ECSE426 Microprocessor Systems

Fall 2012

Tutorial 1: Cortex M4 Programming and Keil Tools

Objective

This tutorial introduces the Keil development tools for ARM Cortex M family of processors, including the basic setup for starting the assembly coding for a given processor, but most of the setup directly extends to the embedded C programming. It will also introduce you to the laboratory exercise, as well as the finer points of the Keil compiler, linker and simulator.

Background

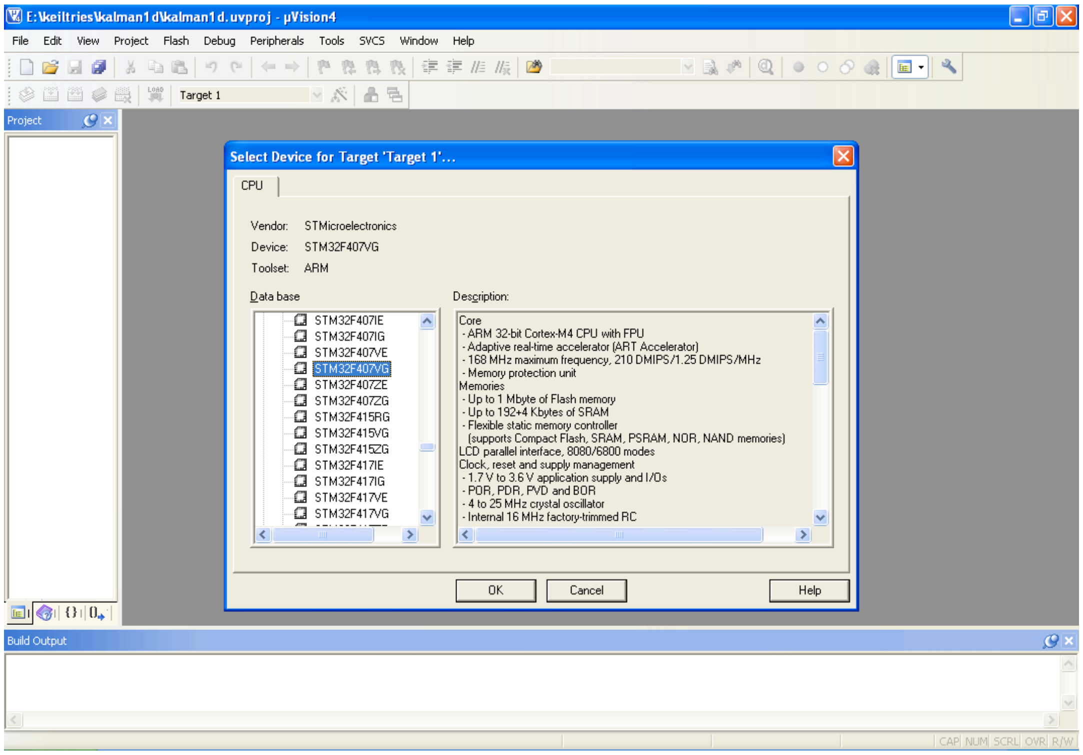
Using the Keil Tool - Requirements

This course will rely on the version 4.53 or higher of the Keil ARM Microprocessor Development Kit (ARM-MDK), also known as Keil µvision. The older versions include much of the features required here, but some critical debug support is only available beyond the version specified.

Creating new assembler project in Keil tool

To learn how to create and define projects, create first a new assembly project. For interfacing the bare processor, it is best to let the tool insert the startup code (written in assembler) that initializes the stack pointer, interrupts and several other things.

To select the proper startup code for the processor used in ST F4-Discovery board, please select device STM32F407VG by ST Microelectronics, as shown by the screen capture below.


E:\keiltries\kalman1d\kalman1d.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Build Output
CAP NUM SCRL OVR R/W

Select Device for Target 'Target 1'...

