E85 Lab 6: C Programming and Microcontrollers

E85 Spring 2016
Due: 3/28/16
Note Lab 7 will be issued 3/23 and due 3/30

Overview:

This lab is focused on the use of C programming and more specifically on the practical use of C for embedded system programming on microcontrollers. In this lab you will practice writing C programs on a regular computer. You will then learn how to program a microcontroller at a low level. Finally you will apply what you have learned to create a hello world application without the use of libraries.

What to turn in:

1. Your answers to the questions in part A
2. Did you complete part B?
3. Your code for part C

Feedback:

Let us know:
What went well in this assignment?
Were there points of confusion?
How long did you need to work on this assignment?
Part A – C Programming

(1)

You are writing code for a CPU that does not have a hardware division or floating point units.

Write a C program that uses only integers that calculates the whole part of the division and the remainder without using % or / (IE using only +,-, control statements, etc.)

Your program should use an appropriate data structure to return the division and remainder as longs.

Check that your code works properly using gcc and the Cygwin terminal.

The command `cd <directory>` changes the directory you are in, the command `ls` lists the contents of the directory you are in.

Recall that you can compile with a line of code like:

```
gcc file.c -o q1.exe
```

Once your code is compiled it could be run by:

```
./q1
```

(2)

Write a program to sort an array of integers from lowest to highest. Use the minimum amount of memory. You should not need to make a copy of the array of integers.

Test its function for various short arrays.

Check that your code works properly using gcc and the Cygwin terminal.

(3)

As you are developing complex applications that use many input files typing the compilation commands at the terminal becomes tedious. To solve this problem tools were created to build/compile projects. A common tool for this function is make. It takes a file called makefile.

Look at the makefile contained in this zip:


What is the equivalent line command that is run when `make` or `make all` is run?

What happens when you run `make clean`?

Modify the file to compile your code for question (1). What did you need to change?
Part B – Tutorial on microcontroller programming:

By now you have put together your Claremont board you designed in the first lab of this class.

The Claremont board is built around the ATSAMD11 Cortex M0+ microcontroller. In this tutorial you will learn how to program this microcontroller from scratch. At the end we will have made a LED blink on and off. This is the normal first “hello world” application for embedded systems.

The most key document to understanding the ATSAMD11 is the datasheet:


Additional resources are available from Atmel here:


Atmel generally issues a single datasheet/manual for their products. Other vendors such as ST Microelectronics often split their manuals into an electrical/physical manual and a programming manual. The former information is mostly what is needed to create the PCB we made in Lab 1, while the latter is what we need for this and the next few labs.

The current version of the manual at the time of writing this lab is 925 pages in length. It can seem overwhelming at first, but these manuals/datasheets are almost always sectioned. The chapters and sections are critical to finding the information you need without reading more than you need to.

3. Block Diagram

One of the most important figures in these datasheets is the block diagram of this integrated circuit.

Reproduced at left the block diagram for the ATSAMD11 can be found in Chapter 3 of the datasheet. It shows all of the modules that compose the integrated circuit (IC) and how they are connected together.

It is an excellent reference for what each module of the IC depends upon as well as where they get their inputs and outputs from.

It also serves as a visual graph of the features of the IC.

From a programming perspective one of the most important figures in any digital system’s datasheet is the memory map. It is normally called the memory map but in this datasheet Atmel has called it
product mapping (Chapter 8, reproduced below). The memory map is almost always near the start of the datasheet/manual.

A memory map specifies where the registers that control the functions/states of all digital circuits “live” in memory.

Our microcontroller is a set of finite state machines, combinational logic and analog circuits. Each finite state machine has an associated memory (register(s)) that contains the state. In a memory mapped system each register is mapped to memory by giving it a unique address. This allows the registers to be modified in a uniform way no matter what the function of the module/finite state machine is.

Let’s look at the memory map of our ATSAMD11 in more detail.

