One Month 2015 Microcontroller Workshop

The purpose of this document is to provide a documented hands on approach to learning basic and advanced microcontroller, MCU, functions. Please go through this document at your own pace. You will cover all the MCU tools necessary to create the code for your One Month payload and learn how to appropriately connect sensors, wires, and other useful things to your MCU.

Contents

Required Supplies

To complete this workshop, you will need:

- Computer with Atmel Studio 6.2 installed
- The Xmega128A1 Xplained Evaluation Board
- An Atmel-ICE or a JTAG-ICE3
- Necessary power cables for programmer and Xplained board
- Patience!

Xplained board

Debugger (same thing, one just has a plastic covering)

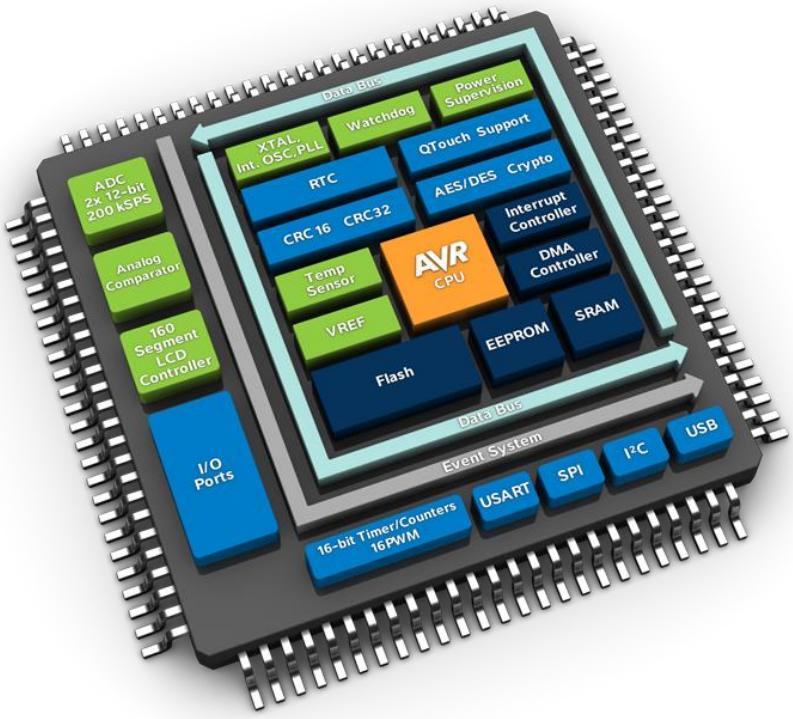
What is a Microcontroller

A microcontroller is an integrated circuit consisting of a CPU core, memory, and programmable input/output peripherals. Now what does this really mean in simpler terms. The MCU is like a small computer that does tasks it is programmed to do. What separates the MCU apart from a regular desktop computer is the size, power consumption, processing capabilities, and most importantly the job the MCU is supposed to do versus a normal computer. A normal computer is meant to crunch large numbers, play games, and run an operating system. The MCU is meant to be interacted with using input/output peripherals. These input/ouput, also called IO, peripherals can interact with the surrounding environment in some of the following ways:

- Determine if a button was pressed (General IO Port)
- Turn on a light at a specific time or interval of time called a period (Timer Counter)
- Read a voltage (Analog to Digital Converter)
- Produce a specific voltage (Digital to Analog Converter)

- Communicate with a computer, radio, or other serial devices (USART)
- Communicate with sensors following a standard protocol (SPI and TWI/I2C)
- Drive a small LCD screen like on a calculator
- Be the brain of a small satellite!

Each bullet point above makes use of a certain peripheral or a collection of peripherals to perform some task. The concept of the MCU having a main core then supporting peripherals is sometimes a hard concept to grasp. The picture below will further help convey this idea.

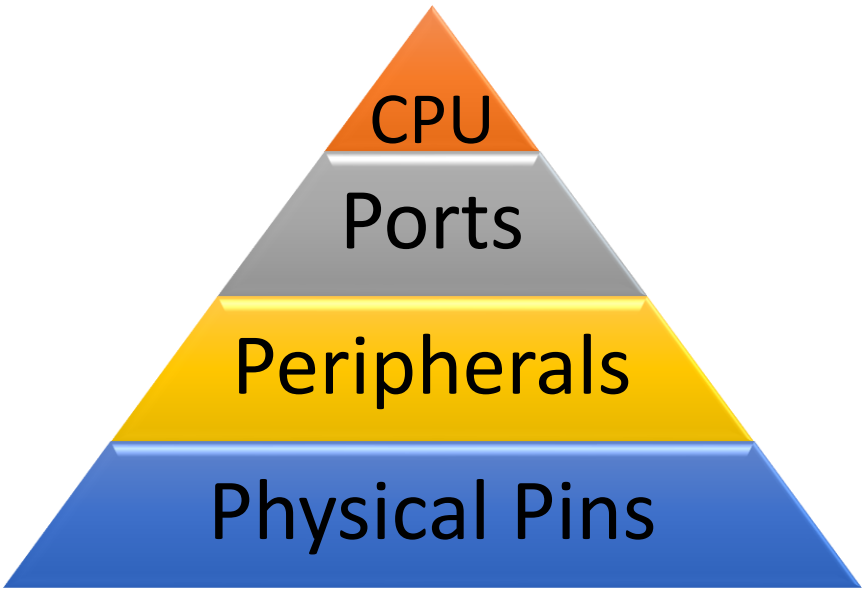


The blocks within the light blue arrows make up the main core of the MCU. The orange block is the CPU. This does the number crunching of the MCU. The CPU is also where information from the peripherals will be used to make decisions on what to do next. Example, The IO port detected a button was pressed, the CPU will then decide if it should tell the port to turn on an LED. The blocks around the left side and bottom of the illustration above represent the peripherals. The peripherals must be setup by the CPU, but then run independently leaving the CPU to perform other operations while waiting for information from the peripherals.

Another important topic to consider when learning about the MCU is the clock. Think of the clock of the MCU as the drummer in a march. The drummer sets the tempo of the marching or action. The clock sets the tempo of the device. On each clock cycle, the CPU will execute an instruction. If the system clock of the MCU is running at 2 MHz, this means the CPU will execute

an instruction 2 million times a second. I introduced a new concept of clock in the previous sentence, system clock. The system clock can be thought of as the master clock. It determines the speed of the CPU. The peripherals also need clocks to run in order to know when to perform tasks themselves. This kind of clock is called the peripheral clock. The peripheral clock can be just as fast as the system clock or can be an even division of the system clock. Most of the time, the peripheral clock is slower than the system clock. Each peripheral has its own clock that must be configured and is specific to that peripheral.

To finish up this section, let's examine the "hierarchy" of the MCU. At the highest level we have the MCU itself as a single unit. Looking at the Microcontroller on the Xmega128A1 Xplained board we see 100 metal legs sticking out of the plastic box. These are pins and these are what we connect to so we can power the MCU and transfer information back and forth via the peripherals. However, there is a more complicated hierarchy that is inside the plastic box that must be understood in order to program the MCU.



The CPU controls the Ports, the Peripherals are associated with a particular port, and the Pins are controlled by its Port and Peripherals. The Xmega128A1 has 11 ports A,B,C,D,E,F,H,J,K,Q,R and are referred to as PORTA, for example, during programming. The Xplained evaluation board has four headers with male pins that can be used for this project. This means you have physical access to Ports A, C, D, and F. Other ports have been already used for other purposes on the board but you can still use these. For example, the pins that belong to PORTE have been connected to LEDs that you can control in software. You will use these LEDs when you learn how to blink an LED and how to use a Timer Counter later in this document.

Each port has a set of peripherals that can be used. Some examples of peripherals were already listed above, however, the following sections teach how to use specific peripherals so they will be discussed in further detail later. Each port has 8 pins associated with it. Issuing a port wide

command will affect all eight pins. You may also issue pin specific commands. Peripherals also have access to these pins such as the Timer Counter. When a peripheral is using some of the pins on the port, the peripheral now has control of those pins and issuing a port wide command may not affect those pins.

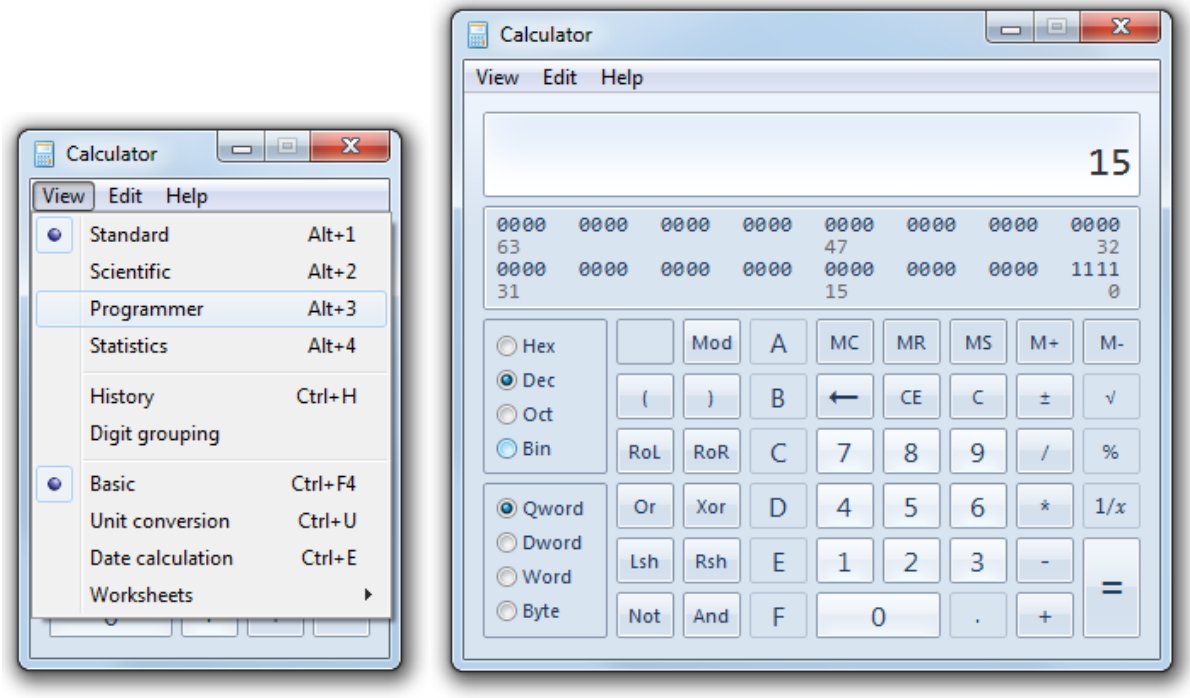
Bits, Bytes, and Registers

As you may have heard or deduced by now, computers do not count like we do. We count in what's called the decimal system so we count up to 9, then roll over back to 0 and put a 1 in the ten's place. This idea of increment until the highest number in a number place, then roll over and increment the next place is pretty universal in all number systems.

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

A computer will count in bits. This number system is different than our decimal system in that there are only 2 unique numbers instead of 10 in the decimal system. We call this counting system the binary system. Binary, as in 2, get it? These numbers are 0 and 1. How do we count in binary? Simple, count up until the largest unique number, then roll over to 0 and increment the next number place.

It is not important you know how to count in binary by heart but rather you know what binary is and can convert to decimal and vice versa. An easy way to convert from binary to decimal and back is to open the standard windows calculator, click on view, then hit programmer. The way you convert is to click on the starting number system, Dec for decimal or Bin for binary, enter the number, then simply click on the number system you want to convert to.



From the table on the last page, we can see that the highest decimal number 4 bits can represent is 15. So 4 bits can represent the numbers 0-15 for a total of 16 numbers. We can mathematically express this relationship by:

$$Largest\ Decimal\ Number = 2^n - 1$$

Where n equals the number of bits. So for example, if I have an 8 bit number, the highest number I can represent is 255. A 16 bit number can represent 65,535 and a 32 bit number can represent 4,294,967,295!

Another concept when it comes to bits is bytes. A byte is simply a grouping of 8 bits. This is similar to saying 12 items make a dozen. A dozen is just a grouping the same way a byte is. Now, the byte is a little more special because much of how the computer stores information and transfers information and computes math is done in multiples of 8 bits. The registers and variable types are good examples of this.

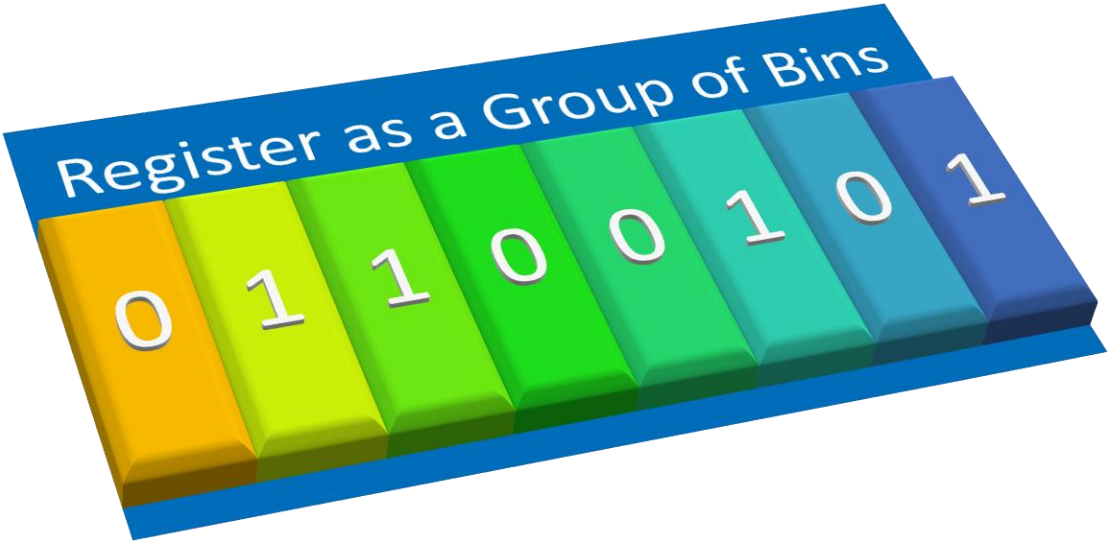
The compiler will convert our C code we wrote in Atmel Studio into binary for the MCU to use. Great! If the compiler does all the work for us, then why do we need to know what bits and bytes are? Well, in order to interact with the pins, peripherals, and ports, you will need to put data in, or read data out from the registers!

