μP Tutorial 1

Introduction to the lab

IDE, first program and lab policy

By: Ashraf Suyyagh

Edited By: Amir Shahshahani

(Sept 2017)

Outline

- Introduction to ARM SoC, Development Boards and SW tools
- Introducing Keil uVision IDE, Notes on installing Keil SW packs and drivers
- ISA and assembly learning resources
- Setting up the IDE for projects
- Writing first assembly program in Keil IDE
- Debugging tools
- Lab Demos and Report Submissions, Grading

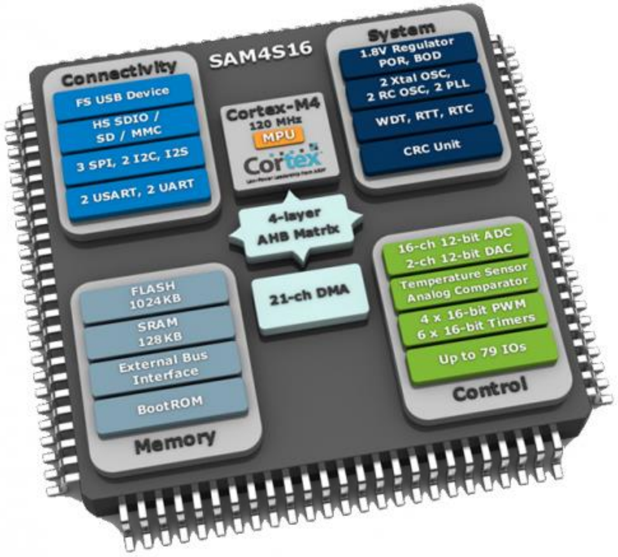
What is a Microcontroller (µC, or MCU)?

- IP core intended for microcontroller applications
- Cortex-M0, M3, **M4**
- In this lab, we will be using the Discovery F4 Board (Lab2+)

- ARM 32-bit Cortex™-M4 CPU with FPU (STM32F407VG)
- Up to 1 Mbyte of Flash memory
- Up to 140 I/O ports with interrupt capability
- Up to 15 communication interfaces
- Advanced connectivity
- General-purpose DMA
- 3×12-bit A/D converters
- 2×12-bit D/A converters
- And much more...

Integrated Circuit (SoC)

- Processor Core
- Memory
- Programmable Peripherals including I/O



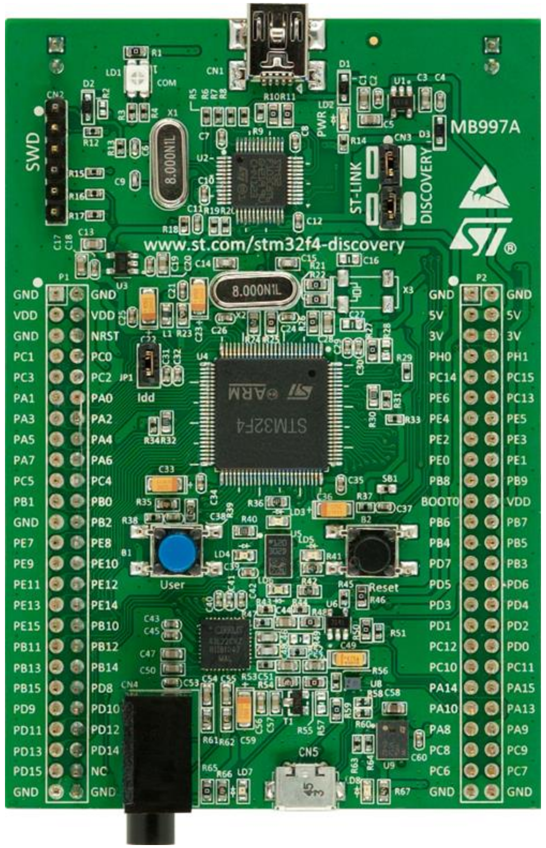
Why ARM?

- **ARM** is a **reduced instruction set computer** (RISC) instruction set architecture (ISA) developed by ARM Holdings.
- ARM does not manufacture its own CPUs. **IP Licensing**

  (Texas instruments, Analog Devices, Atmel, Freescale, Nvidia, Qualcomm, ST microelectronics have all licensed ARM technology).
- Annual shipments estimates of ARM processors are at 7 billion per year [1]. Almost quarter of the total embedded market share is ARM's. ARM leads in 32 bit embedded Market with the most share[2]
- Has three major µC families (profiles): A, R and M

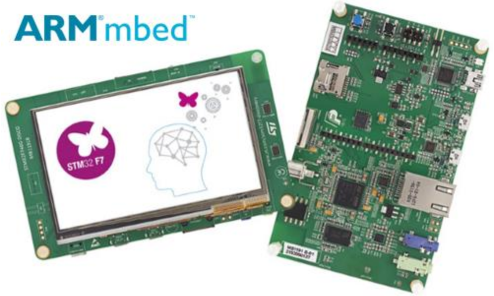
Development Boards

Our main board: F4 Discovery (≈ 17$CAD)



ST 32F429I DISCOVERY
(30$)

ST F3 DISCOVERY
(12$)



ST F7 Discovery
(50$)

CORTEX-M

(Embedded applications)

Raspberry Pi 2
(56$)

BeagleBone Black
ver C(55$)

Odroid-XU: A Samsung Exynos Octa core, with A15 and A7 quadcores in Big.Little configuration (139 – 199$)

CORTEX-A

(Embedded Multimedia and generic applications)

Getting Started with software

You need to download the following SW on your machine OR you can work in the lab (open 24x7)

- Keil uVision IDE Ver **5.xx** (Free download with 32k limitation, enough for labs but may not project)

  http://www.keil.com/arm/mdk.asp

- You can use 5.16a 5.22 5.24(Latest on ARM site)

  ✓ ✓ ✓

CMSIS-Core **5.0** and STM32F4xx_**DFP 2.8(Cube),2.11 works on some project formats,** Driver Packs and MDK-Middleware **7.0.0 or 7.2.0**

- ST LINK V2 driver (for connecting to the board, programming and code debugging)

  http://www.st.com/en/embedded-software/stsw-link009.html

So what are we getting from the SW?

Two modes, different screens and menus (Similar to Eclipse perspectives)

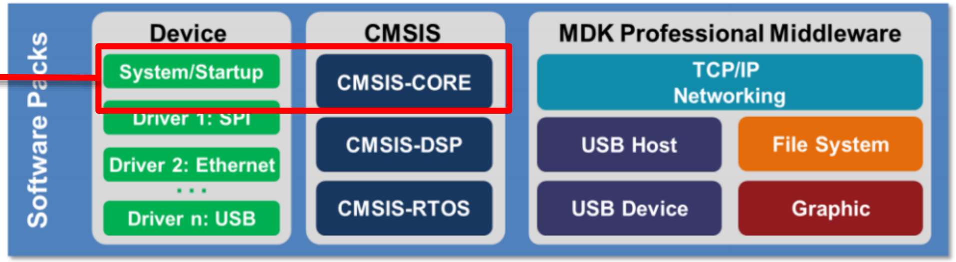
MDK Core

The MDK Core includes all the components that you need to create, build, and debug an embedded application for Cortex-M processor based microcontroller devices. The Pack Installer manages Software Packs that can be added any time to the MDK Core. This makes new device support and middleware updates independent from the toolchain.

Software Packs

Software Packs contain device support, CMSIS libraries, middleware, board support, code templates, and example projects.

System Startup File and CMSIS-Core must be in every Project

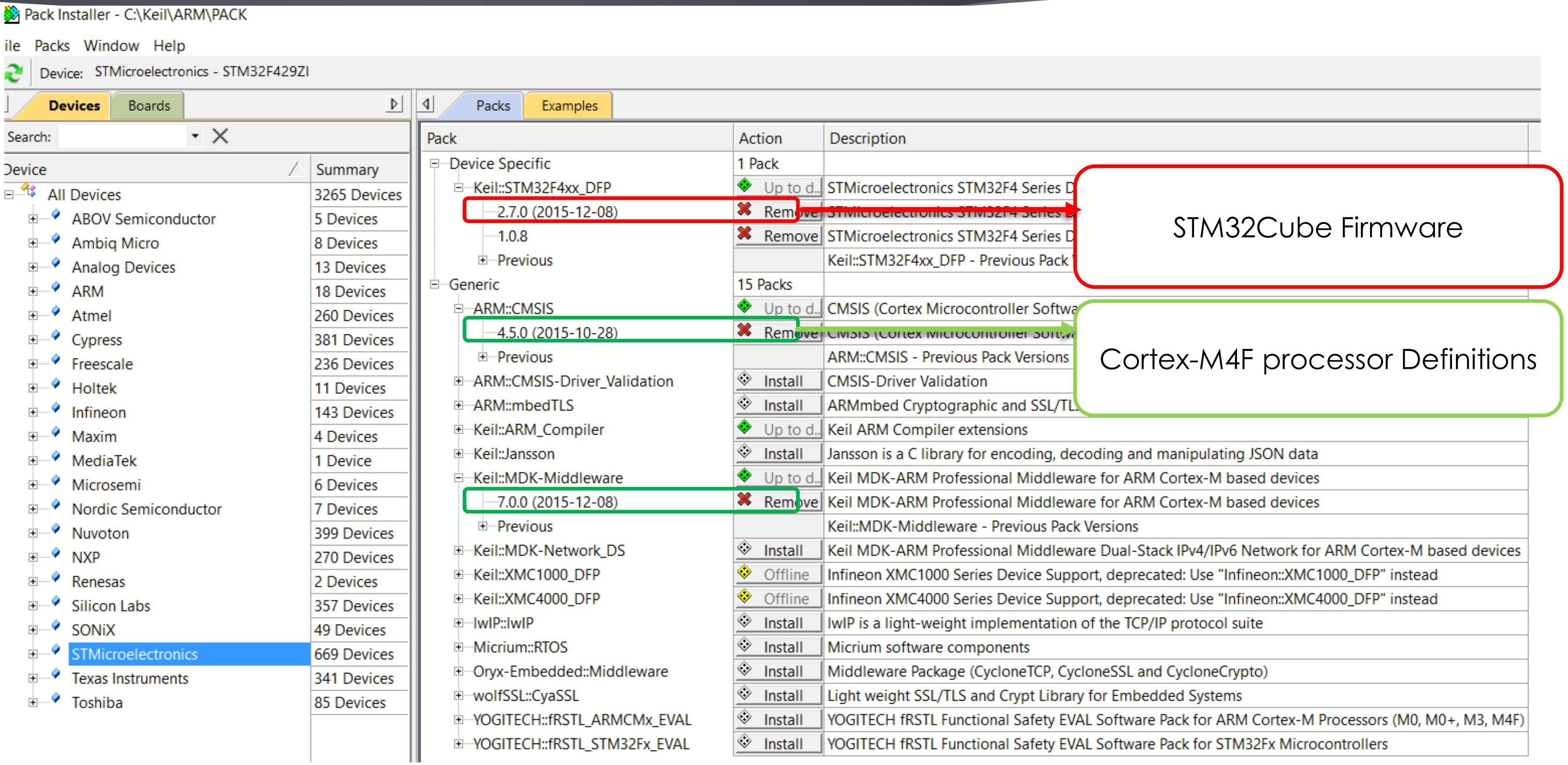


Keil Software Packs – Pack Installation

At the end of Keil uVision IDE installation, the Pack Installer window will show up.

