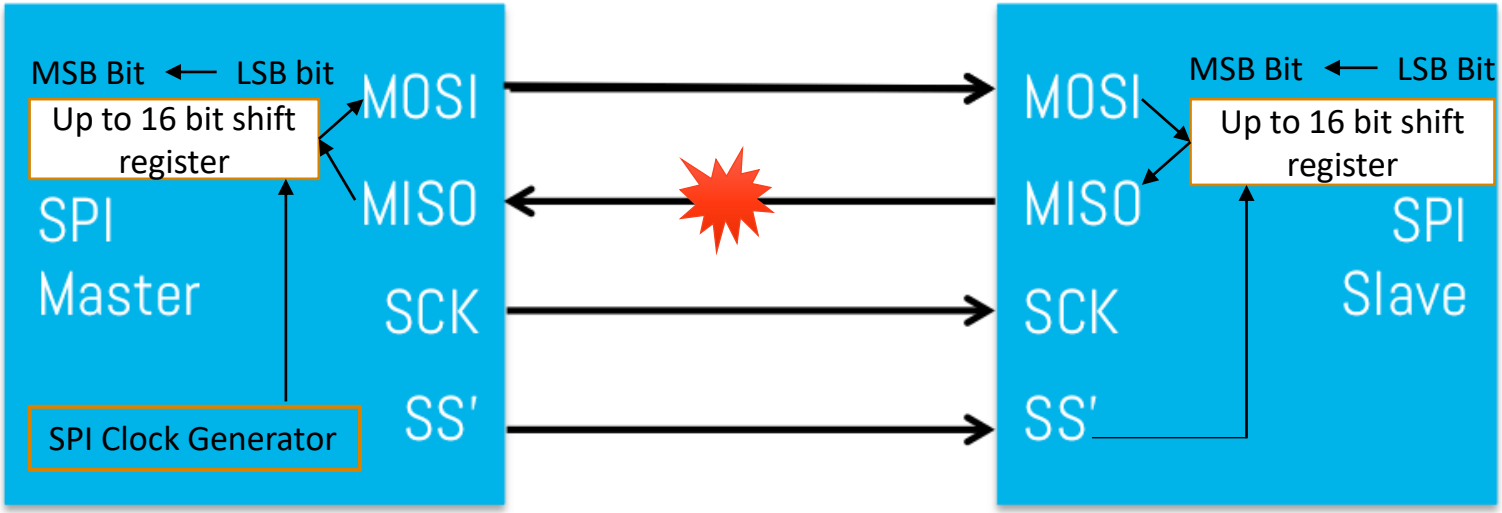
- By default SPI is configured as full duplex communication
- In this configuration, the shift registers of the master and slave are linked using two unidirectional lines between the MOSI and MISO pins.

Half Duplex communication



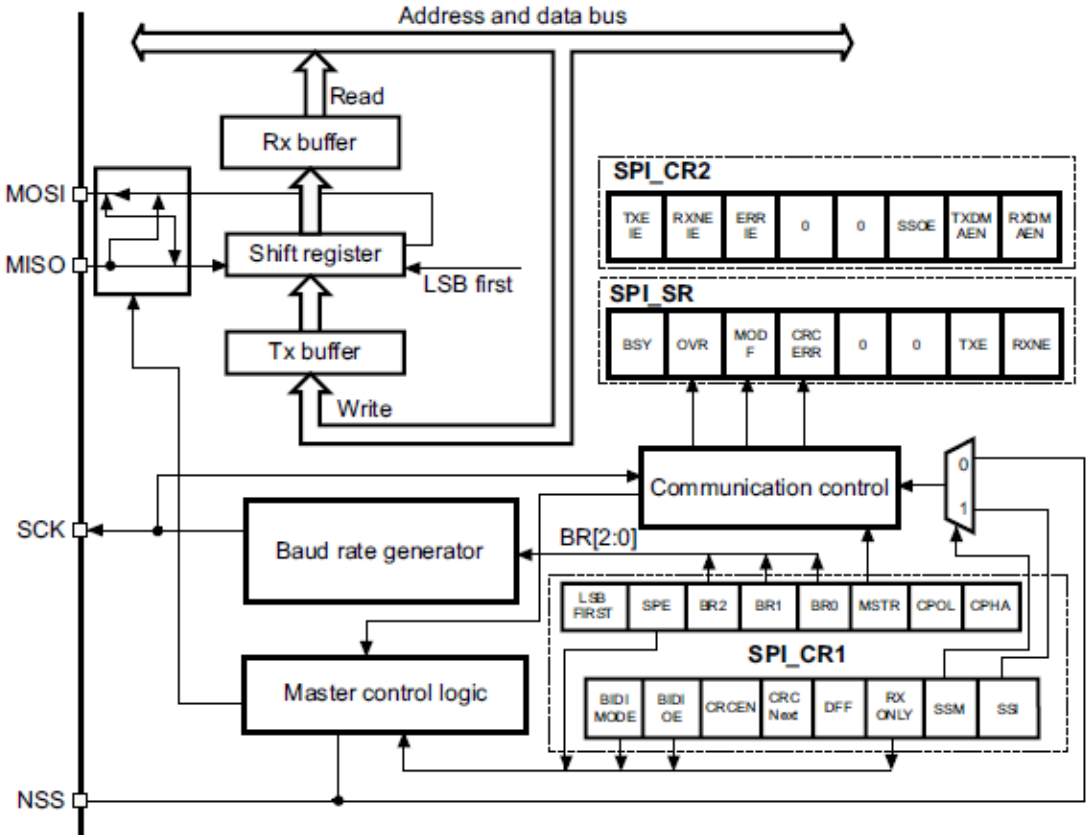
- In this configuration, one single cross connection line is used to link the shift registers of the masters and slave together. During this configuration, the data is synchronously shifted between the shift registers on the SCK clock edge in the transfer direction selected reciprocally by both master and slave.

SIMPLEX Communication



Transmit only, Receive only mode

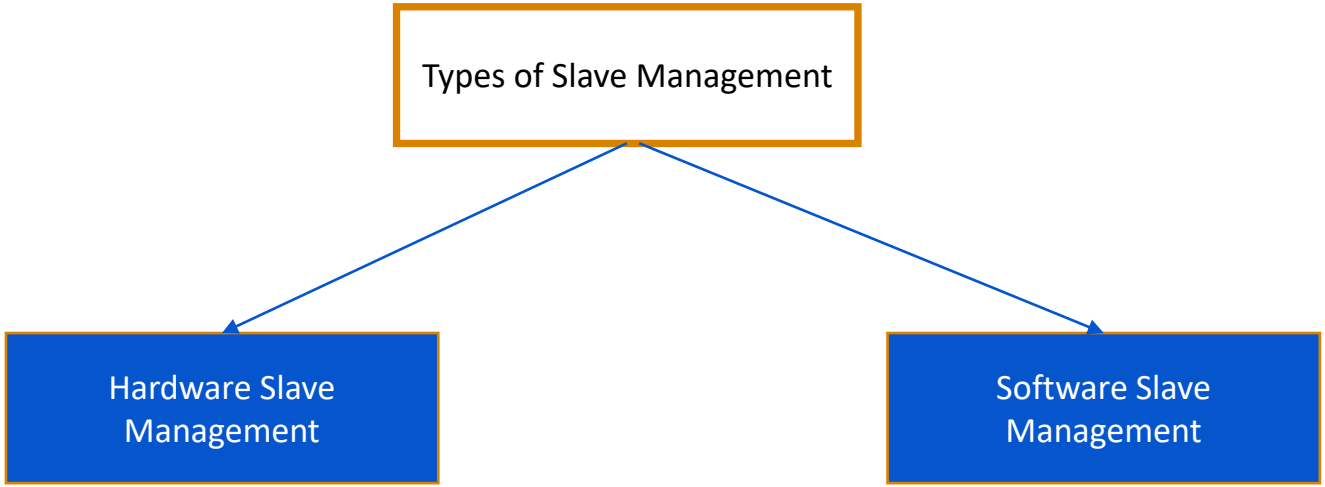
STM32- SPI Block Diagram



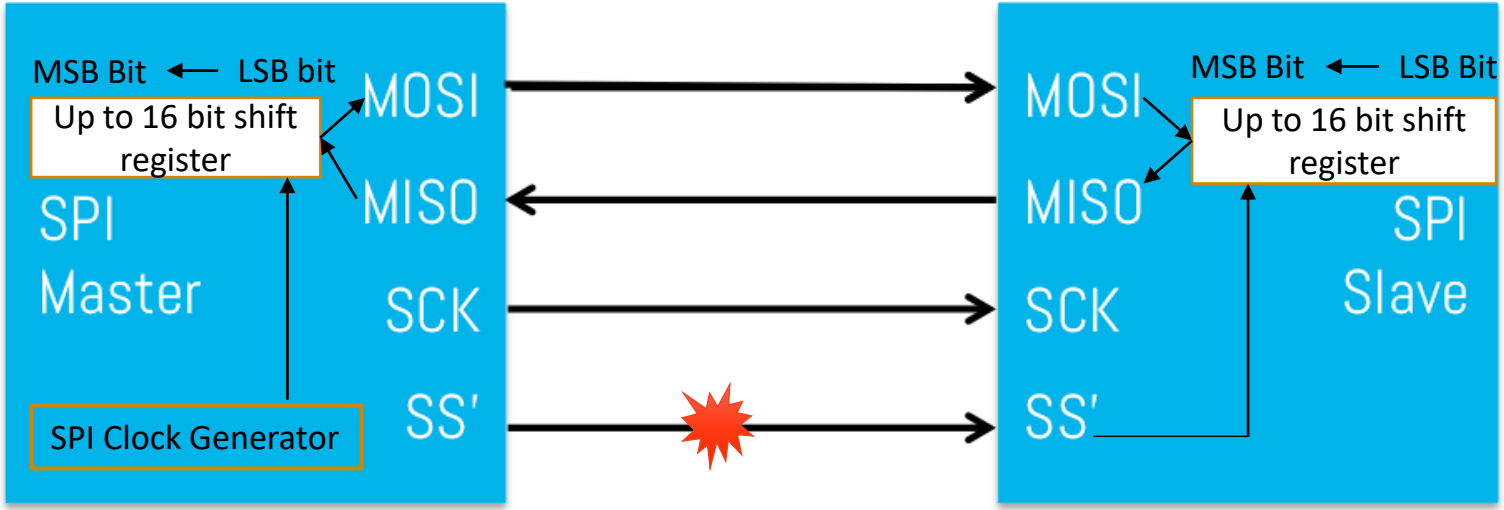
NSS settings in STM32 master and slave modes

- **When the device is in the slave mode:**
  - In Slave mode, the NSS works as a standard “chip select” input and lets the slave communicate with the master
- **When the device is in the master mode:**
  - In Master mode, NSS can be used either as output or input. As an input it can prevent multimaster bus collision, and as an output it can drive a slave select signal of a single slave

