µP Tutorial 1

Introduction to the lab

IDE, first program and lab policy

BY ASHRAF SUYYAGH

VER 2.0 FALL 2013 (MAJOR REVISION)

VER 1.0 WINTER 2013

Outline

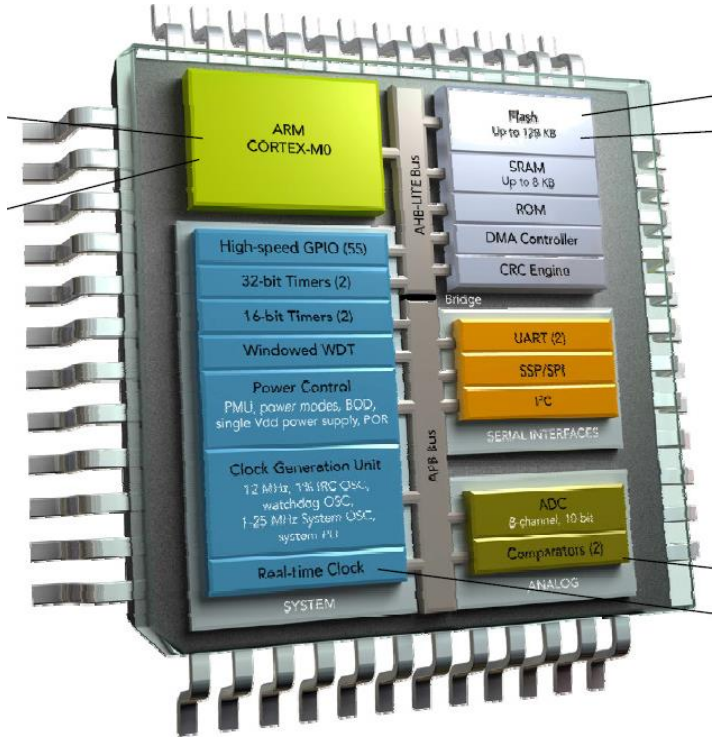
- Introduction to the HW platform and supporting software tools
- Understanding the Programming Model of Cortex M4
- Instruction set quick short review
- Introducing Keil uVision IDE
- Writing first assembly program in Keil IDE
- Debugging
- Lab Demos and Report Submissions, Grading

What is a Microcontroller (µC, or MCU)?

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

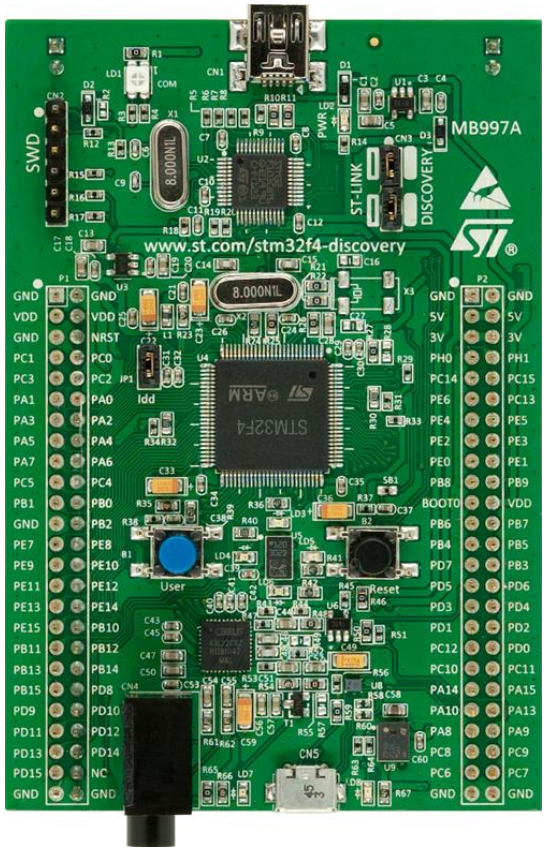
What is ARM?

- **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, STMicroe lectronics and Renesas 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] More News here [http://www.semicast.net/inthenews.html].
- Has three major µC families (profiles): A, R and M



ARM Cortex-M Series

- 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...