Make sure you have the following versions (or newer):

STM32F4xx_DFP: 2.7.0
ARM CMSIS: 5.0.0
ARM Middleware: 7.2.0



Resources to learn more on ISA and assembly?

Resources on Instruction Set Architecture (ISA)

- **Cortex M4 Programming manual (uploaded to myCourses):**
    1. Must read datasheet (Sections 2.1 – 2.4)
    2. Section 3 details each instruction with example code and specific uses (Read subsections as needed)
- **ARM and THUMB 2 Instruction set (uploaded to myCourses):**

  Lists all available assembly instructions in the Cortex-M4F core with brief descriptions.
- ***The Definitive Guide to ARM Cortex M3/M4 Processors (Book available online through McGill library):***
    1. CH4 Architecture(4.2.2, 4.2.4, 4.3, 4.4) covers the register set and status registers in details
    2. CH13 Floating point(13.1.2, 13.2.4, 13.2.5, 13.4.5), Covers all we need about FP

Resource on writing assembly codes (syntax, keywords, rules, errors)

- ARM Real view assembler Guide (All you need about Assembly language)

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/index.html

Setting up your projects

We will always provide you with pre-configured base projects for all lab experiments. However, it is imperative to understand what these options are as you might need to start your own projects, debug issues or change settings.

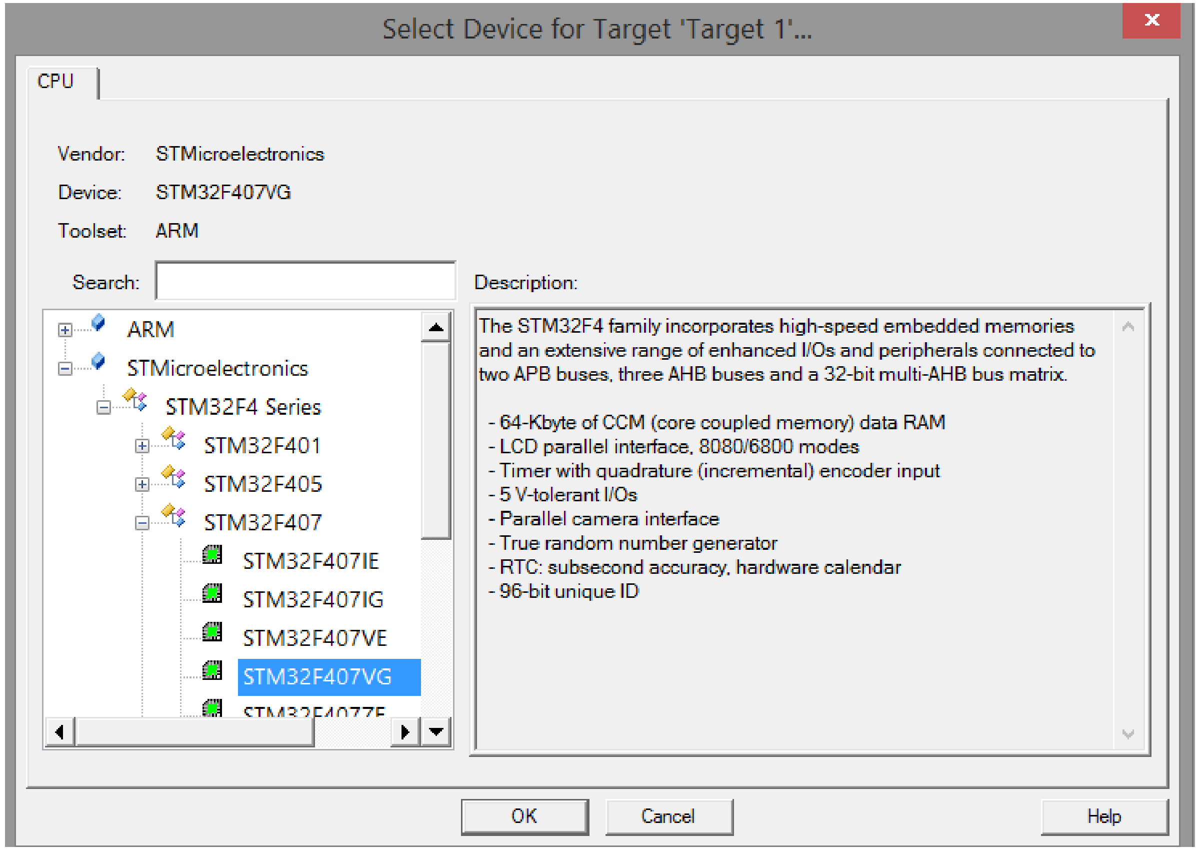
This following slides are just basics

To understand every single option in these upcoming screens and all what there options are, please consult these two references which you can find in my Courses:

1. Introduction to Keil (Getting Started Guide) – 95 pages
2. Debugging with Keil – 32 pages

Introducing Keil uVision IDE #1
Creating a project

- Start Keil
- Project → New uVision Project and save it with the name of your choice
- In Select Device Target box, from STMicroelectronics menu, choose STM32F407VG as the target processor



Introducing Keil uVision IDE #2

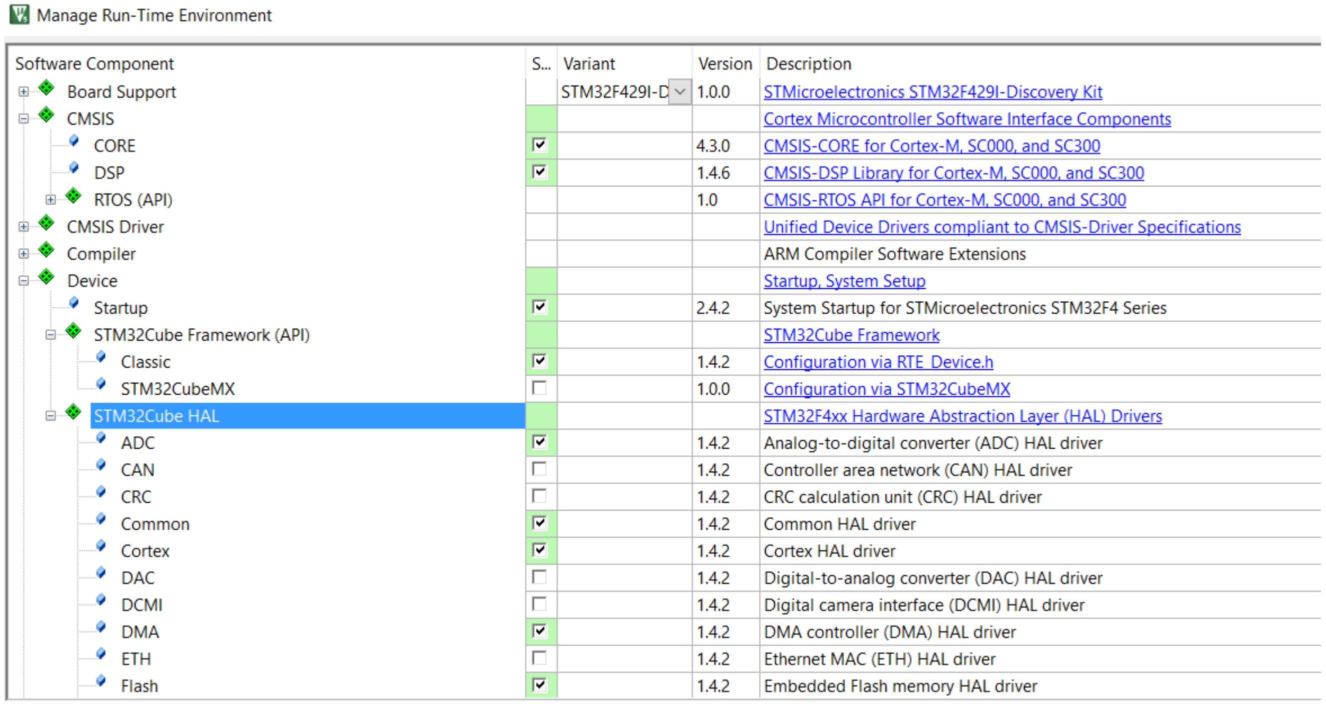
Choosing SW components

In the next screen, we need to select the required driver and startup files for our first assembly project

- From **CMSIS** group → Select **Core**
- From **CMSIS** group → Select **DSP**
- From **Device** group → Select **Startup**

After you click OK, the main IDE window will show up with few Startup files.