STM32 SPI hardware and software slave managements



Software Slave Management

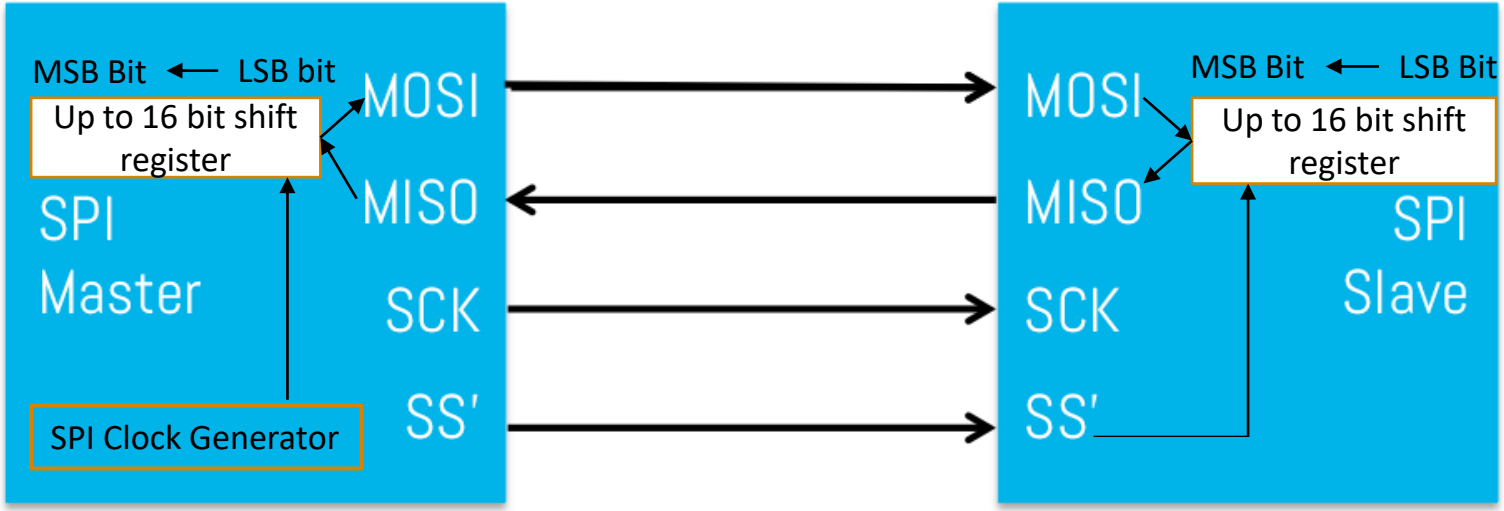


Hardware and software slave select management can be set using the SSM bit in the SPIx_CR1 register

SSM =1
SSI=0

Software NSS Management (SSM =1): IN this configuration, slave select information is driven internally by the SSI bit value in the register SPIx_CR1. The external NSS pin is free for the other application uses.

Hardware Slave Management



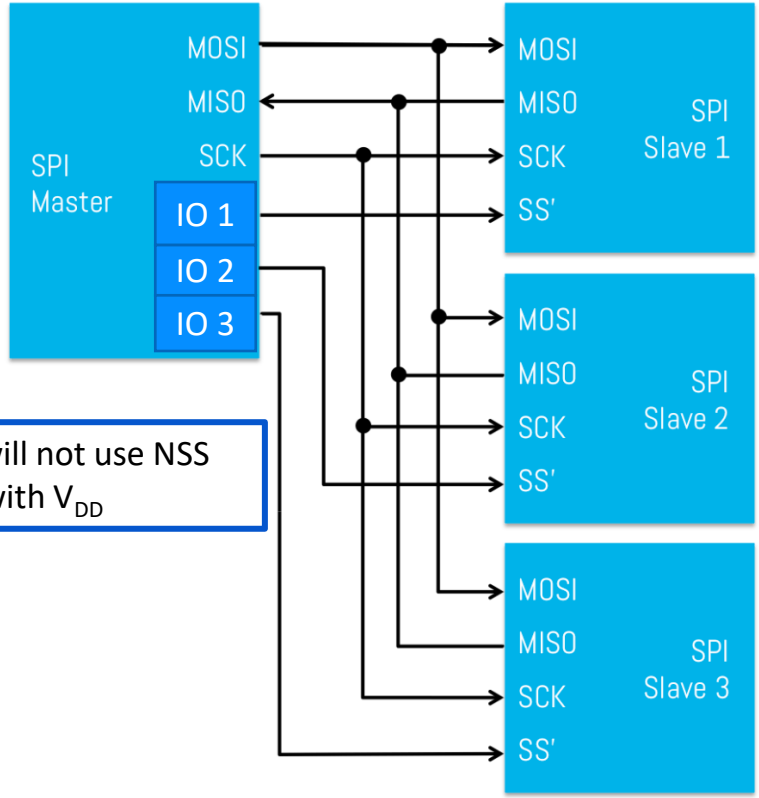
Hardware NSS Management (SSM =0): NSS pin must be in output mode. The NSS pin is managed by the hardware.

SSM =1
SSI=0

Hardware NSS Management (SSM =0): NSS pin must be pulled low to active slave to communicate with master

Hardware and software slave select management can be set using the SSM bit in the SPIx_CR1 register

SSM- Slave select Management

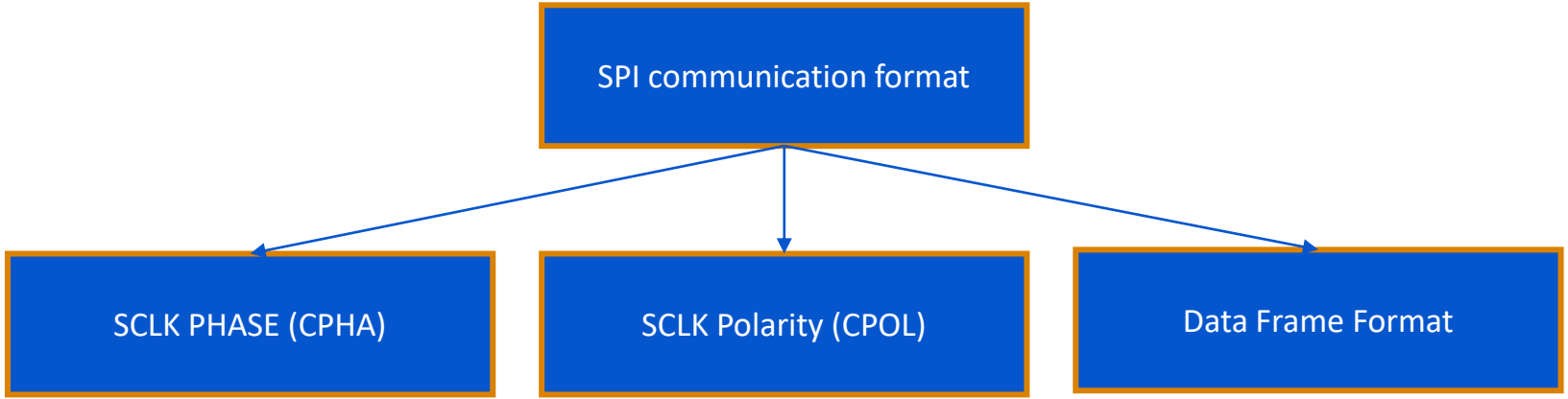


SPI Bus: Single Master
Multiple Independent Slaves

In This application you cannot use software slave management. You have to use hardware slave management

Note : In master side we will not use NSS pin and will connect NSS with V_{DD}

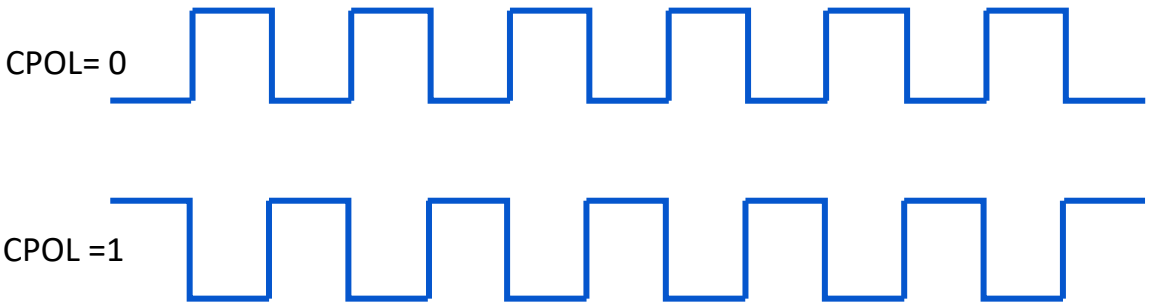
SPI communication formats



- During SPI communication, receive and transmit operations are performed simultaneously.
- The serial clock (SCK) synchronizes the shifting and sampling of the information on the data lines.
- The communication format depends on the clock phase, the clock polarity and the data frame formats. To be able to communicate together, the master and slave devices must follow the same communication format.

CPOL (Clock Polarity)

- **CPOL – Clock Polarity:** This determines the base value of the clock i.e. the value of the clock when SPI bus is idle.
  - When CPOL = 0 (reset), base value of clock is zero i.e. SCK is LOW when idle.
  - When CPOL = 1 (set), base value of clock is one i.e. SCK is HIGH when idle



CPHA (Clock Phase)

- CPHA controls at which clock edges of the SCLK (1st and 2nd) the data should be sampled by the slave
- The combination of the CPOL (Clock Polarity) and CPHA (Clock phase) bits selects the data capture clock edge.
- **CPHA – Clock Phase:** This determines the clock transition at which data will be sampled/captured.
  - When CPHA = 0, data is sampled at clock's rising/leading edge.
  - When CPHA = 1, data is sampled at clock's falling/trailing edge.

