LaACES
Student
Ballooning
Course

Introduction to Programming

Computer programming is the process of writing **code**

Code is executable program instructions that are interpreted by computers to perform specific actions

The World of Computer Logic

Most computers operate on **binary logic** – that is, they utilize bits to perform complex operations

A **bit** is a basic unit of information that can only be one of two values – 0 or 1

Multiple bits can be interpreted together to form larger units of information; for example, 8 bits form a **byte**

Example of a byte: 0 0 1 0 1 1 0 1

These series of bits can be used to represent numbers

Number Representations

A **number representation** is a writing notation for numbers. In everyday, we typically use the decimal number representation to count. We count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. We can show larger numbers by adding these digits together; for example, combining 4 and 2 produces 42

However, computers do not use the decimal number system. They operate on **binary logic** – they only use 0’s and 1’s. This is known as the **binary number system**. It follows the same logic as the decimal number system. As such, it is important to understand how numbers can be represented using binary

Binary Number System

The **binary number system** is a base 2 number representation. Each digit in a binary number is a bit

Binary	Decimal Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Table 1: Binary number system

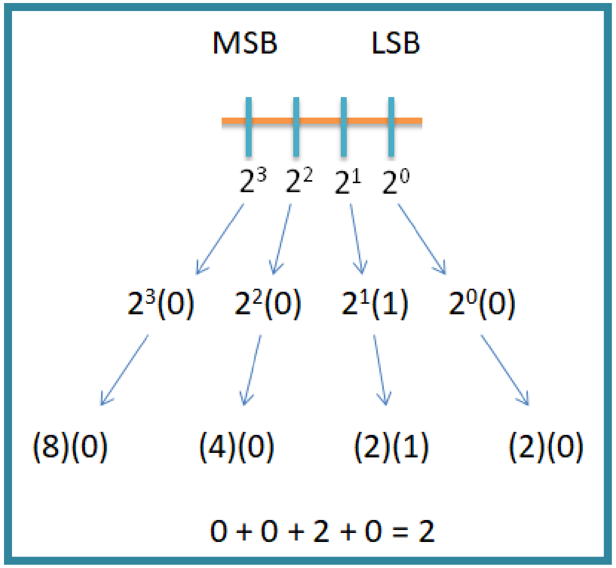
Since all digits are either a “0” or “1”, it can be difficult to interpret what a number is at first glance. It can be useful to compare the number to the decimal number we are more familiar with

Each digit in a binary number can be read as 2^n, where n represents the total number of digits in the binary number

The rightmost digit is known as the least significant bit (LSB) and the leftmost digit is known as the most significant bit (MSB)

Conversion: Binary to Decimal

- To convert from **binary to decimal**, use a base of 2 and powers beginning with 0 at the LSB, counting upwards to the MSB
- This technique is the same in decimal systems; it is similar to how “10” is 10 times larger than 1
- For example, “2” in decimal can be represented in binary as “0010”



MSB LSB

2^3 2^2 2^1 2^0

$2^3(0)$ $2^2(0)$ $2^1(1)$ $2^0(0)$

(8)(0) (4)(0) (2)(1) (2)(0)

0 + 0 + 2 + 0 = 2

Figure 1: Convert binary to decimal

$$\begin{aligned}\text{Example: Convert } (0010)_2 \text{ to decimal} &= 2^3(0) + 2^2(0) + 2^1(1) + 2^0(0)\\ &= (8)(0) + (4)(0) + (2)(1) + (2)(0)\\ &= (2)_{10}\end{aligned}$$

Conversion: Decimal to Binary

- To convert from **decimal to binary,** divide by 2. If dividing by an even number, carry a 0. If dividing by an odd number, carry a 1
- Divide the remaining whole number by 2 and follow the same carry rules; repeat until the remaining whole number is 0
- Build the binary sequence from bottom to top

Divide	Carry
294 ÷ 2 = 147	0
147 ÷ 2 = 73.5	1
73 ÷ 2 = 36.5	1
36 ÷ 2 = 18	0
18 ÷ 2 = 9	0
9 ÷ 2 = 4.5	1
4 ÷ 2 = 2	0
2 ÷ 2 = 1	0
1 ÷ 2 = 0.5	1