So which tools do I need to download to get started?

- Keil uVision IDE ver 4.72 (Free download with 16k limitation, enough for labs)

https://www.keil.com/demo/eval/arm.htm

- ST LINK driver

http://www.st.com/st-web-ui/static/active/en/st_prod_software_internet/resource/technical/software/utility/stsw-link004.zip

- STM32F4 Discovery Board example files and data sheets

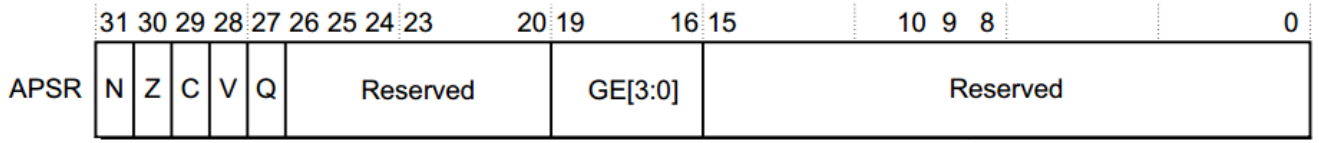
http://www.st.com/web/en/catalog/tools/FM116/SC959/SS1532/PF252419

- Tutorials, datasheets will be added to myCourses as needed

Cortex M4 – Registers Overview

Name		Functions (and Banked Registers)	
R0		General-Purpose Register	Low Registers
R1		General-Purpose Register	
R2		General-Purpose Register	
R3		General-Purpose Register	
R4		General-Purpose Register	
R5		General-Purpose Register	
R6		General-Purpose Register	
R7		General-Purpose Register	
R8		General-Purpose Register	High Registers
R9		General-Purpose Register	
R10		General-Purpose Register	
R11		General-Purpose Register	
R12		General-Purpose Register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

Registers in the Cortex-M



Status Register Bits (APSR)

Bits	Description
Bit 31	**N:** Negative or less than flag: 0: Operation result was positive, zero, greater than, or equal 1: Operation result was negative or less than.
Bit 30	**Z:** Zero flag: 0: Operation result was not zero 1: Operation result was zero.
Bit 29	**C:** Carry or borrow flag: 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
Bit 28	**V:** Overflow flag: 0: Operation did not result in an overflow 1: Operation resulted in an overflow.
Bit 27	**Q:** DSP overflow and saturation flag: Sticky saturation flag. 0: Indicates that saturation has not occurred since reset or since the bit was last cleared to zero 1: Indicates when an SSAT or USAT instruction results in saturation, or indicates a DSP overflow. This bit is cleared to zero by software using an MRS instruction.

Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N != V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N != V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix is specified.

Assembly Programming

- Low-Level Programming Language
- Each statement corresponds to a single machine instruction
- Specific to a particular architecture (Thumb, Thumb-2, ARMx)
- Assembly is converted to executable machine code by the **assembler**
- *How to learn assembly? Sources?*

Assembly instructions Overview

Simply, most take the form of <Operation>{<cond>}{S} Rd, Rn, Operand2

- Arithmetic Operations: **ADD, SUB, RSB .. etc**
- Logical Operations: **AND, ORR, EOR .. etc**
- Comparisons: **CMP, TST (No Results written)**
- Movement operation: **MVN**
- Branching: **B (BX), BL (BLX)**

- Examples:
  - ADD R0, R1, R2
  - CMP R3, R2

How to learn assembly? Resources?

- 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

You will find the following on myCourses page:

