Instrumentation — New Member Orientation

Bruno Almeida, Russel Brown, Shimi Smith & Dylan Vogel

November 7, 2017

Contents

1 Version History

2017-11-07	Dylan Vogel	Created document and added existing write-ups.

2 Introduction

This document is meant to provide a preliminary overview of topics which we feel are useful to a new member joining the Instrumentation subsystem. The topics covered include SPI (Serial Peripheral Interface), reading datasheets, C programming, and hardware in the MP099 lab.

3 C Programming

Written by Shimi Smith

In this document, we will provide a brief overview of the basics of the C programming language, putting emphasis on the most relevant areas. I will assume some previous programming experience and for certain topics universal to all programming language I will only provide an example of the C syntax with a brief explanation. In addition to this document I would recommend "The C Programming Language" by Brian Kernighan and Dennis Ritchie.[1] This book describes all the features of C in great detail and is suitable for all levels of programming experience.

3.1 So, what makes C different than other programming languages?

Well, one thing is that it's not an object oriented programming language like Java or C++ or a whole lot of other programming languages. In addition to that, C is considered a relatively low level programming language since most C operations can be moderately easily translated into assembly. Because of this, C is a great choice for developing software that interfaces directly with hardware. Another unique part of C is pointers and dynamic memory allocation. C allows its programmers to have a generous amount of control over the computer's memory.

Now, let's get into C.

[1] You can find a pdf of this book on the Google Drive in /Instrumentation/Literature

3.2 Variables in C

C includes the following most common types: char, int, long, double and float. A char is always a byte but the size of the other variables is machine dependent. Due to the confusion caused by variable sizes on different machines, in our software we use the types defined in stdint.h. These include the following unsigned types: uint8_t, uint16_t, uint32_t and uint64_t and the following signed types int8_t, int16_t, int32_t and int64_t, where the numbers are the number of bits the variable holds. These types allow us to have better control over the sizes of our variables. So don't use a 64 bit variable in a for loop that loops 10 times.

Here you can see how to initialize and set variables using binary, hex and decimal. The "//" before text is how you make a comment in C.

```
uint8_t x = 0b10101010;  // expressing a number in binary
uint16_t y = 0xFFFF  // expressing a number in hex
uint8_t z = 77;  // expressing a number in decimal
```

You may have noticed no mention of strings or booleans, which are common types in several other languages. While a char represents a byte, it also can represent a character, hence the name char. So a string of characters is just an array of chars with a terminating character (`\0`) as the last element. `char[] s = "Hello World"` is a away to initialize a char array.

As for booleans we just use a char or uint8_t to represent them. 0 represents false and any other number is true. By convention we use 1 to represent true. So the statement `5 == 5` evaluates to 1 whereas `5 != 5` evaluates to 0. Here are the mathematical operators you can do in C:

* (multiplication)
/ (division)
+ (addition)
- (subtraction)
% (modulus)

Here are the comparative operators:

`>` (greater than)
`>=` (greater than or equal to)
`<` (less than)
`<=` (less than or equal to)
`==` (checks equality)
`!=` (checks inequality)

There are also the following logical operators:

`&&` (and) - `a && b` is true if and only if a and b are true
`||` (or) - `a || b` is true if and only if at least one of a and b are true

There are also the following short forms for incrementing or decrementing numbers.

```
a += b;  // this is the same as a = a + b
a -= b;  // this is the same as a = a -b

a++;  // a = a + 1
a--;  // a = a - 1
```

In addition to `a++` and `a--` there is `++a` and `--a` which are indistinguishable without a context.

```
// example 1
if(a++ == 5){
}

// example 2
if(++a == 5){
}
```

In *example 1*, a is used first in the `a == 5` check and then is incremented, whereas in *example 2*, a is first incremented and then used in the `a == 5` check.

3.3 Bitwise Operators

These operators directly manipulate the bits in numbers. A lot of our software is manipulating 8 bit registers on our microcontroller so bitwise operators are very common in our software. The first two are very similar to `&&` and `||` mentioned above. They are the "and" and "or" bitwise operators, represented by `&` and `|`. They both combine two numbers into one. First I will explain the "and" operator.

This is how it is used.

```
uint8_t x = 0b10110001 & 0b00101110;  // x equals 0b00100000
```

It works like this. We'll move from right to left comparing the bits. If both the bits are 1 the result will have a 1 in that location, otherwise it will have a 0. Just like `&&` it needs two 1's to get a 1.

```
10110001
00101110
00100000
```

There is only a 1 in the 6th bit position because that is the only position where there is a 1 in both the numbers.