Table 2: Convert decimal to binary

Example: Convert $(294)_{10}$ to binary = $(100100110)_2$

Hexadecimal System

- Sometimes, it is useful to use a larger number representation
- **Hexadecimal** (a base 16 representation) is often used because it can represent a byte with a single character, and can be much quicker to read and understand
- This is a base 16, alphanumeric system which means that each digit can have one of sixteen different values (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

Hexadecimal	Binary Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

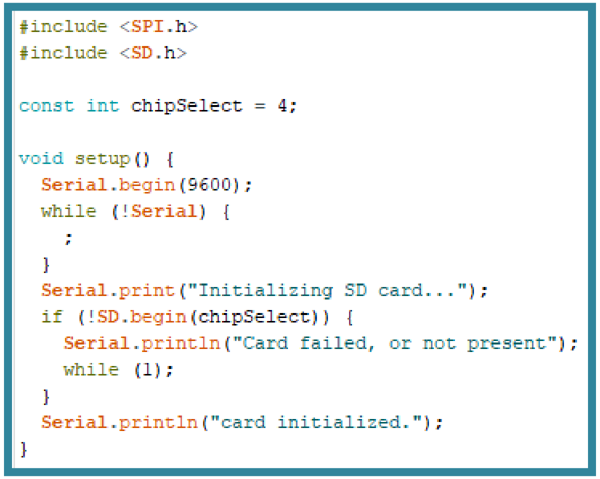
Table 3: Hexadecimal number system

Common Programming Languages

There are many different languages that code can be written in. Each **programming language** varies in **syntax** (structure and format) and **semantics** (meaning)

Common programming languages include:

- C, C#, C++
- Java, JavaScript
- Python



```
#include <SPI.h>
#include <SD.h>

const int chipSelect = 4;

void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ;
  }
  Serial.print("Initializing SD card...");
  if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    while (1);
  }
  Serial.println("card initialized.");
}
```

Figure 2: A example of Arduino Code

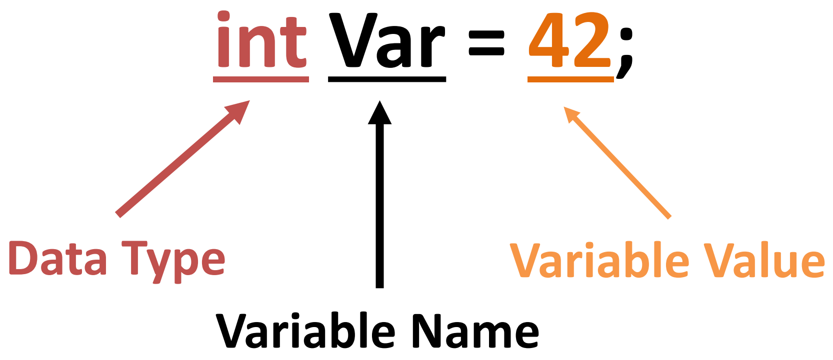
Arduino uses a variation of C++

We will focus on learning Arduino C, the programming language for Arduino hardware

Variables

Variables are **data values** typically saved in memory that can be changed based on code execution

Variables consist of three primary parts – the data type, the variable name, and the variable value



```
int Var = 42;
```

Data Types

There are many different types of variables that store different kinds of data. The type of data stored within a variable depends on the variables **data type**

Data types are typically declared before the name of the variable. They define how you intend to use data and let the computer know how much room to set aside in memory. The amount of memory set aside is measured in **bytes**

Data Type	Keyword	Bytes	Range of Values	Numeric (N) or Alphanumeric (A)
Integer	int	2	-32768 to 32767	N
Character	char	1	0 to 255	N
String	String	varies	varies	A
Boolean	bool	1 bit	0 or 1	N
Floating	float	4	-3.4×10^{38} to 3.4×10^{38}	N
Array	name[]	varies	varies	A or N (depends on array type)