Registers can be a complex subject to somebody who has never heard of them before. It definitely was for me. This is not a problem because we are going to break down this notion of a register into a couple concepts to see how you actually use them.

First and foremost, a register holds data that the programmer has access to. How much data does a register hold? On the MCUs you will be using, the basic length of a register is 8 bits, or 1 byte. This is why these MCUs are referred to as "8 bit Microcontrollers." However, there are many examples where two 8 bit registers work together to form a 16 bit register. This type of register holds 2 bytes of information. The reason for having a 16 bit register vs an 8 bit register is because an 8 bit register can only represent the numbers 0-255 while a 16 bit register can represent the numbers 0-65535. There are cases when larger amounts of data is needed and so registers will be 16 bit instead of just 8.

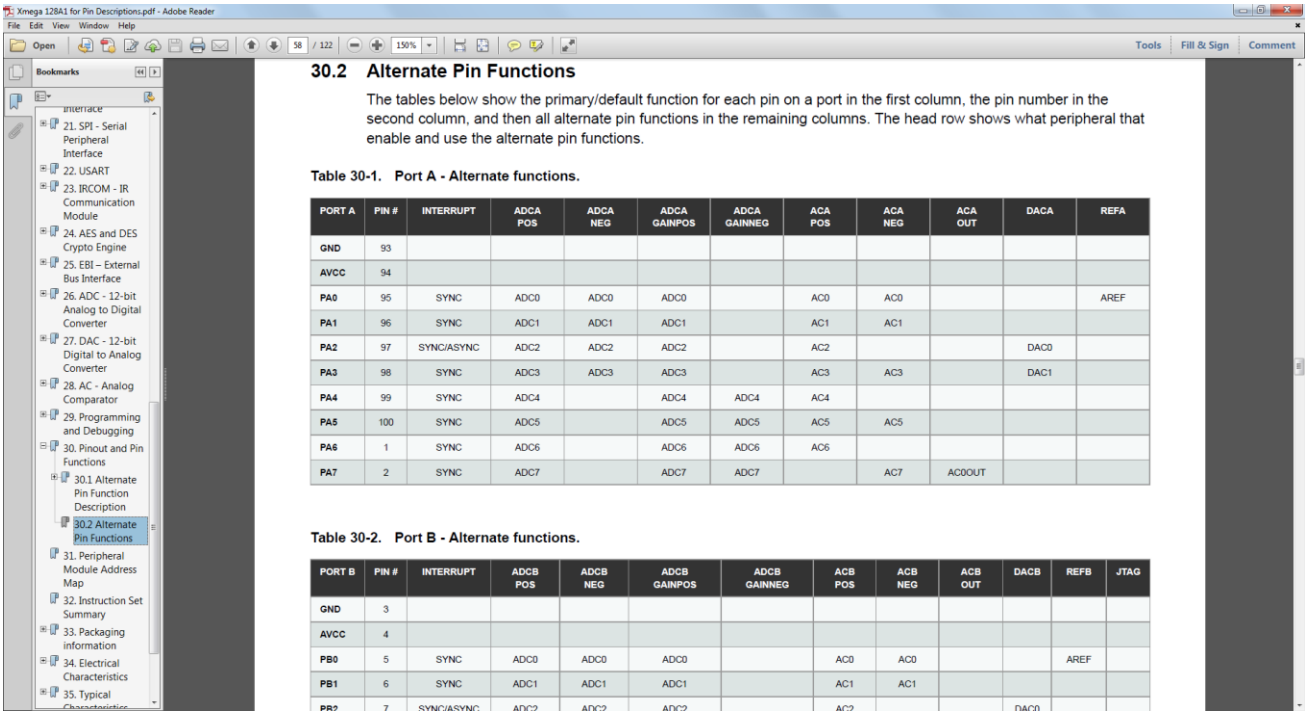
Okay so this register holds data and the programmer can access it, but what data does it actually hold? How do I know how to use it? The answer to these questions will be covered more in depth in the next section "How to Use the Datasheet" but for now I want to describe a register in two ways. A register's data will either visually represent a simple number where all 8 or 16 bits form the number, or where the register is like a group of bins and the 8 bits can be further split up and grouped together to form bins that hold 1,2,3… bits of data but there are multiple bins in each register. While these registers operate in different ways according to the datasheet, on the surface there is no distinction. You will find in the "Timer Counter" section that the MCU can count at a certain tempo to keep track of time. The registers associated with the counting fall into the first category where they represent a simple number. The graphic below shows a register that contains the simple number 10010110 in binary which converts to 150 in decimal.

You will learn in the “Blink an LED” section that each bit in the register is its own bin of information and that actually each bit in the register associated with a particular port corresponds to 1 of the 8 physical pins on the MCU itself. This kind of register is how you set the “settings” of peripherals, ports, and pins. The graphic below shows how each bit in the register is independent of one another because they can be thought of as being in their own bins.



How to Use the Datasheet

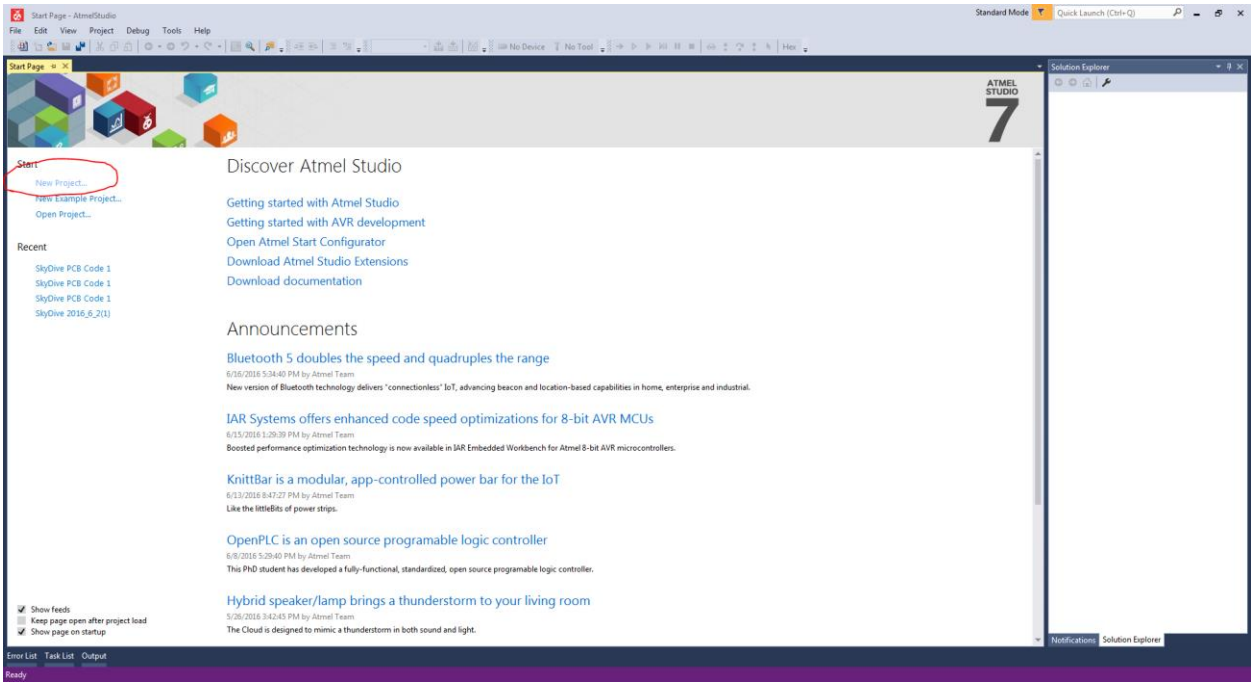
There are two datasheets that you will need in order to program the Xplained board. One is called the Xmega A Family datasheet. This holds all the information you need to actually program the MCU because the MCU on the Xplained board belongs to the A family. The other is called the Xmega128A1 and is specific to the actual MCU on the Xplained board. This is merely used for the names and descriptions of what the pins do on the actual Xmega. The screenshot below is the only section used in the Xmega128A1 specific datasheet. Most of the time you will use the Xmega A Family Datasheet.



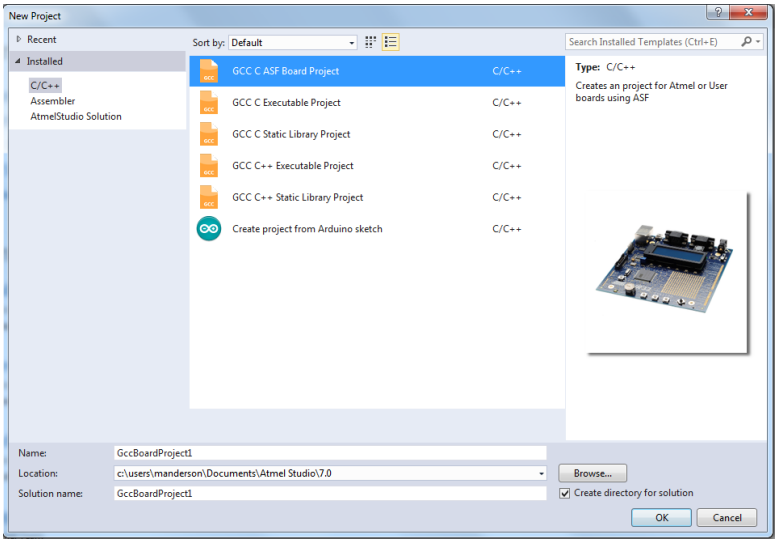
Open all datasheets in Adobe Reader instead of using a webpage as there is usually a nice table of contents that you can click through on the side. This will drastically save you time and frustration. Screenshots of things of interest will be given where applicable during the programming sections later on.

How to use Atmel Studio

To program the Microcontrollers, you will be using the IDE Atmel Studio 7.0. An IDE is just an acronym no one remembers that essentially means a visually pleasing program where actual code is written, compiled, and then uploaded.

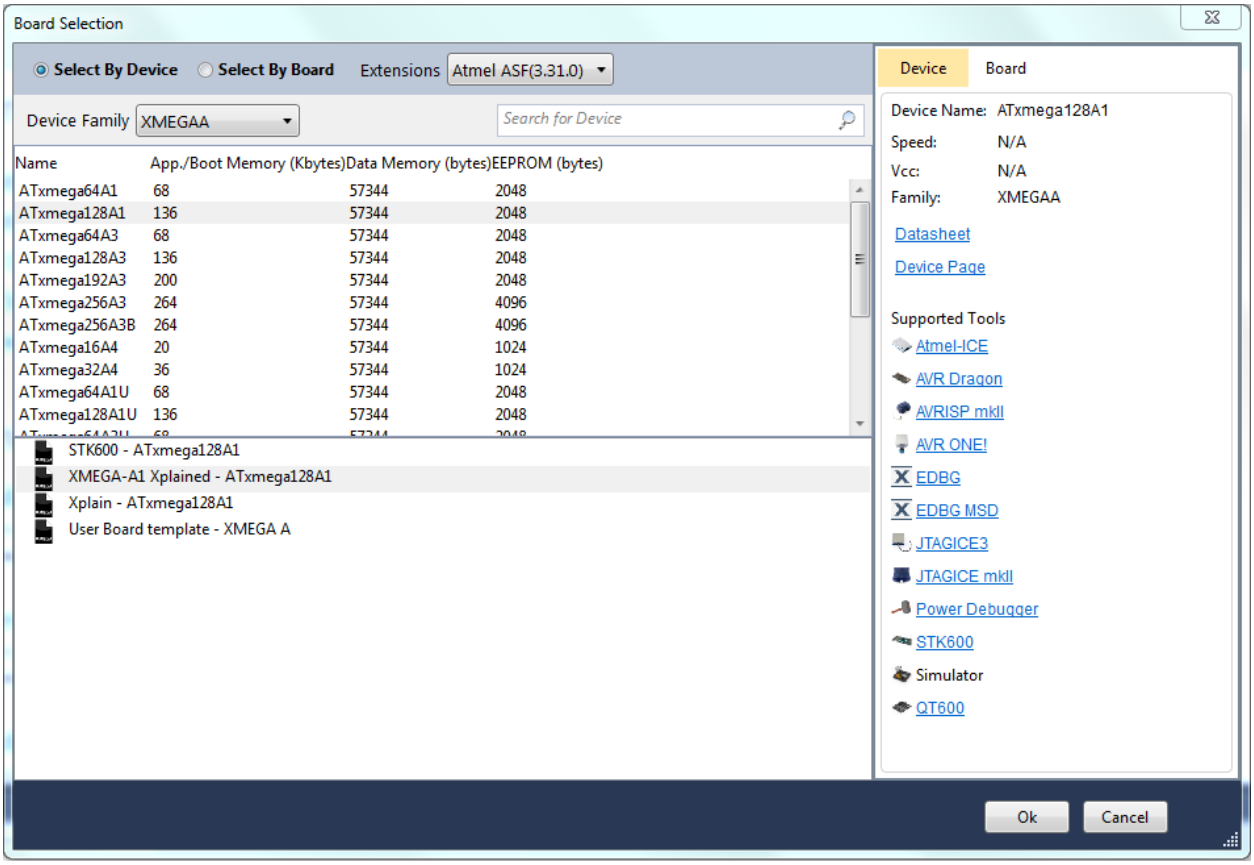


This is the startup page for Atmel Studio 7.0. I'm gonna explain to you guys how to start a project (it's not that hard, but there are a lot of steps). So first things first, click on the **New Project…** button under start (it's also circled in red in the pic above). The next screen should look like this:



You want to create a new ASF Board Project, so make sure the first option is chosen. Then change the name to like "Team 1 One Month Code" or "Code for Team 1" or whatever. Then make sure that the file location is your team's folder on the drive, otherwise it'll be saved on the computer and then you'll only be able to code on that specific computer. Click OK.