The "or" operator works exactly the same except if at least one of the bits is a 1 the result will have a 1 in that position.

```
uint8_t x = 0b10110001 | 0b00101110;  // x equals 0b10111111
```

There is another operator really similar to those two. It's called the XOR (exclusive or) operator and is represented as `^`. If one and only one of the bits is a 1 then the result will have a 1 in that position.

```
uint8_t x = 0b10110001 ^ 0b00101110;  // x equals 0b10011111
```

All these operators can also be used as `&=`, `|=` and `^=`.

The next group of bitwise operators is bit shifts. There is left (`<<`) and right (`>>`) bit shifts. They do what the name suggests, shift the bits in a number.

```
// Consider the following 8 bit number
uint8_t x = 0b11111111;

// To shift the number 4 to the left we do the following
x = x << 4;  // or x <<= 4
```

The bits are shifted to the left and 0's are shifted in. You might think x is now `0b111111110000` but that is incorrect because that is no longer an 8 bit number. It is actually `0b11111111`. With bit shifts you have to consider the size of the variable because if you shift a number to the left past its limit those bits will be cut off.

```
// Now consider the following 16 bit number
uint16_t x = 0b11111111;  // Since this number is 16 bits long it is currently holding 0b0000000011111111

x = x << 4;
```

When we shift this number by four, four of the leading zeros are being cut off so we end up with `0b0000111111110000`, or more nicely `0x0FF0`.

The right bit shift does the exact same thing except in the opposite direction. No matter the size of the variable the bits will be cut off.

```
uint8_t x = 0b11111111;

// To shift the number 4 to the left we do the following
x = x >> 4;  // or x >>= 4
```

In this case the bits are shifted out of the variable to the right and 0's are shifting into the variable on the left. Here we will end up with x as `0b00001111`.

There is one final bitwise operator called the not or compliment operator. It flips all the bits in the number so 1's become 0's and vice-versa. It is represented by the ~ symbol.

Example 1:

```
uint8_t x = 0b10100011;

x = ~x;  // or x ^= x
```

In this example x becomes 0b01011100.

Example 2:

```
uint16_t x = 0b11110000;

x = ~x;  // or x ^= x
```

In this example x becomes `0b1111111100001111`. Don't forget about those leading zeroes.

3.4 Applications of Bitwise Operators

I mentioned that bitwise operators are very common in our software so now I will show you the most common uses in our software.

Very frequently we have to set bits in an 8 bit number to change settings in our microcontroller. We use bitwise operators to change certain bits while not affecting others.

This is how to write a certain bit to 1

```
uint8_t x = 0;

x |= 1 << 5;
```

Now the value of x is `0b00100000`.

And you can set a certain bit to 0 with the following,

```
uint8_t x = 0xFF;

x &= ~(1 << 5);
```

Now the value of x is 0b11011111

You can also switch the value of a bit with the following:

```
uint8_t x = 0xFF;

x ^= 1 << 5;
```

3.5 If statements in C

If statements in C look very similar to if statements in java

```
if(statment here){
    // code here
}
else if(another statment here){
    // other code here
}
else{
    // default code to run
}
```

3.6 Switch statement

Switch statements are used quite often in our software.

```
switch(variable_to_check_against):
    case expression_1:
        // code to run
        // put a break here if you don't want any of the cases below to run
    case expression_2:
        // code to run
        // put a break here if you don't want any of the cases below to run
    default:
        // code to run
```

3.7 While Loop

```
while(condition){
    // code to run
}
```

3.8 For Loop

```
for(uint8_t i = 0; i < 10; i++){  // for loop runs 10 times
    // code here
}
```

There are two useful commands for conditional statements and loops that are worth mentioning. These commands exist in most other programming languages.

> **break** - exits the current statement or loop
> **continue** - skips the current iteration of the loop and continues with the next

3.9 Functions

In C functions are very similar to functions/methods in other programming languages. We'll look at the following code to learn how to do functions in C and see what a C file should look like.

```
#include <stdint.h>  // you have to include this header to have access to uint types

uint32_t sum(uint8_t x[], uint8_t size);  // this is a function prototype

// This is the main method in C
int main(){
    uint8_t x[] = {1,2,3};  // This is declaring an array of uint8_t and filling it with 3 numbers

    uint32_t s = sum(x, 3);  // calls the function and puts the returned value in the variable s
}


// This is a function to add up the numbers in an array
uint32_t sum(uint8_t x[], uint8_t size){

    uint32_t result = 0;

    for(uint8_t i = 0; i < size; i++){
        result += x[i];
    }

    return result;

}
```

First I included `stdint.h`. I will go into header files a bit more later.