- **ARM and THUMB 2 Instruction set:**

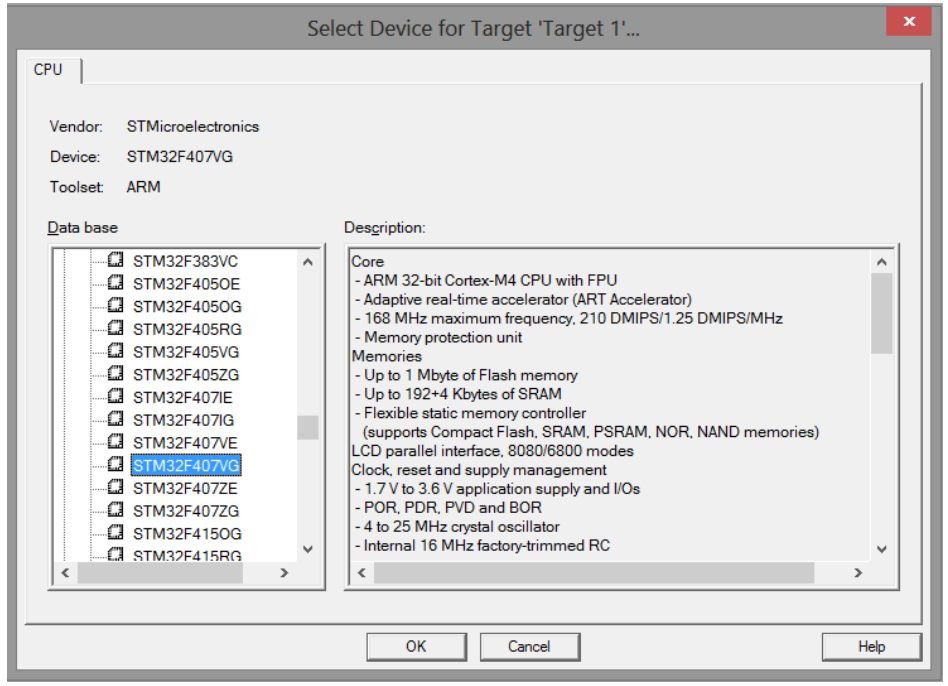
Lists all available assembly instructions in the Cortex-M4 core with brief descriptions.

- **Cortex M4 Programming manual:**

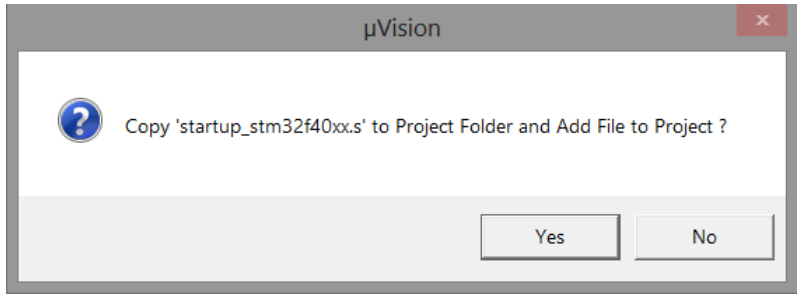
1. A must read datasheet (Sections 2.1 – 2.4)
2. Section 3 details each instruction with example code and specific uses

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



- Click Yes when asked to copy the start-up file



Introducing Keil uVision IDE #2
Organizing your project

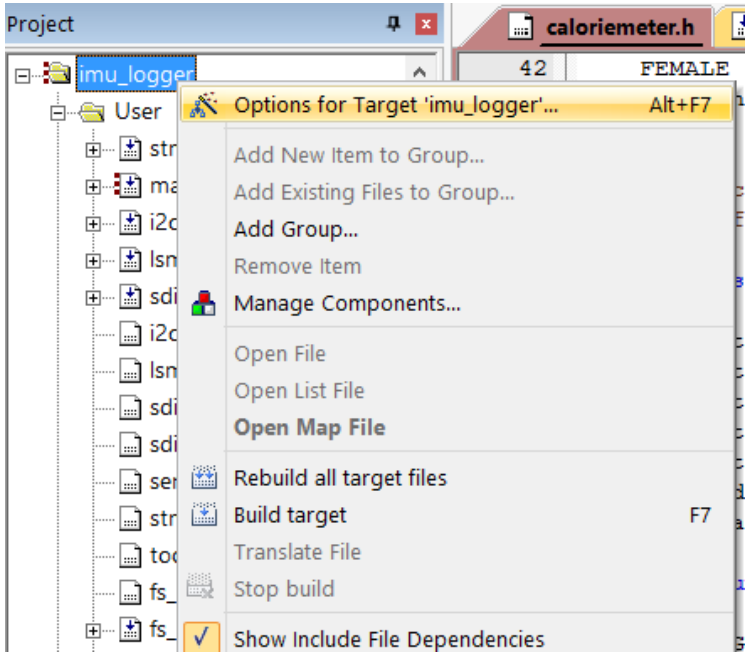
- To the left side, you will find the project pane where you have access to all your source files and project settings.
- Give your target a name by renaming it.