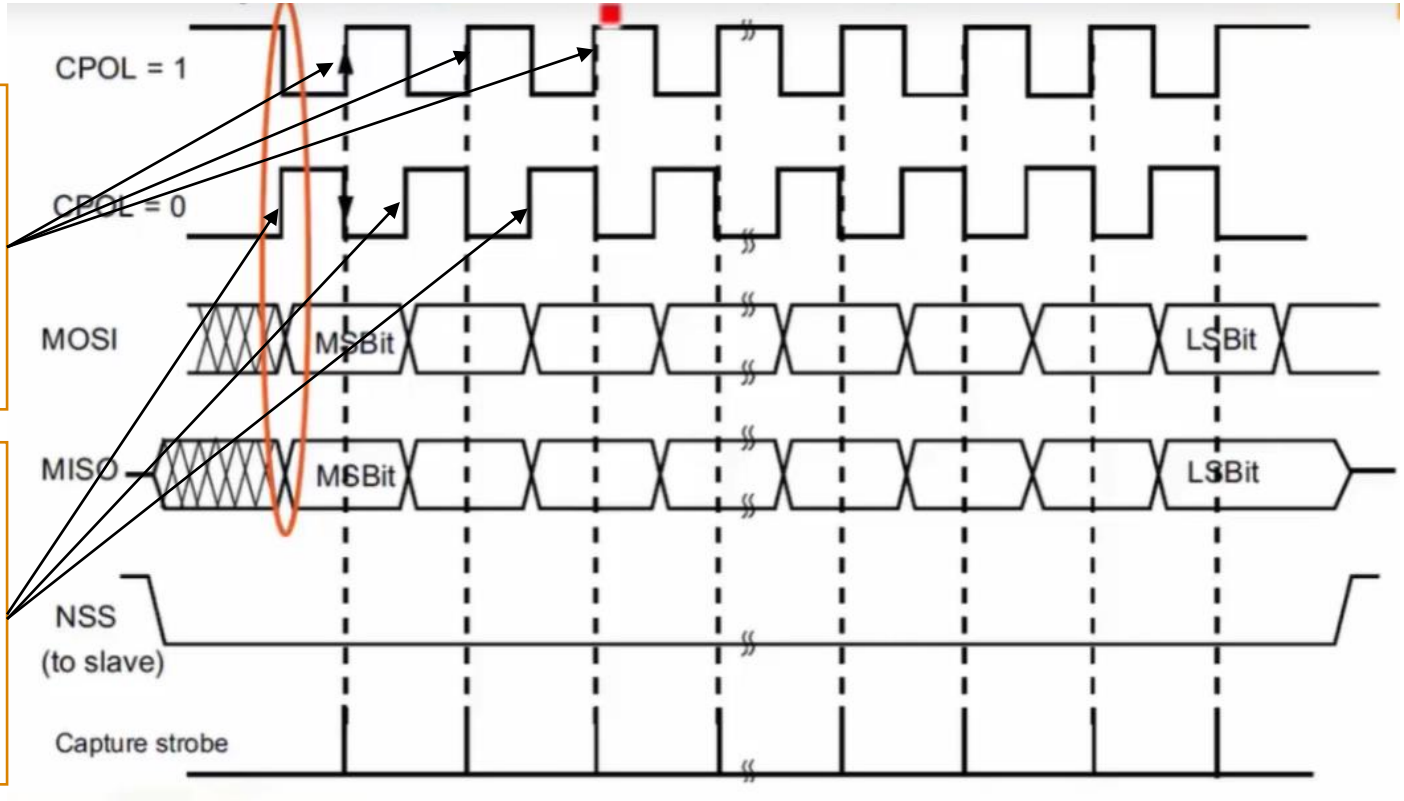
Table 59. CPOL and CPHA Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

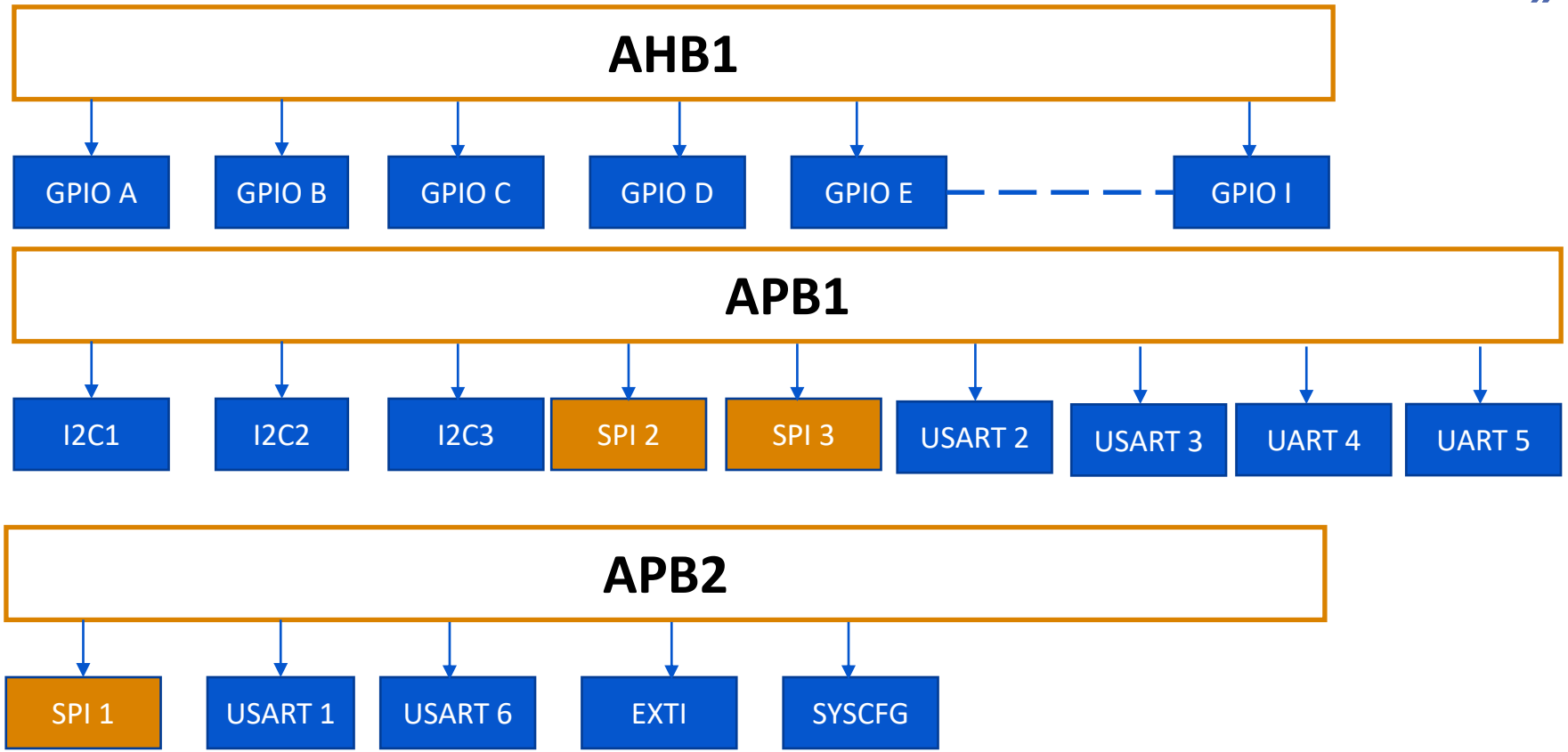
CPHA =1 or 0

CPHA -1
Data Appear in first edge of CPOL but capture the same data in second edge (Trailing edge)

CPHA -0
Data Appear in Second edge of CPOL but capture the same data in first edge (Leading edge)



SPI Peripherals of MCU

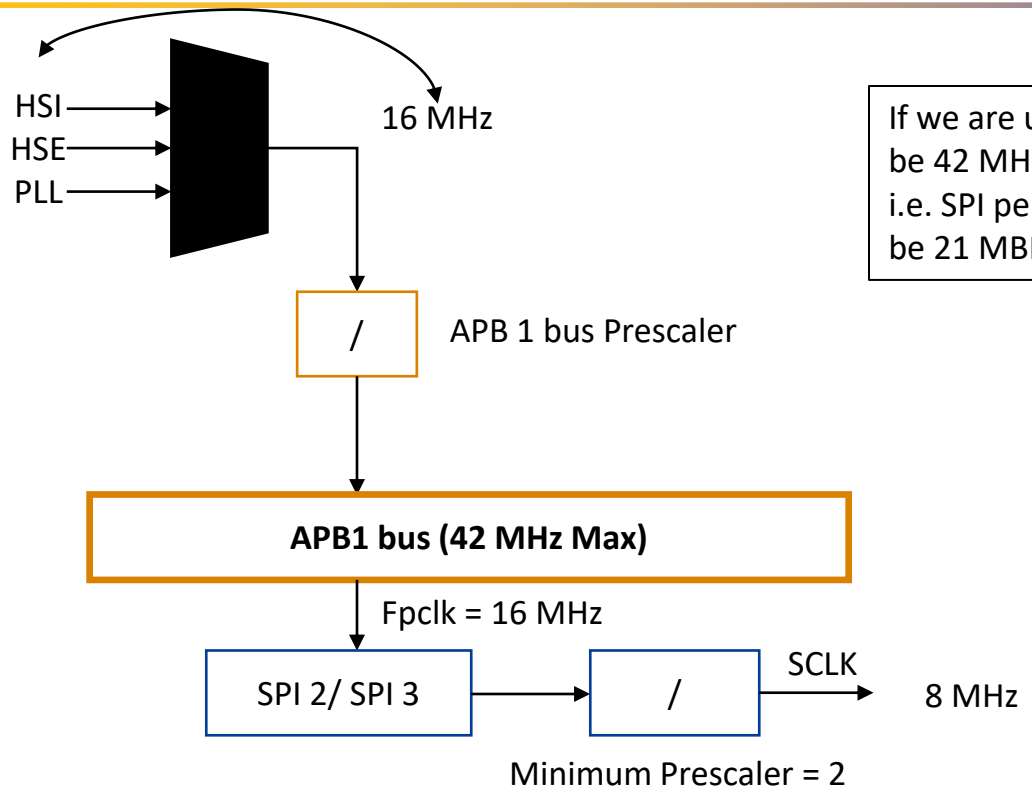


SPI Serial clock frequency

- What is the maximum SCLK speed of SPIx peripheral which can be achieved on a given microcontroller.

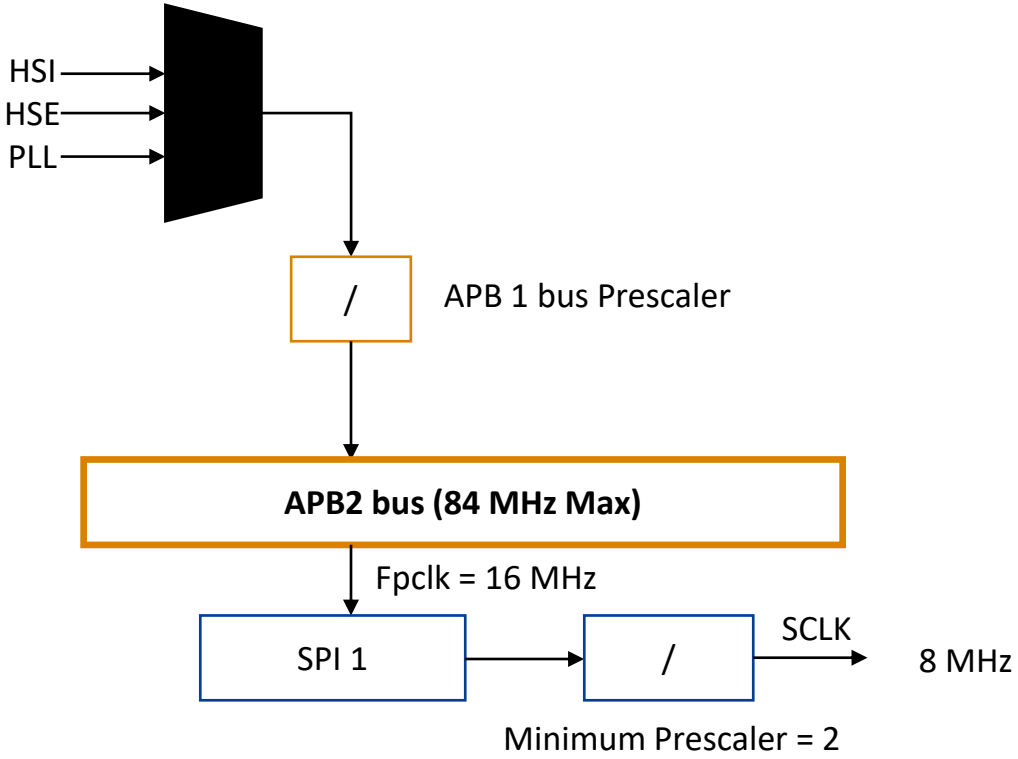
- **Answer to this is** : First we should know the speed of the APBx bus on which the SPI peripheral is connected.