In C the main method returns an int. By default it returns 0. All C projects must contain one and only one main method.

The function sum takes in a `uint8_t` array and it's size and returns a `uint32_t`. In C you have to always pass the size of an array with the array.

The only difference with functions in C is that for the function to be recognized throughout the file it needs to have a function prototype as show above.

3.10 Header Files

A header file that we have already seen is `stdint.h`. The purpose of header files is to have definitions and declarations that can be shared amongst multiple files by including the header. By putting function prototypes in a header file and including it, you can call those functions in other files. For non-library header files make sure to include them using `#include "file_name.h"` whereas library headers are included with `<file_name.h>`. Header files can also include other header files.

You can also put define and typedef statements in header flies. These statements that I will describe shortly, can also be put in the source file but for better organization tend to be put in a header file.

```
#define foo bar

typedef uint8_t boolean;
```

The preprocessor will replace text in the source code based on the define statements. So every occurrence of foo in the source code will be replaced with bar.

The `typedef` statement shown above lets you use "boolean" as a type. Now the following is a valid statement.

```
boolean booly = 5 != 7;
```

3.11 Pointers

A pointer is a type of variable whose value is the memory address of another variable. This is how you declare and use a pointer.

```c
uint8_t x = 7;

/*
This is a pointer to a uint8_t.
That means it holds the address of a variable that is of type uint8_t
*/
uint8_t* ptr;

ptr = &x;  // &x gives the memory address of x

// You access what the pointer points to like this, *ptr

/*
So if you do this you are changing what the pointer points to,
making x 8
*/
(*ptr)++;
```

3.12 Structs

Structs are a useful way of bunching multiple variables together. This is probably the closet C gets to objects. We tend to declare our structs in a header file using typedef for easier use.

```c
/*
This is how we define a struct
Typedef let's us call this struct person
*/
typedef struct{
    uint8_t age;
    char name[10];
} person;
```

```c
person bob = {27, "bob"};  // This is how you initialize a struct of type person
/*
You access the variables in a struct like,
bob.age;
bob.name;
*/
```

I think this should be enough of a crash course on C to get you writing programs. There are more details on pointers and structs that I have left out. I would suggest the C book I mentioned earlier to learn more about those topics.

4 SPI (Serial Peripheral Interface)

Written by Bruno Almeida

In this section, we will describe the SPI protocol and how to use our library with the ATmega32m1, the main microcontroller used on the Heron Mk. II cube-satellite.

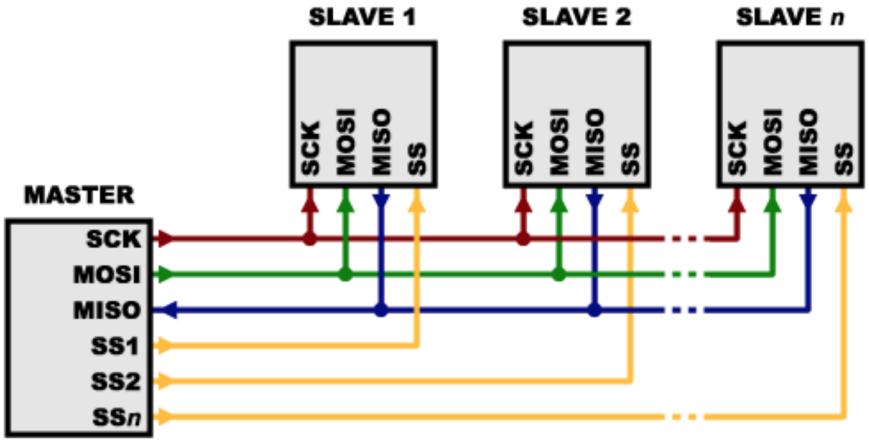
4.1 What is SPI?

SPI is a communication protocol used to communicate between microcontrollers and peripheral devices, such as sensors. Put simply, it's a system that allows us to send a byte to some device and receive a byte in return.

SPI uses what's called a Master-Slave architecture. In this system there is one master device that communicates to multiple slave devices. In our system the master device is our microcontroller and the slave devices are mostly sensors. The master can only communicate with one slave device at a time and the slaves cannot communicate with each other.