Starting with the global memory space we there are areas marked undefined and reserved. Neither of these locations should be used. Reserved generally means it may behave oddly if you write values there. Undefined normally means those address do nothing.

The other blocks in the global memory space are Code, SRAM and Peripherals.

Code where our program we will write is stored. In the block diagram we see 16 KB of NVM (non-volatile memory, in this case its flash). The range of addresses from 0x00000000-0x20000000 gives access to this memory. Notice this is larger than 16KB. In the expansion in the memory map, they show this. Only to 0x0004000 is really available.

The next block is SRAM. This is the memory where variables for programs will be stored. It will contain the stack and the heap of our program. It works like the main memory on a regular computer, but instead of GB of memory we have only a few KB.

The last block is peripherals ranging from 0x40000000-0x43000000. This is in many ways the most interesting block of memory because it contains the registers for all the modules in the block diagram. If the CPU with its SRAM and FLASH is a brain. The peripherals are the arms, legs, eyes and ears. They let the chip interact with the world. This introduction to microcontroller programming is to make the processor turn on a LED. This is the “hello world” program for embedded systems because it establishes
like hello world a basic communication ability. You can ask a binary question with a light. A foothold to
learn more.

Returning to the memory map, we see that inside the peripheral memory there are 3 bridges (A, B, C).
Our task is to determine which block will let us toggle the voltage of a single pin high and low to make an
LED blink.

The importance of the peripherals is born out in the contents of the datasheet. Chapters 11-34 are all
concerned with peripherals. The CPU itself gets very little coverage because it is covered in separate
manuals which were written by ARM to document the functions of a Cortex M0+ CPU.

For now we are looking for the peripheral that will let us turn pins on an off. If you skim the “overview”
of each chapter from 11-34, you will eventually reach PORT: “The Port (PORT) controls the I/O pins of the
microcontroller”. Atmel and some others call this PORT, but most call it GPIO (general purpose input
output). Knowing these two names you will find what you are looking for much quicker.

Now we must:

1. Learn how to use PORT to toggle our pin
2. Determine how to load and run a program onto our microcontroller
3. Write a program that toggles PA4, compile, load and run this program on the Claremont board.

The PORT chapter is 20 pages long. Much more manageable than the 925 of the full manual. Most of it is
tables and diagrams. You should always read the overview of a peripheral and look at the block diagram,
after that is it often helpful to look directly at the register summary.

From this we see there is DIR, DIRCTL, DIRSET, DIRTGL, OUT, OUTCLR, OUTSET, OUTFL, IN, CTRL,
WRCONFIG, PMUX*, and PINCFG*.

Once you gain experience just this list will make it pretty clear what will need to be set to make it work as
desired. For now we can read the register descriptions in 22.8. The other items we need to read are the
function description (22.6) and product dependencies (22.5).

Please read these now.

From the product dependency section we learn:

1. That we need to look at the configuration of the pin multiplexers on page 12.
2. On reset all the pins are set to be inputs. We need to set one pin PA4 to be an output to use it to
   output a voltage.
3. That we need this peripherals’ clock to be on.
4. We also learn we could protect the registers involved and how it operates while the CPU is halted
   etc.

Going to page 12 we immediately read: “Each pin is by default controlled by the PORT as a general purpose
I/O”. Our goal being to use it as GPIO we can stop reading, the default setting is what we want, so have
no code to write here.

Our answer to point 3 is given in table 15-1 on page 109. The clock needed (CLK_PORT_APB ) is on by
default.
From this we can conclude all we need to do is configure the pin as an output (item 2) and set its value. This is written nicely in section 22.6.1.

We conclude we must write to the DIR and OUT registers. DIR to set the direction to output and OUT to set the value.

We look at the register summary to see that DIR is located at *offset* 0x00 and OUT is located at *offset* 0x10. They are listed in this relative manner because the creators of this module want to be able to map it anywhere they want to in the global memory map! This way they can make multiple copies or just reuse their block in another chip and put it somewhere else. They also do not have to carefully update this section of the manual for each chip where it might be at a different memory location. Instead they just give the offset for the registers in this module.