Next, you'll come across a screen that looks like this:

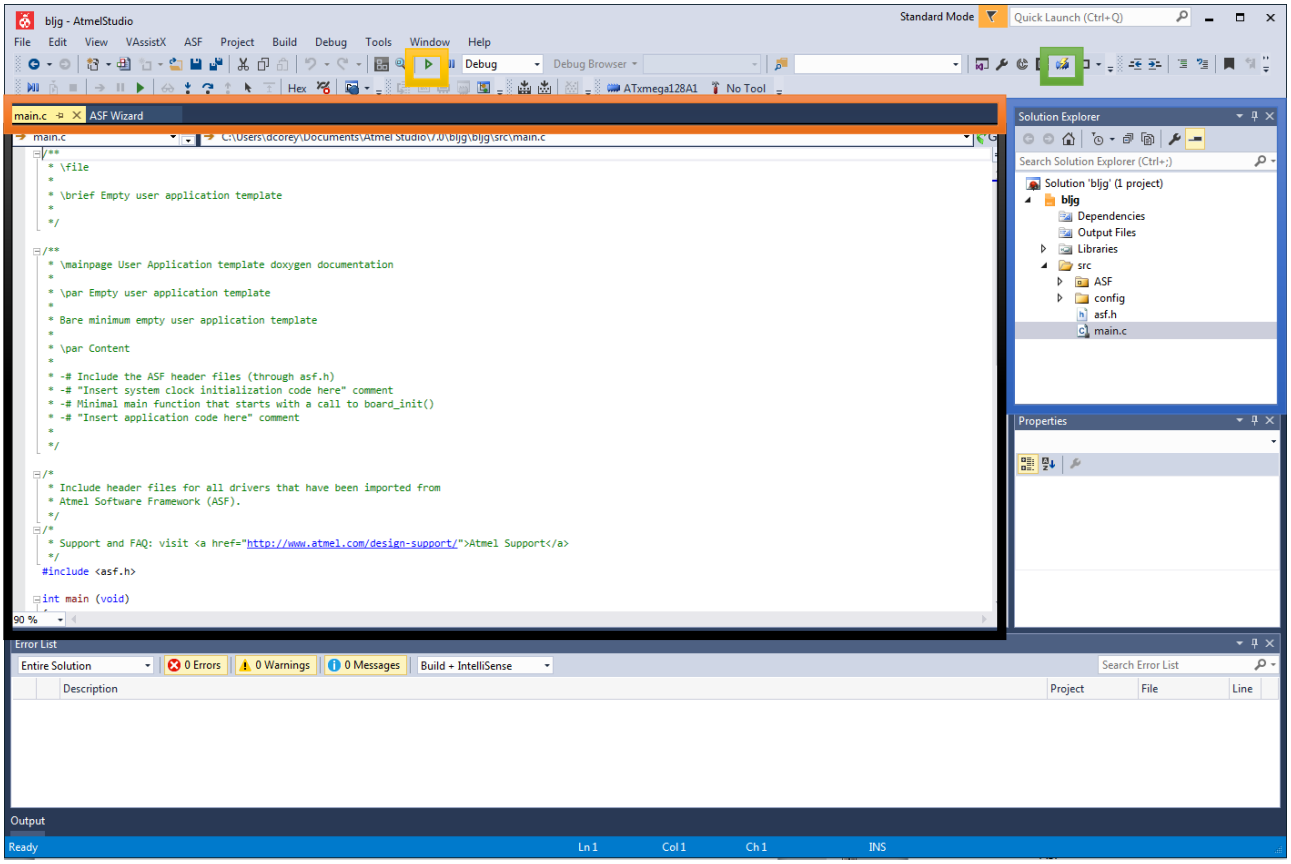


You want to change the **Device Family** to **XMEGAA**. Then click on **ATxmega128A1** and finally click **XMEGA-A1 Xplained – Atxmega128A1**. Then click Ok. There, your Atmel Studio project has been made.

An Atmel Studio project is the collection of files that has all the code for a single MCU. Let's say I am building a satellite and have one MCU as the main brain and another MCU control a rocket thruster. Each MCU in this example would have its own Atmel Studio project. Since your One Month Project has only one MCU, then you only need one Atmel Studio project.

To open the project from file explorer, simply click on "Open Project…" in the left hand side of the screen and browse through the folders to where it is saved. Hopefully you listened to me and put a copy in your particular team's folder. If you didn't, take some time to find it and then save a copy to your folder on the drive. For those of you who have found your folder, you will need to double click on the folder to go inside said folder. You will find another folder with the same name and another file with a ladybug on it. Clicking the lady bug file will actually open the template.

When your project opens, look for the window on the side that says **Solution Explorer**. Click the drop down button next to src and open up the file named main. This is your main file and is where the main flight code for your project will be.



While they don't have a colored box around them, The top section is standard save, open, cut, copy, and paste buttons and the bottom white space with "Output" on the top left is a display what code is being compiled and eventually what errors or warnings occurred. This will be shown later. The Orange Box is where tabs of different files are open like in an internet browser. Currently, main.c file is selected and is shown inside of the Black Box. The Black Box area is where actual code is written. In particular, the main.c (also called just main) file is where the flight code is written and where functions are used from supporting files to support the main flight code. The Blue Box is where the solution explorer is. A solution is the collection of all the files your project is using. The solution explorer helps you navigate to different files within your solution. However, you will never need to open the following:

- Dependencies
- Output Files
- Libraries
- ASF

Changing things within these files may do enough damage to the overall solution that you will have to start over. This leads me to another big point, save often! Just hit ctrl and then s. Please get into the habit of saving every couple of lines of code. The following files and folder are ones which you will be programming in:

- config
- main.c

You can also add folders to help you organize your code. In fact, we really want you to organize your code into different folders and files because otherwise it can get really messy and hard to debug. So do me a favor and go ahead and add a file under src called “drivers” (just right click on src and got to add… folder). This creates a new folder where you can put drivers. Drivers are kind of like software interpreters. The code written in a driver is used to translate the information coming from the sensors into language and numbers we can understand. Basically, any code used to get information from a sensor should probably be stored in the driver folder (at least that’s what I would do).

Another good organization technique is splitting your code into .h and .c files. In ATMEL, these two files have different uses. The .h files are used as initialization files and contain #defines, function prototypes, and maybe some variables that the main file, or some c files, would use. Below is a picture of a typical .h file for a timer counter driver.

```
#ifndef TIMER_COUNTER_INIT_H_
#define TIMER_COUNTER_INIT_H_

/* Place any #defines or function prototypes here */
void TCD0_init (uint16_t period,float duty);//For LED flash cycles
void TCE1_time(void);
void speaker(uint16_t period,int duty);

#endif /* TIMER_COUNTER_INIT_H_ */
```

The functions defined here in the .h file can be called in the main file so long as the main file has a line of code that reads **#include <timer_counter_init.h>.** Also, notice that the only thing in this file is the naming of functions, and not the actual code that performs calculations or executes commands. That is all found in the .c file. The .c files will use the functions defined in its corresponding .h file, but this time it actually gives the functions a purpose. The picture below is from the timer_counter_init.c file that shows what the function TCD0_init does.

```
void TCD0_init(uint16_t period, float duty){

    PORTD.DIR = 0b00000001;//Sets port 0 to out
    TCD0.CTRLA = 0b00000111;//sets prescalar to
    TCD0.CTRLB = 0b00010011;//Enables 1 pin, se
    TCD0.PER = period;//sets period
    TCD0.CCA = TCD0.PER - (TCD0.PER*duty);//The

}
```

There are two important things to remember about the .c and .h files. The first is that their names need to match. In the above example, both the .c and .h files were named timer_counter_init. That’s just proper syntax for ATMEL. Also, the relationship between the .h

file and .c file is not automatically applied. You need to make sure the .c file has **#include <timer_counter_init.h>** just like the main file. Otherwise, you just wasted your time.

Now that we've explained .h and .c files, we need you to make some specific modifications to your code so the board can "talk" to the computer. It's super important. So, please add a file under drivers called **uart_comms.c** Next, open up windows explorer and navigate to the Space Cadet Training -> 2016 -> Admin folder and open up the text file called **One Month UART Starter Code** and copy all of that code into your **uart_comms.c** file. Then, in your **Solution Explorer** navigate to config -> conf_usart_serial.h. This is going to be the corresponding .h file for your usart_comms.c file. When you open it, make sure the file has the code from the below picture in it.

```
#ifndef CONF_USART_SERIAL_H_INCLUDED
#define CONF_USART_SERIAL_H_INCLUDED

//These #defines are the "settings" for the USART communication
//The receiving end, computer or radio, must have the same settings or it won
#define USART_SERIAL_BAUDRATE            9600 //Also called 115k Baud
#define USART_SERIAL_CHAR_LENGTH         USART_CHSIZE_8BIT_gc //Each packet o
#define USART_SERIAL_PARITY              USART_PMODE_DISABLED_gc //No Parity,
#define USART_SERIAL_STOP_BIT            true //One stop bit

//This function prototype lets the compiler know we are going to use this fun
void UART_Comms_Init(void); //This will initialize the uart communications

#endif /* CONF_USART_SERIAL_H_INCLUDED */
```

So what you just did was #define new global variables in a .h file. Now you can use those names as variables within your project. If you look back at your uart_comms.c file you'll notice those variables names are used. So, you just set .baudrate to 9600. Yay you.

The penultimate setup task is to change the system clock frequency. By default, the MCU will run at 2 MHz. If you leave it like this, some of the code later in this workshop will work a little differently than you expect, because the sections on Timer Counters and Digital Communications assume that the MCU is running at its maximum speed of 32 MHz. To change the clock speed, open the file **conf_clock.h** in the config folder. There should be a few lines like this:

```
#define CONFIG_SYSCLK_SOURCE          SYSCLK_SRC_RC2MHZ
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC32MHZ
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC32KHZ
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_XOSC
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_PLL
```

Notice how everything except the line about for the 2 MHz clock is commented out. Comment out that line and uncomment the line for the 32 MHz clock to use that clock speed instead, like so:

```
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC2MHZ
#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC32MHZ
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC32KHZ
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_XOSC
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_PLL
```

Your board will now run at 32 MHz.

One last thing before these files are up and running: go to your main file. There are three lines of code we need you to type into your main file. Type this:

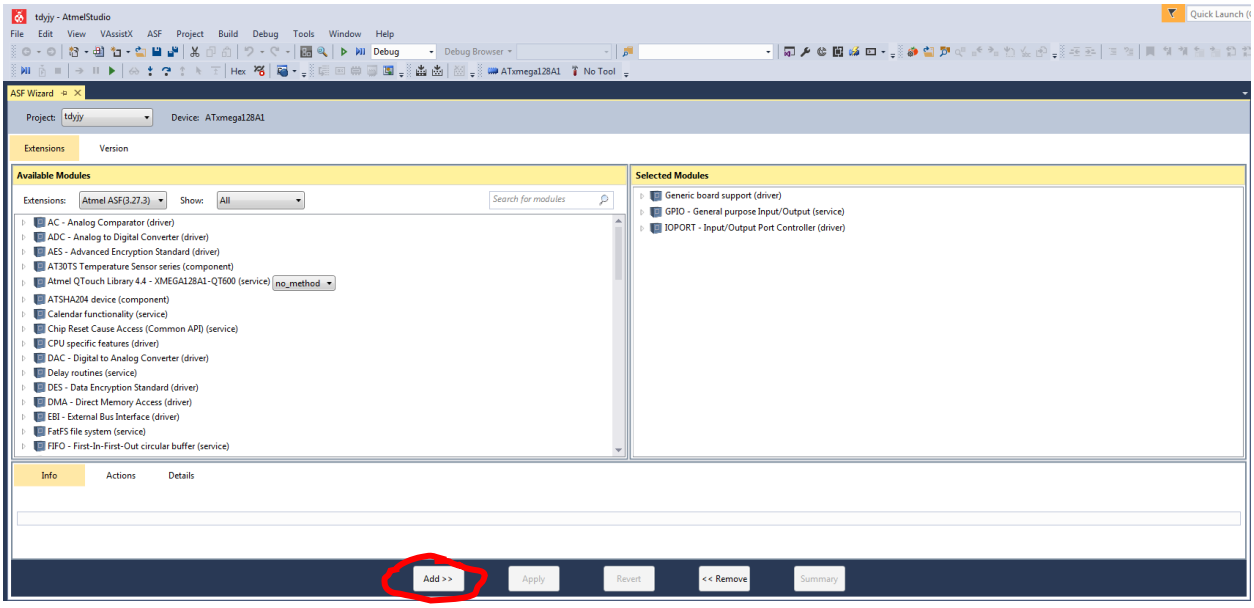
sysclk_init();

sysclk_enable_peripheral_clock(&USARTC0);

UART_Comms_Init();

What did you just do? Well in the first line you disabled all of the unnecessary system clocks that drain your battery. The next line enables the system clock for USARTC0, which is what we set up your USART on. The final line called the function we wrote in your uart_comms.c file so the code in that function actually compiled. Helpful tip: if you don't call your function, then it will never actually get used.

We've got one other thing to do before we move on with the coding. Look at the top toolbar and click on **Project -> ASF Wizard**. The screen that pops up should look like this…



The list you see on the right are available modules with pre-made functions that you can use. First we'll add the modules and then I'll explain them a little more. Scroll through the options on the left side til you find "Delay Routines" then click the Add >> button on the bottom. The module that you just added is incredibly useful for One Month. It allows you to call the function

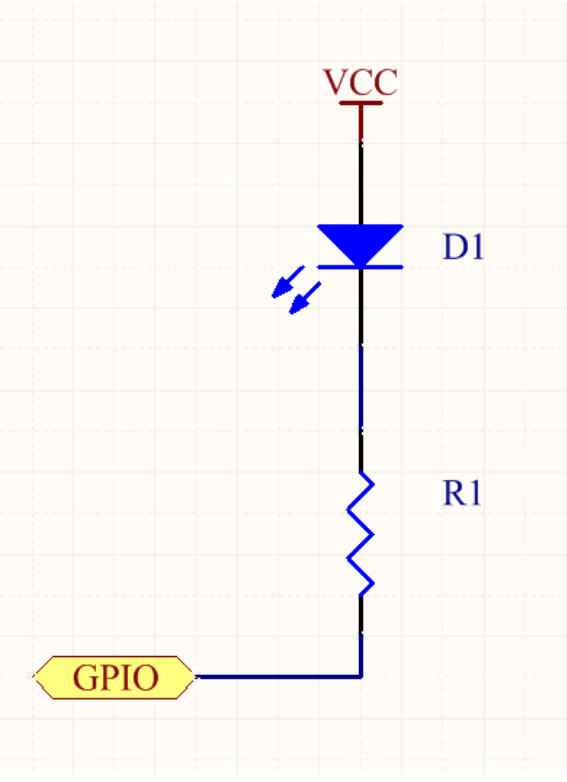
delay_ms() which pauses your code for a given amount of milliseconds. You've got a few more to add. Scroll through and add ADC and Standard Serial I/O driver. Then add the USART ones as well. That's all I got for now. Get ready to blink some LEDs.