SPI Serial clock frequency

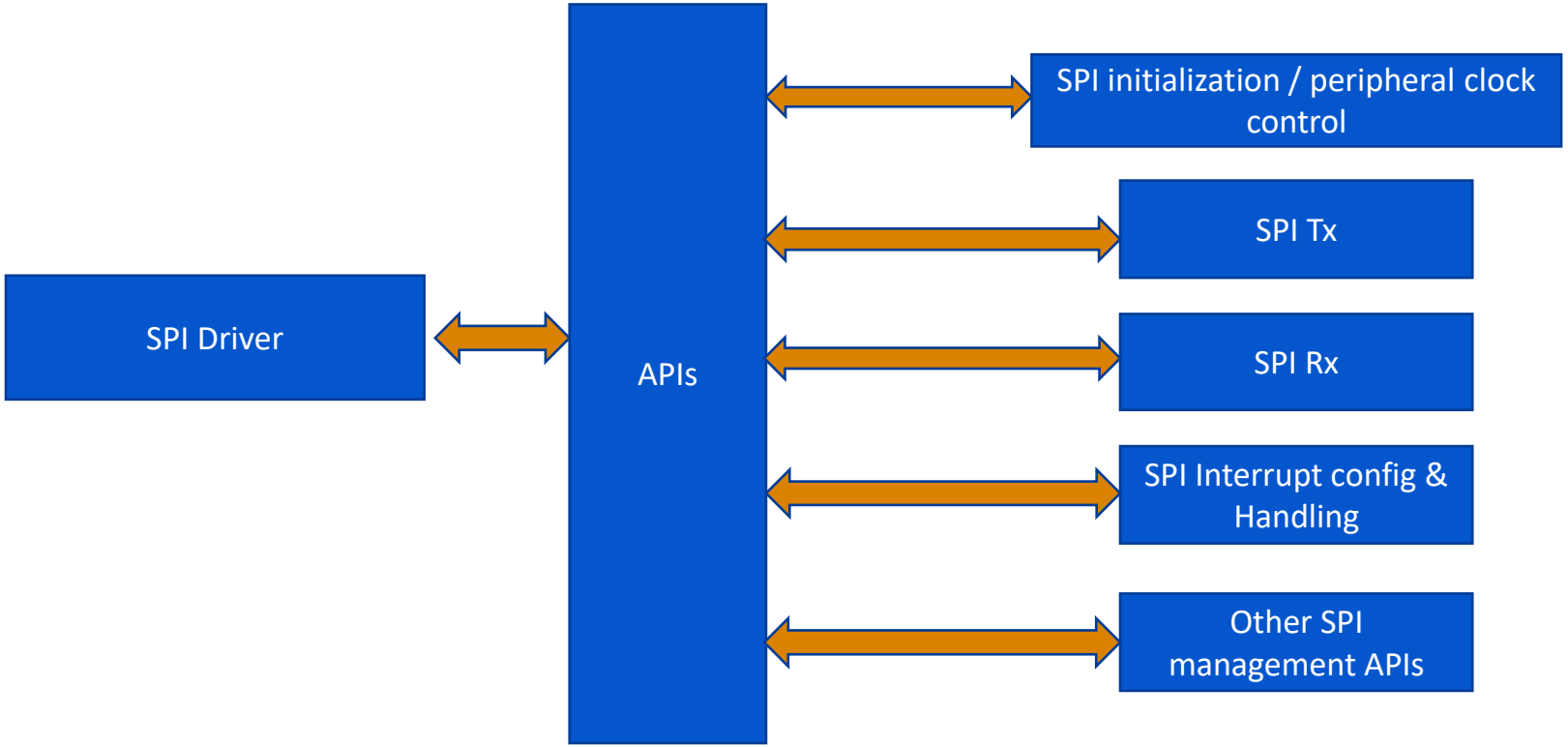


If we are using PLL then the maximum frequency will be 42 MHz and SPI and SPI 3 can produce 21 MHz i.e. SPI peripheral's maximum data transfer rate will be 21 MBPS

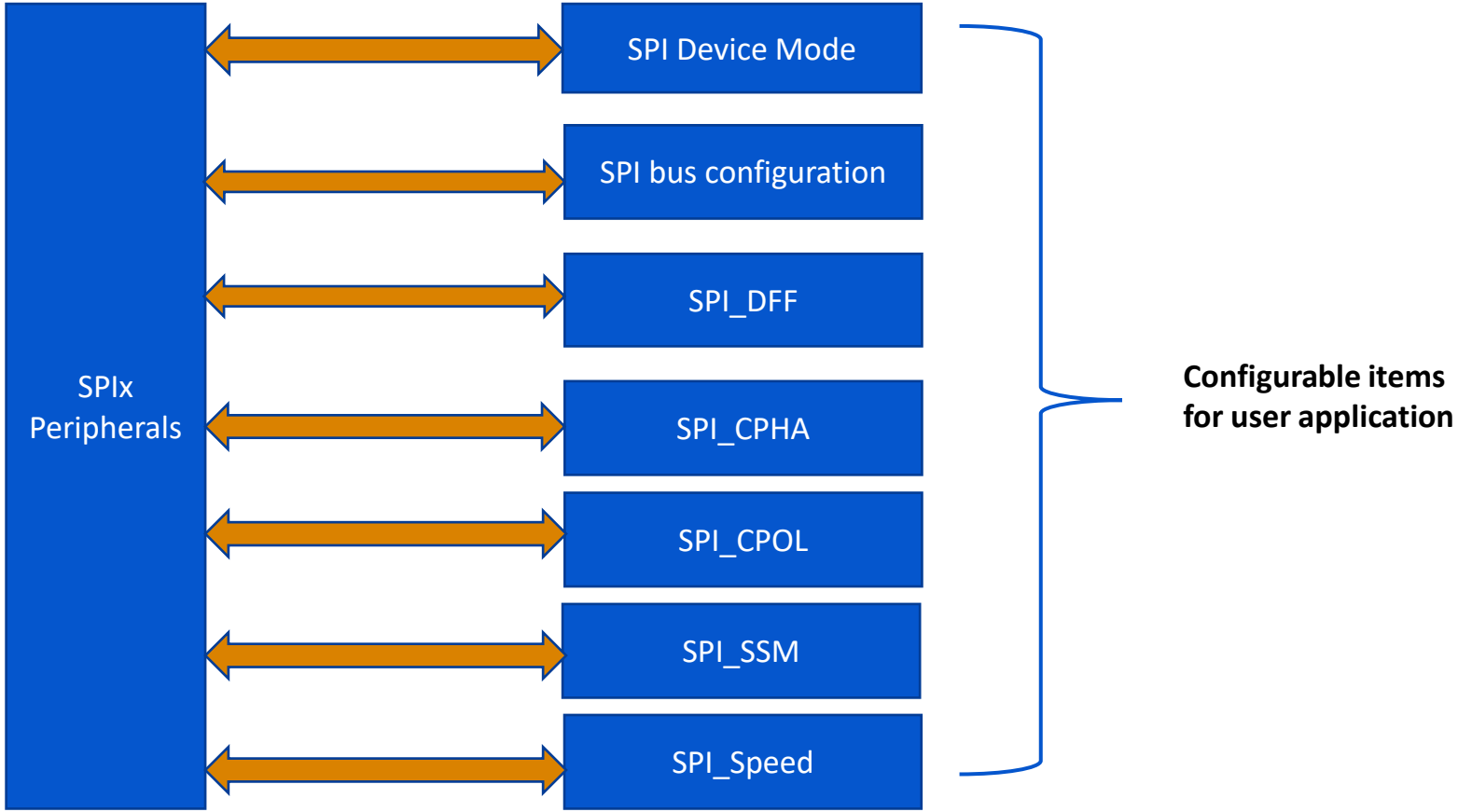
SPI Serial clock frequency



SPI Driver Development



SPI Peripherals



SPI Handle structure and configuration structure

SPI Configuration Structure

```
typedef struct
{
    uint8_t SPI_DeviceMode;
    uint8_t SPI_BusConfig;
    uint8_t SPI_SclkSpeed;
    uint8_t SPI_DFF;
    uint8_t SPI_CPOL;
    uint8_t SPI_CPHA;
    uint8_t SPI_SSM;
}SPI_Config_t;
```

SPI Handle Structure

```
/*
 *Handle structure for SPIx peripheral
 */
typedef struct
{
    SPI_RegDef_t    *pSPIx;    /*!< This holds the base address of SPIx(x:0,1,2) peripheral >*/
    SPI_Config_t    SPIConfig;
}SPI_Handle_t;
```

Exercise

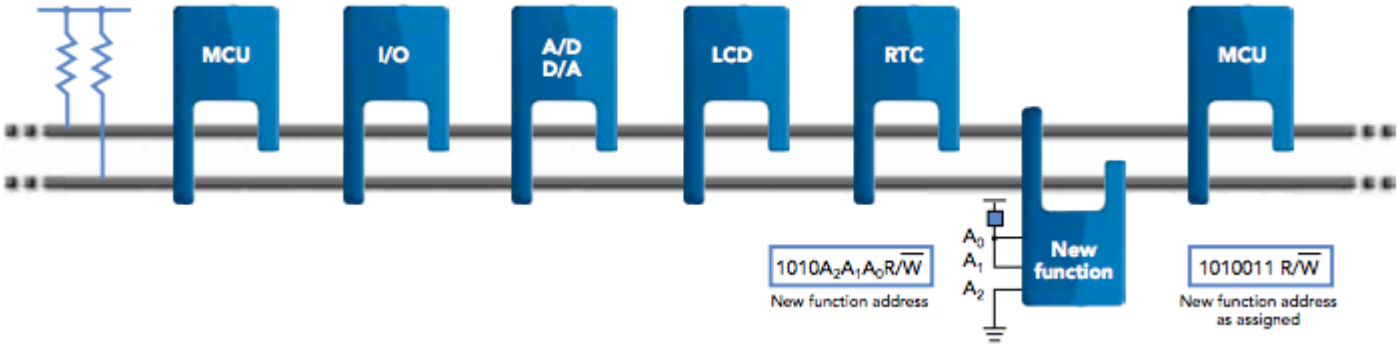
- Complete SPI register definition structure and other macros (Peripheral base addresses, device definition, clock en, clock di, etc.) in MCU specific header file.
- Complete SPI configuration structure and SPI handle structure in SPI header file

I2C Protocol

Inter-Integrated Circuit (I^2C)

- Inter-IC (or the Inter-Integrated Circuit), often shortened as *I2C* , was developed as a communication protocol to interact between different ICs on a motherboard, a simple internal bus system. (Developed by NXP)
- It is used to connect low speed peripherals (like keyboard, mouse, memory, IO/serial/parallel ports, etc.) to the motherboard (containing the CPU) operating at much higher speed.