4.2 SPI Bus

SPI uses four lines, which make up what we call the SPI bus. Three of these lines (SCK, MOSI, MISO) are shared amongst the slave devices. Whereas the final line (CS) must be unique to each slave device. SPI is a synchronous communication protocol, meaning data is sent and received at the same time. This means that two of the four lines (MOSI, MISO) are for data, and one of the lines (SCK) is for timing. One of the data lines is for master to slave and the other is for slave to master. All of this will make more sense shortly. The four lines are shown in the figure below. They'll be explained in more detail shortly.



4.2.1 SCK (Source Clock)

The clock keeps both the data lines in sync. The clock is an oscillating signal produced by the master device that tells the receiving device when to read the data. Depending on the device properties, data is either sent/received on the rising/falling edge of SCK. I'll touch on this in more detail shortly. This line is shared by all slave devices.

4.2.2 MOSI (Master Out Slave In)

The name pretty much explains this line. This is the line where data is sent from the master device to the slave device. This line is shared by all slave devices.

4.2.3 MISO (Master In Slave Out)

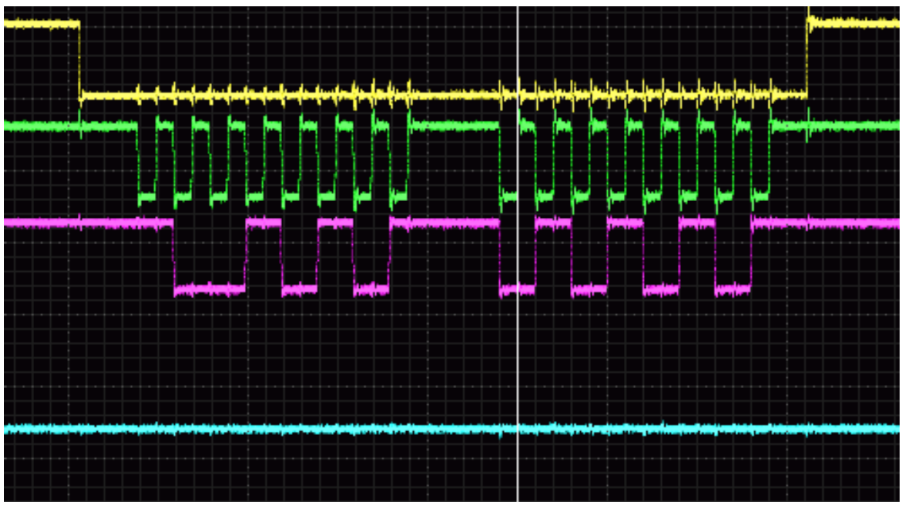
On this line the data is being sent out of the slave device received by master. This line is shared by all slave devices.

4.2.4 SS/CS (Slave Select / Chip Select)

This line is referred to as SS and CS interchangeably. CS is active low, which means the slave device is active when CS is written low. Only one CS can be low at a time or there will be conflicts on the SPI bus resulting in garbage data. We use a pull-up resistor on the CS pin to set a default value.

A pull-up resistor is a large resistor (typically 10K) which bridges between VCC (3V3 in our case) and another pin. When no load is applied to the pin, no current flows through the resistor. This allows us to hold CS at a known (3V3) state when the CS pin isn't being driven by other circuitry. Then, we can drive another pin on the CS line low (GND) to select the device. Current will flow through the resistor and drop 3V3 across it.

Hopefully the diagram above now makes sense. This is what a SPI transfer should look like:



I took this image from online but SPI transfers look very similar on our equipment.

I'll pose the following question:

> *Which lines are which?*

You can't really distinguish between MISO and MOSI in this picture but just pick one to be MOSI and the other to be MISO. What is being sent and received? The answer is given below.

> **Answer:**
>
> 1. The **yellow line is CS**. This is because it is being lowered before and raised after the SPI transfer is complete.
> 2. The **green line is SCK**. It oscillates 8 times for each byte sent.
> 3. The **pink and blue are MOSI and MISO**. The blue line has `0x00` and `0x00` and the pink line has `0b10010101` and `0b01010101`.

4.3 Using our SPI Library

Here are the basic software steps to send SPI messages:

1. Initialize CS as an output pin
2. Set CS high
3. Initialize SPI
4. Send SPI message
5. Set CS low
6. Send message
7. Set CS high

4.3.1 Initializing CS as an output pin and writing CS high or low

On our microcontroller each IO pin has three registers that control it. We will only need to use two of them. There is the data direction register that controls if the pin is input our output and there is the port register that lets you write high or low on the pin.