Blink an LED

Go to the Solution Explorer to open the main file in your Atmel Studio project, if it isn't already open. It should look something like the example below. Notice the helpful little comments, words in green, that describe the layout of the main file. These are the kinds of things you should write comments on.

```
int main (void)
{
    /* Initialize the system clock, 32MHz, this also turns off all peripheral clocks */
    sysclk_init();
    sysclk_enable_peripheral_clock(&USARTC0); //For every peripheral, you must enable the clock like shown here. Ex. Timer counters, SPI, ADCs

    /* Example, Timer Counter on PORTE */
    sysclk_enable_peripheral_clock(&TCE0);
    sysclk_enable_module(SYSCLK_PORT_E, SYSCLK_HIRES); //You must have this line for every timer counter due to a flaw in the design of the chip

    /* Initializations */
    USART_Comms_Init();

    /* Flight Code */


}
```

You will write your flight code, and the example code in this workshop, below the commented section called /* Flight Code */. Now we are going to start programming the MCU to blink the LEDson the Xplained Board! The first question you should ask yourself is what Port are the LEDs on? The LEDs are on PORTE and LED0 being Pin PE0, LED1 being PE1, and so on. The LEDs have a little quirk to them. You would expect if you set the Pin to high (This also means 1 in programming language or if you measured the voltage of the pin, it would be 3.3V) the LEDs would come on because this seems logical. In reality, the makers of the Xplained board set them up a different way. Here is the circuit showing how the LEDs are setup:



Here we can see the LED already is supplied power through VCC, 3.3V on the Xplained board. Where we would expect ground to be, the LED is connected to an IO pin on the MCU. Setting the Pin high will not allow current to flow because there won't be a difference in voltage between the anode and cathode of the LED. Setting the pin to low (0 in programming language) will make the pin be 0V, essentially ground. This will allow current to flow across the LED and produce light. Keeping that in mind, let's write two lines of code to turn on the 8 LEDs. First, we must tell the MCU which direction we want the pins on PORTE. We need to do this because these pins can be both

output and input. Think about this as output is controlling something external to the MCU, like an LED. Input is taking information from something external, like if we want to know if a button was pushed or not. So, we need to set the output of the pins on PORTE as outputs. To find out how to do this, we must refer to the datasheet. I have pasted the part of the datasheet you need to read in this document for this module, but it may not be pasted in the future.

13.13 Register Descriptions – Ports

13.13.1 DIR – Data Direction register

Bit	7	6	5	4	3	2	1	0
+0x00	DIR[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- **Bit 7:0 – DIR[7:0]: Data Direction**

This register sets the data direction for the individual pins of the port. If DIRn is written to one, pin n is configured as an output pin. If DIRn is written to zero, pin n is configured as an input pin.

You may first notice that the 8 pins are labeled 0-7, this is very common in programming to start at 0 so unfortunately you will have to get used to it. The important part of this picture is the paragraph at the bottom. Read it and say what you think it says.

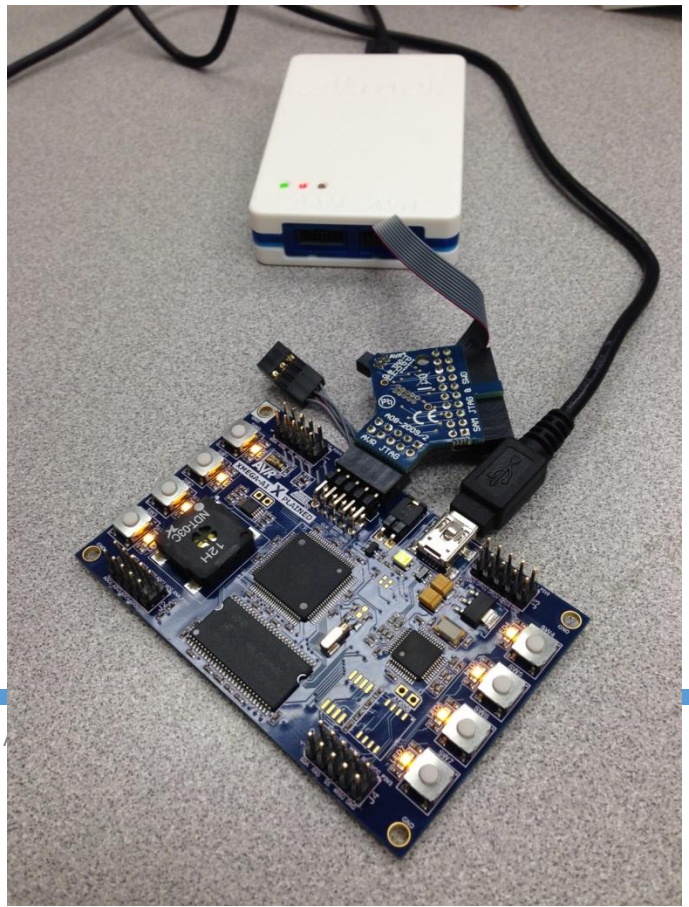
It says that to set a pin to an output, you need to set the corresponding bit to 1. This might sound very foreign but we can break it down. Each pin has a corresponding bit. Pin 0 corresponds to bit 0 and so on. That means this register operates as 8 independent bins. So if I wanted to set all the even pins to outputs, I would want the register to be 01010101. Notice how the far right represents Pin 0. Alright, let's write the line of code to set all the pins on the port as outputs.

```
/* Flight Code */
PORTE.DIR = 0b11111111; //This line sets all the pins on PORTE as an output
PORTE.OUT = 0b00000000; //This sets all of the pins voltage levels to 0V, which is logic 0 in programming usually

while (1)
{

}
```

Let's explain this a little more moving from left to right. PORTE tells the compiler that we are about to change something related to PORTE. The . in between PORTE and DIR is actually accessing a struct. A struct is simply an organizational "tool" in c programming to store data that is related to one another. Don't focus on this right now just know that you need to add a period after you type PORTx. DIR is the name of the direction register, as seen in the previous picture from the datasheet. Then comes the assignment operator, =, to assign the value on the right side to the register DIR on PORTE. Now comes a bit of syntax when entering numbers in Atmel Studio. When entering a binary or hexadecimal value, you need to preface the value with

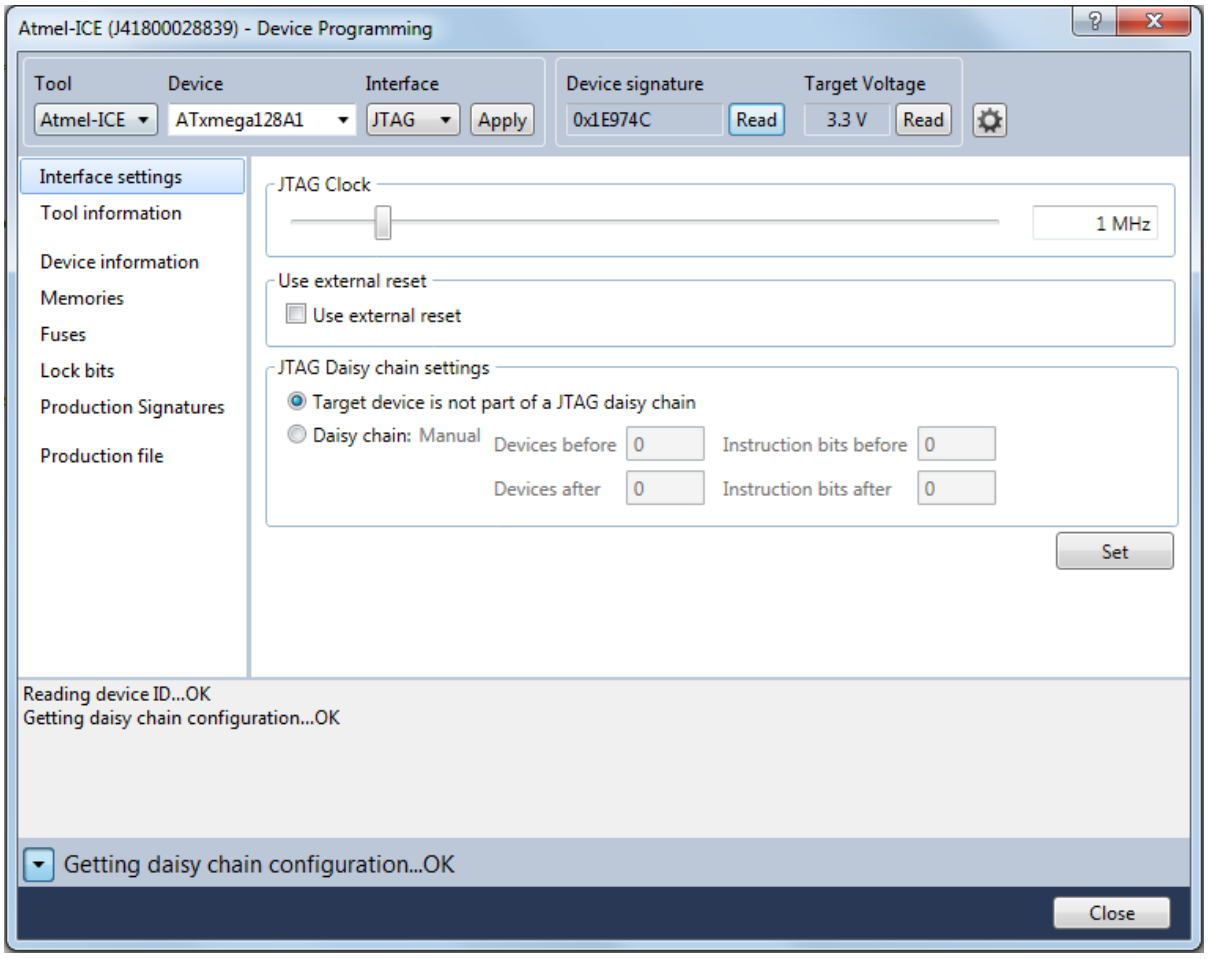
a 0b (zero and b) for binary and 0x (zero and x) for hexadecimal. This lets the compiler know the type of value you are entering. When entering regular decimal values, no preface is necessary. For example, 0b11111111 is the same as entering 255 because 2^8-1 = 255. The OUT register operates the same as the DIR register except a 0 means the pin will output logic low which is the same as 0V in our case. Writing 1s will make the output high, or 3.3V. We end each line of code instruction with a ; to let the compiler know we are done with that line. Below these lines of code there is what we call a while 1 loop. Remember that the while loop runs until the condition within the parentheses is false. When a 1 in in the parentheses, it will always be true and therefore always run. The purpose of this is to make sure the program never reaches the end of main. With MCUs, you don't want your program to reach the end of main because most of the time it will simply loop back to the beginning of main where your first line of code is and begin the whole program again. Imagine launching a satellite with an MCU that restarts its code every couple minutes because we didn't add a way for the code to not loop over and over. So don't forget to add a while 1 loop! Future NASA space probes are depending on you.

Alright, we have written our first bit of code, whew. Let's compile the program and upload it. This is the part where you need to get your programmer, MCU, and a USB-A to USB-mini cable. Hook up the programmer, programming cable, and MCU as shown in the picture below. Note, the pictured programmer is an Atmel ICE, some teams may have a gray Atmel ICE-III. They operate essentially the same. On the white model, there are two places to plug in the small connector, make sure you plug the connector into the AVR port and not the SAM port. Plug the 10 pin JTAG connector on the multi-way connector into the horizontal 10 pin JTAG pins on the Xplained board itself. Plug in the mini USB connector on the board and the 8 LEDs should start doing a pattern. Connect the micro USB cable to power the programmer. If you have done this correctly, a green and red light should come on the programmer. If this is not the outcome you got, ask for help. Also, when first plugged in, the computer will automagically download drivers for the Xplained board and the programmer. You may have to update the firmware on the programmer however. If so, just keep saying yes until it says done.