IMPORTANT: Future labs will require additional SW components and drivers to be checked as shown in the screen



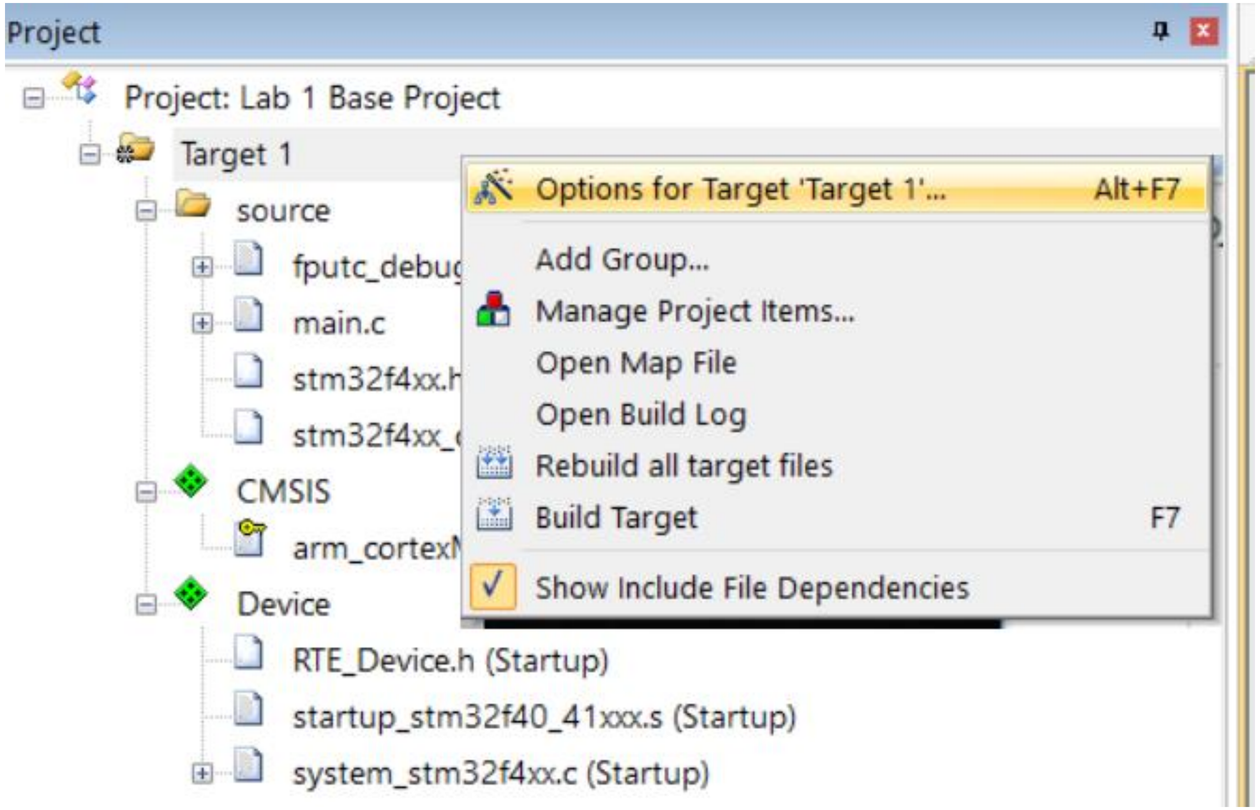
Introducing Keil uVision IDE #3

Organizing your project

- To the left side, you will find the project pane where you have access to all your source files and project settings.
- Give your target a name by renaming it.

  → Select and single click to rename
- You can add multiple folders where you can group your files into categories (main files, drivers, libraries, ..etc)

  → Right click and "Add Group"
- To access Project settings , right click on project name and choose options



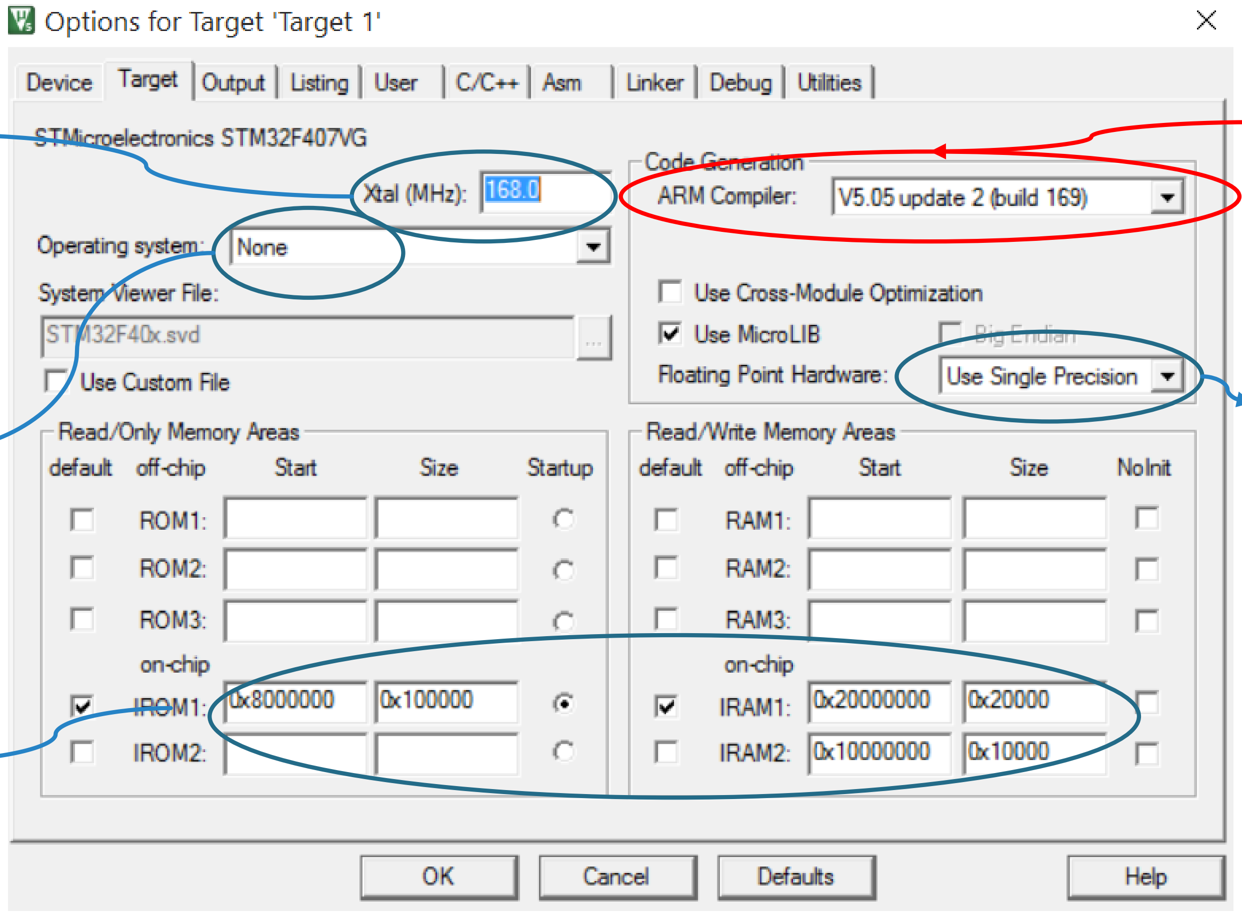
Introducing Keil uVision IDE #4

Project Options / Target

Choose the processor speed. The Cortex-M4 runs at a maximum of 168MHz

In the RTOS experiment, we will choose the RTX Kernel option

Do not change these settings AT ALL!



You can choose the default version.

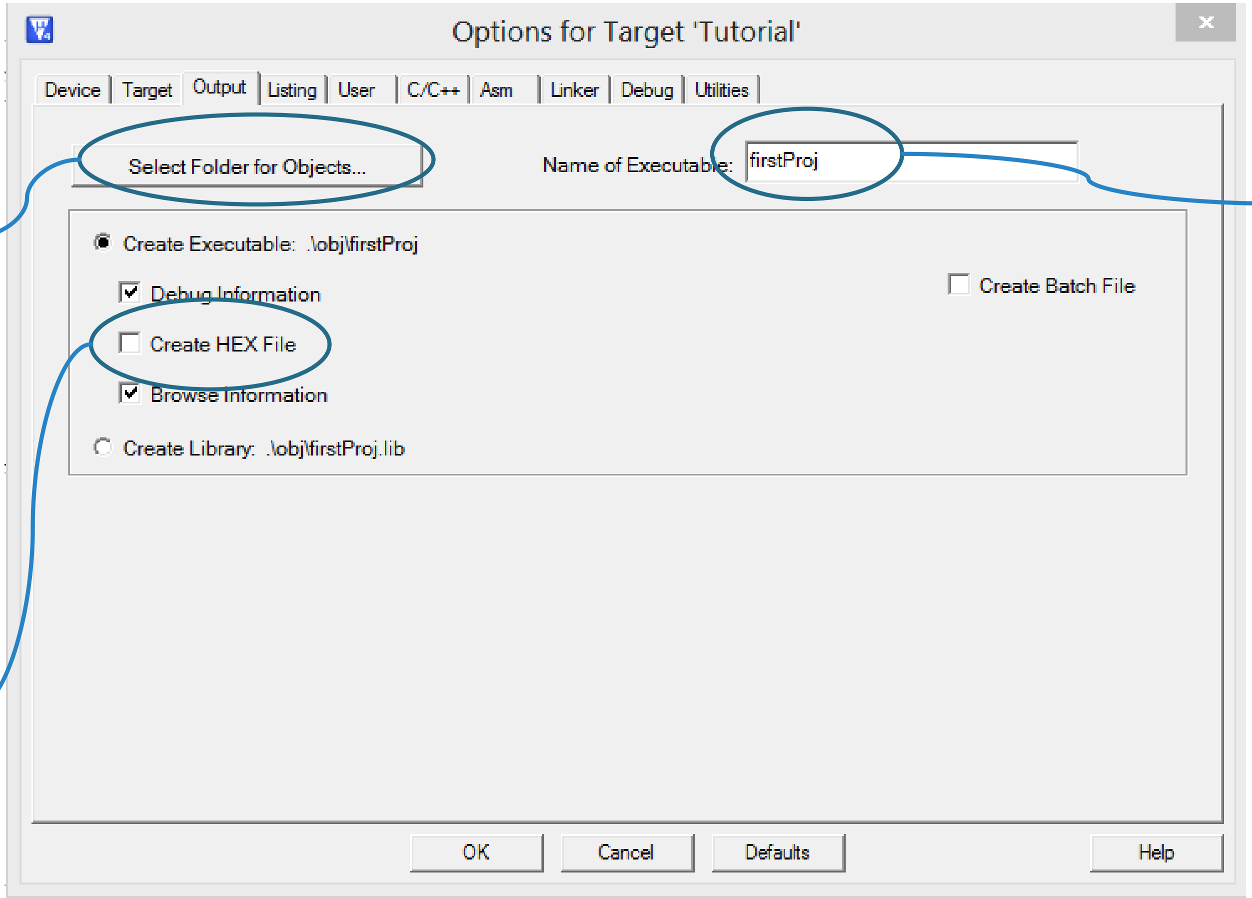
Enables the use of the Cortex M4 hardware FPU unit for accelerated FP performance.