Table 4: Data type specifications

2’s Complement

2’s complement is how negative numbers are stored. The MSB gives the sign of the number. 0 means the number is positive; 1 means the number is negative. To convert from a positive number to a negative number, invert all bits and then add 1

28 = 0001 1100

Invert -> 1110 0011
Add 1 -> 1110 0100

-28 = 1110 0100

Operators

Operators are one of the most common ways of manipulating the value of a variable. They represent a functional operation such as adding or subtracting

Common types of operators include:

- Arithmetic
- Logical
- Conditional
- Bitwise
- Comparison

Arithmetic Operators

- Arithmetic operators are mathematical functions that take two operands, perform a calculation, and provide a result

Operator	Description	Example
+	Adds two operands	A + B = C
-	Subtracts second operand from first	A – B = C
*	Multiplies operands	A * B = C
/	Divides dividend by divisor	B / A = C
%	Modulus operator: Remainder of quotient	B % C = D
++	Increments integer by 1	++A = A + 1
--	Decrements integer by 1	--A = A - 1

Table 5: Arithmetic operators

Logical Operators

- Logical operators use the laws of Boolean logic to compare two conditions and provide one result if true and another if false

Operator	Description	Example
&&	AND – If both operands are nonzero, the condition is true; otherwise, it is false	If A = 1 and B = 0, then A && B = false
||	OR – If either operand is nonzero, the condition is true; otherwise, it is false	If A = 1 and B = 0, then A || B = true
!	NOT – If a condition is true, then !condition is false	If A || B = true, then !(A || B) = false

Table 6: Logical operators

Conditional Operators

- A conditional operator will return one value if a condition is true and another if a condition is false
- Most operators are conditional by nature because they compare entities and then proceed one way if a particular condition is met and another way if it is not

Example:

```
if (expression1) a = a1;            // Test this first
else if (expression2)  a = a2;      // If above was false, test this
else a = a3;                        // If above was also false, do this
```

Bitwise Operators

- Bitwise operators are similar to logical operators, except they compare individual bits instead of the entire operand

Operator	Description	Example
&	Bitwise AND – If both bits are nonzero, the condition is true; otherwise, it is false	If a = 1 and b = 0, then a & b = false
|	Bitwise OR – If either bit is nonzero, the condition is true; otherwise, it is false	If a = 1 and b = 0, then a | b = true
^	Bitwise XOR – If both bits are different, the condition is true; otherwise, false	If a = 1 and b = 1, then a ^ b = false
~	Bitwise NOT – Inverts all bits of a number	If D = 0110, then ~D = 1001

Table 7: Bitwise operators

Comparison Operators

- Comparison operators are used to compare two operands
- These are typically found nested within a function

Operator	Description	Example
==	Equal to	x == y (x is equal to y)
!=	Not equal to	x != y (x is not equal to y)
<	Less than	x < y (x is less than y)
>	Greater than	x > y (x is greater than y)
<=	Less than or equal to	x <= y (x is less than or equal to y)
>=	Greater than or equal to	x > y (x is greater than or equal to y)