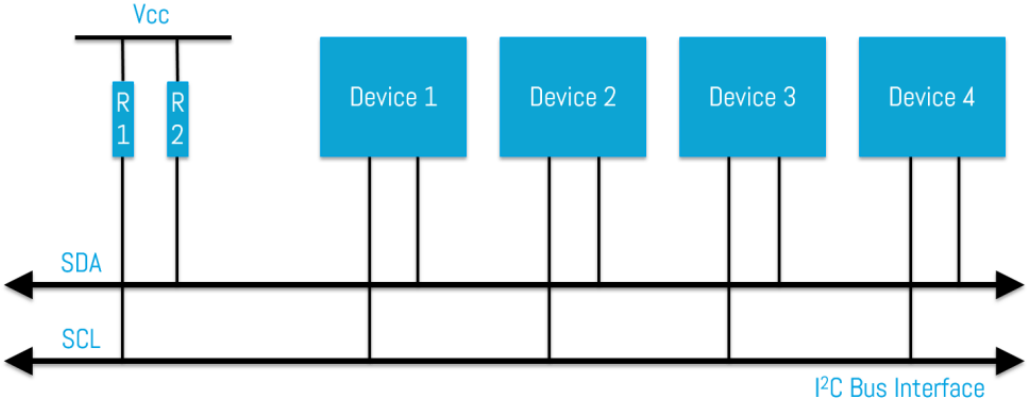
I²C Bus Interface



- The devices are hooked up to the I2C bus with just two pins (and hence it is sometimes referred to as ***Two Wire Interface***, or the ***TWI***)
- All the devices connected to the bus are classified as either being Master or Slave (just like SPI).

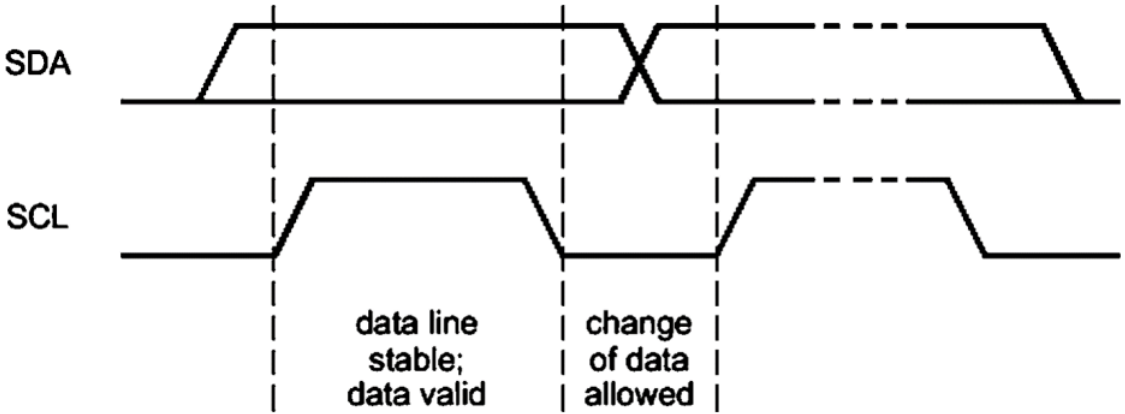
I²C Bus Interface

- The I2C bus consists of two bidirectional “open-drain” lines – SDA and SCL – pulled up with resistors.
- Serial Data Line (SDA)
- The **Serial Data Line (SDA)** is the data line (of course!). All the data transfer among the devices takes place through this line.
- Serial Clock Line (SCL)
- The **Serial Clock Line (SCL)** is the serial clock (obviously!). I2C is a synchronous protocol, and hence, SCL is used to synchronize all the devices and the data transfer together. We’ll learn how it works a little later in this post



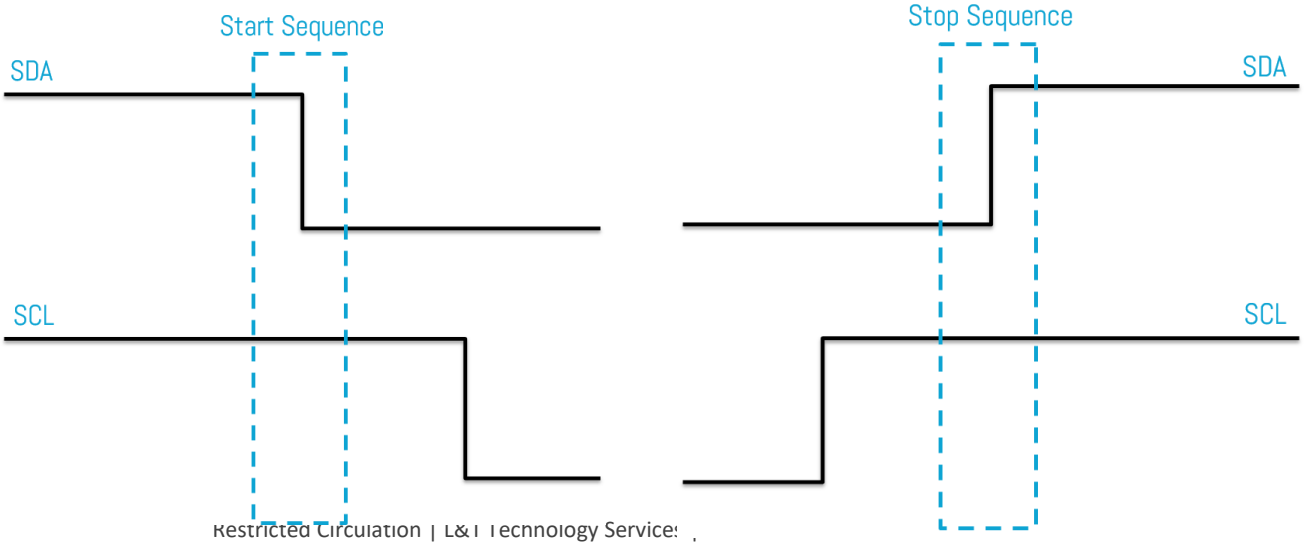
I²C Data Validity

- For the data to be valid on the SDA line, it must not change while the SCL is high. The data on the SDA line should change only and only when the SCL line goes low. If this standard is not followed, the data transfer becomes flawed



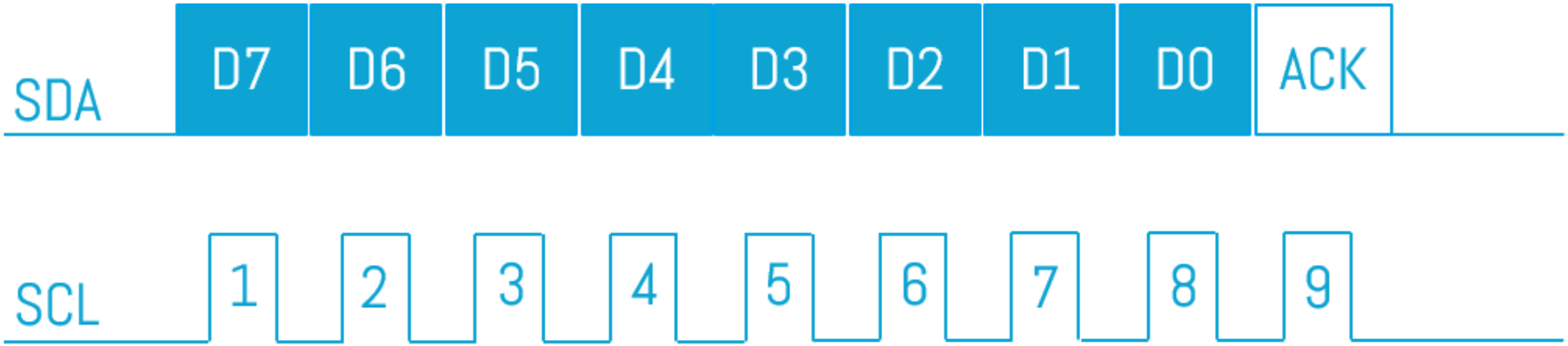
I²C Bus Transaction

- I2C supports unidirectional as well as bidirectional data transfer
  - Unidirectional Data Transfer
    - *Master-transmitter* to *Slave-receiver* (Case 1)
    - *Slave-transmitter* to *Master-receiver* (Case 2)
  - Bidirectional Data Transfer
    - *Master* to *Slave* and *Slave* to *Master* (Case 3)
- Start/Stop Sequence



Acknowledge Scheme

- I2C transfers 8 bits (1 byte) of data at a time. After the transfer of each byte is complete, the receiver must acknowledge it. To acknowledge, the receiver sends an ACK bit back to the transmitter. Here's how it goes–
- The transmitter (could be either *Master* or *Slave*) transmits 1 byte of data (MSB first) to the receiver during 8 clock pulses of SCL, after which it releases the SDA line i.e. the SDA line becomes HIGH for the ACK clock pulse.
- The receiver (could be either *Master* or *Slave*, it depends) is obliged to generate an acknowledge after each byte sent by the transmitter by pulling the SDA line LOW for the ACK clock pulse (9th clock pulse) of SCL.
- So overall, there are 9 SCL clock pulses required to transmit a byte of data. This is shown in the diagram in next slide with the assumption that *Master* is the transmitter.