Yet, if your project is purely assembly, you need to enable the FP unit manually by writing instructions. (Later in the slides)

Introducing Keil uVision IDE #5

Project Options / Target

Use this to create a folder named "obj" and select it. All object files (.o) files of your project will be stored there making your project more organized. Do the same in the Listing screen



Select the name of your output executable

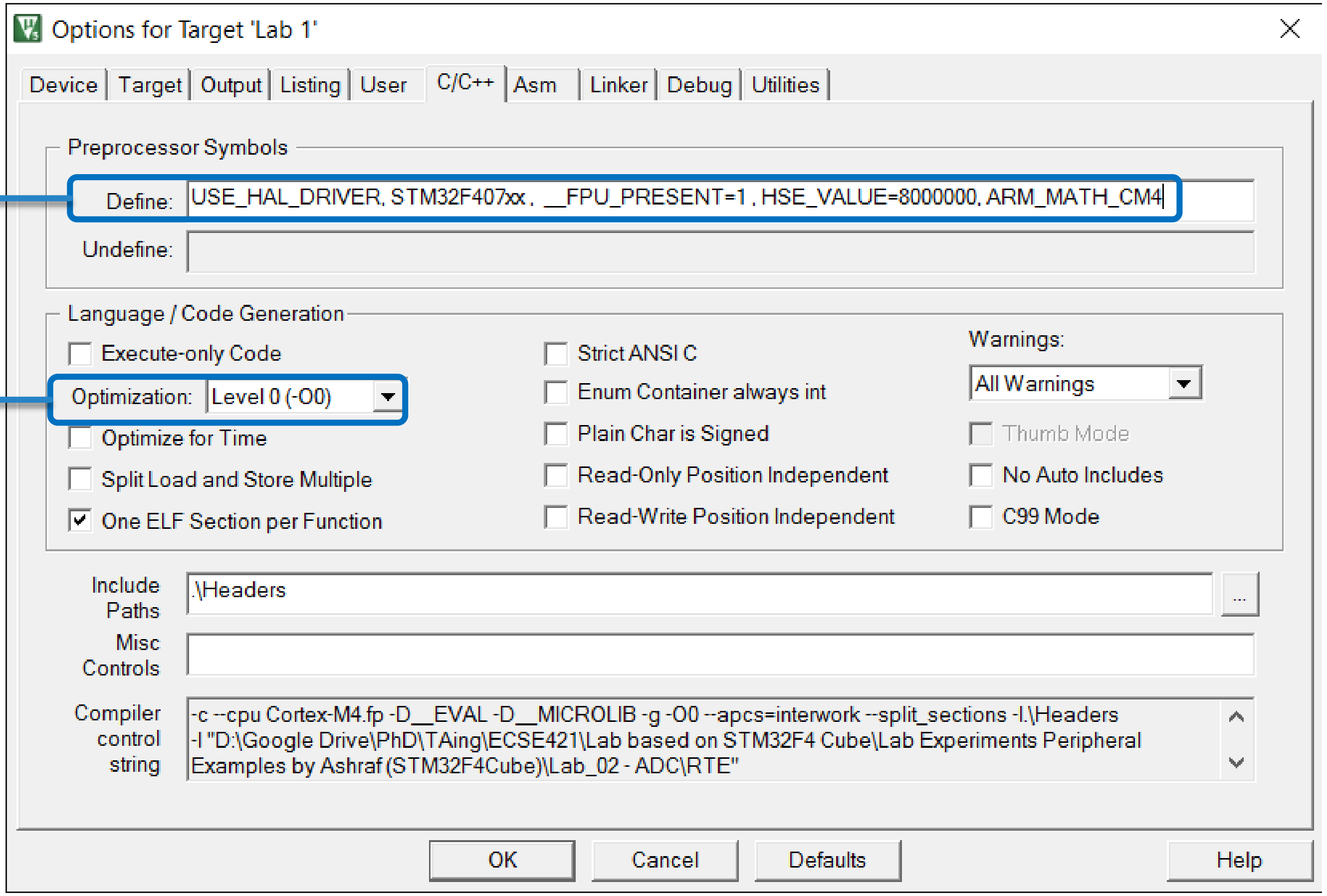
By default, your executable is of the .elf format. Use this to create a hex file instead if needed.

Introducing Keil uVision IDE #6
C/C++ Screen

Always add these preprocessor definitions to set up the oscillator clock, enable use of any drivers, enable the use of the DSP Math functions and state that we are using the FPU hardware unit respectively:

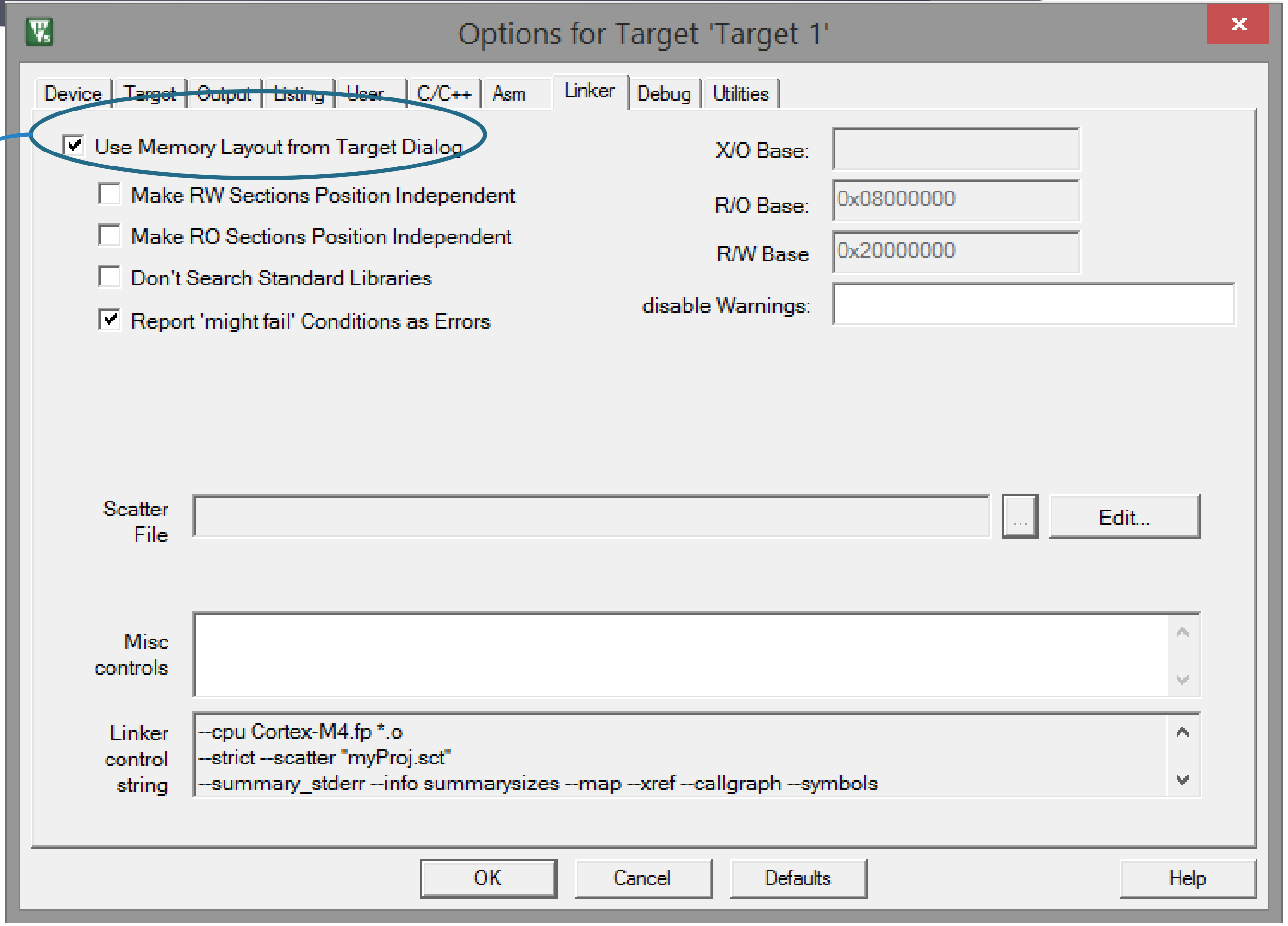
- HSE_VALUE=8000000
- USE_HAL_DRIVER
- STM32F407xx
- ARM_MATH_CM4
- __FPU_PRESENT = 1

In embedded development, we usually choose Level0 (-O0) optimization, which is basically no optimization!. You can choose higher levels of optimization.



Introducing Keil uVision IDE #7
Project Options / Linker

Check "Use Memory Layout from Target Dialog". This allows the creation for data and code memory regions from the memory range specified in the Target tab. Otherwise, you have to manually write a scatter file and load it. This is beyond the scope of this lab.