Once you figure out what pin you are using for CS you can check the microcontroller's datasheet to get the name of the pin. Figure 1 shows the pin configuration for the ATmega32m1 microcontroller.

There are four banks of ports (B, C, D and E) with eight pins on each. There are 8-bit data direction and port registers for each of the four ports. Each bit in the register is for a separate pin. The data direction register is called DDRx and the port register is just called PORTx where x is the port. So if you wanted to initialized PB6 as output the code would be the following.

```
DDRB |= _BV(PB6);
```

`_BV(PB6)` is a macro that expands to `1 << PB6` and PB6 is a macro that expands to 6. Here is the code to write high or low on PB6. "'

```
PORTB |= _BV(PB6);  // write PB6 high
PORTB &= ~_BV(PB6);  // write PB6 low
```

In our SPI library we have functions that will do this for you. This is how you use them.

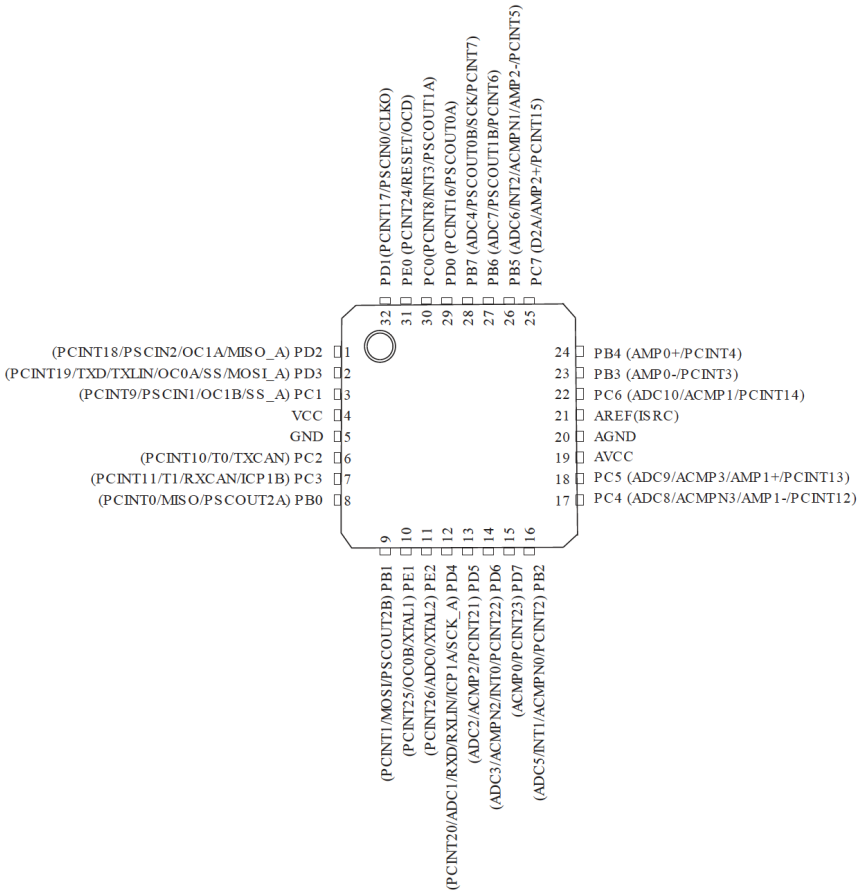


Figure 1: ATmega32m1 pinout

```c
#include "spi.h"  // Include the library's header file

#define CS PB2  // We will use PB2 as the CS pin
#define DDR_CS DDRB
#define PORT_CS PORTB

int main(){

    /* Initializes the CS pin as output
    This takes in a pointer to the DDRx register
    */
    init_cs(CS, &DDR_CS);

    set_cs_low(CS, &PORT_CS);  // Writes CS low
    set_cs_high(CS, &PORT_CS);  // Writes CS high
}
```

4.3.2 Initialize SPI

We have a function, `init_spi()` that does this. It initializes SCK and MOSI as output and sets the SCK frequency to 8 MHz / 64. 8 MHz is the frequency of the 32m1's internal clock.

