Sam Detor

Lab 2

Part 1

I first updated the start-up script, linker script, and header file to the correct files for the STM32F411VE MCU—startup_stm32f411xe.s, stm32f411xe.h, STM32F411VETx_FLASH.ld. I also updated the per-project makefiles and the armf4 makefile to reference the relevant updated files. After that, I used the generic header file in my .c files and uncommented the relevant section in the generic header file to include stm32f411xe.h. One of the struggles I had during this phase of Part 1 was getting the correct .s file for my compilation configuration. I accidentally was trying to use one for a custom arm configuration at first, before seeking help in office hours and then selecting the correct .s file for my gcc set up.

After doing this, I was able to make and run the "timer" example by commenting out "set_sysclk_to_168" , but the timing of the led blinks did not match what it said it should be in the code. This was because my timer was assuming that the system clock was running at 168 Mhz, when it was not. I then updated the system_stm32f4xx.h to define new PLL_M, PLL_Q, PLL_P, and PLL_N based on the recommendations found in this source to set the clock to 100 Mhz, which the board can handle. I then changed the name of the "set_sysclk_to_168" to "set_sysclk_to_100" to more accurately reflect its functioning.

After these changes were made I combined the code from the blinky and external projects from the sample repo given. I enabled both the rising and falling edge trigger on the button so that the interrupt handler would be called both when it was pressed and being let go. In the actual interrupt handler, I checked if the interrupt was coming from the correct line, cleared the interrupt and waited for another trigger. This allowed the falling edge trigger to initially trigger the interrupt, and then it would hang and wait for the rising edge trigger, which paused the led blinking given the high priority level of the interrupt handler. Once the falling edge trigger was detected, the handler reset the led sequence to start at green again, and returned.

The functions of the files mentioned in the lab handout are detailed below.

startup_stm32f411xe.s:

Since C files assume that there is a call stack when they run, some code not written in C has to set up the stack before the C code can be executed. At a high level, this code, which is written in assembly, sets up the call stack and other basics so that a C program will run. This program first defines the reset handler, which sets the stack pointer to _estack. __estack is defined in the linker script as the highest address in RAM. Then, using the section headers and footers defined in the linker script, it copies the data from flash to RAM, and fills the bss section with zeros. Then it initializes the clock, calls static constructors and then calls main. This script also sets the vector table entries for the interrupt handlers and creates aliases for these handlers that can be overwritten by user code. Also, the script sets the program counter to the reset handler, so it is the first instruction run when the program starts.

It is very important that this script and the linker script used the same variable names, as this script used the data segment addresses (like the start and end of the bss section, for example) in the startup code. This was one of the problems I ran into when updating the files.

stm32f411xe.h

This file defines nice macros that the C program can use to access and set peripheral registers without having to directly use addresses. First, this file defines a structs that can be used to access registers nicely. Then all the base pointers for the major peripherals are defined. Finally, the file casts the base addresses to struct pointers and defines them as with names that correspond to the reference manual.

STM32F411VETx_FLASH.ld

The linker first sets the program entry point to be the reset handler. This means that the first thing that will be executed in the code is the reset handler. Then, it sets the location of the stack pointer, the minimum heap size, and the minimum stack size. It then also sets parameters like flash and RAM sizes, their start values, and their permissions. This is a very important section of the linker script because it specifies the memory layout that the board has. Then the script dictates the layout of the ELF file to be generated. The startup code is put first, then the standard text section and rodata sections. After that, there is a "ARM extab" section defined, which according to this stackoverflow post, contains unwiding information. Unwinding, per this source, is how to get rid of local variables and call destructors on a stack when a function call ends. Apparently this information is mostly useful for dealing with exceptions being thrown. After this section, "pre init array", "init array", and "fini array" are mentioned. From the stackoverflow post mentioned earlier, these sections are referring to the constructor and destructor C methods that are called before and after main(). After that, the standard data and bss sections are described, and global symbols are created at the beginning and end of these sections for use in the startup script. Towards the end of the file, there is some memory reserved for the user heap/stack. The last bit deletes any sections that were not specified in the above script (source).

