Now that we've seen some of the computing solutions that could be used for designing a payload, lets dive deeper under the hood of the controller that we've used for our example payload, the Arduino Uno.

The Arduino Uno is a 5V 8 bit microcontroller, when power is connected to it, it runs whatever program that has been uploaded to it. These programs are written in the C++ programming language with some extra bells and whistles added by the Arduino company to make many operations easier. Earlier in our lessons we've looked at some of the capabilities the Arduino Uno has and how some of the program functions work. Generally speaking these programs have a "setup" function that runs once at powerup and a "loop" function that runs repeatedly until power is lost or until the reset button is pressed, which stops the system and starts back from the beginning at the "setup" function. These functions use curly brackets {} to define the contents of the function, and parentheses () that state the inputs of the inputs of the functions. Note that our setup and loop functions don't have anything in their parentheses, this is because these particular functions don't normally have inputs, but a programmer is able to define their own custom functions that can input and output values. Setup and loop can be thought of as the "outside" functions, so there isn't any way to pass a value through these functions at startup. Within the functions, each statement is completed with a semicolon ; and statements are evaluated from top to bottom. When a program is written and the run or upload button is pressed in the arduino development environment (IDE) checks the code to make sure that everything is written following the rules of the C++ language like semicolons at the end of every statement, after that the code is then converted into binary for uploading to the Arduino Uno itself. From that point, the Arduino runs that binary code to execute the program. The conversion from human readable code to binary code is called "compiling". A programmer can add notes to explain parts of their code using comments, which is anything on a line after a pair of slashes // or anything between a pair of slash stars like this /*comment*/. Let's look at the specialized components of the Arduino and the related functions.

[DIGITAL]

The Arduino is a programmable digital electronic computer. By this, I mean that it uses electricity to power its computation mechanism as opposed to computers using physical mechanics to carry out operation, and while there is a world of electronics that purely deals in continuous signals that respond to any variation, the digial devices respond to 2 discrete voltage levels, a digital HIGH level, and a digital LOW level. The UNO is a 5V logic level digital device, so it encodes a binary 1 or HIGH/TRUE to 5V on a pin. And it associates a binary 0 with LOW/FALSE. Digital microcontrollers can accomplish many feats, and every mechanism in at root is encoded in this binary. Since our controller is programmable, we can arbitrarily control its signals using code, that can be easily modified by simply uploading a new program.

Blink §

```
1  void setup() {
2    pinMode(13, OUTPUT);
3  }
4
5  void loop() {
6    digitalWrite(13, HIGH);
7    delay(1000);
8    digitalWrite(13, LOW);
9    delay(1000);
10 }
```

The functions pinMode() and digitalWrite() can be used to create simple digital signals like blinking for an led.

Use pinMode(*pin, INPUT/OUTPUT*) in the setup function to configure pin 13 as an output, we're choosing 13 so that we can blink the led that comes presoldered to the 13th pin of normal Arduino Unos.

In the loop function, use digitalWrite(*pin, HIGH/LOW*) to set the output voltage to either 5V or 0V/ground. Here we're using delay(1000) to make the controller pause 1000 milliseconds or 1sec.

Button §

```
1 void setup() {
2   pinMode(12, INPUT);
3   pinMode(13, OUTPUT);
4 }
5
6 void loop() {
7   // Read value from 12, and write it to 13
8   digitalWrite( 13, digitalRead(12) );
9 }
```

To read a pin, use a pinMode() function to configure a pin to read a digital signal, and use digitalRead(*pin*) to read the signal. When the code is executed, code within parenthesis are executed first and then replaced with the resulting value. So if this code read a 5V signal, the digital write section would be replaced with a digitalWrite(13, HIGH); and the led on the physical circuit board will turn on. If using a pull up resistor like the circuit from our TTL logic section, a LOW would represent the button pressed and HIGH would mean not pressed, since the button directly connects to ground when pressed.