CPU

Vendor: STMicroelectronics
Device: STM32F407VG
Toolset: ARM

Data base

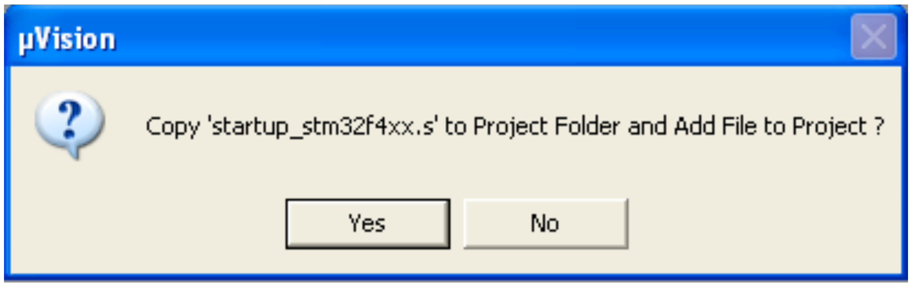
STM32F407IE
STM32F407IG
STM32F407VE
STM32F407VG
STM32F407ZE
STM32F407ZG
STM32F415RG
STM32F415VG
STM32F415ZG
STM32F417IE
STM32F417IG
STM32F417VE
STM32F417VG

Description:

Core
- ARM 32-bit Cortex-M4 CPU with FPU
- Adaptive real-time accelerator (ART Accelerator)
- 168 MHz maximum frequency, 210 DMIPS/1.25 DMIPS/MHz
- Memory protection unit
Memories
- Up to 1 Mbyte of Flash memory
- Up to 192+4 Kbytes of SRAM
- Flexible static memory controller
  (supports Compact Flash, SRAM, PSRAM, NOR, NAND memories)
LCD parallel interface, 8080/6800 modes
Clock, reset and supply management
- 1.7 V to 3.6 V application supply and I/Os
- POR, PDR, PVD and BOR
- 4 to 25 MHz crystal oscillator
- Internal 16 MHz factory-trimmed RC

Figure 1: Selecting the processor model for F4-Discovery board

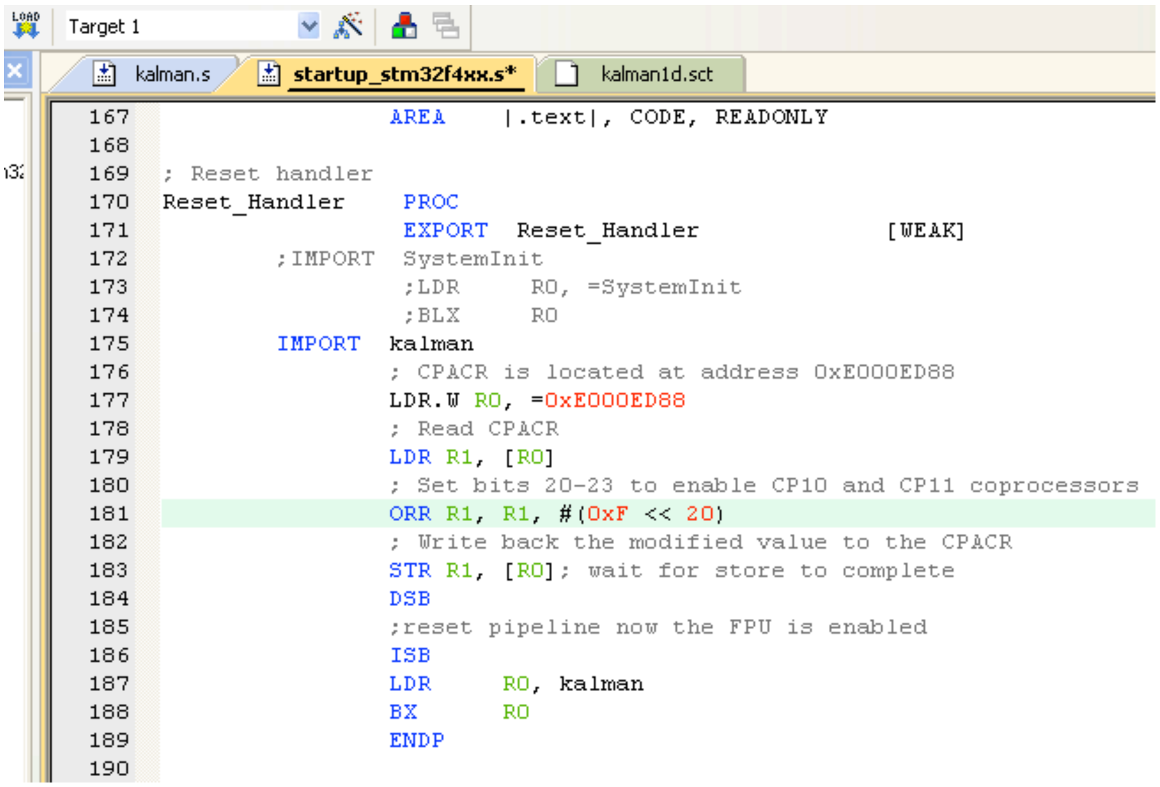
Next, when asked you should specify to include the startup code, so you should select "Yes", Figure 2. This is the assembler code that sets up the processor properly, including the interrupt vectors definition. It provides place for invoking the user-defined SystemInit procedure before the user program starts by branching to the main program.