Table 8: Comparison operators

Functions

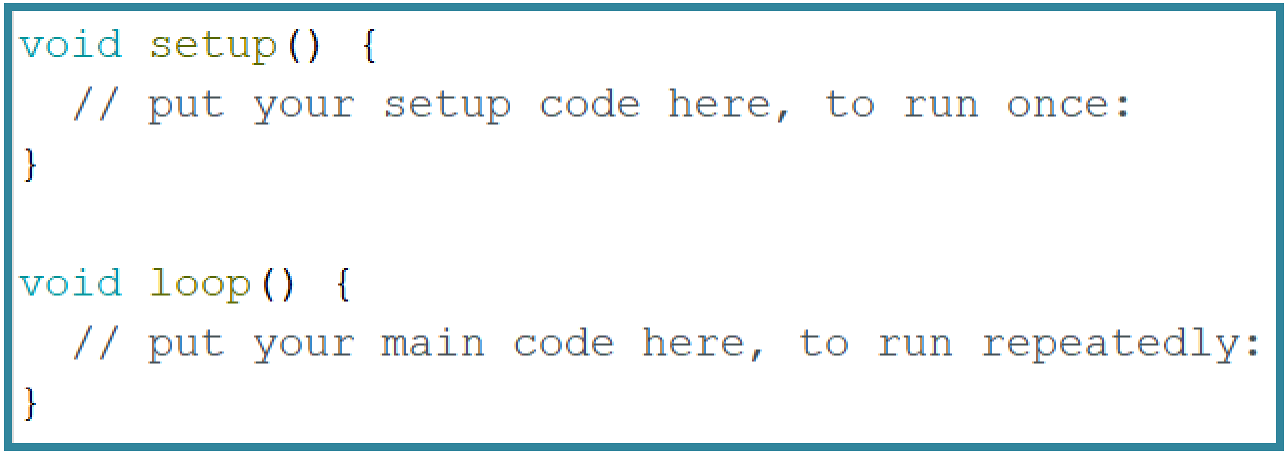
- A function is a code segment in a program that contains instructions the computer will use to perform a task
- To define a function:
  - Specify a data type for the return
  - Provide a unique name followed by a set of parenthesis
  - After the parenthesis, put the instructions that need to be executed inside a set of brackets

```
void setup () {
    <insert instructions>  }
```

Figure 5: Structure of a void function

Void

- **Void** is a special data type used for declaring a function that is not expected to return any information
- Arduino uses two void functions to get you started; the main setup runs one time when the program begins, followed by a loop that runs continuously thereafter



```
void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Figure 6: Image of the Arduino start screen

Conditional Statements: If/Else

- An if statement proceeds one way if a condition is met and another way if it is not met

```
void setup() {
    Serial.begin(9600);   }
void loop() {
    int A = 15;
    int B = 10;
    if (A >= B) {
        Serial.println (A – B);
    }
    else if ( (A < B) && (B !=  0)  )  {
        Serial.println (B + A);
    }
```

Figure 7: In this example, if A is greater than or equal to B, then A – B will print. Otherwise, if B is not zero then B + A will print

```
void setup() {
    Serial.begin(9600);   }
void loop() {
    int A = 5;
    int B = 20;
    if (B >= A) {
        Serial.println (B - A);
    }
    else if ( (A < B) && (B !=  0)  )  {
        Serial.println (B + A);
    }
```

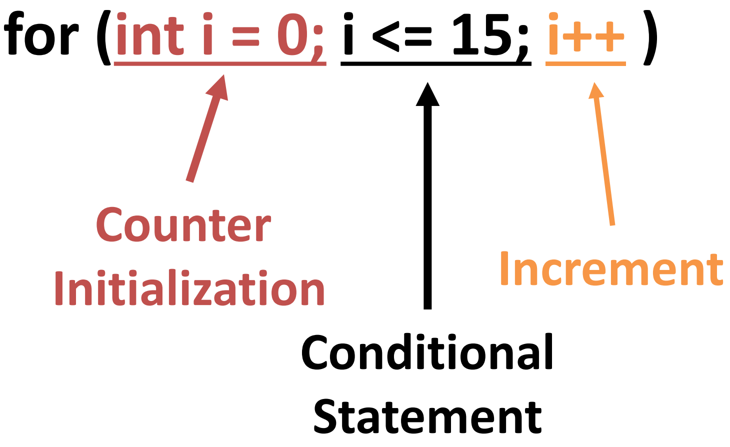
Figure 8: In this example, if B is greater than or equal to A, then B – A will print. Otherwise, if B is not zero then B + A will print

Loops

- A **loop** is useful when repetitive operations are being performed because the instructions will repeat until a particular condition is met
  - Some loop commands pretest, which means they test for a condition at the beginning of the loop
  - Other loop commands posttest, which means they test for a condition at the end of the loop

For Loops

- A **for loop** executes repeatedly and increments a counter variable until the conditional statement is no longer true (pretest condition)



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

void loop() {
    for (int i = 0; i <= 15; i++)  {
        Serial.println(i);
    }
}
```

Output → 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 9: A for loop that counts from 0 to 15. The variable i starts at 0 and increments every loop until the condition stated in the loop is no longer valid.