[ANALOG]

The Arduino is a 5V digital device so it's sensitive to 5V as HIGH, and 0V as LOW, and if the desire is to read values between, you need an Analog to Digital Converter (ADC) to convert the analog signal into a digital value that the controller can understand. Not all controllers have ADCs because they aren't always necessary, but the Uno has 6 pins capable of reading analog input signals. Use analogRead(*pin*) in a similar fashion as digitalRead() to read an analog pin. Those pins can be addressed using A0 – A6 as the pin number. The difference is that instead of getting a HIGH/1 or LOW/0 like how digitalRead() works, analogRead() returns a number between 0 and 1023 with 5V being 1023 and 0 being 0V. The reason for this range is because the ADC has a 10-bit resolution, so the possible values a 10 bit binary number can represent is 2^{10}=1024 and 0 is one of those numbers so the max value is 2^{10}-1=1023. That means that the smallest changes it can read are 5V/1023div=4.89mV/division. If the signal being read has meaningful data smaller than 4.89mV/div, then an Uno would need an external ADC.

[PWM]

There is an analogWrite(*pin*, *value*) function, but it's not quite a direct method to output an analog signal. The Uno is digital, so this still needs to be broken down to binary, so the way controllers often do that is with Pulse Width Modulation (PWM). PWM turns that pin on and off very quickly to simulate an average analog signal, and the pins with a wavy symbol on the Uno are capable of doing this. The PWM driver inside the Uno has an 8 bit resolution, so the maximum value for a full on signal is 2^{8}-1=255 while 0V would be 0. Using pinMode(9, 128) with drive an led on pin 9 around half as bright as just using a digitalWrite() to light up the led. What's actually happening is the pin would flash the led so quickly that the human eye can't see the blinking, and the signal would be a square shaped wave that's on 50% of the time and the other half off. Using analogWrite(9, 64) would make the signal on 25% of the time.

[SERIAL UART]

Many controllers have one or more methods for communicating text characters with other computers. Since 8 bit numbers can represent 256 unique values, this offers plenty of space to use a unique binary number to encode text, which is called ASCII code. An ASCII table shows what number in binary represents each character, for example 'A' is 65 and 'a' is 97. The simplest type of Serial to use for the Uno is Universal Asynchronous Receiver/Transmitter (UART), which has a TX or transmit pin and an RX or receive pin on pins 0 and 1. These pins are connected to a UART/USB converter chip and can be read on the computer the USB cable is connected to using the Serial Port in the Arduino IDE. Serial.begin(*speed*) to startup the serial port at a given signal speed, and use Serial.print(data) to

print to the screen. When the Serial monitor is open, the speed should be set to the same speed declared in the begin function so that the binary won't be misinterpreted, this sets the binary speed the host computer is expecting. The print function can be used with different types of data, and an optional "format" input can be used after the "data" option to specify how the data should be printed. Here's an example;

sketch_jun20a §

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  // print labels
  Serial.print("NO FORMAT");  // prints a label
  Serial.print("\t");         // prints a tab

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.println();        // skip to the next line, "carriage return"

  // Store the value 0 in an integer variable named 'x'
  // Execute the code between the brackets
  // add 1 to x at the end and repeat as long as x is less than 64
  for (int x = 0; x < 64; x++) {

    Serial.print(x);           // print x using its own format, decimal
    Serial.print("\t\t");

    Serial.print(x, DEC);      // format as decimal for an integer
    Serial.print("\t");

    Serial.print(x, HEX);      // format as hexidecimal
    Serial.print("\t");

    Serial.println(x, BIN);    // print in binary with carriage return

    delay(200);                // delay 200 milliseconds
  }
  Serial.println();            // prints another carriage return
}
```