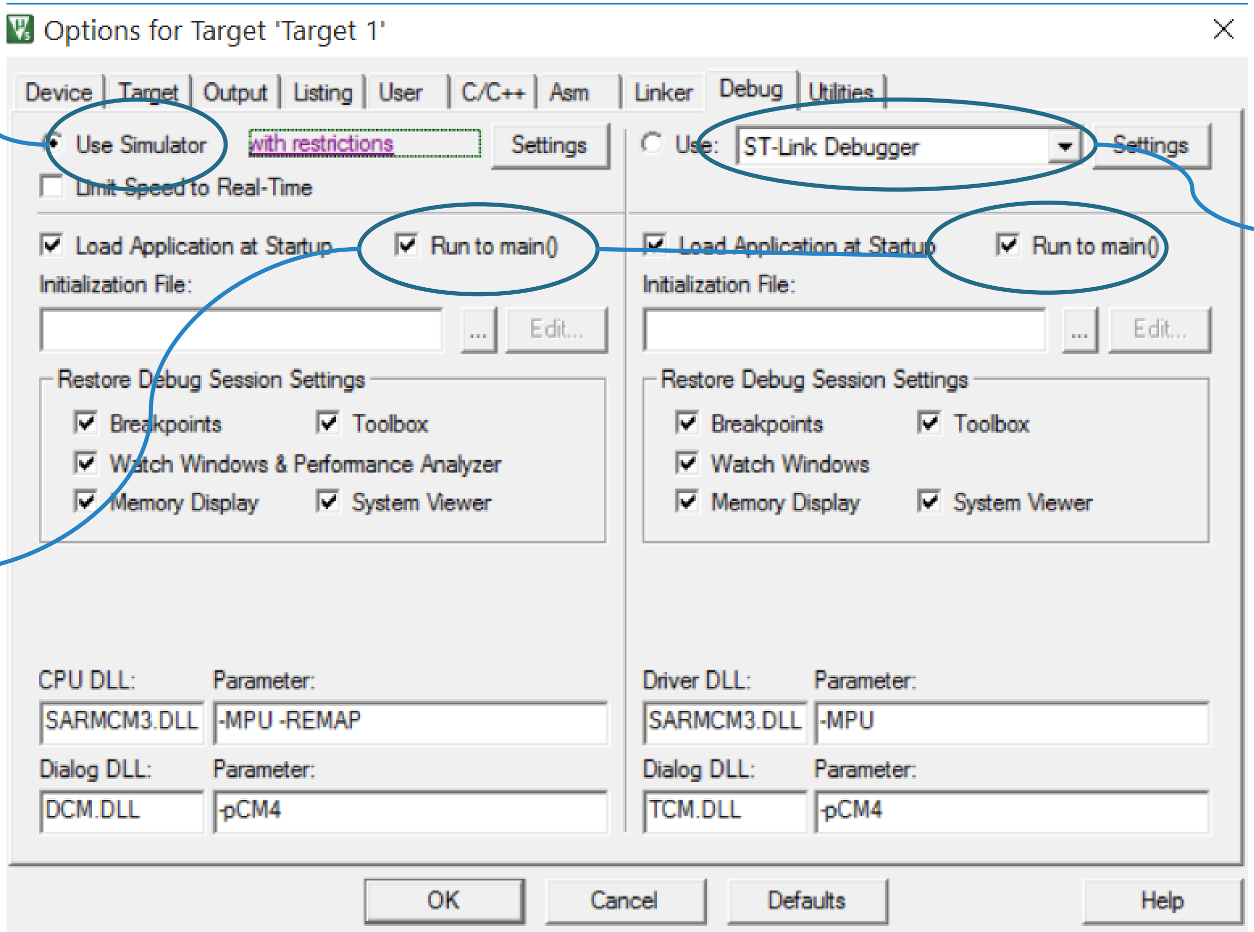


Introducing Keil uVision IDE #8

Project Options / Debug

For the first experiment, we will use the simulator, so keep this checked.

If we have a C program, debugging will immediately start from main program. If not checked, it will start from the reset vector and show you the SystemInit and other Reset_Handler code



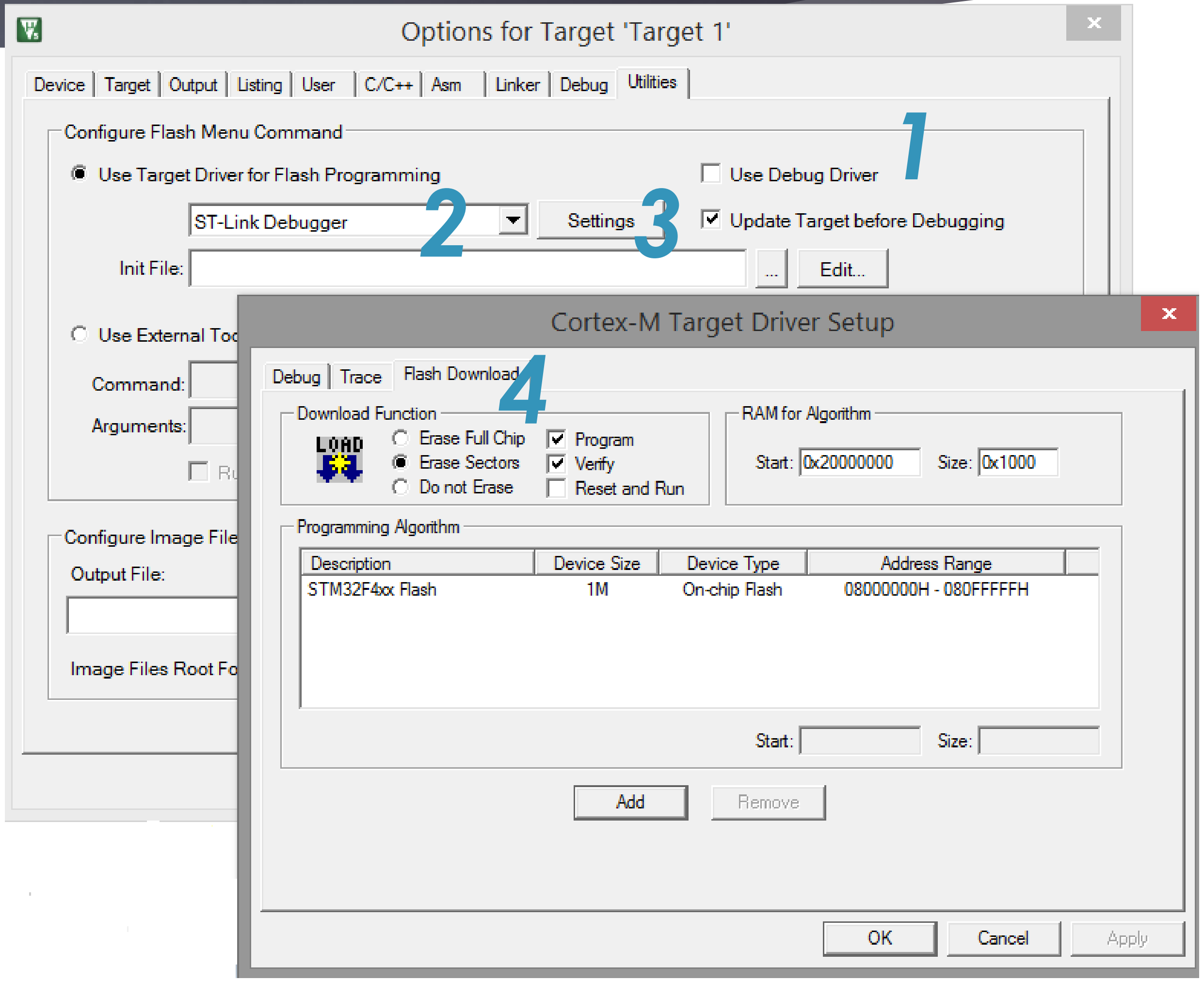
Once we start using the board, we will be using the HW debugging options. We will choose ST_Link/V2 debugger/programmer

Introducing Keil uVision IDE #9

Project Options / Utilities

In the Utilities Screen (Lab 2 onwards):

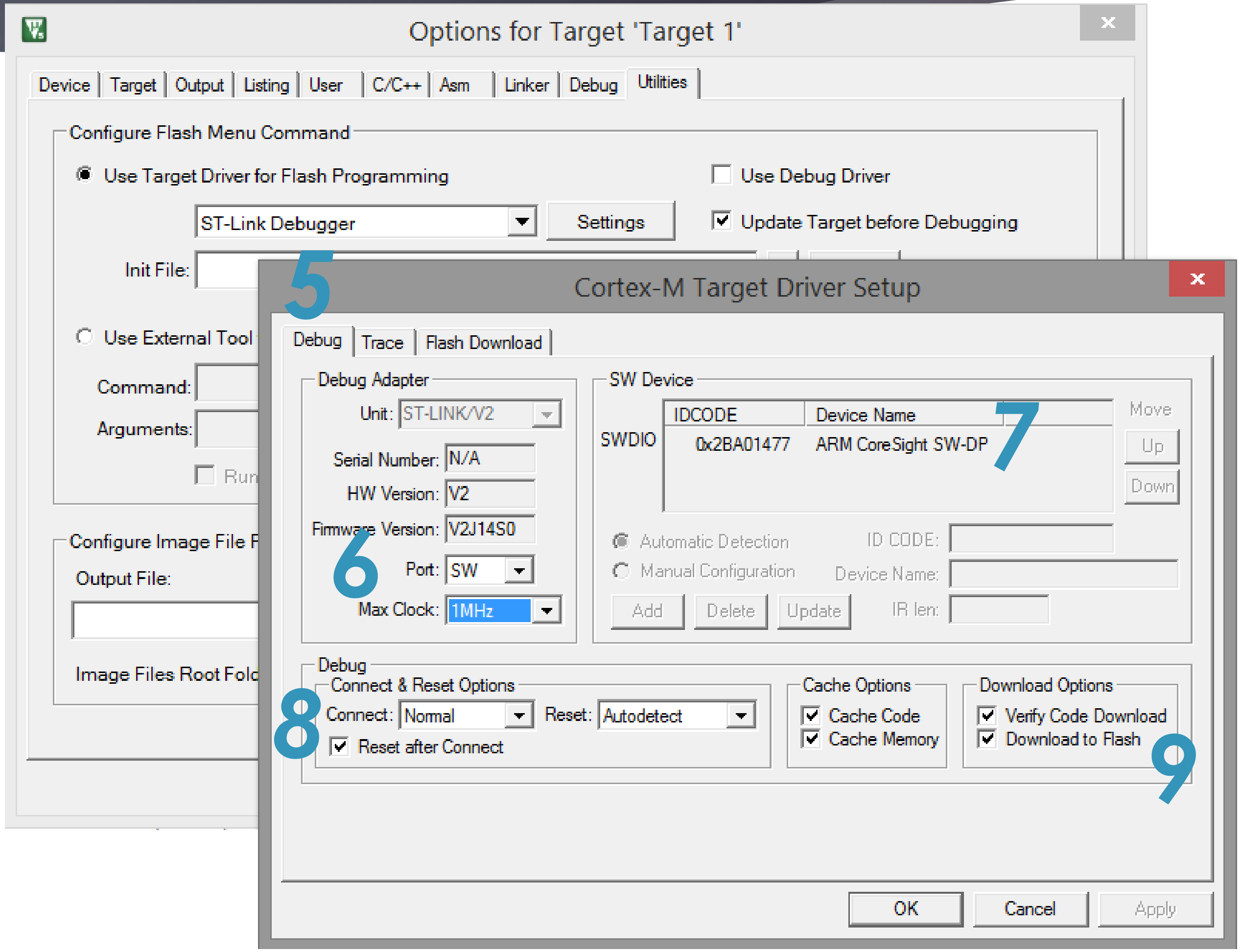
1. Uncheck *"Use Debug Driver"*
2. Choose **St-Link Debugger** from the *"Use target Driver for Flash Programming"* drop down list
3. Click Settings
4. Make sure that **"STM32F4xx"** Flash is added with Device size of **"1M"**. If not, click on the **Add** button to select it