While Loops

- A **while loop** will only run when the conditional statement is true (pretest condition)

while (carrier < 0) { Serial.print (carrier++) }

Conditional Statement **Loop Execution**

```
int carrier = 0;

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

void loop() {
    while (carrier < 20) {
        Serial.println(carrier++);
}
```

Figure 10: A while loop that counts from 0 to 19

Do/While Loops

- A **do/while loop** only checks for a condition after some other action has occurred (posttest condition)

do { Serial.print (carrier) } while (x < 10)

Loop Execution

Conditional Statement

```
int x = 0;
void setup()  Serial.begin(9600);
void loop() {

    do {
        Serial.print(“Waiting…”);
        Serial.println(x++);
    } while (x < 10) ;

    Serial.println(“done”);
    while(1) {};
}
```

Figure 11: A do/while loop that counts from 0 to 9

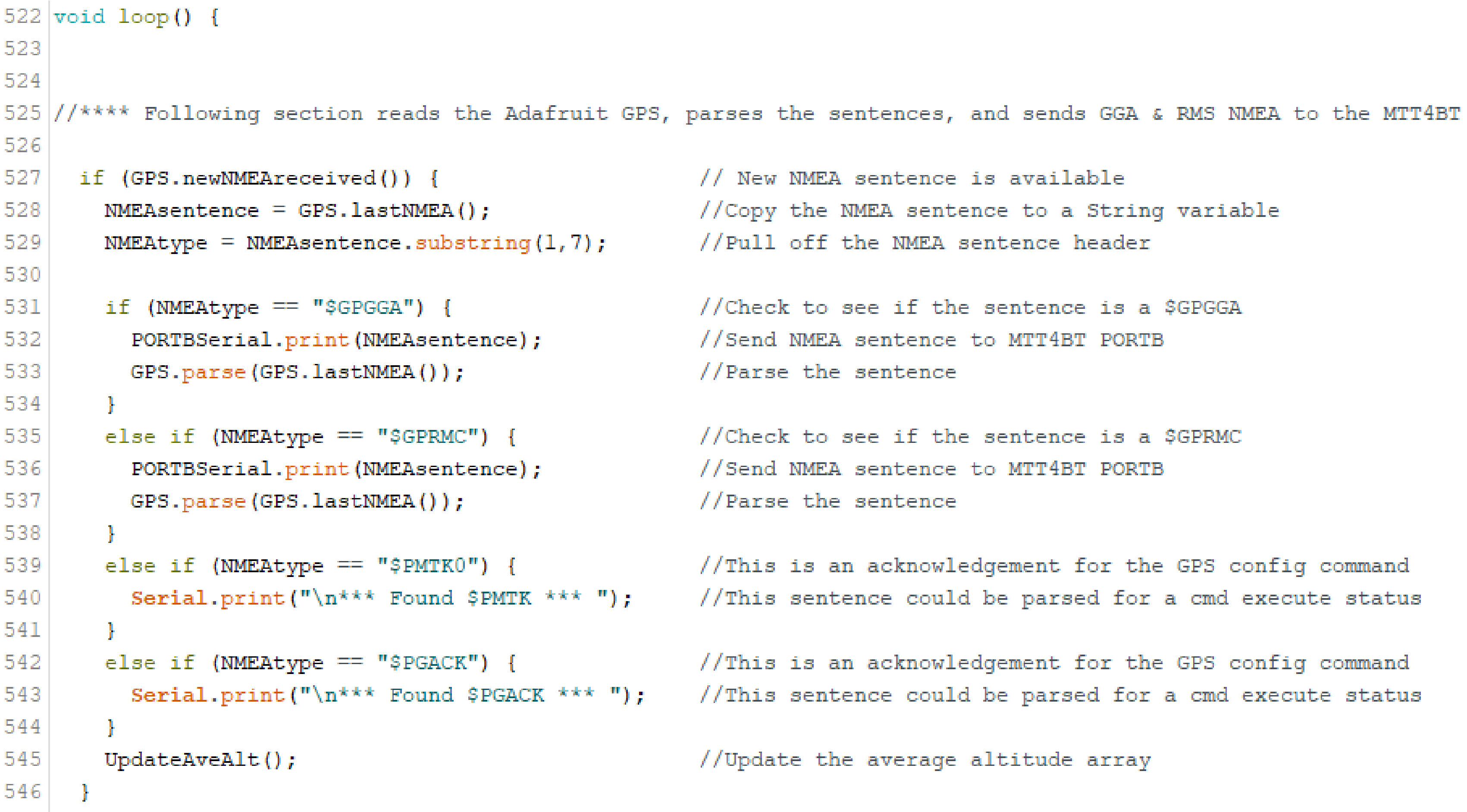
Leaving Comments

- If you can fit your comment on one line, then simply type two backslashes followed by your text
- If you need more room, then use a backslash and asterisk combination to comment over multiple lines
- You can highlight a block of information and press ctrl + backslash to comment the entire block

```
// Leave a one-line comment like this
```

```
/* Use as many lines as needed in order to provide
enough information for someone else to understand
your code */
```

Good Comments



```
void loop() {


//**** Following section reads the Adafruit GPS, parses the sentences, and sends GGA & RMS NMEA to the MTT4BT

  if (GPS.newNMEAreceived()) {                    // New NMEA sentence is available
    NMEAsentence = GPS.lastNMEA();                //Copy the NMEA sentence to a String variable
    NMEAtype = NMEAsentence.substring(1,7);       //Pull off the NMEA sentence header

    if (NMEAtype == "$GPGGA") {                   //Check to see if the sentence is a $GPGGA
      PORTBSerial.print(NMEAsentence);            //Send NMEA sentence to MTT4BT PORTB
