**EA-467 Command and Control – Data Handling** (rev f) 28 Sep 2016

**INTRODUCTION:** Every complex system such as a satellite or even satellite subsystem has a CPU processor. The processor’s job is to monitor conditions and status of its system. It is also responsible for the implementation of algorithms, and the command and control necessary for proper operation. Generally, the CPU of a simple system will perform the following tasks, and the engineer will write code to implement them during satellite design. This lab will teach you how to do the following:

* Monitor sensors, voltages , currents, temperatures, states, and status
* Combine this information into telemetry and transmit it to the ground
* Communicate with the ground for command and control
* Log data and telemetry when out of view of ground station
* Activate/control devices, e.g. to activate a string cutter to deploy an antenna or solar panel
* Schedule spacecraft and payload operations
* Monitor power systems and maintain safe loads
* ****Monitor attitude and implement attitude control algorithms

**Arduino CPU:**

The Arduino CPU is a widely used educational and training system for micro controllers. It is easy to program and very flexible. There is a wealth of information and routines available for this processor. The Arduino board shown here and below is an image of the CPU board attached to a prototyping board where you will add interface circuits.

**EXERCISE OBJECTIVES:** For code development we will use the prototyping board shown to the right. You will program your processor to perform the following mission profile. **(Note: In order to maximize learning, ensure that you enter the lines of code in this handout manually rather than by using copy and paste.)**

****Keep the objectives of your spacecraft in mind as you learn how to program the CPU through the various parts of the lab:

Mission Profile:

* Initialize variables and run self-test
* Begin data/telemetry logging
* Wait for Separation Switch activation
* Delay
* Monitor sun tracker for alignment
* Deploy solar panel when sun aligned
* Record telemetry
* Await ground station acquisition
* Download data log

**PART A – The Basics**

On the desktop, open the Arduino Editor. Connect your CPU board to power and the PC via the provided USB cable. Now plug in the Arduino’s USB cable and check which serial port the board is using in the “Devices and Printers” folder in the start menu or control panel. This should match the serial port under the “Tools 🡪 Serial Port” tab in the toolbar.

Every Arduino program consists of an initial area for declaring global variables and two main functions that can be called to execute program statements. The first is the **setup()** function, which can be used to set initial conditions, the pin modes, and initialize the serial monitor upon startup. It only runs once when the board is powered on or reset. The second is the **loop()** function which continuously runs after setup, controlling the outputs based on changing inputs.

1. **Global Variables:** In Arduino, you can declare different types of variables. To make a variable global, declare it at the start of the program, above the **setup()**function. It is important to specify what type you are declaring, as this exercise shows. Declare the global variables I and X to be integers as shown below:

 **int I = 65; Don’t copy and paste!!!**

 **int X = 'A';**

**2. Setup:** Next, create the two functions as shown below.

 **void setup(){**

**}**

 **Gotcha! Don’t copy and paste please!!!**

**void loop(){**

**}**

All statements will be written between the curly brackets in each function. A semicolon (;) must be used at the end of each statement. To insert a comment, use “//” at the beginning of the line.

Next, add a statement to initialize the serial monitor in the **setup()** function. The serial monitor is used to display information and communicate with the Arduino board. The statement **Serial.begin(9600)** opens this connection, sets the baud rate to 9600 bps, and sends the information to be displayed on the serial monitor. **Serial.print()** is the statement to print out text in the serial monitor. Notice that you do not need quotation marks for variables and the addition of ln adds a new line (carriage return).

 **Serial**.begin(9600); Don’t copy and paste!!!

 **Serial**.println("PART A:");

 **Serial**.print("I is "); Serial.println(I);

 **Serial**.print("X is "); Serial.println(X);

Then press the “upload” (🡪) arrow to upload/execute your program and open the serial monitor on the far right of the Arduino toolbar. The output shows that both I and X are 65. This is because both variables are stored as integers. The serial monitor does not display “X is A” because “A” is an ASCII character, not an integer value. To display ASCII text, the variable type must be a character. Add the following to the global variables and the setup() function appropriately and note the difference.