```
NO FORMAT       DEC     HEX     BIN
0               0       0       0
1               1       1       1
2               2       2       10
3               3       3       11
4               4       4       100
5               5       5       101
6               6       6       110
7               7       7       111
8               8       8       1000
9               9       9       1001
10              10      A       1010
11              11      B       1011
12              12      C       1100
13              13      D       1101
14              14      E       1110
15              15      F       1111
16              16      10      10000
```

The dark window is a screenshot of the output read through the serial monitor. This program uses Serial to print an integer number using different formats. First, a "header" is printed to label each column, and then a "for" loop is used to count a value from 0 to 64, and print that number with no format, decimal, hex, and binary formats.

[SPI]

There are other styles of serial communication that use different styles of binary encoding to facilitate transmitting ASCII text, each with their own pros and cons. UART requires 2 pins, TX and RX, for each pair of devices communicating and they need to be configured in code to the same binary clock speed. This means that an Uno talking to 5 other devices would require 5 UART ports occupying 10 individual pins on the Uno, and it only has one hardware based UART port. Serial Peripheral Interface (SPI) takes a different approach. The SPI protocol assumes that there is a master controller with slave devices like sensors or a micro SD card that don't do anything on their own for the Uno to control. SPI uses a minimum of 3 pins, master-in-slave-out (MISO), master-out-slave-in (MOSI), and serial-clock (SCK). MISO is comparable to the RX pin in UART since that is the master-input pin, and MOSI is comparable to the output TX pin in UART, and instead of each device needing to already know each other's speed the master device tells the slave device the serial speed using the SCK clock pin. When multiple slave devices are used, MISO, MOSI, and SCK pins for all devices are connected together, with one additional slave-select (SS) pin for every additional slave device so the Uno can specify which device should listen to the commands being sent out while all others ignore the data. This means SPI consumes 3 pins minimum on the host controller, and for every additional device, only one additional pin on the Uno is required. To compare this to UART that needed 10 pins to talk to 5 devices, SPI could talk to 7 devices using those 10 pins. Often times SPI devices come with a library made by the selling company, so lots of the time direct use of the SPI programming library is not

necessary, instead using the sensor library functions. Our micro sd card example uses SPI and the SD library which are included in the Arduino IDE automatically, but other libraries would need to be manually included, more on that later. On the Uno, MOSI is pin 11, MISO is pin 12, and SCK is pin 13.

[I^2C)

Inter-Integrated-Circuit (I^2C), often also referred to as Two-Wire-Interface (TWI), is the last type of serial that the Uno has dedicated hardware for. I^2C uses a different approach at communicating with multiple slave devices using only 2 pins as the name implies, serial-data (SDA) on pin A4 and serial-clock (SCK) on pin A5. SPI uses additional selector pins so the Uno can tell which device should listen to its commands, I2C instead transmits a 7 bit binary number called an address that acts as an identity number for each slave device. Every slave device knows its own address and is usually set by the manufacturer and sometimes can be modified by changing physical pins on the circuit board if they're available. Since the address is a 7 bit number, I2C can theoretically allow an Uno to control 2^7-1=127 unique devices using only 2 pins, though many times as the number of slave devices grow, so does the likelihood that one of the devices will have identical addresses which means they would both respond to the Uno at the same time and interrupt each others data. Since the address is usually set in hardware the only way around this address problem is to make sure that any devices that are meant to be used in the same system don't have the same address prior to buying them. The other downside to I^2C is that it's slower than the other serial protocols, though not all applications would be affected by this slower speed. I^2C uses the Wire library that is built into the Arduino IDE but similar to SPI, a lot of the time the Wire library isn't used directly and instead a library made by the manufacturer specifically for that device is used. The barometer from our example payload design uses I2C to talk to the Uno, so the two "include" statements are there to import the Wire library and the library for the sensor made by Adafruit with all the sensor interactions being done with functions from the Adafruit library.

[CLOCKS]

Arduino has several clock and timing related functions that are very useful. The simplest one is delay(*milliseconds*) which makes the controller pause for the specified number of milliseconds. This however is inefficient since the controller is doing nothing during the pause when it could be moving on to some other meaningful task, though not all applications are negatively affected by this. Another way to deal with timing is to use the millis() function, which outputs the time since startup in milliseconds. Another pair of functions delayMicroseconds(*microSeconds*) and micros() have the same functionality except on the microsecond scale instead of milliseconds. Just to show the difference in these functions behavior, lets look at a code comparison;

delay