µVision

Copy 'startup_stm32f4xx.s' to Project Folder and Add File to Project ?

Figure 2: Inclusion of the proper startup code for STM32F4 processors

The startup code by default branches to the procedure SystemInit and __main, which is the default entry point of any C program. To branch to some other entry points, the startup code will need to be slightly modified. The place in the code where this is done is shown in Figure 3.



```
167                        AREA    |.text|, CODE, READONLY
168
169  ; Reset handler
170  Reset_Handler      PROC
171                     EXPORT  Reset_Handler             [WEAK]
172          ;IMPORT    SystemInit
173                     ;LDR     R0, =SystemInit
174                     ;BLX     R0
175          IMPORT    kalman
176                    ; CPACR is located at address 0xE000ED88
177                    LDR.W R0, =0xE000ED88
178                    ; Read CPACR
179                    LDR R1, [R0]
180                    ; Set bits 20-23 to enable CP10 and CP11 coprocessors
181                    ORR R1, R1, #(0xF << 20)
182                    ; Write back the modified value to the CPACR
183                    STR R1, [R0]; wait for store to complete
184                    DSB
185                    ;reset pipeline now the FPU is enabled
186                    ISB
187                    LDR      R0, kalman
188                    BX       R0
189                    ENDP
190
```

Figure 3: Entry point redirection – enabling FPU and starting with the assembly routine "kalman"

Please note that two underscores are used by convention to interface the C compiler, and for branching to your C code, it is not needed to modify this. When there is no C code with main, some further editing of startup code is needed to remove compilation errors due to the missing variables expected from the compiled C code, as explained in the last LDR command in Figure 3. When C code is present, as will be the case in the second lab, no such redirection in the startup code will not be necessary and the processor will jump to the beginning of the C main routine.

One other modification is needed when using the floating-point hardware, as will be in the case of Lab 1 and Lab 2. In that case, the FPU should be enabled. The modification to execute the enabling of the FPU is shown in the first part of the modified startup code, Figure 3, specifically between lines 176 and 186 in the modified startup code.

After this initial setup, the required source file(s) need to be added to the project. The example of adding the source code module is shown in Figure 4.