→ 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

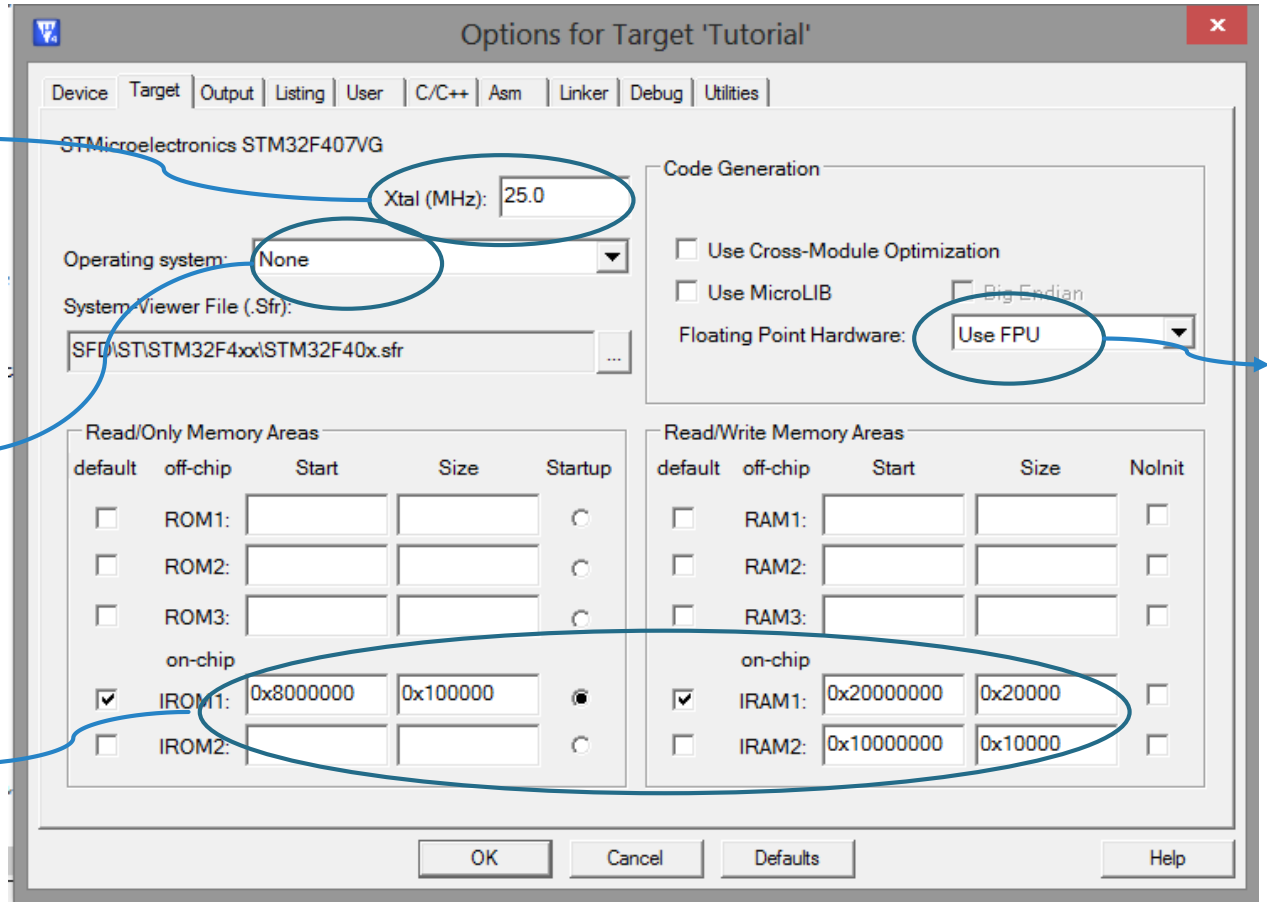


Introducing Keil uVision IDE #3
Project Options / Target

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