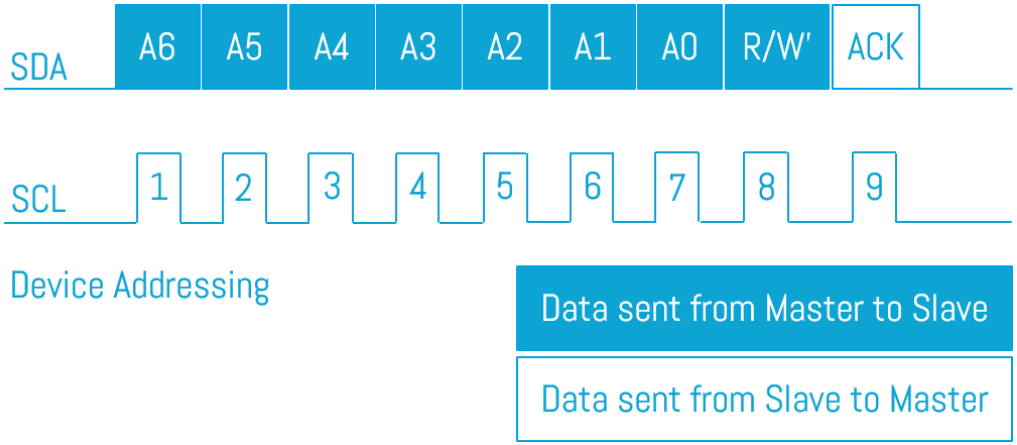
Acknowledge Scheme

Data sent from Master to Slave

Data sent from Slave to Master

I²C Device Addressing

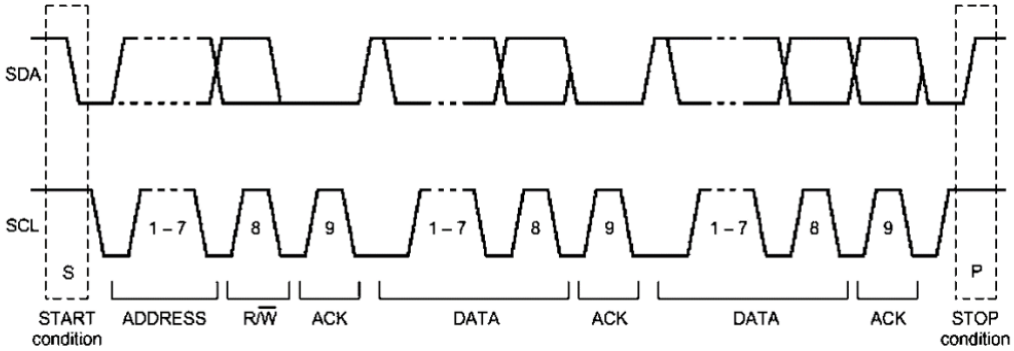
- Every device hooked up to the bus has a unique address. As per the I2C specifications, this address is either 7 bits long, or 10 bits long.
- When we have 7-bit address, we can have up to a maximum of 2^7 = 128 devices hooked up to the I2C bus, with addresses 0 to 127.
- when the *Master* calls out to the *Slave(s)*, it still needs to send out 8 bits of data. In this case, the *Master* appends an extra Read/Write (R/W') bit to the 7 bits of address (note that W' means Write complemented). Thus, the R/W' bit is added to the LSB of the data byte.



Example study of EEPROM

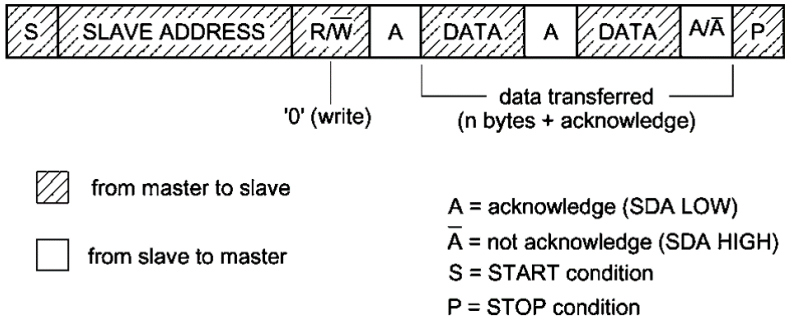
- If we need to do write and read operation in EEPROM then what should address we send?? Assuming the address of EEPROM is 0x50..

I²C Data Transfer Protocol



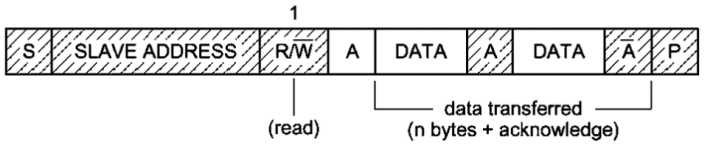
- The transaction starts with a START sequence.
- After the START sequence, the *Master* has to send the address of the *Slave* it wants to talk to. That's the 7-bit ADDRESS (MSB first) followed by R/W' bit determining whether you want to read from the device or write into it.
- The *Slave* responds by acknowledging the address and begins the data transfer and the receiver has to send the ACK signal
- When the transaction is over, the *Master* ends it by generating a STOP sequence

Case 1: Master (Transmitter) to Slave (Receiver) Data Transfer



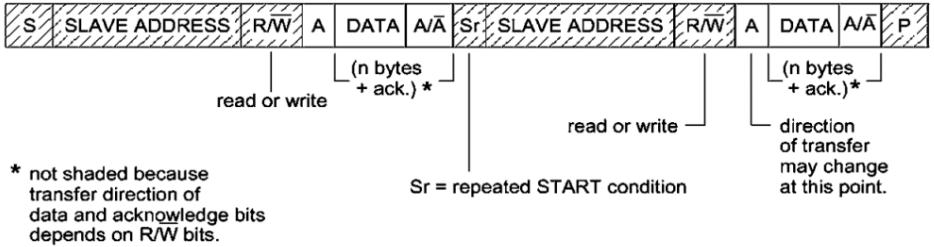
- The *Master* sends the START sequence to begin the transaction.
- It is followed by *Master* sending 7-bit *Slave* address and the R/W' bit set to zero. We set it to zero because the *Master* is *writing* to the *Slave*.
- The *Slave* acknowledges by pulling the ACK bit low.
- Once the *Slave* acknowledges the address, *Master* can now send data to the *Slave* byte-by-byte. The *Slave has to* send the ACK bit after every byte it recived.

Case 2: Slave (Transmitter) to Master (Receiver) Data Transfer



- The *Master* sends the START sequence, followed by the 7-bit *Slave* address and the R/W' bit set to 1.
- We set R/W' bit to 1 because the *Master* is *reading* from the *Slave*.
- The *Slave* acknowledges the address, thus ready to send data now.
- *Slave* keeps on sending data to the *Master*, and the *Master* keeps on sending ACK to the *Slave* after each byte until it can no longer accept any more data.
- When the *Master* feels like ending the transaction, it *does not* send the ACK, thus ending with the STOP sequence.

Case 3: Bi-directional Read and Write in same Data Transfer



- The *Master* sends out the START sequence, followed by the 7-bit *Slave* address and the R/W' bit.
- The *Slave* acknowledges the address.
- Depending upon the value of the R/W' bit, read/write operations are performed (like the above two cases).
- Whatever the case it may be, it always ends with the receiver *not* sending the ACK.
- Until now, in the previous two cases, we have seen that the *Master* would close the connection. But in this case, the *Master* attempts a repeated START.
- And the entire process repeats again, until the *Master* decides to STOP.

Coding guidelines and optimizations

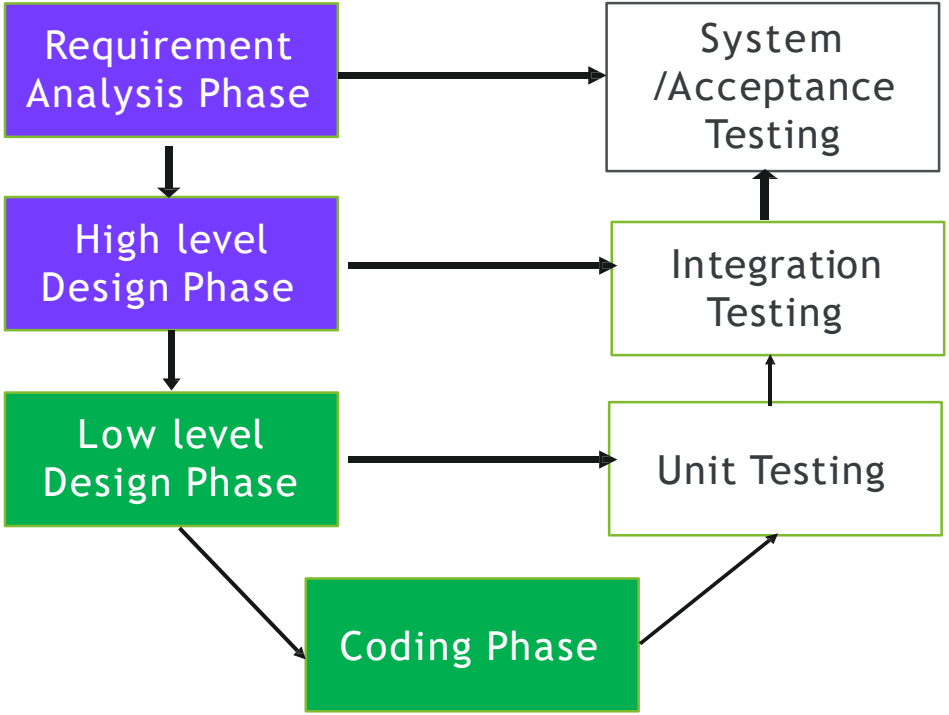
Code quality characteristics

- Software product quality is defined in detail and organized systematically in ISO/IEC 25010. Source code quality characteristics can be divided into eight metrics.

Source code quality characteristics

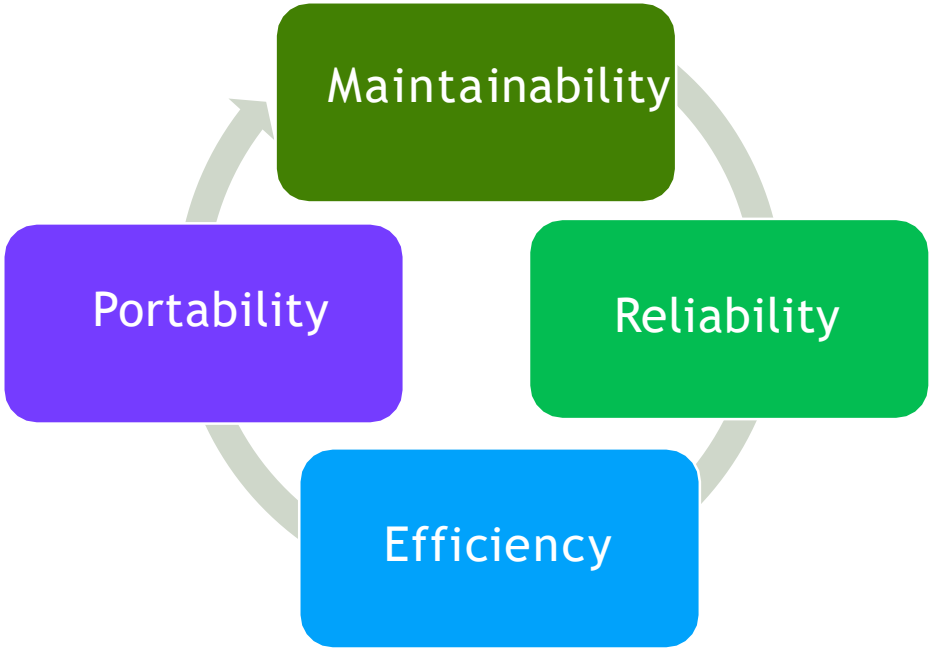
I. Reliability
II. Maintainability
III. Portability
IV. Efficiency
V. Security

VI. Functionality
VII. Usability and
VIII. Compatibility



Embedded developer's perspective on code quality

- Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components
- Degree to which a code can be used in other programs
- Degree of Modifiability and Testability
- Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments
- Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled or replaced in a specified environment



Important code metrics

Reliability	**R**	Practices to improve the reliability of software that has been developed fall under this category. Main points taken into consideration include: Main points considered include: - Minimizing problems arising while using the software; - Increasing tolerability against bugs and interface violation.
Maintainability	**M**	Practices to create source code that is easy to modify and maintain fall under this category. Main points taken into consideration include: - Making the code easy to understand and modify; - Minimizing the impact of modifications on the entire code; - Making the modified code easy to check.
Portability	**P**	Practices to port the software program that has been created on the assumption of being used to operate under a certain environment to another environment as efficiently as possible without error fall under this category.
Efficiency	**E**	Practices to effectively utilize the performance and resources of the software that has been developed fall under this category. Main points taken into consideration include: - Coding that is processing time-conscious; - Coding that takes account of memory size.