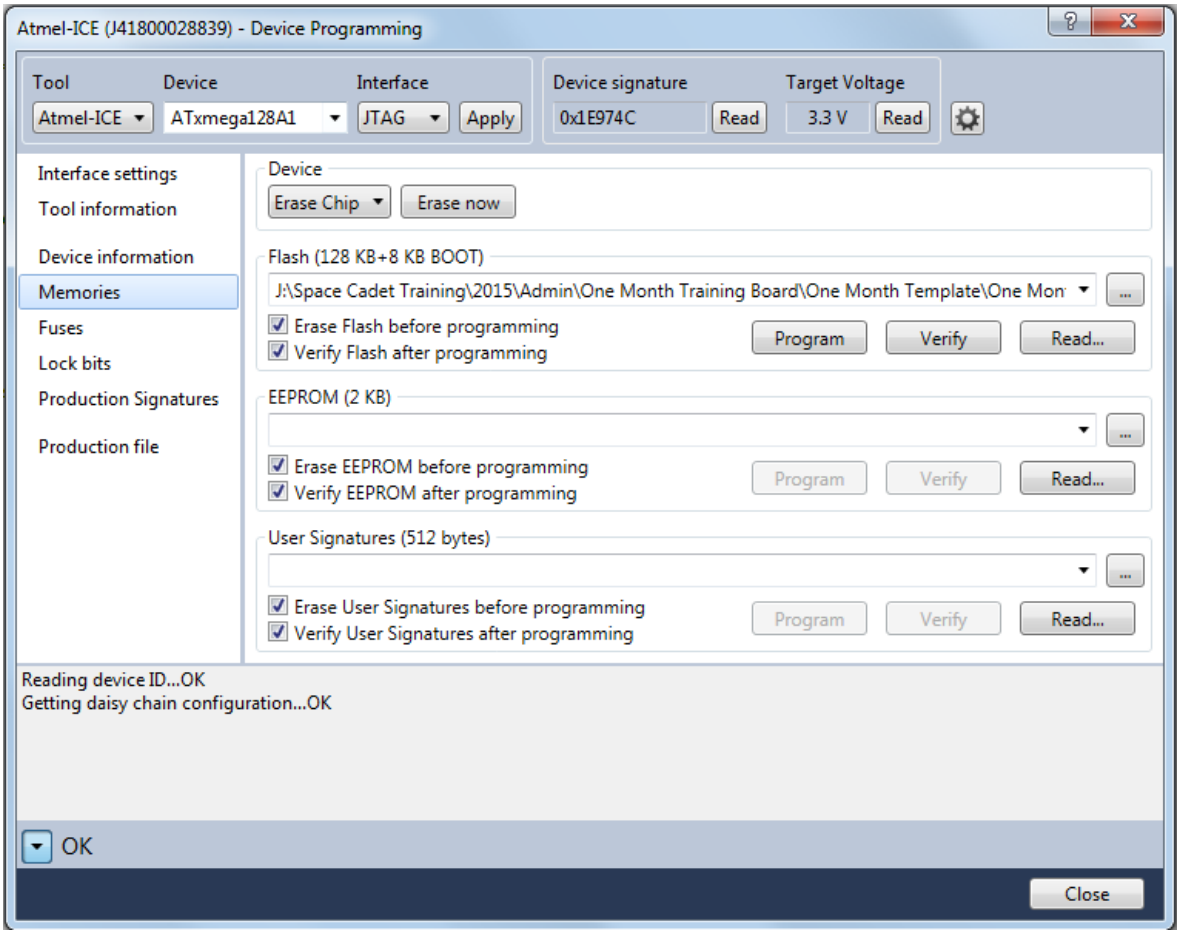
Now we must compile and upload the code. To compile, also known as build in Atmel Studio, press the F7 key and let it do its thing. After this process, look at the output window at the bottom of Atmel Studio. This is where it will tell you if you got an error, warning, or message. An

error usually means your code syntax is incorrect in one or multiple places and can be remedied by fixing the syntax error. An error could also mean other things and the compiler will try to tell you what the problem is so you can fix it. A warning means your code will run, just not in the way you might expect it too. Most warnings need to be fixed, others can be noted and ignored but please don't always assume it is going to be okay. I don't consider my code complete until I have fixed all warnings. Messages are rare and can mostly be ignored. When the code compiles correctly and there are no warnings or errors, it will build successful. This is good. To upload the code, press the button in the Green box that looks like an electrical component getting zapped by lightning. This screen will pop up.

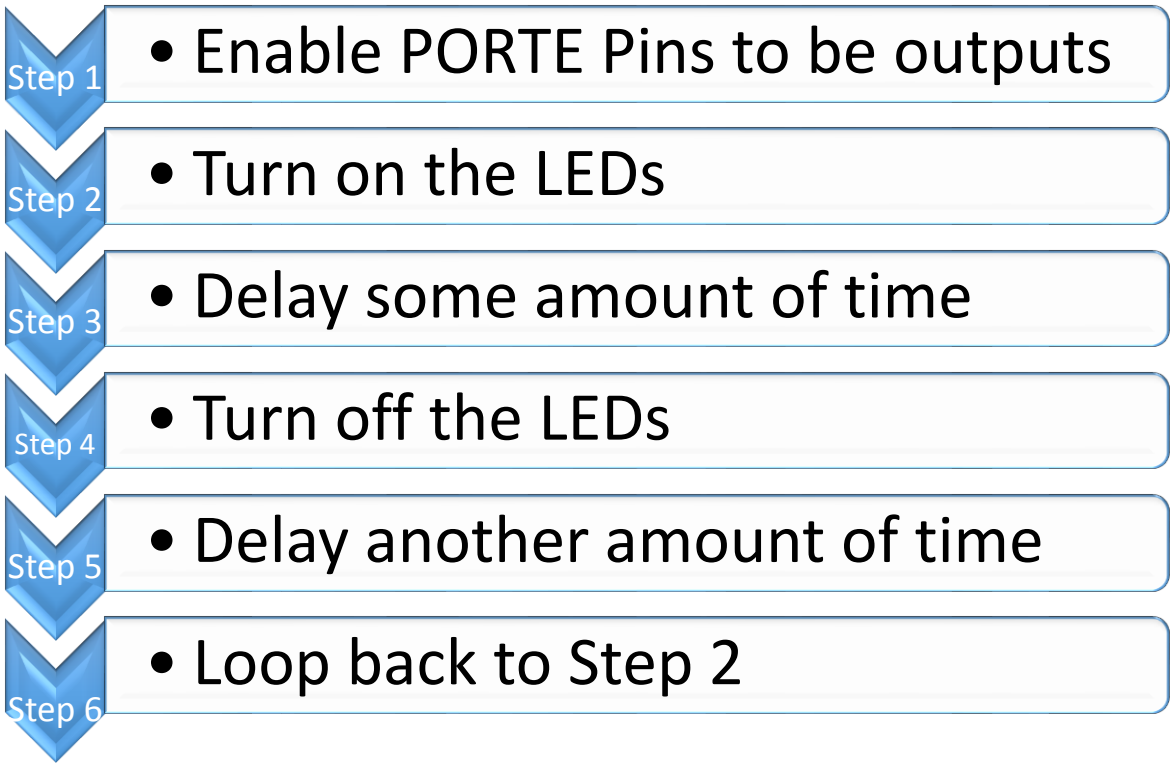


Select your programming tool from the dropdown menu. If it is not present, the computer is still downloading the driver, updating the firmware, or it's not plugged in. Under the device dropdown, select the MCU as shown in the picture since this is the MCU on the board. Select JTAG under the Interface menu, then hit apply. If you got a pop up menu saying there was an error, then something is not plugged in or you selected something wrong. Every now and then a problem will not be your fault but most of the time it is, just saying, happens to me too. Next, hit the "Read" button under device signature. If you got an error, I refer you to my previous

comments about plugging things in and letting the drivers install. If your screen looks like mein, good job! Click on the “Memories” tab on the left hand side to bring up something that looks like this.



Ensure that the address located under the name “Flash (128 KB+8 KB BOOT) is actually where you saved your code. This address is the location of the file you are about to upload. For the most part, this is almost always correct and you shouldn’t have to touch it. No touch. Click on the program button and a loading bar should appear at the bottom of the window. It will tell you when it is done and if your code is correct, the LEDs will turn on like in the previous setup picture! Woo, you turned on the LEDs, now how to blink them. Well writing the command PORTE.OUT = 0b00000000; turned them on so if I write PORTE.OUT = 0b11111111; after I turned them on it will turn them off. This is correct, but you turned them on, then within nanoseconds you turned them off. Your eyes may have not even seen them blink. Well now what. We need a way to delay the MCU when we turn them on and off as well as loop these commands to blink the LED indefinitely. To have the MCU delay some time, use the command “delay_ms();” You put how many milliseconds within the parentheses you want the MCU to delay. Let’s think how we want this flow to work.



Using this flow diagram, consult with your teammates to write code to make the LEDs blink on and off every 0.5 seconds. A screenshot of the code I wrote is on the next page but it is in your best interest to try itself yourself first. I'm not always going to be there to hold your hand.

```
/* Flight Code */
PORTE.DIR = 0b11111111; //This line sets all the pins on PORTE as an output


while (1)
{
    PORTE.OUT = 0b00000000; //This sets all of the pins voltage levels to 0V, which is logic 0 in programming usually
    delay_ms(500); //0.5 second delay
    PORTE.OUT = 0b11111111; //This turns the LEDs off
    delay_ms(500); //0.5 second delay
}
```

Your code may look a little different from mine but that's okay as long as it has the same output. Let's examine what we say. At the instant the LEDs turned on, they were on for 0.5 seconds, and then they turned off for 0.5 seconds. The entire cycle of LEDs blinking from on to off lasted 1 second. To find the frequency of blinking, we take 1 cycle and divide it by the time it took for the cycle to complete. In our case, we have 1 cycle per second. This cycles per second unit is called Hertz, Hz. So our LEDs blinked at 1 Hz. Also notice that for 50% of the period, the LEDs were on. This is called duty cycle. If we have a period of 1 Hz and we want our LEDs to have a duty cycle of 25%, what delays should we program? Why don't you go ahead and try this.

That's it for this section. Please ask questions and go back over the material if you are confused.

Communicate via UART

Communicating between the MCU and other devices is what makes our embedded MCU projects so cool. The first form of communication we are going to learn is called UART, Universal Asynchronous Receiver Transmitter. This is a simple communication involving a transmission line, TX, and a Receive line, RX. This module is going to involve you sending data back and forth to the computer to learn how UART works. UART is how you are going to communicate with your Xbees so this is very applicable to your project.

Before getting to the code, there are some more concepts about UART that you need to know. When we talk about digital communications, we need to know the speed at which we can transfer data. For all intents and purposes during this project, we will refer to the data rate as Baud rate which is simply bits per second. Unlike later forms of digital communication, UART is a form of communication only between 2 devices; in this case the MCU and the computer. Since the A in UART mean asynchronous, the two devices aren't necessarily synced up together meaning there isn't a third line that signals to the other device when data is about to come and at what rate. Because data can show up at any time, asynchronous, both devices must be using the same Baud rate or the data will be wrong. What does a stream of data look like? The figure on the next page shows a UART frame.

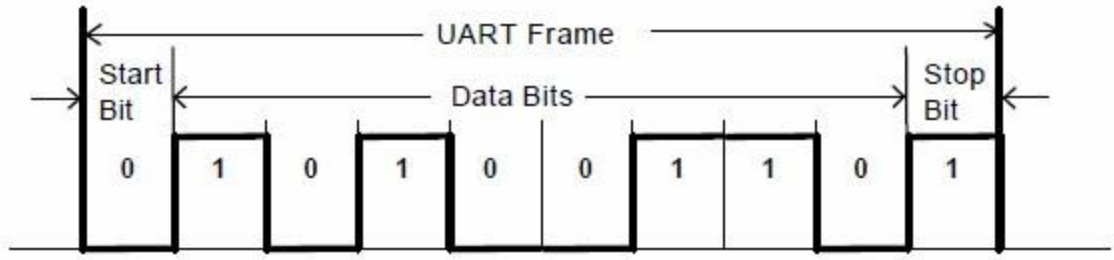


Figure 11a. UART Frame

A UART frame is a set of 8 bits of data beginning with a start bit and followed by a stop bit. The UART peripheral on the Xmega takes care of start and stop bits for you, you just need to make sure you configure them correctly once. The code will help explain this in a second. So, we can see we are sending 1 byte, or 8 bits, of data at a time. How do I take my altitude of my payload and send it over UART? How would I send letters and stuff? The answer to both is the same. UART communication is very good at sending letters and numbers using the same scheme. This scheme is called ASCII. ASCII is a standard form of representing every key on the keyboard as a single 8 bit number. Notice in the table below that the numbers 0-9 can be represented as the decimal value 48-57. The alphabet starts at decimal 65.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Okay that's cool, but a bit of confusing. Can you give me an example?

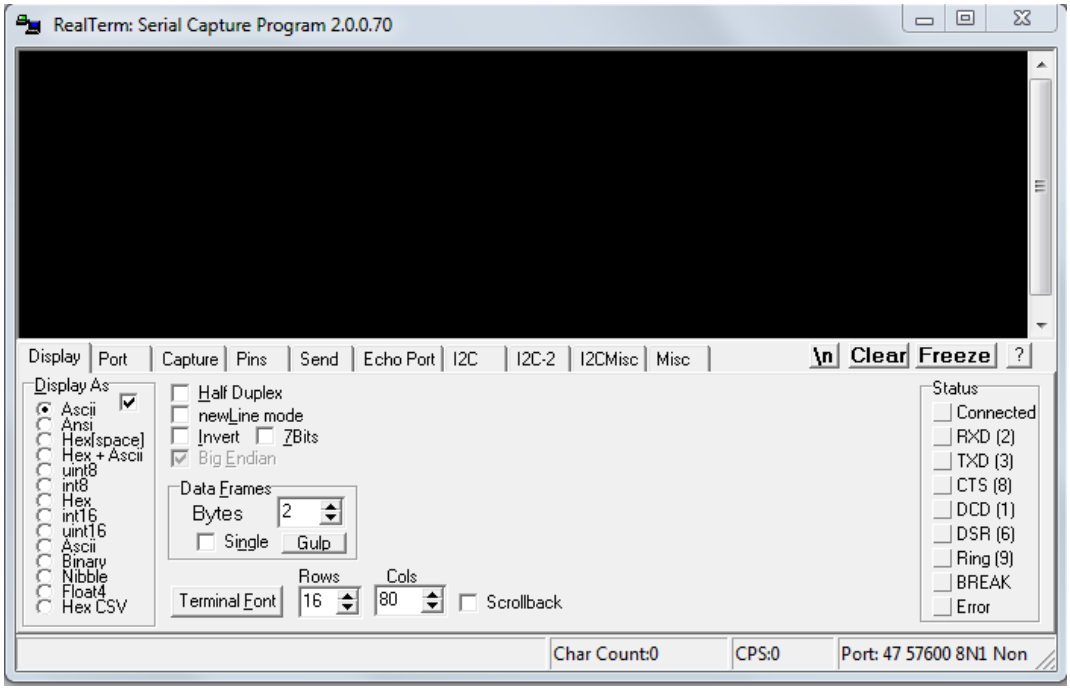
No.

Okay, fine. Let's send the data "Altitude: 1002" using ASCII. Note, spaces and other characters like the : require their corresponding data value.