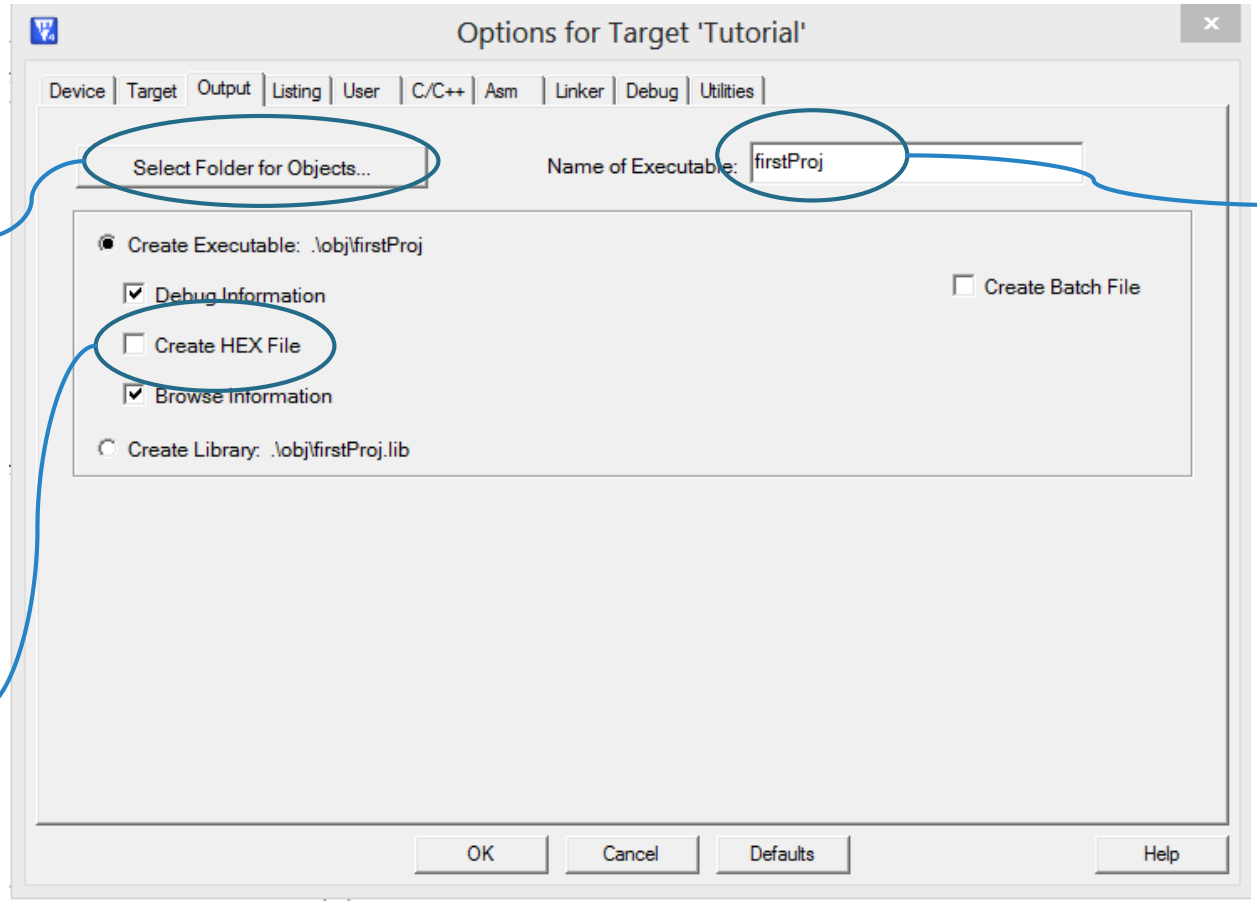


Enables the use of the Cortex M4 hardware FPU unit for accelerated FP performance

Introducing Keil uVision IDE #4
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