4.3.3 Sending a SPI message

SPI sends 8 bit messages. If you want to send more than a byte you can send consecutive SPI messages. This is how you do it.

```c
send_spi(0b10101010);
```

4.3.4 Example SPI Code

This a full SPI program that I used to make sure SPI was running correctly on the microcontroller.

```
init_cs(CS, &DDR_CS);  // initialize CS

set_cs_high(CS, &PORT_CS);  // Write CS high so the slave is not active
init_spi();  // Initialize SPI

while(1){
    set_cs_low(CS, &PORT_CS);  // Write CS low so the slave is in an active state
    send_spi(0b10101010);  // Send a SPI message
    set_cs_high(CS, &PORT_CS);  // Write CS high so the slave is no longer in an active state
}
```

This repeatedly sends `10101010`.

4.3.5 More Clock Settings

As I touched on earlier in this document, there are more clock settings.

Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)	Output Edge	Data Capture
SPI_MODE0	0	0	Falling	Rising
SPI_MODE1	0	1	Rising	Falling
SPI_MODE2	1	0	Rising	Falling
SPI_MODE3	1	1	Falling	Rising

The two clock settings introduced here are Clock Polarity and Clock Phase. Clock Phase determines whether data is shifted in and out on the rising or falling edge of the data clock cycle. Clock Polarity determines determines whether the clock is idle when high or low.

This is the end of this SPI tutorial. Hopefully you you are now a SPI expert.

5 Lab Equipment (WIP)

Written by Russel Brown

Here are some of the types of equipment in our lab (MP 099).

5.1 Breadboard

Used to quickly build and change circuits for prototyping Has holes to insert through-hole components and wires Particular sets of holes are connected under the board The long rails (two on each side) are all connected. Generally you will

want to use the red rails for power and the blue rails for ground. Each row of 5 holes in the main part of the board is connected together. These are generally used to connect components together. Follow the lines, and notice the breaks in the lines (diagram/photo with lines indicating the connected sets of holes)

5.2 Printed Circuit Board (PCB)

Used to create final circuits or major prototype versions of a circuit Getting a functional PCB requires many steps: Design it using CAD software (we use one called KiCad) - create a schematic of the components then create a PCB layout of those components Send the design to a manufacturer, who prints the board with just the connections/traces/wiring Order all of the actual components After receiving the board from the manufacturer, solder the components onto it The ordering and soldering process alone usually takes at least a week, which is why PCBs are not used for fast prototyping A PCB's connections can't be changed after it is ordered and printed (with rare exceptions, ask Dylan for a good story/photos) It is possible to use through-hole (TH) components on PCBs, but we use surface mounted (SMD/SMT) components, which are much smaller

5.3 Protoboard (not entirely sure about this, need someone to check it)

Somewhat between a breadboard and PCB Has connected tracks like a breadboard, but components need to be soldered and are more likely to stay in plane

Use through hole components

5.4 Tools

5.4.1 Multimeter

Used to measure voltage, current, and resistance in specific parts of a circuit and check for short circuits

- Voltage - must be measured across a component because voltage measures the relative energy between two points
- Current - you have to break (disconnect) your circuit at the point you want to measure, then insert the multimeter like another series component in the circuit
- Resistance - must be measured across a component because resistance is measured between two points

- Short mode - use this mode to determine if there is a short circuit (direct connection with no resistance) between two points in the circuit. This is important to check that you have not made extra connections that change your circuit. A short circuit produces a very high current that can damage components.

5.4.2 Oscilloscope

Used to measure waveforms (signals) over time in a circuit This is useful for viewing the raw data/signal in a wire, such as a sensor's output or communication lines

5.4.3 Function Generator/Power Supply (are these the same thing? Check AC vs DC)

Used to generate power or a signal with a specific voltage, current, and/or waveform

5.4.4 Wire Cutters

Used to cut specific length of wire

5.4.5 Wire Strippers

Used to remove some of the insulation off the ends of a wire so it can be connected

5.4.6 Soldering Iron

5.5 AVR/ATmega32M1

Our subsystem uses/will controlled by the ATmega32M1 microcontroller (how to define this, like a processor maybe?) on the satellite as the subsystem's main controller Programmed in C Need to install the AVR software to compile and upload code to it (we will help you with this)

5.6 Arduino

Arduino is a platform of microcontrollers, which we use to test components and code quickly Faster to write and upload code to test individual components than using the main AVR computer, and Arduino already has more built-in code which is easier to get running (but this isn't available on AVR)

6 Reading Datasheets

Written by Dylan Vogel

6.1 Reading Datasheets for Electrical Engineers

6.2 Reading Datasheets for Computer Scientists