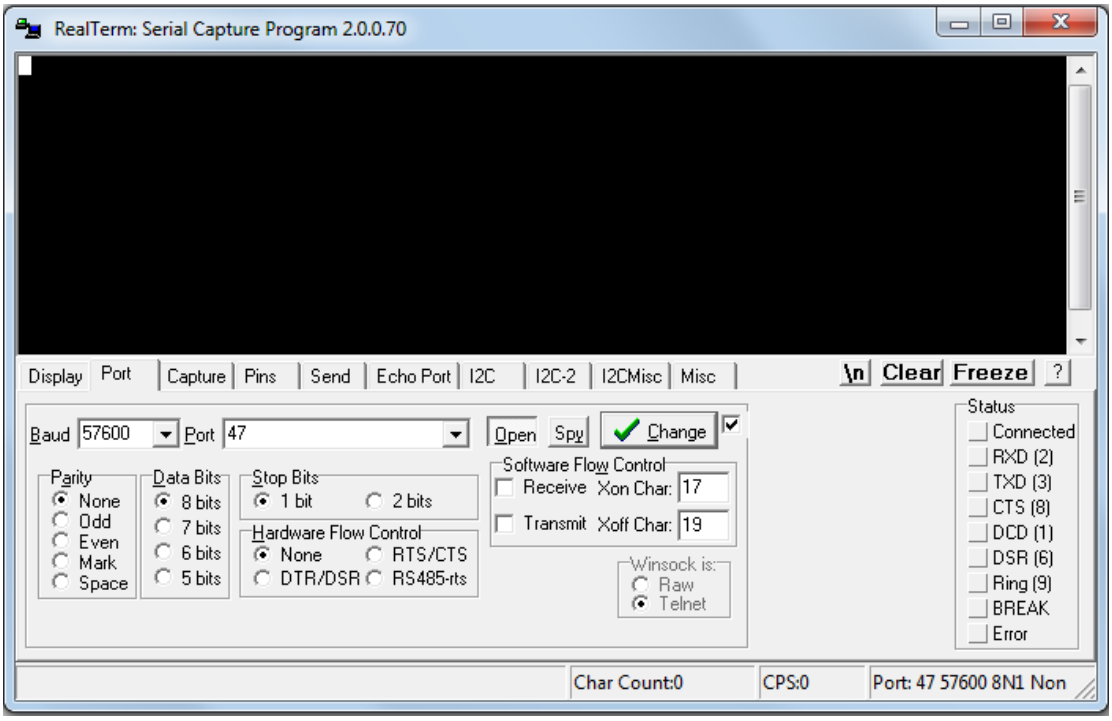
A	l	t	i	t	u	d	e	:	(space)	1	0	0	2
65	108	116	105	116	117	100	101	58	32	49	48	48	50

We call each byte of data a character, or char for short. When strung together to make a statement like the one above, we call it a string. So, do we have to remember what the ASCII value is for everything? Thankfully not, the MCU will do this for you. You just have to type your message.

Go to the One Month Template code and open the "conf_usart_serial.h" file under the config folder in your solution explorer. Scroll past wall of green liability text until you see the purple #defines. Read the comments. These #defines are the settings of the UART module. The device you want to send the data to must have these exact settings as well. Next go read "uart_comms.c" under the Drivers folder to familiarize yourself with how the function "UART_Comms_Init()" works. Now go back to the main file. Find the "UART_Comms_Init()" function in main. Alright, now let's setup the computer to accept the data. Open a file on the computer called "Realterm." This is a free download if you are on your computer.

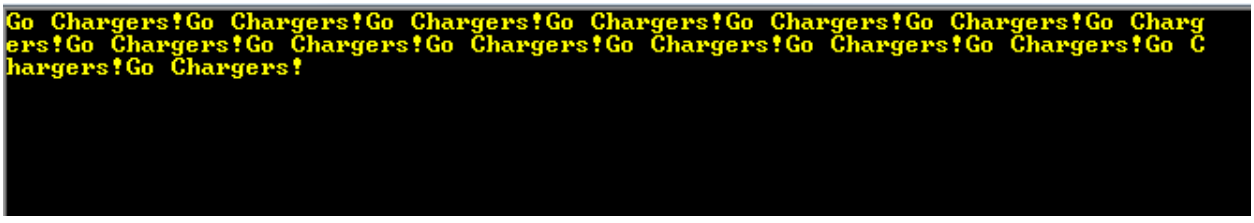


Only the display tab, as shown in the picture, ensure ASCII is selected in left column. Also check a box called newline mode. Click on the Port tab to change the settings of the computer's UART.



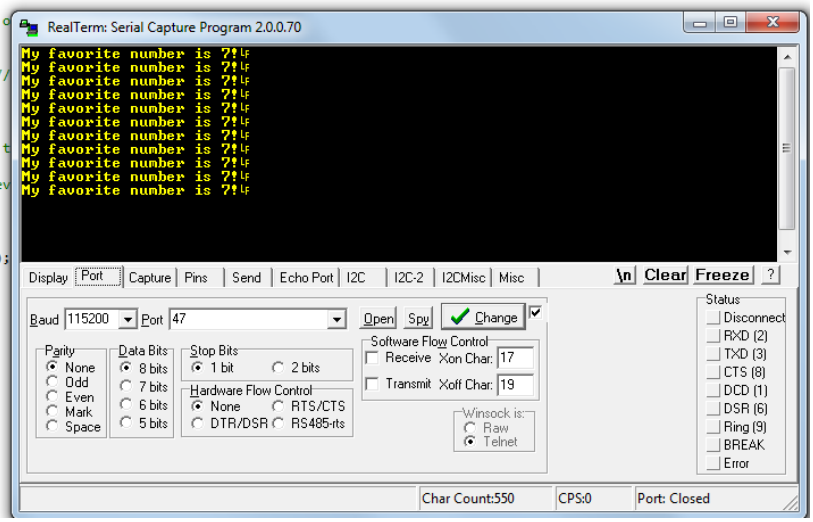
First click the Open button before changing the settings. Change Baud to 115000, set parity as none, set Data Bits to 8, click on 1 stop bit, and have no Hardware Flow Control. Most of these you already saw in the programming template. Now click the change button to save these settings. Hit the Open button again to get the computer ready to receive data. If it says bad port ID or something like that, try powering off your Xmega and powering it back on or reopening Realterm. Back to the code!

To write a string of data, we use the function called "printf();"To write a string, simply write what you want to say between double quotes "". An example would be this, printf("Go Chargers!"); You can save the code you typed for blinking the LEDs into a word doc or text file, but go ahead and remove the lines of code to blink the LEDs but keep in the while loop. Within the while loop, use printf() to send a message to the computer, add a 500ms delay afterwords, and compile and upload you program. Your Realterm screen may look like this.



Well that's cool, but I don't like how the messages are right next to each other. To put each line of data on its own line, put a "\n" right at the end of every message like this, printf("Go Chargers!\n); This is called newline and will put every new message on a newline as long as you checked this box in Realterm. To show you how to print variables to the screen, before the while loop create an 8 bit unsigned (only positive numbers) variable called num and assign your favorite number between 0 and 255 to it. Change your message within the printf to this printf("My favorite number is %i!\n", num); notice we put "%i" in the printf to indicate we want the value of the variable "num" to be in the statement.

```
/* Begin Interrupt Service Routine Section */
//This is where you make ISRs when using interrupts, o

/* End Interrupt Service Routine Section */
////////////////////////////////////////////////////////

int main (void)
{
    /* Initialize the system clock, 32MHz, this also t
    sysclk_init();
    sysclk_enable_peripheral_clock(&USARTC0); //For ev

    /* Example, Timer Counter on PORTE */
    sysclk_enable_peripheral_clock(&TCE0);
    sysclk_enable_module(SYSCLK_PORT_E, SYSCLK_HIRES);

    /* Initializations */;
    UART_Comms_Init();

    /* Flight Code */
    uint8_t num = 7;
    while (1)
    {
        printf("My favorite number is %i!\n", num);
        delay_ms(500);
    }

}
```



To learn all the syntax on how to print floats, larger ints and stuff like that, you will need to google.

Timer Counters

Timer counters are very useful peripherals that you can use to keep time and to blink your LEDs at a specified frequency and duty cycle. This module involves you reading much of the section Timer Counters in the Xmega A Family datasheet. Yay more datasheets… But before you begin, some helpful explanations to make things more clear are in order. Remember back early in this document that peripherals run off of their own clock. Timer Counters are no different. The datasheet refers to this idea that the peripheral clock is a "prescaled" clock. This just means we are going to take the main CPU clock and divide by some power of 2 to make it run slower. Locate the Timer Counters section within the Xmega A Family Datasheet, please be reading the Xmega A Family Datasheet. If you think that the Timer Counter part of the Datasheet starts at chapter 14 then you are correct. Please read sections 14.2 and 14.3 to start off. Don't try to understand everything but just try to get the just of what's going on.

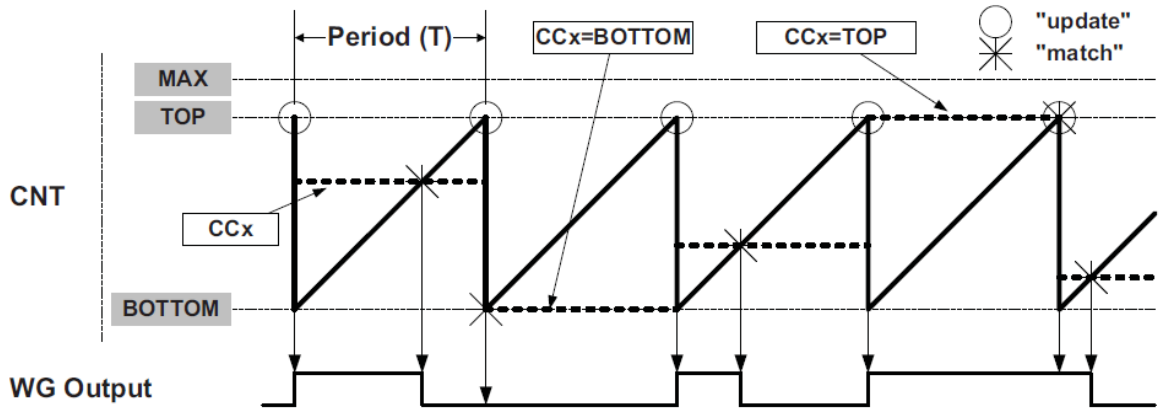
Don't proceed till you read. Just don't. You are only hurting yourself. Why are you hitting yourself? Why are you unfit to live in society like the rest of us? Okay, key concepts you should've come away with.

- The Timer Counter is normally 16 bit
    - This means the Timer Counter CNT, the count register, can hold the values 0-65535.
- The Timer Counter's clock must be a division of the main clock
- There is a period register that when the count registers equals the period register, the count register resets to 0
- There are these things called CCx that can output data from the Timer Counter, more on this later

The Timer Counter can run in different modes. For this project, and almost any project you will ever do, the Single Slope PWM mode is what you want. To learn about this mode, read section 14.8.3.

If you are proceeding without reading the section you obviously don't care about your future. Why do you want to be homeless? So this section showed us a great figure and some equations we can use. I want to discuss the figure for a second.

Figure 14-15.Single-slope pulse width modulation.



On the left axis is the CNT register which can hold a value between 0 and 65535. We will write a value to the period register that is equal to or less than the maximum of the CNT register. This is how many counts the CNT register will count up to until it resets to 0. In the figure, the value in the period register is shown as "TOP." So the period registers sets the period of the Timer Counter, imagine that. This also sets the frequency of PWM. So I guess I should explain PWM now. PWM stands for Pulse Width Modulation. This is fancy talk for changing the length of a "pulse" (On time) with respect to a frequency. Hmm… This sounds like the period and duty cycle discussion we had during the Blink an LED module. That's because you have already performed PWM when you blinked the LED at 1 Hz at 50% duty cycle! Okay so the period register sets the frequency, what sets the duty cycle? The CCx register holds a value that is meant to be equal to or less than the period register. Simple reasoning tells us if the value in

the CCx register is half of the value in the period register that we would have a duty cycle of 50%. Use the equation below to calculate the frequency of the Timer Counter where:

$$f_{PWM\_SS} = \frac{fclk_{PER}}{N(PER + 1)}$$

fclkper is the CPU clock of 32MHz

PER is the period register value

N is the prescalar used

But wait! What prescalar should I use? What prescalar options are there? To learn this knowledge, we must consult the datasheet. Let's go to section 14.12 "Register Description" and read the first register section 14.12.1 "CTRLA." You may first notice that bits 4-7 are grayed out and are labeled "Reserved" underneath the figure. This means they do nothing and we should just write 0s to these bits. The bits we care about are the lower 4. These set the prescalar according to the table at the bottom of the section. The larger the prescalar, the less counts it takes to equal the same amount of time passing as opposed to a smaller prescalar. To illustrate this, we will perform math. Well, you will. With a CPU clock of 32MHz, a desired frequency of 10 Hz, find the PER value needed when N, the prescalar option, is set to 1024 and 256. Which prescalar yielded the higher value in the PER register? If you answer 256 you are correct. If you also answered with, "I'm a little confused. I'm going to try to re-read that section of the datasheet. Then, if I am still confused, I will ask my wonderful teammates. If the entire team is confused, then I will ask my mentor. If my mentor is also a little bit confused, then I will ask another mentor until I understand what's going" that is also correct. I like the way you think. While both prescalars of 1024 and 256 will work just fine, 256 will have increased resolution though during this project that's not a big deal. If you run a calculation to find the period and your answer is greater than 65535, you will need to use a larger prescalar. Also, if you calculate a period with a decimal point you will need to either round it up or down. The register only accepts 16 bit unsigned int values, of which a decimal is not allowed. So don't try. Bad things will happen. Probably not but you never know. Don't chance it. Save #YOLO for stunts involving motor bikes and not spacecraft. While we have the CTRLA register fresh in our minds, let's go ahead and start writing some code. The flow for setting up the Timer Counter to blink LEDs at a specific frequency and duty cycle is only 5 simple steps. Go ahead and complete steps 1-2.

1. If using PWM, make sure the pins for PWM are set as outputs on the register level
2. Setup control register A, CTRLA
3. Setup control register B, CTRLB
4. Set the timer counter period. Note, max value is 16 bit or 65535.
5. Set the CCx for each pin being used as PWM. CCx/PER = Duty Cycle %

Your code might look this.

```
/* Flight Code */
PORTE.DIR = 0b11111111; //All LEDs on PORTE as outputs though only 4 are connected to Timer Counter
TCE0.CTRLA = 0b00000110; //0110 is prescalar 256

while (1)
{

}
```

Note, you don't need to set all the LEDs as outputs but I did just because, you know, LEDs. But wait, what does "TCE0" stand for? Well TC is short for Timer Counter, E because it is on Port E, and 0 because there are 2 TCs per port usually, 0 and 1. Timer Counter 0 is the 16 bit one while Timer Counter 1 is only 8 bit. According to the nice list I gave you on how to set up the Timer Counter, Step 3 involves setting up CTRLB. Does this mean back to the datasheet? Why yes it does. Read the CTRLB register description.