 **char Z = 65;**

 **Don’tSerial.**print("65 is "); **Serial.**println(Z);

In addition to integer and character variable types, there are many other data types. Use the handy Arduino “Help 🡪 Reference” buttons to see a list of these data types including **byte** (8 bits), **word** (16 bits), and **float** (floating-point decimal).

By declaring a variable as a specific type or setting the format in the **Serial.print()** statement, you can display a wide range of variables. Use byte to store an 8-bit unsigned number or float to display decimal values. Add the lines below in the appropriate places and notice that B is used as a formatter to allow you to enter a binary value in 1’s and 0’s and the 2 formatter in the last line signifies that two decimal places should be displayed.

 byte Y = B1000001;

 float Pi = 3.14159; Don’t copy and paste!!!

**Serial**.print("Y is "); **Serial**.println(Y);

**Serial**.print("Pi is "); **Serial**.println(Pi,2);

Upload this code and open the serial monitor to see the result. A byte variable can be interpreted either as a character in the ASCII table or as a number. For example with Y set to 65 it can display as the letter “A” which is the 65th entry in the ASCII table or simply be 65, or Binary 1000001. With the serial monitor still open, you can verify that your program is now permanently in the Arduino board by pressing the RESET button on the circuit card to force the CPU to restart the program.

**3. Loops:** Remember that the global declarations and the **setup()** function run once upon startup while the **loop()** function runs continuously. If we want to repeatedly increment the variable I by one (as with a counter), we can use the statement **I = I+1 which in “C” is just “I=”;.**. Type this in the loop() function (don’t forget the semicolon). Then print the value of I in the serial monitor using

 **Serial**.print("I = "); **Serial**.println(I);

Notice how the response is nearly instantaneous because the processor is running at a clock speed of 16 MHz. Actually, we do not need this high data rate in this function because it would consume huge amounts of memory if we try to store all those samples later on. Therefore, insert the statement delay(1000); to slow the loop down to one sample a second (or 1000 ms, hence the 1000).

**4. Conditional Statements**: Conditional statements are what allow CPU’s to take on control applications, adapt to inputs, and generate outputs based on what is needed in each situation. Add a three second delay at the end of the setup() function and insert these lines after the line that increments I in the loop() function.

copy!!!

 if (I == 70) **Serial**.println("Found I equal to 70!");

 if (I >= 75) **Serial**.println("I is Greater than 75!");

Notice that a double equal sign is used (I == 70). This is an equality comparison operator to test if I is equal to 70. In contrast, a single equal sign is the assignment operator, which sets I to a value of 70. See also the use of **>=**  to test for “greater than or equal to.”

**Part B – Digital Inputs**

Single digital inputs (and outputs) are called **discretes.** Inputs are used to determine the state of sensors and systems such as payloads, switches, actuators, circuits, etc. Any of the 14 digital I/O pins can be used as an input (On the Arduino, pins 0 and 1 are used for communicating with the computer’s serial port and pin 13 is connected internally to a yellow LED). On these pins, an input voltage above about 3 volts will indicate a 1 and below 2 volts will indicate a 0. In this experiment you will use a switch (sep sw), a photoresistor (Rp) and a thermistor (Rt) as inputs as shown in the diagram above. Add the resistors and sensors as shown below. The sensors are connected as R1 in an R1/R2 voltage divider between 5v and ground. If the sensor has a low resistance, it pulls the input pin high. If it has a high resistance, the R2 pulls the pin low. An unused input has an indeterminate state since it is not pulled either way. The 300 ohm resistors protect the CPU pins from accidental shorts.

You are now ready to program the circuit to read and store the values on the pins (sensor inputs for example).

**First, open a new file in the Arduino Program.**

We need a variable to store the value of each of the input pins. The switch, photoresistor, and thermistor (plus a string cutter variable we will use later) will each be declared as individual global variables with initial values of 0. Type the following lines above the setup() function.

int SwState = 0;

int RpState = 0;

int RtState = 0;

int CutState = 0;

Additionally, we want to keep track of time. Add a variable integer called T and set its initial value to zero.