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

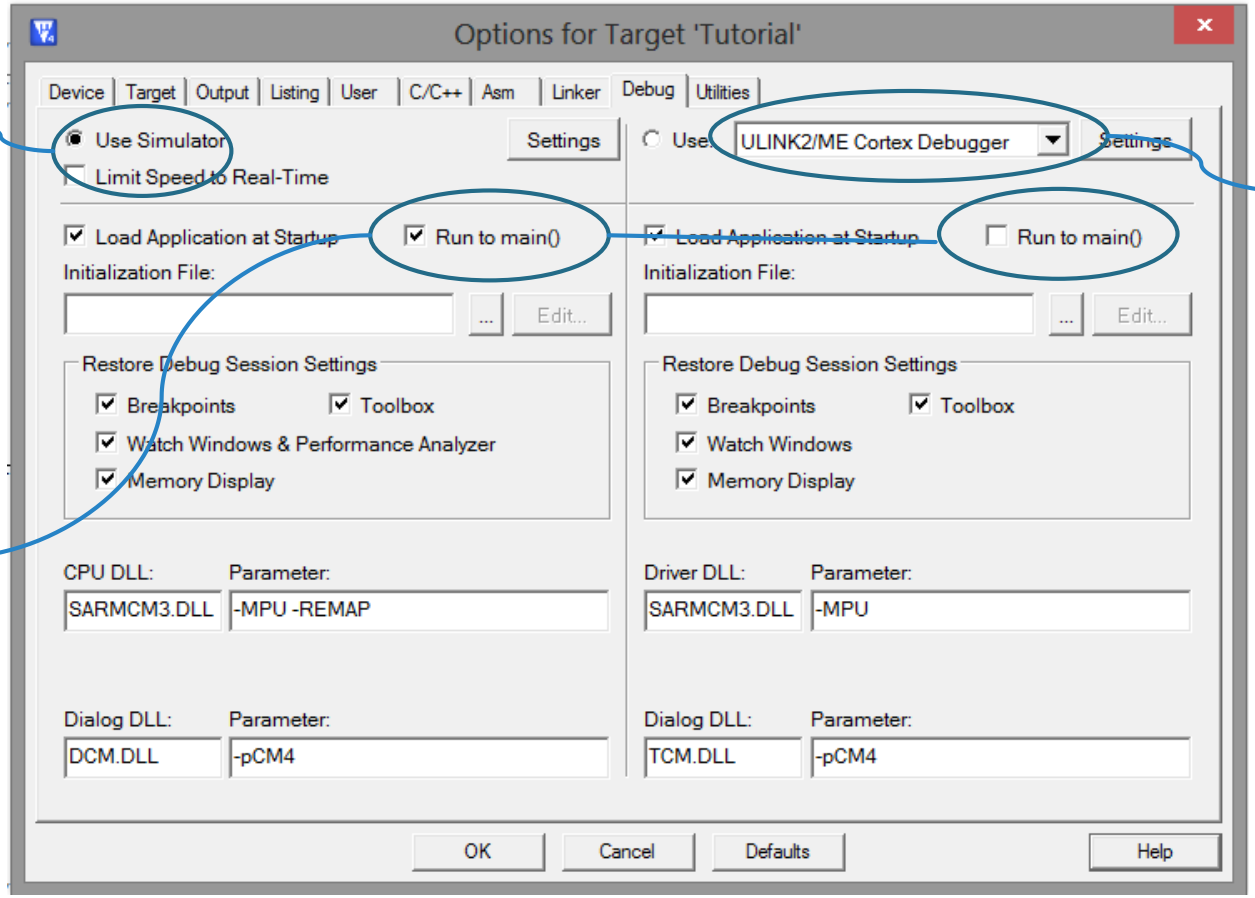


Select the name of your output executable

Introducing Keil uVision IDE #5
Project Options / Listing

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

Starting up with Assembly Programming #1

The startup file (startup_stm32f40xx.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.
- C functions are always preceded by double underscore
- 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