```
void setup() {
    pinMode(13, OUTPUT);
    pinMode(12, OUTPUT);
    pinMode(11, OUTPUT);
}


void loop() {
    digitalWrite(13, HIGH);
    delay(750);
    digitalWrite(13, LOW);

    digitalWrite(12, HIGH);
    delay(500);
    digitalWrite(12, LOW);

    digitalWrite(11, HIGH);
    delay(250);
    digitalWrite(11, LOW);
}
```

millis §

```
// Store pin numbers in integer variables
int led1 = 13, led2 = 12, led3 = 11;

// Large size integers for storing result from millis
unsigned long lastBlink1 = 0, lastBlink2 = 0, lastBlink3 = 0;

// Set the blink period for each led (in ms)
int period1 = 250, period2 = 500, period3 = 750;

// boolean variables to store each leds state
bool state1 = 0, state2 = 0, state3 = 0;

void setup() {
    // set all 3 pins to outputs
    pinMode(led1, OUTPUT);
    pinMode(led2, OUTPUT);
    pinMode(led3, OUTPUT);
}

void loop() {
    // if current millis time is longer
    // than period since last blink, run
    if(millis() - lastBlink1 >= period1){
        state1 = !state1;            // flip the led state
        digitalWrite(led1, state1); // set the pin state
        lastBlink1 = millis();      // update the blink time
    }

    if(millis() - lastBlink2 >= period2){
        state2 = !state2;
        digitalWrite(led2, state2);
        lastBlink2 = millis();
    }

    if(millis() - lastBlink3 >= period3){
        state3 = !state3;
        digitalWrite(led3, state3);
        lastBlink3 = millis();
    }

}
```

On the left is a program blinking 3 leds, but using different delay values. If someone was trying to blink the 3 leds at their own independent frequencies, this would fail, because the controller just pauses in place for the entirety of the milliseconds given to the delay() function. This means there's valuable time that the controller could be using capturing data from other sensors that can serve up information faster. This series of delays controls how long the controller will pause before moving on to the next led, and the time wasted per loop goes up.

On the right, we instead use a series of variables stored as integers for general numbers, positive (unsigned) long integers since many milliseconds can pass while running any controller, and booleans which are 1/true, 0/false. As the variable names indicate, we are controlling 3 leds to control their pin, blink period, time since the last state change, and the state of the led. The setup the 3 led pins are configured as outputs, and then the program moves on to the loop function. We set up 3 if check statements for the 3 leds, and if the current millisecond time minus the last state change is greater than or equal to their period, then execute the body of the code. If the check evaluates to false because it isn't time to flip the led state, then that if check is skipped, avoiding wasting time just waiting idle for a device's behavior to pause. This gives the visual effect that the 3 leds are are blinking at their own given frequencies independent of each other.

Here’s another program that does the exact same thing as the millis program above but using integer arrays of 3 to reduce the amount of typing needed.

millisArray

```
int leds[] = {13, 12, 11};
unsigned long lastBlinks[] = {0, 0, 0};
int periods[] = {250, 500, 750};
bool states[] = {0, 0, 0};

void setup() {
    for( int x=0; x<3; x++){
        pinMode(leds[x], OUTPUT);
    }
}

void loop() {
    unsigned long currentTime = millis();
    for(int x=0; x<3; x++){
        if(currentTime - lastBlinks[x] >= periods[x]){
            states[x] = !states[x];
            digitalWrite(leds[x], states[x]);
            lastBlinks[x] = millis();
        }
    }
}
```