Alrighty, we have 2 things we have to set in this register, the pins we want to be controlled by the Timer Counter and the waveform. I already told you we will be using the single slope mode so this is easy to program. But which pins should you enable to be controlled by the Timer Counter. If you answered "All the pins!" then you would be correct. If not, try again. Your CTRLB register should read like this, PORTE.CTRLB = 0b11110011; Before you go off and complete Steps 4 and 5, eager beaver, we should at least learn which LEDs we just enabled and how to check if another port also has a Timer Counter Module. Open the "Xmega128A1 for Pin Descriptions" Datasheet. We only use this datasheet for pin descriptions. Please go to section 30.2 "Alternate Pin Functions." This lists all the ports and what each pin is capable of doing. Note, while every pin can be an IO pin, only certain pins can be a UART or a Timer Counter and they usually can't be both at the same time. This isn't Hannah Montana, you don't get the best of both worlds. Shout out to those that got that Disney reference. Anyways, she's crazy now, we should just forget her. Starting now. Moving on.

Scroll down to PORTE. Look under the column labeled "TCE0." The pins that have OC0A-D next to them are the ones we turned on. In the CTRLB register setup you can only enable one of these pins to be on if you would like on your actual project. Let's write the rest of the code. Calculate the PER you need blink the LEDs at 10Hz. Add the line of code to your main file. Now we need to complete Step 5 by adding separate lines of code putting the CCx value in the CCA-D registers. So if we want a duty cycle of 10%, we would take our TCE0.PER/10 and store it into the TCE0.CCA register then do this CCB, CCC, and CCD. Observant people may notice that a 10% duty cycle means that the LEDs would be on 10% of the time, however, remember that our LEDs are kind of wired in backwards. So a 10% on time is actually a 90% on time for the LEDs. How do we remedy this? I took the value in TCE0.PER and subtracted 10% of its value to equal 90%. The code I wrote is below.

```
/* Flight Code */
PORTE.DIR = 0b11111111; //All LEDs on PORTE as outputs though only 4 are connected to Timer Counter
TCE0.CTRLA = 0b00000110; //0110 is prescalar 256
TCE0.CTRLB = 0b11110011; //All LED outputs and Single Slope
TCE0.PER = 12499; //This will make the LEDs blink at 10Hz
TCE0.CCA = TCE0.PER-(TCE0.PER/10); //90% Duty cycle equates to 10% on time for LEDs
TCE0.CCB = TCE0.PER-(TCE0.PER/10);
TCE0.CCC = TCE0.PER-(TCE0.PER/10);
TCE0.CCD = TCE0.PER-(TCE0.PER/10);

while (1)
{

}
```

See how nice and commented my code is? Yours should be too or you are doing it wrong. We are going to call you Eric if you don't comment. It's an inside joke that you won't get. Don't want to feel left out? Comment you code. Upload your code and marvel at the blinking! If it doesn't blink, double check your work. Even a single 1 or 0 that is wrong may cause it to not work. Sheesh, computers are picky. You may play around with the period and duty cycle. If you make the frequency of the blinking very fast, you may not that the LEDs appear to not be blinking anymore? What gives? I want my money back. It's not broken, it is just switching faster than the measly human eye can perceive. In fact you may notice that the LED appears dimmer than the other ones. You have discovered how light dimmers work. Ta da! The beautiful thing about the Timer Counter is once you set it up, it runs by itself. This frees you CPU to handle other tasks like calculating altitude or discerning the meaning to life.

Alright, you wrote some pretty baller code that you might want to save for later. If only there was a way to organize your different blocks of code into similar sections. Oh wait, there is! Copy your code, not the while loop just the Timer Counter stuff, and paste it into a file called "timer_counter_init.c" below the steps I already wrote for you. It would be nice to put this chunk of working code into a function so we can just call it from main and it will configure the Timer Counter for us. Well, let's try it. Make a function called TCE0_init() with a return type of void and an argument of uint16_t period and move your code inside. Don't forget the opening and closing {}.

```
void TCE0_init(uint16_t period)
{
    PORTE.DIR = 0b11111111; //All LEDs on PORTE as outputs though only 4 are connected to Timer Counter
    TCE0.CTRLA = 0b00000110; //0110 is prescalar 256
    TCE0.CTRLB = 0b11110011; //All LED outputs and Single Slope
    TCE0.PER = 12499; //This will make the LEDs blink at 10Hz
    TCE0.CCA = TCE0.PER-(TCE0.PER/10); //90% Duty cycle equates to 10% on time for LEDs
    TCE0.CCB = TCE0.PER-(TCE0.PER/10);
    TCE0.CCC = TCE0.PER-(TCE0.PER/10);
    TCE0.CCD = TCE0.PER-(TCE0.PER/10);
}
```

Cool, TCE0_init turned red because it's a function. Before we change up the function a bit, copy the first line void TCE0_init(uint16_t period) and paste it into the file "timer_counter_init.h"

and add a ; to the end. This is a function protoype. This simply lets the compiler know you want to use a function called TCE0_init() later on. So, why have an argument called period? It would be nice to simply call the function and include the period value so we can change the frequency of PWM on the fly. Currently, TCE0.PER is still being assigned 12499, so change this out with our new variable argument period. You can simply write TCE0.PER = period; Alright let's go back to our main file. Call the function in the initializations section above the flight code section. You must also include the .h file into the main. There are so many files in the project, you have to explicitly tell the compiler which ones to include. You don't include the .c file but it won't work without the .h file.

```
/* Initializations */;
UART_Comms_Init();
TCE0_init(12499); //10Hz

/* Flight Code */
```

```
/* Begin #include section */
//This is where you will need to include the header files that you have written code in to use the code
#include <asf.h> //This will mainly include behind the scene code and all header files within the config folder
#include "Drivers/timer_counter_init.h"
```

If the LEDs are still blinking at the appropriate rate then you done good. If not, try again and ask questions. We can also use the Timer Counter for count register. You can calculate the length in time of each increment in the count register by taking the inverse of the CPU clock/prescalar. This will be a very short amount of time usually. You can simple read the CNT register by saving the line TCE0.CNT to a 16 bit unsigned variable. I am going to let you decide if you want to do this in your full project and implement it.

Read a Voltage with the ADC

The dreaded ADC. You've been hearing about it, dreaming about it… Wait what? You've been dreaming about the ADC? Weirdo, go get some fresh air… The Analog to Digital Converter is to read a voltage and convert it into a number that you must interpret. Using math, we can convert the reading from the ADC into the voltage that we read. Start by reading sections 25.2 through 25.3. There is going to be a lot of things that won't make sense immediately, like life, but it will become clearer once we get to the registers.

```
if (reading == done)
{
    break; //This breaks from the statement meaning you can proceed
}
else
{
    //You shouldn't know what this says because you were supposed to read
}
```

Alright, things you should've taken away:

- ADCs are complicated
- The ADC has a resolution of 12 bits
- There is single ended and differential modes
- There is signed and unsigned mode
- There are different channels and something about a mux?
- There are internal inputs
- There are gain stages that you can use on the differential mode
- Is it about time for a Starbucks break?

Let's clear up some confusion. Just keep in mind the ADCs resolution of 12 bits we will use this later. A single ended measurement means we are taking the measurement with respect to ground. A differential is when we take the voltage difference between two pins. For example, if Pin 0 is at 2V and Pin 1 is at 1V, the differential measurement between Pin 0 and Pin 1 is 1V. This also means the differential measurement between Pin 1 and Pin 0, backwards now, is -1V. This isn't a negative voltage with respect to ground, just with respect to each other. When using differential mode, the signal is generally less noisy but the ADC has to be in signed mode. Signed mode is different than unsigned mode because in unsigned mode the value from the ADC can be between 0 and 2^(12)-1, or 4095. There's the resolution showing up again. In general this means at 0V the ADC should read 0 and at the maximum voltage input the ADC should read 4095. In signed mode, the maximum is actually 2047, the minimum is -2048, and 0V is still 0. Signed mode allows us to read negative voltages, but at the cost of reduced resolution over the full scale range. This makes sense when you think about it like this. If the positive voltage rang is between 0V and 2V, would you rather have 4095 divisions or 2047 divisions? You would want 4095 resolution over the positive range to give you double the resolution. Now this assumes you will always have a positive voltage reading with respect to ground. But in your project this is a safe assumption. Read section 25.5 on voltage reference. In order to determine what voltage the input pin is at, the ADC must have a known voltage to reference to. From experience, the best method in this project is to use the Vcc/1.6 as the voltage reference. In a 3.3V system, this yields a voltage reference of 2.0625V. You can also read section 25.6 to get a better understanding of the conversion process. You can read/skim over the sections leading up to 25.16 "Register Description." Follow the process below to setup the ADC. Note, we start lines of code with the ADC as ADCA because we will be using the ADC on Port A.

- CTRLA Register
  - Write a line of code to enable the ADC only and nothing else
- CTRLAB Register
  - Write a line of code to ensure the ADC is in unsigned mode, not in freerun mode, and that the resolution is 12 bit right adjusted (right adjusted is normal)

- REFCTRL Register
  - Write a line of code to have the voltage reference be Vcc/1.6, leave the other stuff on this register disabled
- PRESCALER Register
  - The from experience, the higher prescalar divider, the more accurate and stable the ADC reading is, but making it take too long to take a reading lowers the speed at which you can sample the ADC. This is more of a concern in other fast, data collecting projects but for this project set it at DIV128
- Read the manufacturer calibration data into the CAL register
  - Use this line of code:
    - ADCA.CAL = adc_get_calibration_data(ADC_CAL_ADCA);

After doing all of this, your code should look something like this.

```
/* Flight Code */
ADCA.CTRLA = 0b00000001; //Enables the ADC
ADCA.CTRLB = 0b00000000; //Unsigned 12 bit mode
ADCA.REFCTRL = 0b00010000; //Voltage Reference of Vcc/1.6V
ADCA.PRESCALER = 0b00000101; //This is automatic prescalar of 128 on the clock
ADCA.CAL = adc_get_calibration_data(ADC_CAL_ADCA); //Retrieve stored calibration data about the ADC
```

Now move to section 25.17. This section is for settings specific to ADC channel. Remember, the ADC 4 channels although for this project we will only need to use channel 0. To change settings in channel 0, start your lines of code with ADCA.CH0.

- CTRL Register
  - Write a line of code to only change the input mode. Remember our conversion mode is 0, unsigned mode
  - Notice that bit 7 is called the START bit. When we are ready, so not now, we will flip this bit to 1 to begin the conversion. But not now.
- MUXCTRL Register
  - This register controls what's called a mux, or multiplexer. Think of a mux as a person that directs traffic. PORTA has 8 pins that can be connected to the ADC peripheral, but only one pin can be read one at a time. The MUXCTRL determines which pin is connected to the ADC at any given time. Meaning, if you need to take two ADC measurements, you will need to change this register every time you need to take a measurement on a different pin.
  - Alright, we have already set up unsigned single ended mode. Look at table 25-12. This table matches our settings we have already put into the previous registers. Select which pin you want to read a voltage from first. The ADC pins on Port A are labeled on the top of your Xplained board at one of the 4 pin headers for your convience. You're welcome.

Alright, having done all that your code may now look like this.

```
/* Flight Code */
ADCA.CTRLA = 0b00000001; //Enables the ADC
ADCA.CTRLB = 0b00000000; //Unsigned 12 bit mode
ADCA.REFCTRL = 0b00010000; //Voltage Reference of Vcc/1.6V
ADCA.PRESCALER = 0b00000101; //This is automatic prescalar of 128 on the clock
ADCA.CAL = adc_get_calibration_data(ADC_CAL_ADCA); //Retrieve stored calibration data about the ADC

ADCA.CH0.CTRL=0b00000001; //single ended input
ADCA.CH0.MUXCTRL=0b00000000; //Reading ADCA pin 0
```

You may notice that we have some registers that are essentially 0. Could you skip writing them or make simply write 0 instead of 0b00000000 every time? Yes, but we want to write them all out since you are still learning what all registers need to be used in order to take an ADC measurement. So now the ADC is all set up and ready to do some conversions. Woo. We must now write code to take the measurement. The ADC measurement takes some time though to complete. How much time you say? Well there was a timing diagram earlier that said it took about 8 clock cycles. Well how much time is 8 clock cycles? We can go down this rabbit hole, or we can take advantage of a feature of the ADC in the interrupt flags. What this means is when the ADC is done with a conversion, it raises a little flag, it really flips a bit but I like the flag analogy, to let you know it is down. Where is this little flag you say? Why it is in section 25.17.4. This section tells us when we start a conversion, this bit goes to 0, and then when the conversion is done it will flip to 1. So alls we need to do is wait in a loop until this bit flips to 1. This line should do nicely for us: while(ADCA.CH0.INTFLAGS==0); Let's break it down. We can read this line of code in English like this. While the interrupt flag in channel 0 is equal to 0, check to see if the interrupt flag is 1. So the CPU will sit at this line of code until the conversion is complete. The conversion should only take micro or milliseconds so not long at all.

So that is how to check to see if the conversion is done, but how do you start a conversion. Well, the must set bit 7 in the CTRL register. However, if we simply write ADCA.CH0.CTRL 0b10000000; this will overwrite our 1 at bit 0 which is how we set the ADC to be single ended. We can simply write 0b10000001 instead and this will do the job and fix the problem. A nicer way of writing it would be to do a bitwise or operation and write |=0b10000000. What is this sorcery and how does it work. Using the |= command changes only the bits set to 1 in the new command without changing the bits in the old command. Either way works it's up to you.