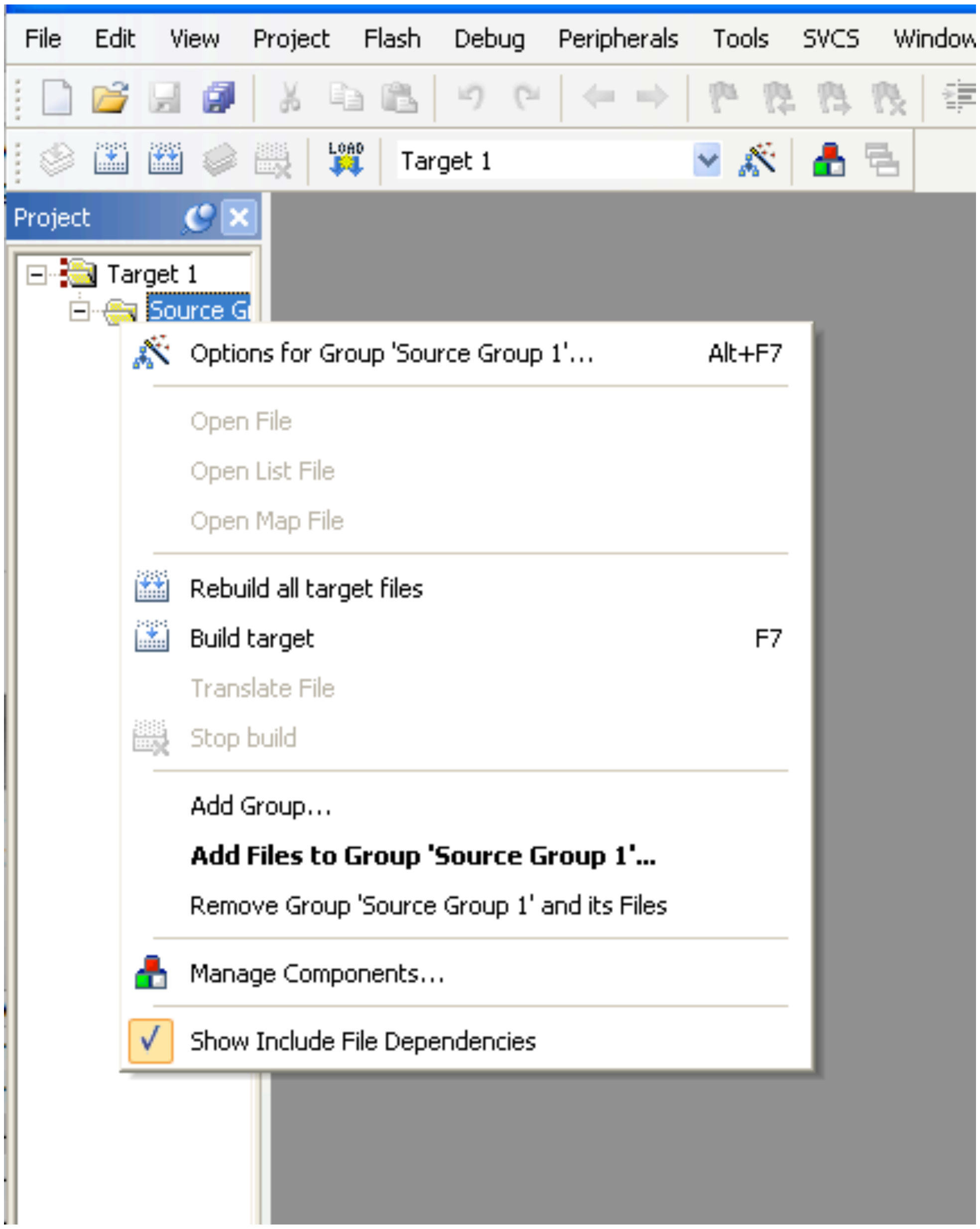


Figure 4: Adding source file to the project

For C language file inclusion, which you will do later, in lab 2 and beyond, the procedure is equivalent, but please keep in mind that the entry point need not be modified, as it will be kept at __main.

In either case, the inclusion of the file does not create it, so the file has to be created and written by either the Keil or an external text editor.

Please note that if only the assembly file is to be included in the project, the given startup code will report an error at the compilation time. To eliminate that error, you should comment the line in the startup code that imports the identifier __use_two_region_memory. Again, in normal uses (i.e. C program) that problem will go away.

Keil compiler, assembler and linker

Upon including the files, you will then need to start writing your assembly code based on the algorithm that you elaborated beforehand. It is simplest if you start the file anew and include in it the statements defining the segment (AREA), an entry point and externally visible variables, as presented in lecture 2 or in the code included at the end of this handout.

Keil simulator

Once the program compiles, you will proceed to the debugging and verification stage, to ensure that the program runs and that all the cases are processed correctly. The simulator models all the processes that take place on the target chip, so you just need to focus on inspecting that the intermediate results are generated correctly. During the debugging, you can "single step" through each instruction, continue up to some point in the program or run the whole test case, depending at which stage in the debug process you are.