To get the absolute address for where DIR and OUT are located we refer to the main memory map. We see at once that PORT is in bridge B at address 0x41004400. This address is called the *base address*.

We want PA4 to be an output so, counting the bit places from 0, we must write 1 to the 4\(^{th}\) bit of memory location to 0x41004400 (DIR). To turn PA4 on we must write 1 to the 4\(^{th}\) bit of memory location 0x41004410. To turn PA4 off we must write 0 to the 4\(^{th}\) bit of memory location 0x41004410.

We have seen how pointers let us write to memory locations directly using the indirection operator *.

We can now make use of this to write our C code:

```c
#define BasePortA 0x41004400
#define OffsetOut 0x10
#define OffsetDir 0x00

int main()
{
    *(unsigned int*) (BasePortA + OffsetDir) = (1<<4); //DIR PA4 to output

    while(1)
    {
        *(unsigned int*) (BasePortA + OffsetOut) |= (1<<4); //PA4 on
        nopDelay(5);
        *(unsigned int*) (BasePortA + OffsetOut) &= ~(1<<4); //PA4 off
        nopDelay(5);
    }

    return 0;
}
```

The * at the left of each line is the indirection of the pointer. The item in brackets tells the compiler that this should be treated as an unsigned integer (an int on this CPU is 32-bits on this CPU as are the DIR and OUT registers).

In place of typing out the hex for the bit to set and clear (1<<4) is used to clearly indicate it is PA4 being configured. The |= and &= operations allow the setting and clearing of the correct bit when coupled with the negation of (1<<4). In total each line sets one memory location. while(1) means the code will run forever without the return being reached.
The final piece of the program is the delay function:

```c
// Burn CPU cycles and spin doing nothing
void nopDelay(unsigned long delay)
{
    unsigned long i, j;
    for(j = 0; j<delay; j++)
    {
        for(i = 0; i<0xFFF; i++)NOP();
    }
}
```

This code does nothing but count in some loops. Most optimizing compilers will remove this code on compilation because it does nothing. To defeat this we defined a NOP() macro:

```c
#define NOP() __asm volatile ( "mov r0,r0" )
```

This is called inline assembly (__asm) the volatile keyword tells the compiler not to remove this instruction in any sort of attempt to optimize the code as it may be changed by some process the compiler does not know about. Here it really is not but we do not wish the code to be deleted. We will be doing some assembly programming soon but for now just know this copies r0 to r0 and therefore does nothing. Because the loops contain a volatile instruction the compiler will leave them in. The result is the nopDelay function creates a delay based on how fast the CPU can count, and the value of `delay`.

From code to binary to board:

This concludes the C programming to do our task. Our final task (3) is to actually load it onto a Claremont board.

To do this we need a set of tools called a toolchain. This is normally a compiler suite, plus some means of programming the chip.

The means of programming the chip is a serial wire debugger (SWD) sometimes called a SWG JTAG (Joint Testing Action Group) or even just JTAG.

Our SWD debugger is specific to ARM chips and is called a CMSIS-DAP (Cortex Microcontroller Software Interface Standard Debug Access Port).

We selected an IBDAP from Armstart. Debug tools often cost 100s to 1000s of dollars we picked this debugger because it is <20 dollars, open source and compatible with multiple toolchains (Keil, gcc/openOCD, etc.)

Setting up a toolchain can be a significant amount of work. Selecting a toolchain is like picking CAD software. Like PCB CAD software there are a lot of toolchains available for ARM microcontrollers. More now than ever before. IAR, Keil (now part of ARM itself), CrossWorks, CooCox, GCC/Eclipse and many vendor specific tools. In this case the vendor specific tool would be Atmel Studio which is very nice. Vendor specific tools often are free and very convenient. Sometimes they cost money and sometimes they are unrestricted. Industry tools can cost between 500-5000 dollars a seat.