Remember that the setup() function runs once upon startup and this is where we set the pin modes. The pinMode(*pin,mode*) function assigns the component connected to a digital I/O pin to be either an input (if the component is a sensor) or an output (if the component is an actuator). In this case, all of the pins used will be inputs. In the setup()function, set pins 9-11 and 6 to be inputs and initialize the serial monitor using the following statements:

**Serial**.begin(9600);

pinMode(11,INPUT);

pinMode(10,INPUT); Don’t copy and paste!!!

pinMode(9,INPUT);

pinMode(6,INPUT);

We will make a separate function for the reading and printing of the sensor values, so that we can use this function later on. Between the setup() and loop(), type in the following:

void readSensorData() {

}

In the readSensorData()function we will check the state (voltage) of the input pins which will assign the value 1 or 0 using the digitalRead(*pin*) function. For example, the value of the switch on pin 11 will be assigned to the SwState variable as shown below. Pin 10 will be assigned the state of the photoresistor and pin 9 the thermistor.

SwState = digitalRead(11);

RpState = digitalRead(10); Don’t copy and paste!!!

RtState = digitalRead(9);

We want to sample the sensors and output the time and states of each sensor on the display. Write a function called printSensorData()with the following lines.

**Serial**.print("Time: "); **Serial**.print(T); **Serial**.print(", "); an

**Serial**.print(SwState); **Serial**.print(RpState); **Serial**.println(RtState);

Now, we need to run these functions continuously by placing them inside the loop(). Type the function name inside the loop() as shown below. Don’t forget to also add a one-second delay.

void loop() {

 T = T + 1;

 readSensorData();

 printSensorData();

 delay(1000);

}

Verify it displays continuous data and verify the sensor bits individually respond to your inputs, pushing the switch, blocking light to the photoresistor and using your fingers to warm the thermistor.

We can use the photoresistor with narrow field of view as a Sun detector. Use your hand to block the light to the sensor to verify it changes from a 1 to a 0. To demonstrate the usefulness of this detector, insert the conditional statement (below) in the bottom of the printSensorData() function to send out an alert that the sun has been found (sensor is illuminated) and re-run your program.

if (RpState == 1) **Don’t copy and paste!!!**

 **Serial**.println("Sun aligned!");

For the next section (writing to EEPROM), we also need to control how long this loop executes. To stop after say 30 seconds, insert a while(T < 30){ condition after the start of the loop and a } at the end of the loop to end the WHILE condition.

***Verify that this loop operates for 30 seconds and then stops before proceeding to the next section. Save this program if you like, as we’ll build on this more later.***

**Part C – Writing Data**

Note: This section can be completed using the same program as Part B, but when you start changing the program for this section, save it as a new program “Part C”.

On a spacecraft, you usually have to store continuous telemetry inputs so that you can downlink them later when the spacecraft is over a ground station. In the Arduino, we can use non-volatile program memory for this purpose. Electrically Erasable Programmable Read Only Memory (EEPROM memory) stores values that are kept even when the board is turned off. This memory is somewhat limited, though; the Arduino Uno can only store 512 bytes in its EEPROM memory. To store data such as a log of the state of our 3 input devices and corresponding time values, we use the EEPROM.write(*address,value*) statement. Here, the address is an *integer* value that is the location of the stored information. (Like an index in an array.) The value stored is a single *byte* value (therefore 0 to 255) that will be written to memory. To store your one second data samples insert the following before the setup()function, add:

#include<EEPROM.h>; //this includes the EEPROM libraryn’t

 int addr = 0; //declares the variable addr as an integer

Write a function called void writeROMData()and include the following two statements:

EEPROM.write(addr,T); caddr = addr+1;

These two statements store the initial time value in the first address location (0) and then increments the address pointer to the next location. Insert additional EEPROM.write and addr = addr+1 lines for each of the 3 sensors (the SwState, RpState, and RtState values) and increment to the next address each time. Since we have the WHILE loop set to T<30 we will only write 30 samples. But since the loop uses 4 bytes per second it could operate as long as about 128 seconds before the EEPROM is full. Add this function call in the loop() function, before the delay(1000):

void loop() {

 T = T + 1;

 readSensorData();

 writeROMData();

 printSensorData();

 delay(1000);

}

In the last section we made sure this loop would terminate and not run infinitely because the EEPROM memory can perform only 100,000 write/ erase cycles before it is used up and the CPU can do this in a few seconds in an infinite loop, so be careful with your code to make sure it stops after 30 seconds (or unplug the board).