Introducing Keil uVision IDE #10

Project Options / Utilities

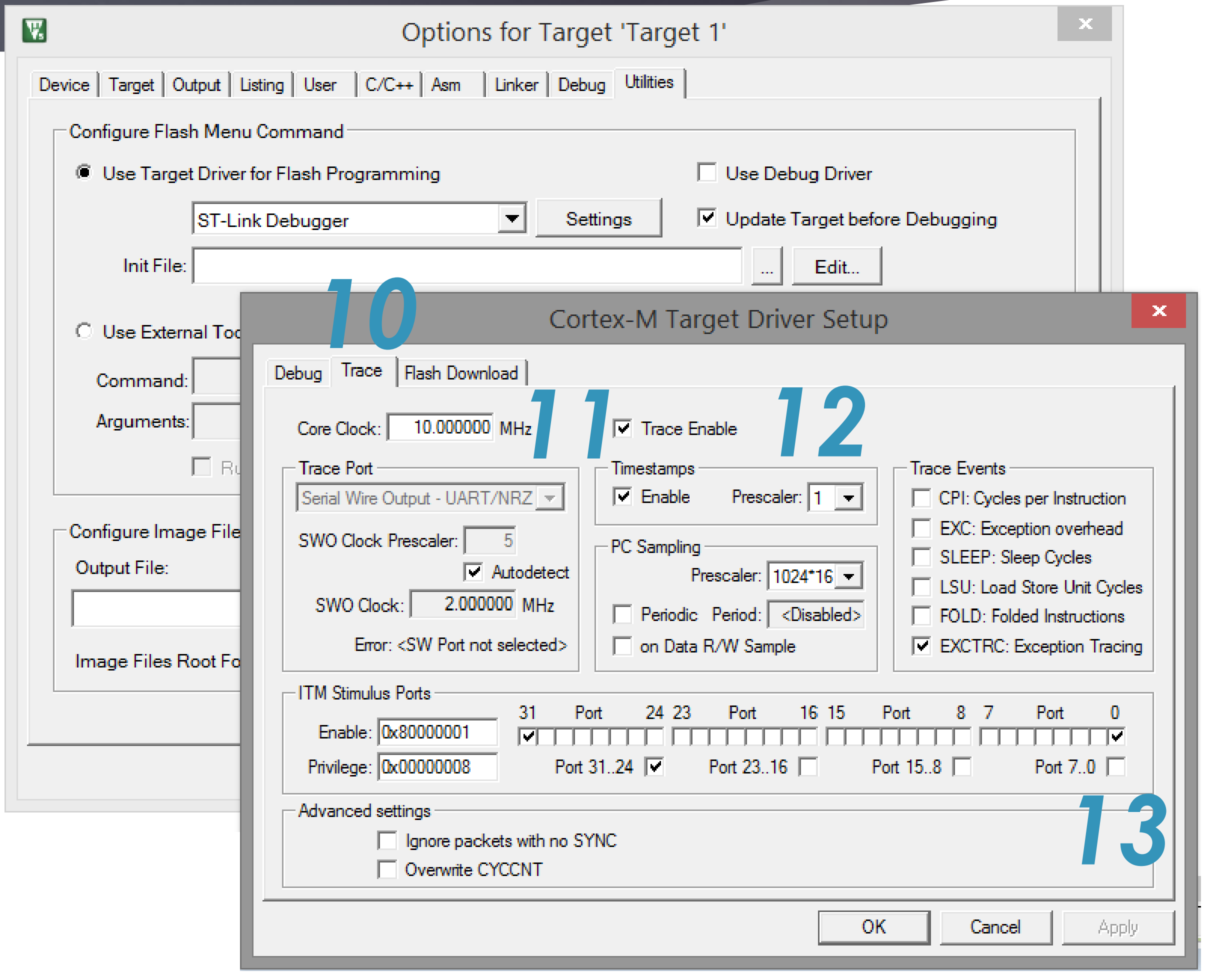
5. Switch to the **Debug** Tab
6. For **Port**, choose "**SW**", which is for ARM's ***Serial Wire Debug "SWD"*** Protocol
7. If the board is already connected and ST-Link driver correctly installed, then you should see that the IDE detects the board, this shows by displaying the device name as **ARM Core-Sight**
8. For connect and reset options, choose **Normal**, **Autodetect** and Check "**Reset after connect**" as shown in the figure
9. Make sure that **Verify Code** and **Download to flash** options are selected.



Introducing Keil uVision IDE #11

Project Options / Utilities

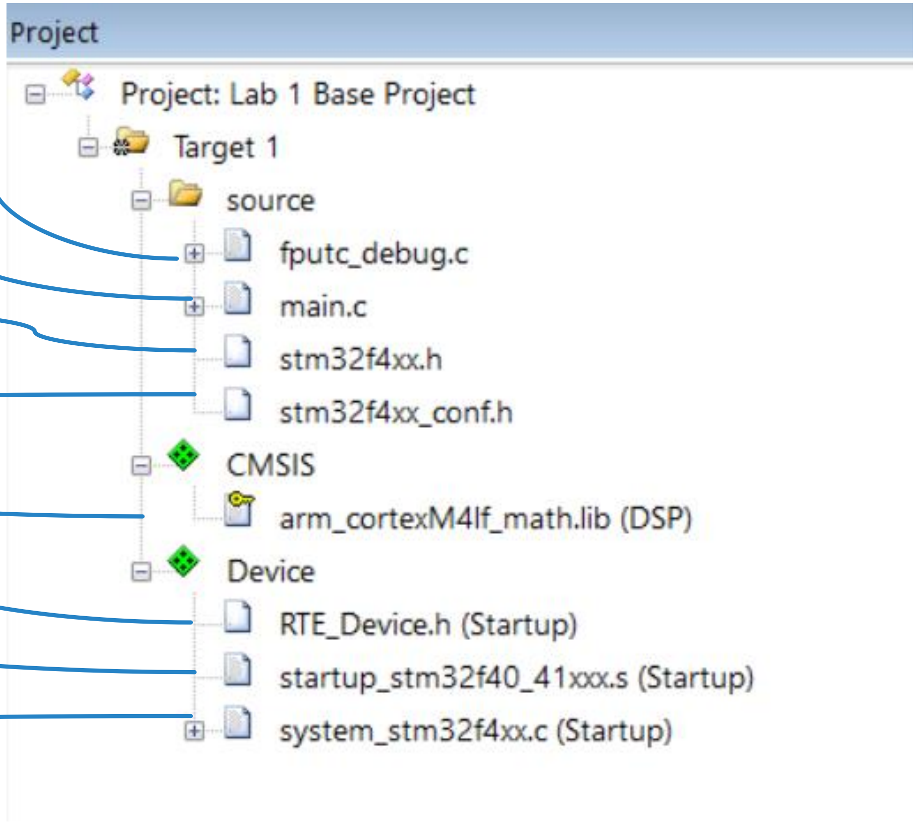
10. Select the **Trace** tab
11. Set Core clock to 168MHz, if you do not do this, you will see communication errors or overflows
12. Check **Trace Enable**
13. Make sure that ITM Stimulus *Port 0* and *Port 31* are checked
    - ***Port 0:*** Allows for **printf-style** debugging (Lab 2 Onwards)
    - ***Port 31:*** Allows to Debug RTOS events (Lab 4 onwards)
    - Leave all other options as is



Our First Assembly Program

The base project

- Enables the use of "printf" statements in C when using the HW (Lab 2+)
- Your C code goes here
- Part of the CMSIS core files, defines processor registers, options .. Etc.
- Selects which drivers APIs to be compiled (Lab2+)
- The CMSIS DSP library functions
- Peripherals and registers definitions (Lab2+)
- Assembly startup file (Lab1)
- C startup file (has the implementations of functions called in the assembly startup file)



Starting up with Assembly Programming #1

The startup file (startup_stm32f40_41xxx.s)

This file sets up the stack and heap memory sizes. Also the exception and interrupt vectors.

Our starting point is the Reset_Handler? Why?

If you are no expert, don't change anything in this file unless told to

How the default reset handles looks like:

```
Reset_Handler    PROC
      EXPORT  Reset_Handler          [WEAK]
      IMPORT  SystemInit
      IMPORT  __main
      LDR     R0, =SystemInit
      BLX     R0
      LDR     R0, =__main
      BX      R0
      ENDP
```

- Using = before the function/procedure name means you need to load the starting address of that function. Failing to include the = will result in not branching to your subroutine or going somewhere else.
- Difference between BLX and BX is that BLX stores the address of the next instruction in LR While BX does not. Be careful which to use. The first is similar to a call while the other is a jump

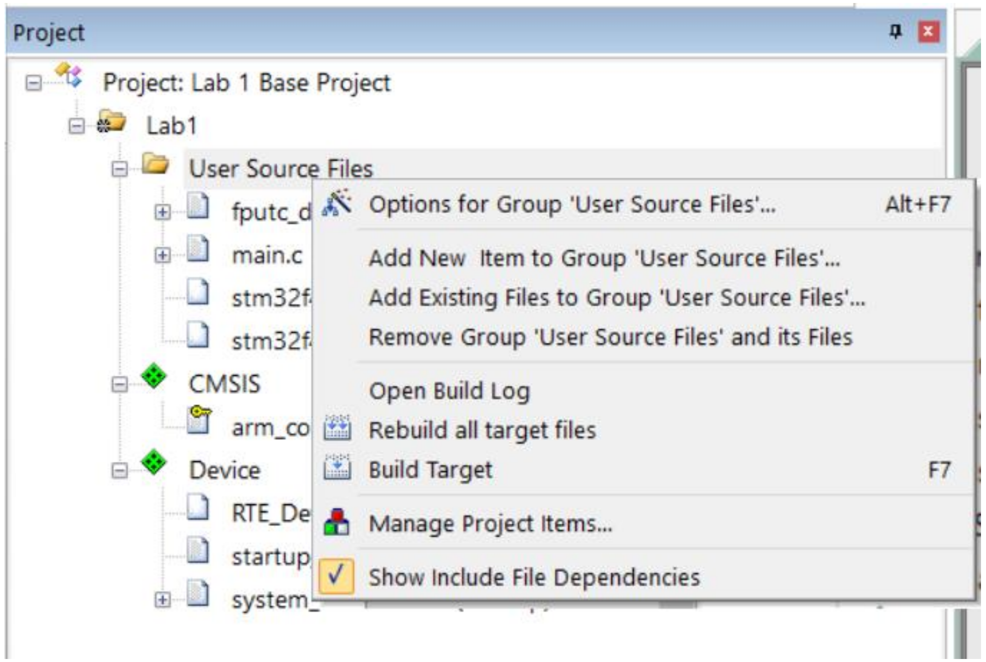
When you compile your code, if you receive an error Undefined symbol __use_two_region_memory (referred from startup_stm32f40xx.o), simply comment the line IMPORT __use_two_region_memory at the end of the startup file by placing a ; at the beginning

Learn more?
Assembler directives, EXPORT, IMPORT, and more
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Babeagih.html