In this lab we will use a barebones GCC toolchain to see the basics of how compilation works. In the next two labs we will use Keil’s uVision Lite environment which is free but code size limited.
Our simple barebones toolchain consists of:

1. A text editor (Notepad++)
2. A GCC ARM cross compiler
3. OpenOCD
4. Build tools/Cygwin/Python
5. IBDAP SWD Debugger

The second item is a C compiler for ARM processors. It is called a cross compiler because it is written to run on x86 computers but compiles another machine language namely ARM and its sub-dialects.

You can get these tools from several places including:

https://launchpad.net/gcc-arm-embedded
https://gnuarmeclipse.github.io/

Source code is also available. OpenOCD binaries are available from the second link. Source is available from:

http://openocd.org/getting-openocd/

We have already met Cygwin and the build tools in part A of this lab.

Essentially the process for compiling a program for our microcontroller is the same as it was for the regular computer. There are a few differences though.

Please download: http://pages.hmc.edu/bbryce/E85S16/files/lab6tut.zip

Extract the contents to new folder for this lab on Charlie.

We will now look at the files contained using a text editor such as Notepad++.

Look at main.c, this is our program of interest as written above.

Next look at the Makefile. It is heavily commented a full reading is encouraged.

The makefile is not fundamentally different from the simple one you made in part A of this lab. It is longer though. Three things standout. First, the number of options needed to compile the code properly. Second, it makes use of a “linker script”. Third, that there is an external call to handle programming via OpenOCD.

The additional flags and options are largely needed because the ARM cross compiler can create many flavors of machine code for different variants of ARM CPUs. This occurs on x86 as well but because the compiler is not running on the system it is compiling for it cannot detect the right flavor, you have to tell it.

The linker script (samd11d14as_flash.ld) is provided by Atmel and tells the compiler what memory addresses should be used for the start of each type of code the compiler makes. Atmel’s is more complex than it perhaps has to be but it is comprehensive and requires no effort to reuse. Generally the linker script is just a set of labels to tell the compiler where to put chunks of binary machine code.
Finally the external call to OpenOCD is handled via a python script called flash.py. OpenOCD is a tool to communicate over JTAGs and debugger interfaces. There is a local file called openocd.cfg that contains the information needed to talk to this particular chip. The external python script handles the commands to flash the chip. If you wish you can look at these files contents.

The remaining files are system_samd11.c and startup_samd11.c as well as an include directory with many header files. These come from the CMSIS software packages that Atmel provides. ARM microcontrollers and processors have interrupt vector tables. This tells the CPU what to do when something external happens. The startup code fills out these tables. They also allow the CPU to setup functions to allow for standard C system calls. Atmel did this with the entire CMSIS library plus the line __libc_init_array(); I have commented this line out in startup_samd11.c so that only the peripheral header files are needed. The ramifications of this are that some standard library functions will not work (printf does not have a standard out to print to for instance, malloc will not work either).

**Programming Claremont with the makefile environment:**

Having got a quick feel for the contents of our makefile toolchain we can now make our board blink.