MISRA C standard

- A set of "Guidelines for the Use of the C Language in Vehicle-Based Soft-ware," or "MISRA
- C," are developed with following objectives:

- Writing a safe program in C
- Remove base language issues and use subset as per ISO standard C
- Make C readable and portable to other platforms

- Guidelines are set of advisory rules while writing C programs

Initialize areas and use them by taking their sizes into consideration.

- Use areas after initializing them.
- Describe initializations without excess or deficiency.
- Pay attention to the range of the area pointed by a pointer.

Automatic variables shall be initialized at the time of declaration, or the initial values shall be assigned just before using them.

Compliant code

```
void func() {
int var1 = 0; /* Initialize at the time of declaration */
int i;
var1++;
/* Assign the initial value just before using it */
for (i = 0; i < 10; i++) {
    ... const variables shall also be initialized at the time of
                                    declaration.
    }
    }
```

Non-Compliant code

```
void func(){
int var1;
var1++;
...
}
```

Describe initializations without excess or deficiency

- Arrays with specified number of elements shall be initialized with values that match the number of the
- Elements.

- char var[] = "abc";
- - or -
- char var[4] = "abc";

Pay attention to the range of the area pointed by a pointer.

- Rule 1. Integer addition to or subtraction from (including ++ and --) pointers shall not be made; Use Array format with [] for references and assignments to the allocated area.

```
#define N 10
int data[N];
int *p;
int i;
p = data;
i = 1;
Compliant example of (rule 1) and (rule 2)
data[i] = 10; /* compliant */
data[i+3] = 20; /* compliant */
Compliant example of (2)
*(p + 1) = 10;
```

- Rule 2: Integer addition to or subtraction from (including ++ and --) pointers shall be made only when the pointer points to the array and the result must be pointingwithin the range of the array.

```
#define N 10  int
data[N];  int *p;
p = data;
Non-compliant
example of (rule
1)
*(p + 1) = 10;  /*
Non-compliant */
p += 2;  /* Non-
compliant */
Non-compliant
example of (rule
2)
*(p + 20) = 10;  /*
Non-compliant */
```

Subtraction between pointers shall only be applied to pointers that address elements of the same array. [MISRA C:2012 R18.2]

```
ptrdiff_t off; /* ptrdiff_t is a type of result of subtraction between pointers defined in
<stddef.h> */
int var1[10];
int *p1, *p2;
p1 = &var1[5]; p2 = &var1[2];
off = p1 - p2; /* Compliant example of this rule */
```

Comparing which pointer is greater or less than the other pointer shall be used only when two pointers are both pointing at either the elements in the same array or the members of the same structure.

- The restrict type qualifier shall not be used. [MISRA C:2012 R8.14]

```
void f(int n, int *  restrict p, int *  restrict q){
while (n-- > 0) {
*p++ = *q++;
}
}
void g(void) {
extern int d[100];
f(50, d+1, d); /* Undefined behavior */
```

Examples

- Arithmetic operator in logical operations
- Warning detail: The left hand operand of this logical AND (&&) or logical OR (||)operator is derived from a bitwise or arithmetic operator. This is probably a mistake; perhaps a bitwise AND (&) or bitwise OR (|)was intended ?

For example

```
r = (a+b) && c; // warning

r = ((a+b) != 0) && c  // correct

r = (a+b) || c; // warning

r = ((a+b) != 0) || c; // correct
```

Rule 13

- Application should not directly use basic types such as char, int, float etc.
- The most common case is the type int that could be seen as 16-bit wide by one compiler and 32- bit wide by another
- The C standard, as of the 1999 version, contains a header file stdint.h that provides typenames on the form
- int8_t and uint32_t

```
uint32_t flag32; /* Use unit32_t if 32bits is assumed */
int i;
for (i = 0; i < 10; i++) { … }
/* i is used as an index. It can be 8bits,
16bits or 32bits and a basic type in the
language specification can be used for i*/
```

Rule 23 (advisory):

- All declarations at file scope should be static where possible
- Prevents you from unintentionally exposing internal help functions and file-local variables

Rule 33

- The right hand side of a "&&“ or "||“ operator shall not contain side effects
- Consider the code below:

```
if ((x == y) ||(*p++ == z))
{
/* do something */
}
```

Use Explicit conversions

- Favor explicit conversions to avoid loss of data
- When a variable is converted to a type smaller in size
- When a signed type is converted to an unsigned type
- When an unsigned type is converted to a signed type
- When a floating-point type is converted to an integer type
- When a constant expression is converted to a type that cannot hold the value of the constant

```
unsigned int a ; unsigned
int b;  unsigned char
result;


result = a/b ;    // Not OK
result = (unsigned char)(a/b);  // OK
```

Tests of a value against zero should be made explicit

```
int pl = 5;

if (pl)  /* Not Ok */

{

}

if (pl != 0) /* OK */

{

}
```

```
// If(    TRUE == m_bIsAvailable )
// {
// // something
// }
// // Instead of  If(m_bIsAvailable == TRUE )
// {
// // something
// }
```

General Coding guidelines

1. Avoid code duplication. Put duplicate code in the function. This makes the program readableand flexible for future change.
2. Place the machine dependent code in a special file so that it may be easily located when porting code from one machine toanother.
3. Write code in such a way that makesit easy to change the prefix for global identifiers.
4. If any module needs to be executed in specified time, consider coding in C and assembly
5. The code size or its footprint must not exceed 75% of available flash memory or ROM in order to accommodate last minute changes /debugging.
6. If application is calculation intensive, check what floating point precision isrequired
7. Check whether the target processor has a separate Floating point Unit to speed up floating point operations
8. Consider using compression algorithms if huge data isinvolved

Optimization and code efficiency ?

- Program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources.
- Optimization is a process of improving efficiency of a program in time (speed) or Space (size) or Power

Why optimization is needed ?

- Limited microcontroller resources (ie., memory and CPU loading).
- Performance is time-critical (real-time behavior).
- Characteristics of Embedded Systems that require better optimizations:
  - system timing,
  - code size,
  - RAM usage,
  - and energy consumption

Definition- Optimization

- Application Perspective
- "Optimization" is a procedure that seeks to maximize or minimize one or more performance indices including:

Throughput (or execution speed)

Memory usage

I/O bandwidth and

Power dissipation

Developer's Perspective on optimization

- **Optimizing the CPU architecture**
- **Optimizing the compilers**
- **Optimizing the algorithms**

Optimization: some trade-offs

- **Speed Optimization**
- May consume less computing power or energy
- Often Scarifies memory
- **Memory Optimization**
- May reduce un-necessary memory access and hence consume less clock cycles or energy But slows down the execution speed of code
- **Power optimization**
- Speed of execution can be optimized

Optimization Methods – Developer's perspective

- Compiler settings
- Programming Algorithm and Techniques
- Rewriting program in Assembly

Some trade-offs: GCC optimization levels

Option	Optimization level	Execution time	Code size	Memory usage	Compile time
-O0	optimization for compilation time (default)	+	+	–	–
-O1	optimization for code size and execution time	–	–	+	+
-O2	optimization more for code size and execution time	– –		+	+ +
-O3	optimization more on execution time or speed	– – –		+	+ + +
-Os	optimization for code size		– –		+ +
-Ofast	O3 with fast none accurate math calculations	– – –		+	+ + +

How to optimize?

- Data types usage and their significance
- Variables and address optimization
- Data Alignment – Arrangement & Packing
- Pass by Reference
- Flow control
  - Use of switch statement
  - Inline function or macros
  - Loop unrolling
  - Loop Hoisting
  - Loop Overhead
- Other Handling
- Use of operators

Use Interrupts and utilize low power modes

Data Type usage

- The use of correct data type is important in a recursive calculation or large array processing.
- The extra size, which is not required, is taking up much space and processing time.

- Example:
  - If your variable range is varying in between 0 to 200 then you should take variable type of unsigned char.

  - unsigned char count = 0;

- Use unsigned types for:
- Division and remainders
- Loop counters
- Array indexing
- Use signed types for:
- Integer-to-floating-point conversion

Usage of inline functions

- Do not use preprocessor directive #define to obtain more efficient code, instead use inline functions.

```
#define SQUARE( x ) (( x ) *  ( x ))
int     a = 2;
intb = SQUARE( a++ );    //b = ( 2 *  3 ) = 6 !!!
```

```
inline
int     square( int a_x )
{
return ( a_x *          a_x );
};
int c = 2;
int d = square( c++ ); ////d = ( 2 *  2 ) = 4
```

- *Inline functions* are faster because you don't need to push and pop things on/off the stack

Inline Vs Macro functions

- Macro functions shall be used only in parts related to speed performance as inline is implementation or compiler dependent

```
extern void func1(int,int); /* func1: called only once */

#define func2(arg1, arg2) /* func2: called many times */  func1(arg1,

arg2);

for (i = 0; i < 10000; i++) {

func2(arg1, arg2); /* Speed performance is critical for this process. */

}
```

Macro expansion increase the objectsize

- The technique will cause the compiler to replace all calls to the function, with a copy
- of the function's code.
- This will eliminate the runtime overhead associated with the function call.
- This is most effective if the function is called frequently, but contains only a few lines of code.
  - Important: In-lining large functions will make the executable too large.
    - Trade off between speed Vs code optimization

Variables and Address based optimizations

- Referencing a global variable by constant address requires more instructions, while referencing the same variable through a pointer requires less. Even on architectures with full-address-precision addressing modes, it may be faster to use a pointer and offset.
- int_Global_pool[3]; // Global pool to hold 3 integer variables
- int *_Global_ptr= Global_pool[0]; // Declare pointer to Global pool

```
void function_access( )
{
* __Global=3;
*(__Global_ptr+1)=5;
*(__Global_ptr+2)=10;



return;
}
```

Avoid Type Conversion

- Programmers should plan to use the same type of variables for processing. Type conversion must be avoided.
- Otherwise, precious cycles will be wasted to convert one type to another (here Signed and Unsigned variables are considered as different types.)
- Const Keyword
- The “const” keyword is to define the data as a constant, which will allocate it in the ROM space. Otherwise a RAM space will also be reserved for this data.
- This is unnecessary as the constant data is supposed to be read-only.

Compiler optimization and the volatile keyword

- Mostly, higher optimization options in compiler optimize the variable if we have missed volatile qualifier.
- When a variable is not declared as volatile, the compiler can assume its value cannot be modified in unexpected ways.