Upload/execute your program to verify your syntax is correct and exercise your inputs to obtain a nice pattern of changing inputs for later replays. Save this program as Part-C. The data once written remains in the Arduino EEPROM for later reading in the next section.

**Part D – Retrieving Data**

Having now stored 30 seconds of activity on your sensor inputs, now let’s revise the program to read it out from memory and display it. **(**Rename this existing program and save it as Part D**).**

To read from EEPROM memory, we will use the ***value* = EEPROM.read(*address***) statement. Since WRITING and READING are single byte functions, it is important to understand that although we wrote integer variables to the byte memory locations, only the least-significant byte of the two-byte integer was actually written. This worked in this case, since the values in those integers were less than 256. If the values had been larger, it would not have worked. In that case you would have to separately write out each of the two bytes of the two-byte integer and increment the address by 2 for each new write.

During READ we can also cheat and read these bytes into an integer variable because the values were less than 256. To read the four values from memory, first, we have to read in the value of T. Comment out the T=T+1 line, readSensorData(), and writeROMData(), and instead insert a new line readROMData() at the top of the loop.

Now create a new function called readROMData() which is similar to the readSensorData() function that includes the commands to get the value of T from EEPROM:

T = EEPROM.read(addr); addr=addr+1;

And, that similarly, replaces each of the **digitalRead(pin)** statements with a comparable EPROM read:

X = EEPROM.read(addr); addr=addr+1;

Where X is each of the three sensor variables. Notice that after each READ, the address is incremented to point to the next location.

Execute your program and verify that it prints the data saved during the previous 30 second WRITE test. When satisfied, save this modified code as Part D and move on to the next section.

**PART E – Digital Outputs**

Digital outputs are used to turn spacecraft hardware (e.g. sensors, subsystems, torque coils, release mechanisms) on and off. In this lab we will use the outputs to drive our count-down LEDs and to fire a burn resistor to release our antennas.

Any of the 14 I/O pins on the Arduino can be enabled as a digital output. This means a pin can be set to a logical 0 or 1 translating to 0-2 or 3-5 volts output on the pin. This use of the CPU for making simple controllers is so easy and useful not just for spacecraft, but for controlling anything we want to in the lab. To demonstrate this capability we will use some LEDs in a typical circuit.

**NOTE:** Unfortunately the Red LED’s used in the Arduino kit and the Red LED’s we have by the hundreds in the lab are opposite polarity and it is impossible to tell them apart. So if your Red LED does not work try reversing it. Or, when the green LED is working, test the polarity of the Red LED in its place before placing it in its proper place.

Add the LEDs to the CPU board, taking care with the polarity of the flat side and including a transistor with the red one as shown. (The flat side of the LED should face away from the resistor/power source. An output of “1” will source 5 volts through the resistor and LED and turn it on. An LED typically lights brightly with 5 to 10 mA current, so a series resistor is ***always*** required to serve as a current limiter for the given voltage. From the 5 volts, there is a 1.4 volt drop across the LED when lit, so when the LED is on, what is the current in this circuit with the 430 ohm resistor \_\_\_\_\_\_\_\_\_\_\_\_\_\_?

Although each pin can source or sink up to 20 mA, there is a 40 mA total for all pins. Thus, we use only 3 LEDs for this experiment. To output to a pin, you first have to define the pin as an output using the pinMode(*pin,mode*) statement similar to what we did in Part B. Use pins 2-4 for the LEDs.