Almost done here. We need a 16 bit unsigned variable to store the result to. The result from the ADC is located in the RES register which belongs to the channel 0. Create a variable and store the ADC result to it. Then, write a printf statement to print out this uint16_t variable to Realterm. Delay 0.25 seconds and start the whole conversion process over again (this means while loop). Your code may look like this.

```
/* Flight Code */
ADCA.CTRLA = 0b00000001; //Enables the ADC
ADCA.CTRLB = 0b00000000; //Unsigned 12 bit mode
ADCA.REFCTRL = 0b00010000; //Voltage Reference of Vcc/1.6V
ADCA.PRESCALER = 0b00000101; //This is automatic prescalar of 128 on the clock
ADCA.CAL = adc_get_calibration_data(ADC_CAL_ADCA); //Retrieve stored calibration data about the ADC

ADCA.CH0.CTRL=0b00000001; //single ended input
ADCA.CH0.MUXCTRL=0b00000000; //Reading ADCA pin 0


while (1)
{
    ADCA.CH0.CTRL|= 0b10000000; //Start the conversion
    while(ADCA.CH0.INTFLAGS==0); //Wait until conversion is done
    uint16_t adcReading = ADCA.CH0.RES; //Save the result into variable called adcReading
    printf("%i\n",adcReading);
    delay_ms(250);
}
```

Before you upload the code, add this line towards the top of main with the other similar lines of code. This enables the clock to the ADC on Port A. Without this, it will not run.

```
sysclk_enable_peripheral_clock(&ADCA);
```

Once you upload the code and look at the output on Realterm, you may notice that the numbers are varying wildly. That's because there is not voltage being applied to the pin you are reading so it is floating. Using a power supply, apply a voltage between 0 and ~2V to the pin. Now look at the number. What gives! It's not giving me the voltage and it is still fluctuating a couple numbers here and there. I quit. Doing hard things is stupid. Well don't get your jimmies rustled, we haven't converted the ADC reading into a voltage yet. The simplest way to do this is to apply a voltage of 0.5V to the ADC and record the average reading. Then increase the voltage to 1.5V and record the corresponding ADC reading. Now that you have 2 points of data, using algebra, find the equation of the line that passes through these points in the form y=mx+b where y is the voltage and x is the ADC reading. Now write a new line of code right after you store your ADC result and convert it to an actual voltage. Note, this will have to use floating point math. So create a variable of type float to represent the voltage. Then typecast this float back to a uint16_t in the printf function while also multiplying by 1000 to read millivolts to read the actual voltage! That probably sounded really confusing. Look at the code and it may help.

```
while (1)
{
    ADCA.CH0.CTRL|= 0b10000000; //Start the conversion
    while(ADCA.CH0.INTFLAGS==0); //Wait until conversion is done
    uint16_t adcReading = ADCA.CH0.RES; //Save the result into variable called adcReading
    float voltage = 0.0005*adcReading - 0.0941; //This converts adcReading into an actual voltage based off of my slope of a line
    printf("%i\n",(uint16_t)(voltage*1000)); //voltage*1000 converts it into millivolts
    //We if we hadn't multiplied by 1000, the typecast would've truncated the voltage reading to just the one's place
    //So a voltage if 1.523 would've been displayed as 1. By multiple by 1000, it will output 1523
    delay_ms(250);
}
```

So now we have some good chunks of working code. What to do with them. Well we have one block that sets up the ADC, this could be an initialization function. We have another block that actually does the reading and returns the voltage. This could be a function to return the voltage on the pin. I'm going to leave this up to you. Don't let me down.

Alright that's how the ADC works. Feeling lost, re read this section. Ask a friend. Ask a mentor. Get a coffee. Like a boss.

Digital Communications

The Digital Communications section only applies to teams that have a digital pressure sensor for the One Month Project. If that's you, then you will need to use digital comms to talk with your sensor. I will only cover one type of digital communications in this section because more than likely the pressure sensor you guys picked uses SPI, or Serial Peripheral Interface. If a team selected one that only uses something called I2C, please let me know. I'm going to offload much of the instruction of this section by providing two links for you to read.

https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

http://www.atmel.com/Images/doc8057.pdf

You also need to read pretty much the entire datasheet section on SPI withing the Xmega A Family for Programming datasheet, section 20. Don't worry don't worry, I'm not assigning you a novel to read. The SPI section is rather short as far as datasheet stuff is concerned so it's all important.

If you are reading this sentence, good job you can read, but you also need to have read the previous reading assignments and understand how SPI works before moving on. If you are still very confused, read the other recommended readings from the Sparkfun reading, watch a YouTube video, or perform a google search. These are invaluable engineering tools that they just don't teach you in class.

Your program flow should be something like this for initialization (Maybe this could be a function (It should be a function)).

1. Enable the peripheral clock for the SPI peripheral similarly to how it's done for the UART module. Don't know which port it's on? Go to the other Xmega datasheet for pinouts as explained in an earlier module (section 30.2).
2. Initialize the 4 pins for SPI with appropriate directions and initial high or low values.
3. Setup the SPI CTRL register. The syntax is SPIx.CTRL = … where the x is the letter of the port the SPI peripheral is on. The clock speed of the SPI peripheral should be what you sensor's datasheet calls for as well the SPI mode selection. Also the MCU should be run in master mode

Once initialization is done, you must now send and read certain data from your sensor to calibrate it. You will need to read the sensor datasheet to learn what data to send and what data to read. This usually involves sending a command, then reading calibration data to be used during the pressure and temperature conversions. This flow should be used for sending data and then waiting on new data from the sensor.

1. Set SS (Slave Select) low
2. Write the data (8 bits at a time remember) into the SPIx.DATA register
3. Using the interrupt flag on the STATUS register, wait until it flips to know when your data sent. This is similar to how we waited for the ADC to finish doing the conversion.
4. Write 0xFF, which is the same as 255 or 0b1111111, into the DATA register to set it up for new data to come in. The point here being we don't exactly know when the sensor will write back the data we requested so we immediately set the DATA to something we know, 0xFF, then use Step 4 below to know when new data arrived.
5. Using the interrupt flag on the STATUS register, wait until it flips to know when new data has arrived. This is similar to how we waited for the ADC to finish doing the conversion.
6. Read the DATA register and store the data to a variable.

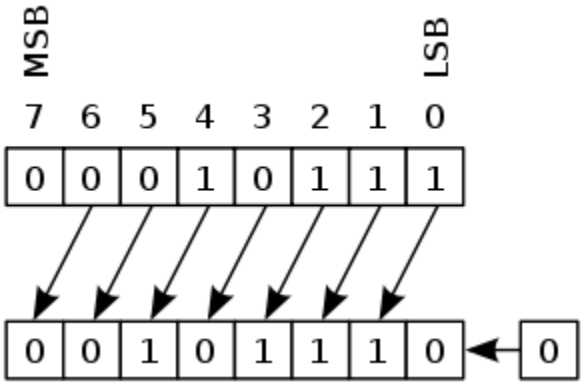
You can see we used the DATA register to both send and receive data. This is because it is a shift register. Data gets shifted out and new data gets shifted in its place. Steps 3 and 4 are best used together in a function where you return the data from the DATA register like this:

```
uint8_t spi_read(void)
{
    SPIC.DATA = 0xFF; //Set the data to something we know
    while(!(SPIC.STATUS>>7)); //Wait until new data comes in by monitoring the Interrupt flag
    return SPIC.DATA; //Returnt he data
}


void spi_write(uint8_t data)
{
    SPIC.DATA = data; //Take data from function argument and write the data
    while(!(SPIC.STATUS>>7)); //Wait until data has sent
}
```

Using these steps within functions, SPI is made easy! You must know read your sensors datasheet on how to calibrate and communicate with you sensor! I will mention one programming trick that will come in very handy during this part of coding, bit shifting. As you can see, the DATA register is only 8 bits, so what if I have to read a 16 bit value? Well, SPI will only shift in 8 bits at a time. More times than not, it will shift in the first 8 bits of the 16 bit value, this is called Most Significant Bit or Byte (MSB). The code below is an example of reading the MSB first, storing it to the upper 8 bits of the variable, then saving the next 8 bits to the lower 8.

```
uint16_t ms5611_read_adc_32(void)
{
    uint16_t data; //Temporary data variable
    spi_select(); //Select the SPI device
    spi_write(0x00); //Write data to the SPI device
    data = ((uint16_t)spi_read())<<8; //Typecast the 8 bit data to 16 bit, the move it 8 places to the left
    data+= ((uint16_t)spi_read()); //data += add the new data to the old saved data but in the lower 8 bits
    spi_deselect();
    delay_ms(1);
    return data;
}
```

This image below shows what bit shifting one place looks like. The code example above bit shifted 8 bits to the left instead of 1 like the example below. If the data was 0b11111111; for example, an 8 left bit shift would look like this: 0b1111111100000000; Then by adding this number and an 8 bit number, the entire 16 bit number can be stored!



Saving Data to EEPROM

So far in your programming, you've used two kinds of memory, flash and RAM. The MCU also includes a 3rd type of memory, called EEPROM. Here is how they compare:

Flash	RAM	EEPROM
• Can only be changed with an external device attached to special pins ○ Your JTAG-ICE programmer is one of these • Always has the software you wrote on it when it starts up	• Gets wiped every time your MCU is powered off • This is where all your variables are stored and your calculations are done	• Doesn't change when the MCU gets turned on and off • BUT, your program can read and/or modify the data stored here if it needs to

So, if you want your data to still be there after a battery pops out when your payload lands, or something else like that goes wrong, you can stick it in EEPROM. Then you can just power your payload back up and take the data off.

There is a set of functions available in the ASF (Atmel Software Foundations) library for reading and modifying EEPROM. Because doing it manually is complicated, and requires a lot of additional work, we suggest using this library. To do this, you'll need to open the ASF wizard (ASF -> ASF Wizard), then add the library called "NVM – Non Volatile Memory (driver)". Once you do this, you have two functions you can use, both of which are demonstrated here:

```
int main (void)
{
    board_init();
    sysclk_init();

    UART_computer_init();

    // This will tell you what's in location 0 in EEPROM, then increase it by 1
    // It'll do this every time it's run
    // So you should see a new number every time you run it, one higher than the last
    // Once it hits 255 it'll wrap back around to 0, since a single byte can't hold numbers any bigger than that
    uint8_t myfirstbyte = nvm_eeprom_read_byte(0); //myfirstbyte now has the data from address 0 in EEPROM

    printf("EEPROM byte 0 is %x\n", myfirstbyte);
    nvm_eeprom_write_byte(0, myfirstbyte + 1); //Replace the 0th byte in EEPROM with the old value + 1

    while (1); //Wait forever
}
```

There are two important things here

- `nvm_eeprom_read_byte(address_of_byte)` tells you what the byte of data at a given address is
- `nvm_eeprom_write_byte(address_of_byte, value_of_byte)` replaces whatever is at the given address with the 8 bits of data in the 2nd argument

For either one of these, you need an address. What can this be? The answer can depend on the exact MCU you're using. On the XMEGA128A1 MCUs you're given for One Month, the address can be any number between 0 and 2047. You can figure this out by looking at the "Xmega128A1 for Pin Descriptions" datasheet. In chapter 8.5, "Data Memory", it says the MCU has 2 KB of EEPROM. 2 KB means there are 2048 (or 2^{11}) bytes, which we call 2 kilobytes because it's sorta-kinda close to 2,000 bytes. Because we're counting from 0, instead of 1 (programming!), we can choose addresses from 0 (the beginning of EEPROM) to 2047 (the end of EEPROM).

If you want to store more than 8 bits of data (like, a 32 bit variable of some sort), you can split it up into 4 separate bytes by bit-shifting, like you did in the Digital Communications section.

Saving data is the kind of thing that can take a little bit of extra thought.

1. What data are you saving?
2. How many times do you need to save it for every flight?

3. How many flights do you want to save?
4. How is all this data organized?

"Wait," you're thinking, staring at your screen and questioning my sanity, "why would I need to save more than one flight's worth of data?" You don't, really. The answer could be 1. But, it might come in handy to have more than one around, if you have time to program it in.

Regardless of how many flights you save, you need to figure out a way to keep old data from getting overwritten. Imagine your flight happens, your payload does its thing and looks pretty, it lands, you find it, and you stick the RBF pin back in to turn it off. Then, when you plug it into a computer to get the data back off, it turns back on, reads in the current pressure and temperature, and overwrites your ground data from the flight with the data from next to the computer. This would be bad™.

So, how do you do plan this out? Start with a list of the variables you're saving and their size/type. For, say, a rocket test stand, this might look like this:

Variable	Variable Type	# of Bytes Required
Time (in ms)	uint32_t	4
Force measured by load cell (N)	uint16_t	2
Pressure in motor (psia)	int16_t	2
	Total	**8**

Now, suppose this rocket test stand is sampling its sensors at 100 Hz, and will be collecting data for 30 seconds. Then the total amount of space required can be calculated by multiplying all of these together:

$$(30\ s)\left(100\ \frac{samples}{s}\right)\left(8\ \frac{bytes}{sample}\right) = 24000\ bytes$$

This would take a little bit more (ok, almost 12 times more) space than your 12 KB EEPROM provides. Luckily, your One Month flight only has to record 4 sets of data, so you probably won't need much at all (exactly how many depends on what variable types you use).

So, now that you know how much space your data requires, how can you avoid overwriting old data? There are a lot of approaches. A few are:

1. Include a switch that you flip after landing. If the switch is flipped, the payload does not save data
2. Have one place in memory for the data you are collecting and saving for the current flight, and another for the data from the last flight. When your payload starts up, it reads the data in the "current flight" location and moves it into the "old flight" location, then starts collecting measurements and storing new data in the "current flight" location

3. Save the oldest data early in memory, then the next oldest after that, and so on and so forth. When you run out of memory, loop back around to the beginning. Include a "flight counter" variable in the data saved so you know what flight is the newest

Any of these will work just fine. There are also many, many other ways you could do this. Ask me (Daniel) if you want to hear 5000 more.[1]

How to Write a Program to Perform a Mission

You have now completed all the programming tools you need to complete this project! It may not seem like it, but you are now smart. You task is to know combine all these tools you have learned together to write your mission code. Your mission is broken down into flight states. You must determine which loop or statement to use to stay in a flight state until the condition is met. This could be a switch statement where the variable you are evaluating is the flight state. This could be multiple while loops in series or even "nested" where the while statement is your condition on going to the next flight state. Remember that your pressure sensor is not going to read the exact altitude. This doesn't matter because you don't care how high you are above sea level, only above ground. So save your initial altitude on startup and reference your current altitude against this initial altitude. Don't forget when making your conditions on going from flight state to flight state that you may have cases where you may have a negative number. Like if initial altitude reads 700 ft and for a brief moment before lift off the altitude goes to 698 ft, be able to handle these situations. You must transmit data every 0.5 seconds, think about how you want to do this either with a Timer Counter or a delay statement. Test, test, test this in a pressure chamber or outside. Use the printf() function as a debug tool to show that your code progressed to certain sections and is not getting hung up on something. Good luck Space Cadets, we are counting on you.

[1] Ask me about my CanSat filesystem!