Code debug

The simulator provides extensive debug support. You will be able to inspect memory, program execution and observe or even modify registers while in simulation mode. You can use that to your advantage to perform efficient debugging, as shown in the tutorial. The user interface is very intuitive for all the debug functions – clicking left to the line number creates a breakpoint, and the icon performing single-stepping, going to next breakpoints etc. are intuitive and backed up by text that appears upon covering the icon. The debug functions are not accessible

Useful files

When compiling your assembly code, the compiler will generate object files, and the linker will produce the executable code. The linker uses the object files (.o), along with those generated for C-files (part B) to create an executable and eventually loadable ELF (.axf) files for simulation and the actual device. You will not need the loadable files for this exercise, as everything is done in the simulator.

Some other useful files include:

- .lst, the assembly and machine code listing.
- .map, the file listing symbols, memory locations all data specifying the code layout

Recall that the included startup code might rely on some external variables that will not be present in your own assembly code. To avoid compilation errors, those variables can at this stage be safely commented out, as shown in the tutorial. You can use the above files to locate all questionable symbols and deduce their significance.

Generic Function Requirements for Subroutines written in Assembler

1. All registers' contents and the stack position should be unaffected by the subroutine call upon returning.
2. The calling convention will obey that of the compiler.
3. No memory location should be used to store any intermediate data. Not only that it should be faster, but no memory allocation is needed that way.
4. The subroutine should be location independent. It should be able to run properly when it is placed in different memory locations.
5. The subroutine should not use any global variables.
6. The subroutine should be as fast as possible, but robust and correct for all cases of positive and negative numbers, as well as the overflows. Grading of the experiment will depend on how efficiently you solve the problem, with all the corner cases being correct.

Hints:

ARM Assembly Code

1. It helps to solve the problem conceptually at first, taking into account all corner cases, including the arithmetic overflow. Move to assembly code when the algorithm is well defined.
2. Follow the examples of codes and instructions elaborated in classes, tutorials and material posted online for getting quickly to the working code. When optimizing for speed, some better ARM Cortex instructions could exist, such as those replacing directly bits between two words.
3. Document assembly code thoroughly and thoughtfully. It takes only a few hours for you to forget what each register holds and what implicit assumptions were used, not to mention the communication about the code with the team partner.

Linking

1. When creating a new project, it is best to include the startup code, for which the tool will ask you whether to include it. Then, modify that code to branch to `kalman` rather than __main for testing assembly code alone, prior to embedding into C main program. Please note that you will need to declare as exported the subroutine name in your assembly code.
2. If the linker complains about some other missing variable to be imported in startup code, you can either declare it as "dummy" in your assembly code, or comment it's mention in the startup code.
3. For linking with C code that has main, none of the above two measures are needed.

Useful Notes

For more information refer to help contents and reference material posted on the course web site**.** Also, you can consult application notes, such as **Cortex^{TM}-M4 Programming Manual** that are available on course web site. Please keep in mind that for any code and examples available online the calling convention and syntax, including even the assembly language might be those of other compilers or previous version of ARM processors, where the compatibility has not been preserved.

The Linker

You may find it useful to separate your functions into separate files. This will definitely be the case for later experiments. The Assembler and C Compiler will be used to assemble each assembly file into a location independent object file. The *linker* will then link these individual object files to produce a single absolute file.

In projects where teams of programmers construct larger programs, it is actually necessary to separate the work into smaller units. These smaller pieces of code are compiled and tested individually, and are joined together by the linker program.

When working with these individual pieces of code, it is important to keep them location independent. By postponing the decisions of memory allocation to later times, collisions in memory spaces among different programs can be avoided. Memory for code, data and stack will be allocated when the linker generates the absolute, loadable file.

Appendix: Assembly Code – Sample Template

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;kalman.s

; comments, purpose of code, inputs, outputs, errors.

     AREA      FILTERCODE, CODE                  ; Define area to be code
     EXPORT moving_average                       ; Declare symbol to be exported
moving_average
                                     ;your BCD ADD assembly routine here (Lab 1)
     BX R14

     END
```