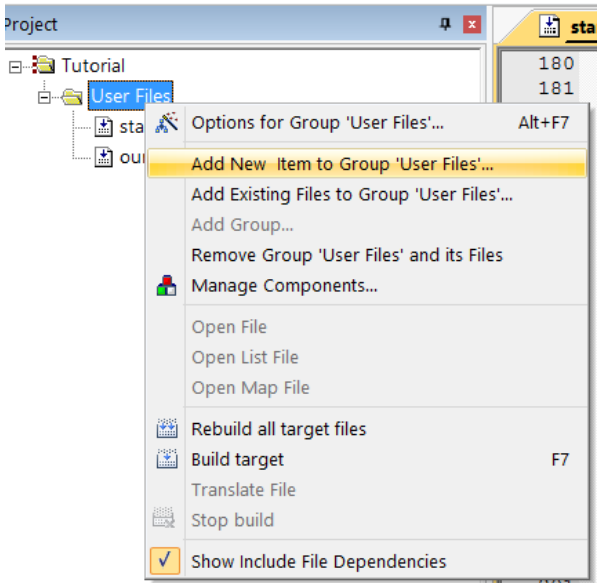
When you compile your code, if you receive an error Undefined symbol __use_two_region_memory (referred from startup_stm32f40xx.o), simply comment the line IMPORT __use_two_region_memory at the end of the startup file by placing a ; at the beginning

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

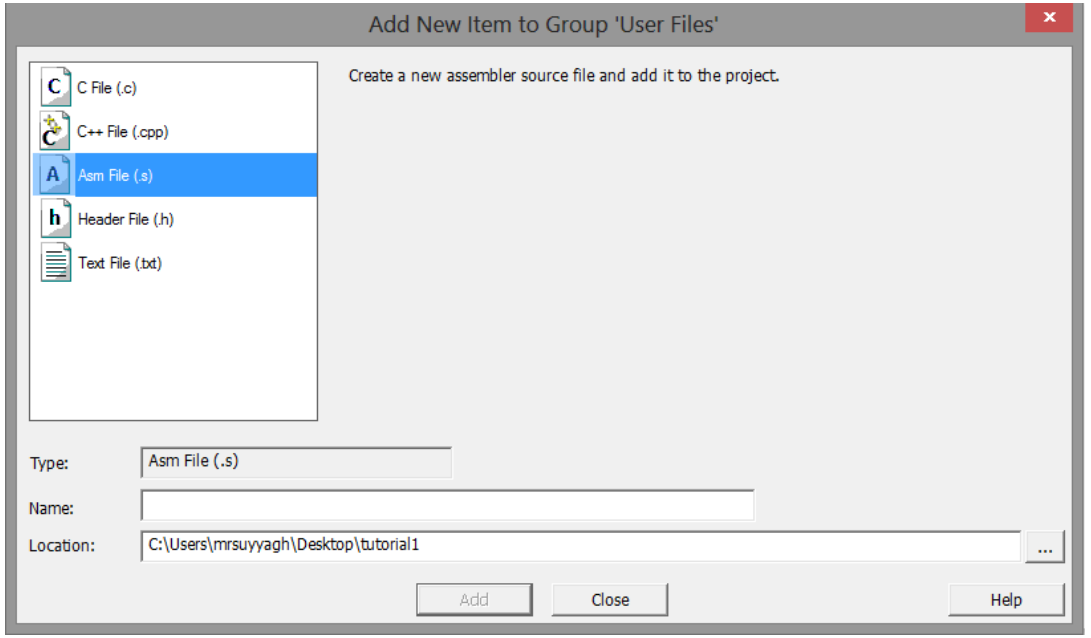
Starting up with Assembly Programming #2

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.



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, R1, 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 gies here ..
END

Forgetting the commas will result in this error: "Code generated in data area"

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 16)

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

Mixing assembly and C

- To call and assembly subroutine from within C is no different than calling any other C function.
- However, to make this work, there are a certain set of rules that 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

Lets agree on some things, shall we?

- How demos go? What is expected from you? Grading demos? Partner involvement?
- Communicating with TAs, discussion board or emails? When to send an email, when not to?
- Lab Reports. How to write your report? Grading Reports? Fairness?

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