Makefile

At a high level, the initial per-project makefile defines the some variables used in the overall makefile (armf4) and then calls the overall makefile. Some of these variables that were important to change were the TARGET and SRC variables, which are responsible for the name of the files generated and the files that need to be compiled, respectively. The linker script was also assigned to a variable as well as the processor type, for further use in the "overall makefile".

When "make" is called, it calls three sub commands, clean, build, and size. Clean is pretty self explanatory, as it removes all files/folders generated from a previous call to make. Build is more complicated, as it generates a .elf, .bin, and .lst file with the name specified by the TARGET variable, in a sub folder named by the variable OBJDIR. This command first generates the .elf file. First, all the relevant .c and .s files are compiled into .o files. All the .o file names with their corresponding file paths in OBJDIR are stored in the variable OBJS. So, the makefile compiles all the corresponding .c and .s files for the

.o filenames in OBJS with the specified CFLAGS and with all the included libraries necessary (in the case of the .c files). Then these .o files are linked together to create the .elf file based on the linker flags, one of which is the linker script mentioned earlier. Once this .elf file is created in the OBJDIR directory, then the .bin and .lst files can be created by calling "arm-none-eabi-objcopy" and "arm-none-eabi-objdump" on the .elf file created earlier with the correct flags. The "size" command then reads the elf file and prints the sizes of all the text, data, bss, sections as well as other information to the screen after you are done generating all the necessary files. "make burn" then writes the .bin file to address 0x8000000 using the st-flash command with the "write" option from the stlink-tools library. 0x8000000 looks like the standard address to write .bin files to on an STM32 board.

Source code for part1

I got my startup_stm32f411xe.s, STM32F411VETx_FLASH.ld, and stm32f411xe.h from the STM23CubeF4 repository of the STMicroelectronics github. I did not change any of these files.

From the sample repo given, I used the contents of the libs folder (the CMSIS_5 repo), and the stm32f4xx.h file without making any changes to them.

The makefile in the Part1 folder was taken from the "blinky" project from the sample repo. I changed the name of the TARGET and SRC variables to be Part1 and Part1.c respectively. I changed the processor CDEF to -DSTM32F411xE from -DSTM32F407xx and changed the address of the linker script to point to the updated linker script: STM32F411VETx_FLASH.ld.

I also used the overall makefile (armf4). I changed the path to the startup script and to point to the updated script mentioned above. I also changed the paths to the CMSIS library and the system_stm32f4xx.c to reflect their new location in my repository.

I also used the system_stm32f4xx.c from the sample repo. I changed the name of the "set_sysclk_to_168" function to "set_sysclk_to_100" to more accurately reflect its functioning.

I also used the system_stm32f4xx.h file, but changed the PLL_M, PLL_Q, PLL_P, and PLL_N values to better fit with the STM32F411VE MCU. I also changed the name of the function declaration of the old "set_sysclk_to_168" function to "set_sysclk_to_100".

My Part1.c was a combination of the "external.c" and the "timer.c" files from the external and timer projects.

I combined the main functions from external.c and timer.c to create my main function. I then deleted any repeated parts (like setting up the LEDs) and called set_sysclk_to_100 instead of set_sysclk_to_168. Since the board I was using had a different clock speed, I updated the prescaler (TIM2->PSC) from 8399 to 4999 based on the formula listed in the comment on lines 69-75 in timer.c. I also changed the auto reload time from 10000 to 5000 to trigger the switching of the LEDs every 0.5s instead of every 1s.

I based my EXTI0_IRQHandler on the one found in external.c. I used the function definition, if statement used to check if the interrupt was coming from EXTI0, and line 50, which was used to clear the interrupt status. I deleted everything else in that function and replaced it with my solution, which was described above.

I used the TIM2_IRQHandler from timer.c in my code. The only changes I made were declaring "i"—the variable used in timer.c's TIM2_IRQHandler—as a global variable and changing the variable name from i to ledVal.