Inside the setup() function, set each of the pins 2, 3, and 4 to be an OUTPUT using the statements:

pinMode(2,OUTPUT);

pinMode(3,OUTPUT);

pinMode(4,OUTPUT);

To tell the pin when to turn on and off, use the **digitalWrite(*pin,value***) statement, where the value can be HIGH or LOW. In the loop() function, insert the following line to turn on the green LED (pin 2):

 digitalWrite(2,HIGH);

Next, insert additional **digitalWrite()**statements in the **loop()** routine to cause your program to turn on the Yellow LED (pin 3) when I >= 5, and the red LED (pin 4) when I==10 (along with printing “Fire!”). In the next section, this red LED will be rewired to turn on a transistor switch to activate a burn resistor. A thin fishing line around this resistor can be used to deploy your solar panel. For example, to turn the Yellow LED on at its designated time, use the statements below. Notice when the “if” statement needs to execute multiple statements, the block of statements must begin with a { and end with a } as shown below for the Yellow LED:

 if (I >= 5){

 **Serial**.println("Greater than 70!");

 digitalWrite(3,HIGH);

 }

Though these multiple statements may also be written on one line with the braces if desired as written here for the Red LED:

 if (I==10) {**Serial**.println("Fire!"); digitalWrite(4,HIGH);}

Remove any …ROMData() function calls from your loop(). Make sure that T=T+1 line, readSensorData(), and printSensorData() are already included. Test and debug your code.

You can make the Green LED blink on each pass through the loop by delaying another second and then turning the LED off again. But notice this impacts the overall 1 second loop timing by doubling the time. A better way is to only turn the Green LED on when I is even and off when I is odd. You can use a MODULO division by 2 function (J=I%2) to do this since the remainder J is 0 when I is even and is 1 when I is odd. Then use an if (J == 0) statement to set the green LED HIGH. You can either use an “else” statement to turn it off when (J == 1), or simply always turn the Green LED off after the 1 second delay at the end of the loop.

To make sure we do not waste power in the string cutter after 5 seconds have been allowed to burn the string, add a 5 second delay and then output a LOW value on pin 4. Finally, it is a good idea to add a statement to turn off the Yellow LED after the firing with another condition statement:

 if (I >= 15) digitalWrite(3,LOW);

**PART F – String-Cutter Deployer**

Now we add a transistor to amplify the 10 mA Red LED current to several hundred mA to the burn resistor (bulb) as shown here to deploy a solar panel. The Red LED cathode is connected to the transistor Base where it turns on the transistor and counducts through the Emitter to ground. When pin P4 goes high, the current through the 430 ohm resistor and LED turns on the Base of the transistor and causes current to flow from the 5V to ground through the burn resistor (light bulb).

In an actual burn resistor from 5 volts we would use maybe an 8 ohm ¼ watt resistor. How much current flows through this resistor? \_\_\_\_\_\_\_\_ How much power are we burning in the resistor to make this work? How much are we over-rating this component during the burn?\_\_\_\_\_\_\_\_\_%

Since this kind of deployer circuit is so critical to the mission, there is usually a sensing circuit to detect continuity through the resistor to make sure it is ready to go and has not been previously burned out. In the circuit to the right, a 11k resistor is added between the burn resistor and an INPUT pin P6 so that we can see if there is continuity through the burn resistor to the Vin source. If there is continuity through the resistor, then pin P6 will be pulled high by the Vin source. If not, then pin P6 will be pulled low to ground through the second 11k. To read the input condition on pin P6 you can use another digitalRead(6) into a BrState variable. Don’t forget to add a line pinMode(6,INPUT); with the other pin mode statements at the start of the setup() code and then add the following statement to your program as the last lines in the setup() function to indicate the state of the burn circuit.

 BrState = digitalRead(6);

 if (BrState == 1) {**Serial**.print("Burn Circuit is GO!");

 }

 else {Serial.print("Burn Circuit NOT ready!");

 do{

 }while (1==1);

 }

Notice how we have introduced an IF… ELSE construct to choose between the two outcomes and we have inserted an infinite DO…WHILE loop to effectively halt execution until this mission critical circuit is operating correctly.