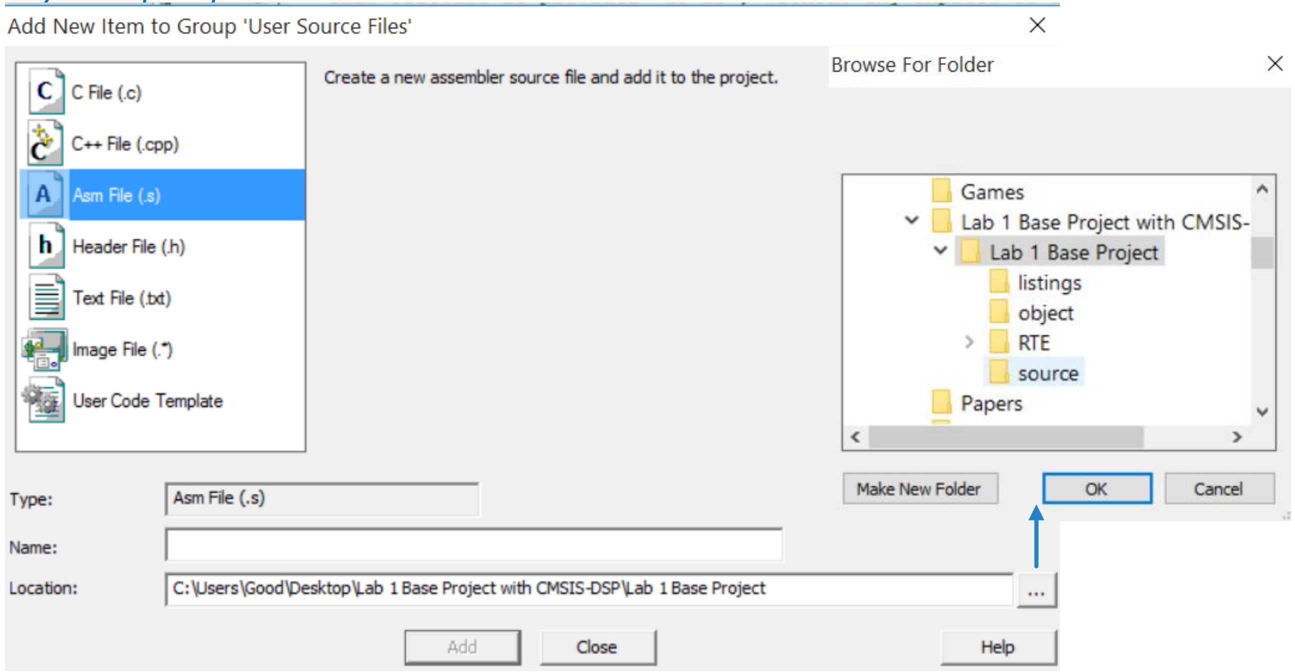
Starting up with Assembly Programming #2

Adding an assembly file to the project

Open Keil, and in the project sidebar to the left, right click on your project and select "Add New item to Group .."



Give your assembly file a name and add it. Explicitly add the extension .s. *Keep your files stored in the source folder in your project*



Starting up with Assembly Programming #3

Writing your first assembly program

```
        AREA text, CODE, READONLY
        EXPORT example1
example1
        MOV  R1, #25
        MOV  R2, #75
        ADD  R1, R2, R1
        BX LR ;
        END
```

First you need to create a code area to place your code within, the syntax is:
AREA AnyName, CODE, READONLY
… your code goes here ..
END
Forgetting the commas will result in this error: "Code generated in data area"
P.S. In this experiment, we don't need the data memory

You need to export the procedure name. This will make your subroutine visible to the linker since it is in a separate file. You need to import it wherever you intend to use it (in this case the startup file, see next slide).

Numbers are in decimal format by default. To use hex, use this format 0x25
If not preceded by # a warning will be issued (# not seen before constant expression). But the code still works.

Remember that after we call a function, we need to resume execution after the point we branched from.
Since we branch using a Branch with link (BL or BLX), the return address is stored automatically in R14 (LR) register.
Ending your procedure by **BX LR** means you return and start executing from where you stopped.

Except for procedure names, all other lines must be Tabbed. Otherwise, you will have the error
Unknown smthg, expecting opcode or macro

Starting up with Assembly Programming #4

Modifying the start up file to call your procedure/function

```
Reset_Handler    PROC
        EXPORT  Reset_Handler          [WEAK]
        ~~IMPORT  SystemInit~~
        ~~IMPORT  __main~~
        IMPORT example1
    ~~LDR     R0, =SystemInit~~
        ~~BLX     R0~~
        ~~LDR     R0, =__main~~
        LDR     R0, =example1
        BX      R0
        ENDP
```

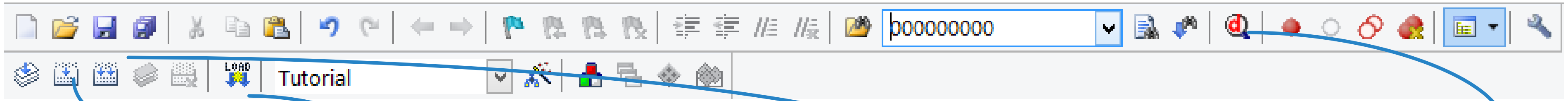
The SystemInit is a subroutine called to communicate with the board, set the clocks and other internal HW things. Since we are not using the board and we are using simulation mode, we don't need it.

To make the procedure visible to the linker, you need to import the procedure name. The EXPORT/IMPORT pair will resolve all references.

Simply call your subroutine by passing its address and branching to it

What if there is more code to follow? Should we use BX or BLX in this case?

Building and Basic Debugging #1



Build (F7)
Only builds your changes.
Always use this after you build your project the first time (much faster)

Rebuild: Builds each file in your project regardless if you changed it or not (slow)

Programs the board without starting the debugger

Debug (Ctrl – F5)
Programs the board with the latest build you did and STARTS the debugger
Note that if you edit the code and do not build it, then the board and debugger will not see the new changes, make sure to build before flashing and debugging
Press again to switch back to Build mode

Build Output

```
Build target 'imu_logger'
linking...
Program Size: Code=37652 RO-data=1060 RW-data=480 ZI-data=61464
FromELF: creating hex file...
".\build\obj\imu_logger.axf" - 0 Errors, 0 Warning(s).
```

Building and Basic Debugging #2

Reset
Resets the CPU
Starts from the Reset handles code
(or from main C subroutine if Run to Main() is checked in settings – see slide 24)

Single stepping, instruction by instruction

Press again to exit and switch back to Build mode

Run the code (F5), if breakpoints are placed, stops at next breakpoint

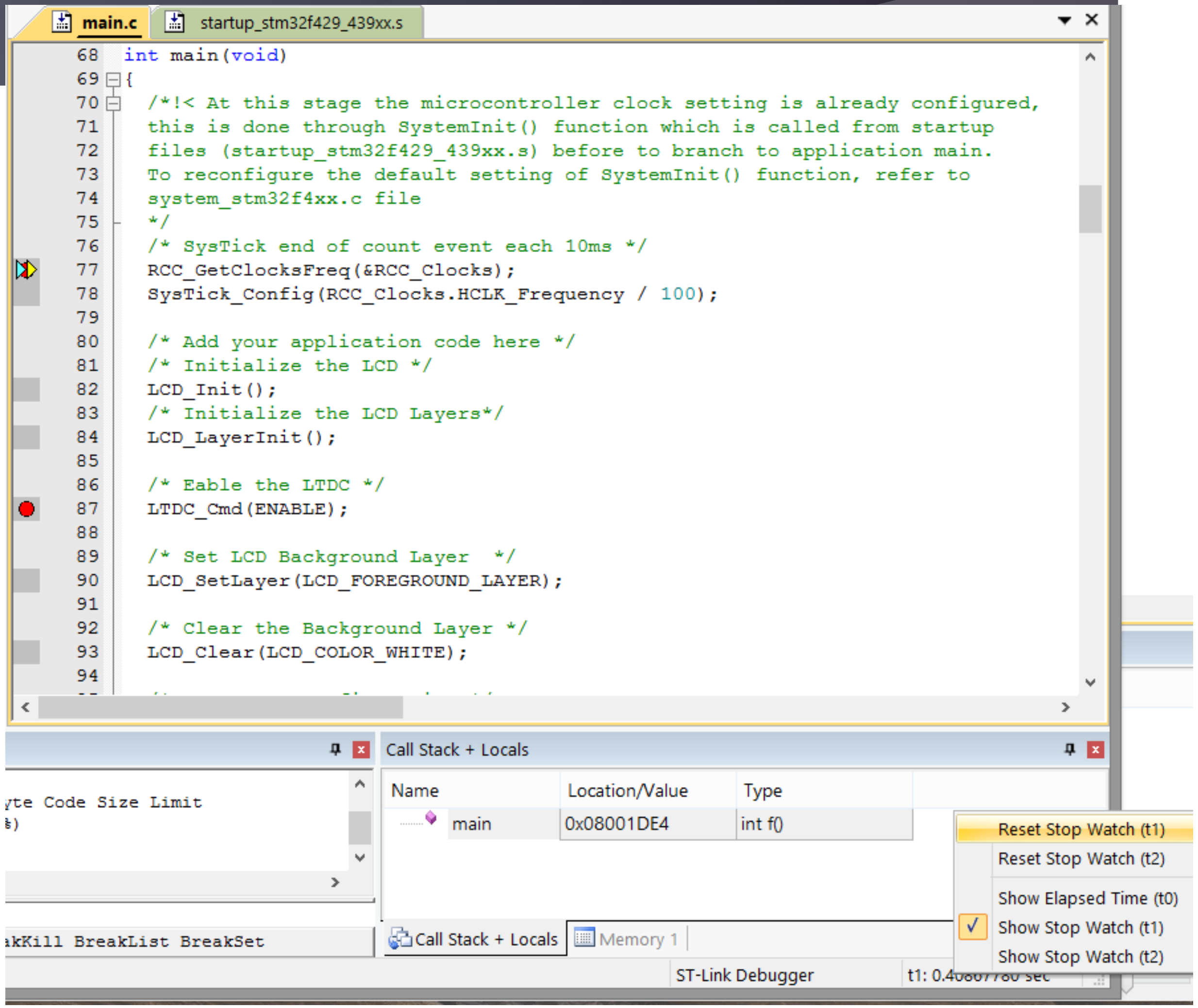
Debugging tools?