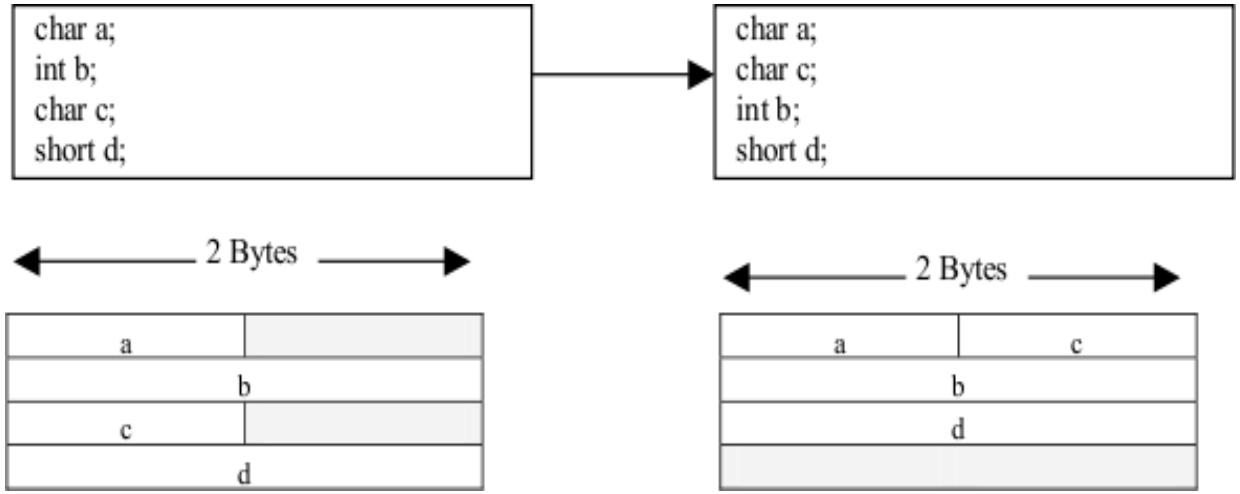
When to declare variable as volatile ?

- writing multi-threaded programs
- accessing variables inside ISR's or signal handlers
- Accessing memory mapped peripherals

Source: http://www.keil.com/support/man/docs/armcc/armcc_chr1359124222941.htm

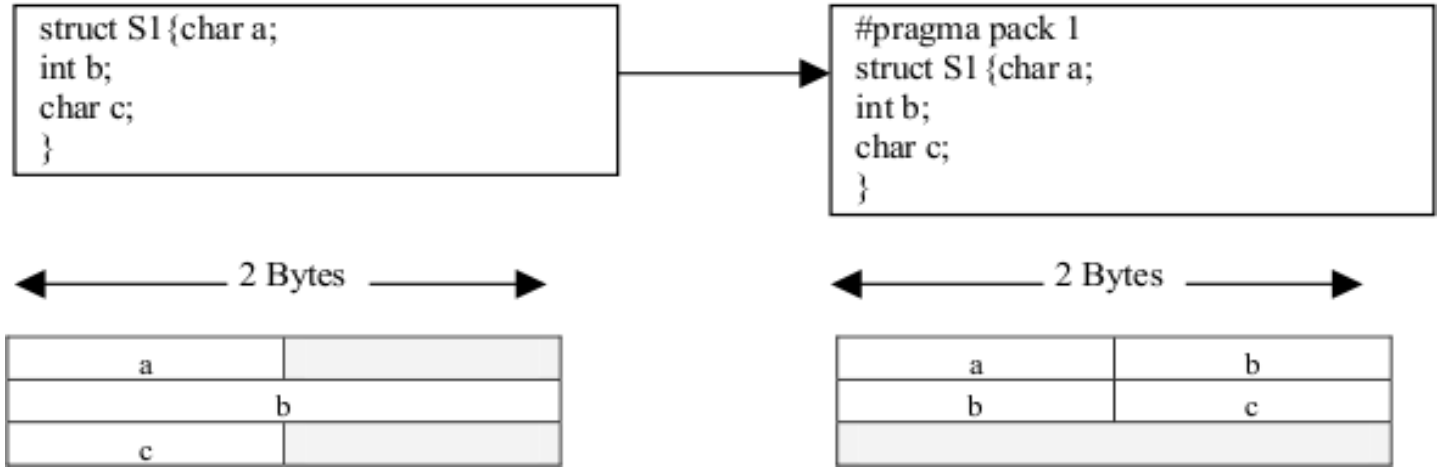
Data Alignment :Arrangement and Packing

- The declaration of the component in a structure will determine how the components are being stored.
- Due to the Memory alignment, it is possible to have dummy area within the structure. It is advised to place all similar size variables in the same group.



Structure Padding

- As the structure is packed, integer b will not be aligned. This will improve the RAM size but operational speed will be degraded, as the access of ‘b’ will take up two cycles.



Pass by Reference

- Larger numbers of parameters may be costly due to the number of pushing and popping actions on each function call.
- It is more efficient to pass structure reference as parameters to reduce this overhead. **Return Value**
- The return value of a function will be stored in a register. If this return data has no intended usage, time and space are wasted in storing this information.
- Programmer should define the function as “void” to minimize the extra handling in the function.

```
total (long a, long b, long c, long d);
```

```
struct sum{
long a;
long b;
long c;
long d;
}all;

total (&all);
```

Instead of structures, pointers to structures shall be used as function parameters.

Compliant Example

```
typedef struct stag {
int mem1;
int mem2;
…
} STAG;
int func (const STAG *p) {
return p->mem1 + p->mem2;
}
```

Non- Compliant Example

```
typedef struct stag {
int mem1;
int mem2;
…
} STAG;
int func (const STAG p) {
return p.mem1 + p.mem2;
}
```

Switch & Non-Contiguous Case Expressions

- Use if-else statements in place of switch statements that have noncontiguous case expressions.
- If the case expressions are contiguous or nearly contiguous integer values, most compilers translate the switch statement as a jump table instead of a comparison chain.
- Example of the rule:
- ***switch statement shall be used instead of if statement when:***
- a process branches according to the value of the expression (integer value), and the number of branches is three or more.
- However, this rule shall not apply if:
- - using the switch statement causes an efficiency issue that impedes the improvement of program performance.

```
switch (i) {
   case 1: Task1; break;
   case 2: Task2; break;
   case 3: Task3; break;
}
```

```
void Task1() { }
void Task2() {  }
void Task3() { }

typedef void (*pfunc)(void);

pfunc functions[3] = {Task1, Task2, Task3};


if ((unsigned)i<3)
    functions[i]();
```

Loop Unrolling

- Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
- If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

```
// old loop
for(int i=0; i<3; i++)
{
        color_map[n+i] = i;
}
```

```
// unrolled version
int i = 0;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
```

Portability

- Various compiler options for unrolling the loops:
- For GCC
- #pragma GCC optimize (“unroll-loops”)
- For keil:
- #pragma unroll or
- #pragma unroll (n)

- where n is the number of iterations to unroll.

Loop Hoisting

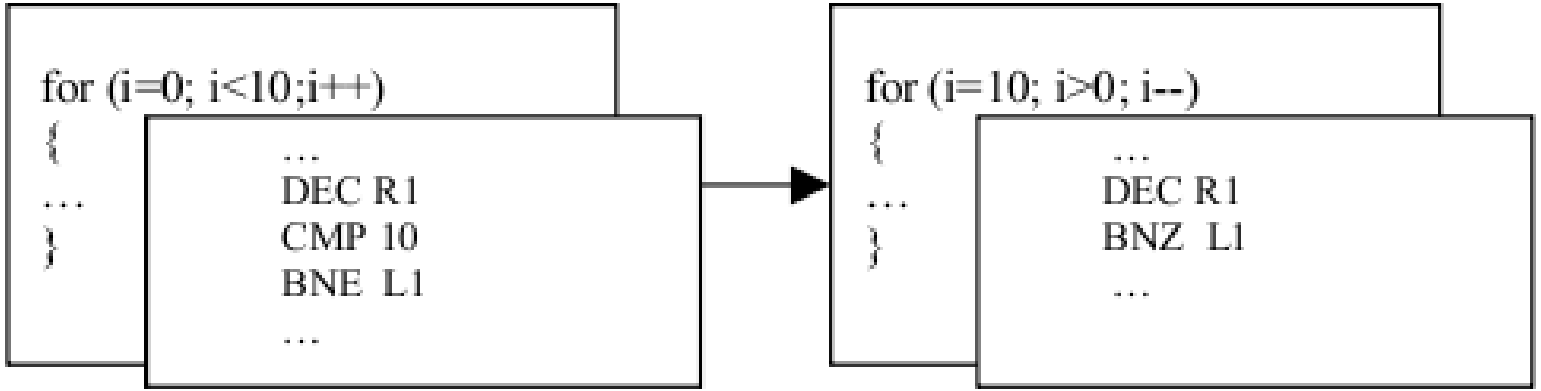
- Moving computations outside loops
- Saves computing time

```
/* Old code */
for (i....)
{
        if (CONSTANT0)
        {
                DoWork0(i);
        }
        else
        {
                DoWork1(i);
        }
}
```

```
/* Optimized Code */
if (CONSTANT0) {
        for (i....)
        {
                DoWork0(i);
        }
}
else {
        for (i....)
        {
                DoWork1(i);
        }
}
```

Loop Overhead

- The MCU have a conditional branch mechanism that works well when countingdown from positive number to zero.
- It is easier to detect “Zero” (Branch non zero – single instruction) than a “predefined number” (Compare and perform jump – two instructions).



Use of operators

```
/* Old Code */
int i, j;
if (i > CONSTANT)
{
        j = 1;
}
else
{
        j = 0;
}
```

```
/* Optimized Code */

int i, j;
j = (i > CONSTANT) ? 1 : 0;
```

Use Formula

- Use formula instead of loops
- What is the formula for sum of naturals numbers?
- Example

```
n=100;
for (x=0, y=1; y<=n; y++)
        x +=y;
```

→

```
n=100;
x = n* (n >>1);  //n^2 /2
```

Example 1: Data size/type Effect on size optimization

- Run optimizer assistant and see the variation in size of code.
- Check Memory allocation for whole program

Analyse Effect with

char_____8-bit

unsigned int ------16-bit

long _____32-bit

Example 2: Local and Global variable Effect on size optimization.

- const unsigned int PORT_value = 0x1234;

- Make it global and check for code size

- Compare code size while declaring it local

Example 3: Loop based optimization

- If we use a loops, an increment or a decrement loop index generates different code size.
  - It is more efficient to count the loop from the maximum value to zero (decrement).

- Run optimizer assistant and see the variation in

size of code

- Check Memory allocation for whole program

Embedded Microprocessor Benchmark Consortium (EEMBC)

- Role of benchmarks:
- Study Impact of memory bottlenecks, efficiency of thread synchronization w.r.t multicore processors.
- Microcontroller benchmarks that focus on energy consumption rather than performance is still inprogress.
- Presently, MCUs from different vendors have their own approach to sleep state and have varied wake up time from low power.

- The Embedded Microprocessor Benchmark Consortium (EEMBC) has released version 2.0 ofits suite for measuring the performance of typical automotive powertrain tasks on multicore processors.

Thank You !

L&T Technology Services