**Test Fire:** To test your circuit prior to the final flight, we are using a light bulb (similar impedance) in place of the burn resistor. Again test your software to verify that it is fully operational and ready for flight (lamp lights for 5 seconds). You only get one shot in space.

The first person finished will get to hook up to a real burn resistor at the end of the lab after all systems are GO and actually deploy the panels.

**Part G – Mission Flight**

Save your program as your final part-F **test** code. Now, s*ave it again with a new name for your* ***flight*** *code*. Now begin modifying, commenting-out, or writing new code to implement the following mission objectives*. Include lots of comments to help identify the sections of code and what each one does.* Each stage has a loop until its conditions are met. Your spacecraft is attached to a rocket. It will be powered up several seconds prior to *separation.* Separation will not occur unless all temperatures are nominal, the system is in eclipse, and the separation burn circuit is OK. After separation, wait for sun alignment and then count down 10 seconds using the previous LED code to deploy the solar array string-cutter. (Note: Your code should write a log of the input sensors in Memory during the Flight test. However, comment out the actual WRITE statement until all code is debugged and working.)

Mission Requirements:

* Stage 1: Initialize variables and setup serial port
* Pause 2 seconds before entering stage 2
* Check Burn Circuit continuity and Proceed only if burn circuit is OK.
* Stage 2: check status of all inputs every second. Proceed only if Temperatures are not in extreme cold (not 0), the spacecraft is in the sun, (sun = 1) and the pyro circuit is OK (1). Light Green LED and proceed.
* Stage 3: Wait in a loop, flashing Green and checking inputs while waiting for the Separation Switch (push button). When detected, announce SEPARATION and “waiting for Eclipse”. Proceed to stage 4.
* Stage 4: While flashing the green LED, and checking inputs every second, and confirming temperatures and pyro circuit OK, wait for sun to go into eclipse. Then, announce “Eclipse!” and proceed to stage 5.
* Stage 5: Begin 5 sec count-down, flashing yellow LED every second.
* Fire the RED LED and burn resistor for 2 seconds
* Then stop execution (in an infinite “do {}while(1=1)” for example

**FLIGHT!** The instructor will help the first team finished that passes the Flight Readiness Review (FRR) to replace their test light bulb with wires going to a 39 ohm burn resistor on a model Cubesat connected to an external 12v power supply. The change in resistor and voltage is because the small Arduino, powered from the 5v USB cannot actually produce the needed burn current. When the burn resistor fires, a string around the spacecraft will release the solar panels.

Comment your program for understanding. Cut and paste your log data for your report.

**Post Lab Assignment:** Prepare a short memorandum-style report (3 page maximum) detailing your experiments, key learning elements, and implications of this lab to spacecraft designers. In addition, comment on your progress towards accomplishing mission flight objectives. Use portions of your code in your report as necessary to convey understanding. Also, address how you will use what you learned in this lab and apply it to your Capstone project. Include a printout of your entire “flight code” program as an appendix to your report. Within your program, add suitable comments to so that you can remember what you did and come back to this software in the future as needed in this course and in your design courses. Your memo will be graded as follows: progress towards mission objective (20%), presentation of key learning elements (40%), discussion of potential application to Capstone project (40%).

**C&DH Lab Inputs (pre-installed)**

300 ohm long red red brn 3

430 resistors grn blu brn 3

4.7 K resistor red viol red 1

7.5 K resistor viol grn red 1

(or 8.2k in colder weather)

Thermistor 1

Photo resistor 1

Push Button 1

Black tube for shading sensor 1

0.4” jumpers 2

0.5” Jumpers 3

**Digital Outputs:**

11 K resistor brn brn org 3

LEDs Grn,Yel,,Red 3

2N2222 transistor 1

Light Bulb 1

**For Telemetry I**

11K resistors brn brn org 2

10k resistor (blu) brn blk org 1

200k restor (brn) red blk yel 1

4.7 Ohm resistor yel vio gold 1

150 ohm BIG R brn grn brn 1

Serial cable to yel/blk pin cable 1

**For Telemetry II**

GPS unit per group 1

Male/male RS232 adapter 1