- Register view, breakpoints, time calculation ..
- More to be introduced later

Timing your code – the stop watch

- To measure how much time a piece of code needs to execute, **use break points** and the **stop watch.** Follow these steps in order. The stop watch is found in ***Keil status bar (called t0 or t1)***

1. Make sure that your clock settings are set to 168MHz in your project options (slide 21)
2. Place breakpoints on the beginning and end of the code you wish to measure its execution time
3. Run the code until it stops at the first break point
4. Right click on the stop watch and reset the time
5. Run the code again until it stops at the second breakpoint
6. Read the time



Same code but using FP

```
    AREA text, CODE, READONLY
     EXPORT example1
example1
     VLDR.f32 S1, =2.5
     VLDR.f32 S2, =7.5
     VADD.f32 S1, S2, S1
     BX  LR
    END
```

Floating point version of the same instructions

Floating point numbers are loaded using an equal sign

Consult following references on FP instructions:

Listing: Vector Floating Point Instruction Set Quick Reference Card

Details: Section 3.11 from the Cortex™-M4 Devices Generic User Guide

Enabling the FP unit

- The previous code will compile correctly, however, once you run the code, a HardFault will occur! Your code will get stuck in an infinite loop inside the HardFault_Handler

Problem

```
                B       .
                ENDP
HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler           [WEAK]
                B       .
                ENDP
```

- Solution: Enable FPU hardware unit by writing the following instructions right at the beginning of your reset handler subroutine (lines 182 – 187)

Solution

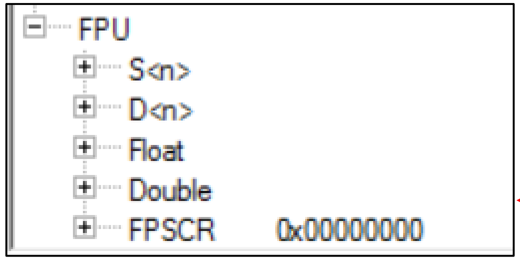
```
178                 EXPORT  Reset_Handler
179         ;IMPORT  SystemInit
180         IMPORT   __main
181         IMPORT example1
182                 LDR.W R0, =0xE000ED88
183                 LDR R1, [R0]
184                 ORR R1, R1, #(0xF << 20)
185                 STR R1, [R0]
186                 DSB
187                 ISB
188                 ;LDR     R0, =SystemInit
189                 ;BLX     R0
190                 LDR     R0, =example1
191                 BLX      R0
```

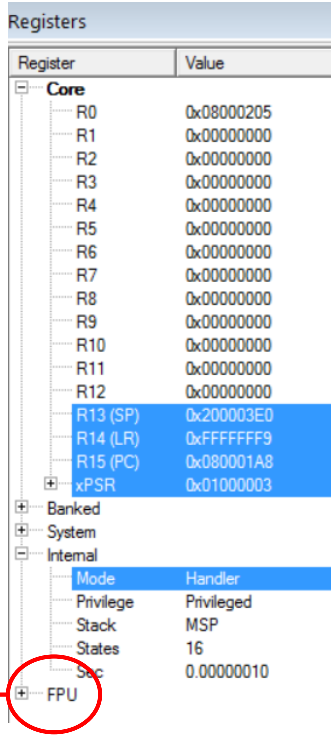
- DO NOT COPY/PASTE THE CODE FROM ANY SOURCE (I.E. pdf). Hidden characters will make the code fail at compiling. Just write those lines yourself

Debugging FP instructions

- Once you go into debug mode, expand the FPU registers. You can see the FP values in either the IEEE 754 format (***S<n>*** for floats and ***D<n>*** for doubles (if supported))
- For human intelligible floating point representation, expand the ***Float*** and ***Double*** lists

You can only use FPSCR ***ONLY*** after copying it to APSR. Use the VMRS instruction





How to store and use data in assembly?

- **Registers** → Not recommended at all, should be minimum use and used for operations
- **The stack** → You can push your data (say program constants), or just push some zeros (free space) into the stack before calling your assembly function. You can access the data through **Stack Pointer** manipulation. You have to manually keep track of where your pointer is pointing throughout your program execution.
- Both **Code Memory** for Read only *constants*, and **Data Memory** for read-write variables. However, this is **not as straightforward** for the first time assembly programmer

Lets take a deeper look

Data Memory (1)

- Defining Data memory is similar to Code memory definition
- Should be in a separate file (See example data memory definition)
- We have written a code to access and load first value of each string
- The expectation is that we have five integers 1 – 5 stored in data memory followed by fifty 0xAB values!
- We expect to see that **R1** has the value **of 0x04030201**, that is loading the first four bytes into **R1**, and that **R2** has the values of **0xABABABAB**
- **But it is not the case!**
- **Both R1 and R3 have a value of ZERO**

```
        AREA myData, DATA, READWRITE
        export myString1
        export myString2
myString1   DCB     1, 2, 3, 4, 5
myString2   FILL    50,0xAB
        END
```

```
        AREA text, CODE, READONLY
        export example1
        import myString1
        import myString2
example1
        LDR R0, =myString1
        LDR R1, [R0]
        LDR R2, =myString2
        LDR R3, [R2]
        BX  LR

        END
```

Data Memory (2)

- The data memory is always initialized by zeros, no matter what directive you use (DCB, DCD, FILL or SPACE)
- You only use these directives to reserve space and give it a name
- So how do we get to store constants? → Code Memory
- Right after your code, start with ALIGN directive followed by DCB, DCD, FILL .. Etc
- BUT YOU CAN NEVER OVERWRITE THESE VALUES. THIS IS A READ-ONLY AREA, SO? → NEED TO COPY TO DATA MEMORY (SEE Example)
- To access content of the data memory and see the values there, you have two options:

1. Right click on myString1 and Select "Add to Watch1"
2. Or, see the memory address of myString1 that is loaded into R2, then Go to View→MEMORY Windows→Memory 1 and write the address there (press enter)

```
    AREA text, CODE, READONLY
    export example1
    import myString1
    import myString2
example1
    LDR R0, =myString3
    LDR R1, [R0]
    LDR R2, =myString1
    STR R1, [R2]
    BX  LR

    ALIGN
myString3 DCB 1, 2, 3, 4

    END
```

Watch 1

Name	Value	Type
myString1	0x04030201	uint

Memory 1

Address: 0x20000000

```
0x20000000: 01 02 03 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000003B: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000076: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Mixing assembly and C

- To call an assembly subroutine from within C is no different than calling any other C function.

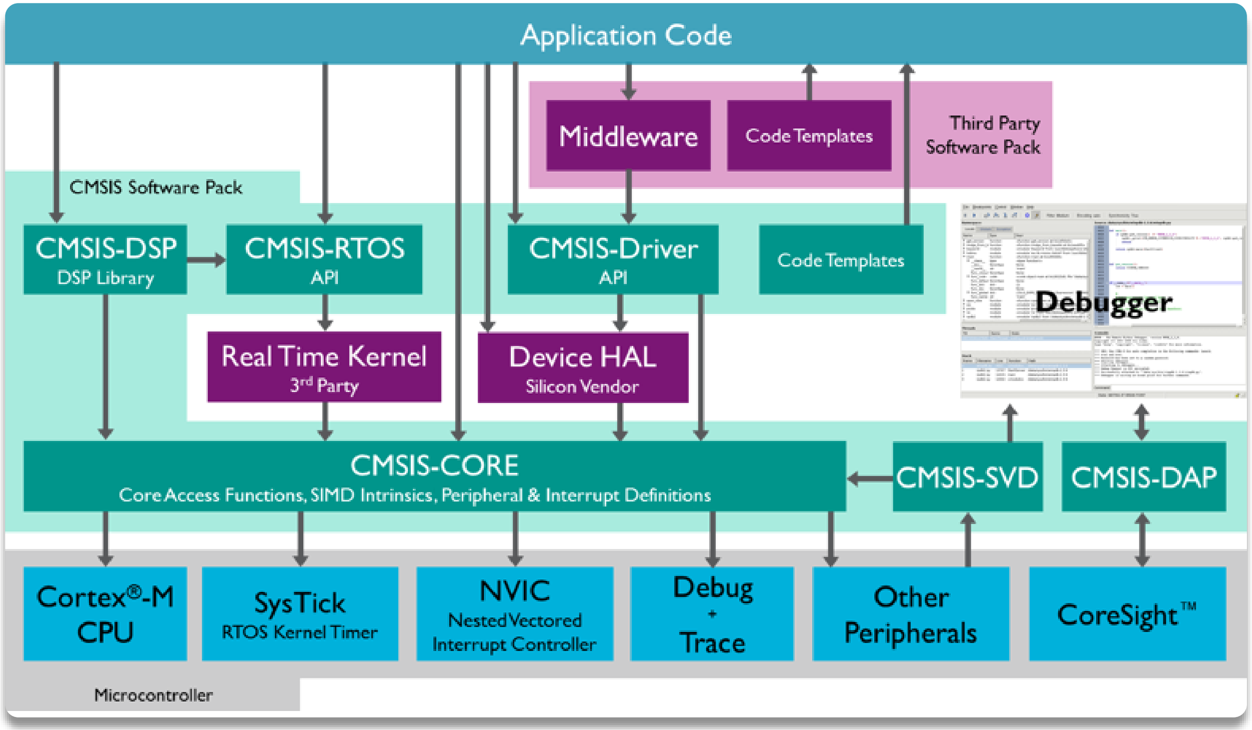
- However, to make this work, there is a certain set of rules which you must follow in writing your assembly program

- These rules are called the calling convention. For example, they specify which Registers to use when doing certain actions (like passing parameters).

- You are required to read "**Procedure Call Standard for ARM Architecture**" section 5 to familiarize yourself with these rules as you will need them in Lab 1. This document will be uploaded to myCourses
- Review Prof. Slides for examples

CMSIS DSP Library

- The **Cortex Microcontroller Software Interface Standard (CMSIS)** is a vendor-independent hardware abstraction layer for the Cortex®-M processor series.



CMSIS DSP Library

- The CMSIS DSP software library consists of different functions in categories like:
  - Basic, complex and fast math functions
  - Filters
  - Matrix functions
  - Transforms
  - Motor control functions
- For more information, visit the Keil website on CMSIS library:

  http://www.keil.com/pack/doc/CMSIS/General/html/index.html

References

- [1] http://www.ecnmag.com/news/2011/05/arm-passes-x86-and-power-architecture-become-leading-mcu-empu-architecture-2010
- [2] http://www.design-reuse.com/news/27468/arm-processors-annual-shipments-forecast.html?sf2308980=1
- The Definitive Guide to ARM Cortex Programming, Joseph Yiu, 2013.