*Follow the following steps:*

1. Plug the IBDAP into the computer you are using
2. Plug the ribbon cable into both IBDAP and your Claremont board as shown in the next photo
3. Plug your Claremont board into a USB port to power it
4. Launch Cygwin and use cd to go to the directory where you unzipped the example program
5. Type make flash (you may get an administration request, it is okay to ignore it, it should still program)
You should get something like:

```
$ make flash
"C:/Program Files (x86)/GNU Tools ARM Embedded/4.9 2015q3/bin/arm-none-eabi-
objc" -0 binary main.elf main.bin
python flash.py main.bin "C:/Program Files
(x86)/GNU ARM Eclipse/OpenOCD/0.9.0-
201505190955/bin/openocd"
GNU ARM Eclipse 32-bits Open On-Chip Debugger
0.9.0-00073-gdd34716-dirty (2015-05-19-09:56)
Licensed under GNU GPL V2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: JTAG Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 1.0
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO
= 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd14d14.cpu: hardware has 4
breakpoints, 2 watchpoints
Info : accepting 'telnet' connection on
tcp/4444
Info : SWD IDCODE 0x0bc11477
Info : at91samd14d14.cpu: hardware has 4
breakpoints, 2 watchpoints
Info : accepting 'telnet' connection on
tcp/4444
Info : SAMD MCU: SAMD11D14ASU (16KB Flash, 4KB RAM)
Info : SAMD MCU: SAMD11D14ASU (16KB Flash, 4KB RAM)
wrote 1024 bytes from file main.bin in 0.246184s (4.062 KiB/s)
Open On-Chip Debugger
> reset halt
Info : SWD IDCODE 0x0bc11477
Info : at91samd14d14.cpu: hardware has 4
breakpoints, 2 watchpoints
Info : accepting 'telnet' connection on
tcp/4444
Info : SAMD MCU: SAMD11D14ASU (16KB Flash, 4KB RAM)
wrote 1024 bytes from file main.bin in 0.246184s (4.062 KiB/s)
> reset run
The light should be blinking!
```
Part C – Say Hi over a UART

In part B we setup GPIO to turn on and off a light. In this part you will setup a UART to send the message “Hi”. UART uses two pins: TX to send data, RX to receive data. Both transmission and reception are done asynchronously.

The ATSAMD11 has 3 SERCOM modules built into it each is capable of performing as a UART. This is in chapter 25 of the manual. Reproduced below is how a UART (Universal Asynchronous Receiver Transmitter) transmission works in the form of a timing diagram:

![ UART Timing Diagram]

The module in the ATSAMD11 is actually a USART (S = synchronous), but we will use it as an asynchronously and thus just say UART. The diagram above is a flavor of RS-232 like transmission. This type of transmission uses a start and stop but to let the receiver see where the data is. The clump of data is called a frame or packet. Devices communicating over a UART link agree ahead of time on the format, this consists of the arrangement of bits and the time between them.

In the UART transmission above once the receiver sees the falling edge of the start bit it begins timing. Edges between the bits may not exist (for instance two consecutive 0s or 1s), but by looking at data line at the correct intervals of time from the falling edge of the start bit the receiver can sample each of the binary bits 0-8, then parity if it exists. Transmissions end with one or more stop bits which guarantee a transition from high-to-low for the start bit.

Notice in the diagram above that the first bit sent is the least significant bit 0.

In the datasheet notice that not all registers are 32-bits.

To send our desired message “Hi” message you will need to apply the knowledge you gained in part B to configure the SERCOM and send the needed message. Please configure the UART 8-N-1 at 9600 BPS (8-bits, no parity, 1 stop bit). Use PA14 as TX and PA15 as RX.
The general structure of the program will be:

```c
int main()
{
    setup peripherals
    while(1)
    {
        Send "Hi " over the UART
        Delay
    }
    return 0;
}
```

Setting up the UART is significantly more complex than GPIO. Please study the dependencies before looking at the provided code base so that you can understand it.

To complete this part you will need to finish the provide code.


There are many lines missing. In some places there comments as hints of what needs to go there. You do not have to use this code at all if you do not like it. You just have to make the UART say “Hi”.

Remember you have two LEDs you can turn on/off to have the CPU tell you something about what is going on with your code.

**TeraTerm + USB/UART Bridge:**

To check if your code works physically attach the USB to UART bridge cable to your Claremont board. Attach the white wire to Tx on your Claremont board and the black wire to GND on your Claremont board.

Start TeraTerm and connect to the USB serial port (It will be called Prolific, but the COM port number will be different on different computers) at 9600 Baud.