      GPS.parse(GPS.lastNMEA());                  //Parse the sentence
    }
    else if (NMEAtype == "$GPRMC") {              //Check to see if the sentence is a $GPRMC
      PORTBSerial.print(NMEAsentence);            //Send NMEA sentence to MTT4BT PORTB
      GPS.parse(GPS.lastNMEA());                  //Parse the sentence
    }
    else if (NMEAtype == "$PMTK0") {              //This is an acknowledgement for the GPS config command
      Serial.print("\n*** Found $PMTK *** ");     //This sentence could be parsed for a cmd execute status
    }
    else if (NMEAtype == "$PGACK") {              //This is an acknowledgement for the GPS config command
      Serial.print("\n*** Found $PGACK *** ");    //This sentence could be parsed for a cmd execute status
    }
    UpdateAveAlt();                               //Update the average altitude array
  }
```

Figure 12: This an example of good commenting. Notice the comments explaining each step and the use of white space to help the user understand the code.

Bad Comments

```
String GetCUTResponse() {
  char  temp[35];
  char  X = ' ';
  int   templen = 0;
  unsigned long StartTime = millis();
  unsigned long TimeOut = 100000;         // Time out in microseconds
  unsigned  long  DeltaTime = 0;          // Elapsed time in micros since start
  boolean Beg = false;
  boolean End = false;

  StartTime  = micros();
  DeltaTime  = 0;
  for (int i = 0; i < 35; i++)  temp[i] = '\0';
  while(!End && (DeltaTime < TimeOut))  {
    if  (XBee.available())  {
      X = XBee.read();
      switch(X) {
        case  '$':
          Beg  = true;
          break;
        case  '#':
          End    = true;
          break;
        default:
          if (Beg && isPrintable(X)) temp[templen++] = X;
          break; }}
    DeltaTime = micros()  - StartTime; }
  if (End) return(String(temp));
  else return("TimeOut");}
```

Figure 13: This an example of bad commenting. The lack of comments make the code difficult for a user to follow and understand the purpose of this function.

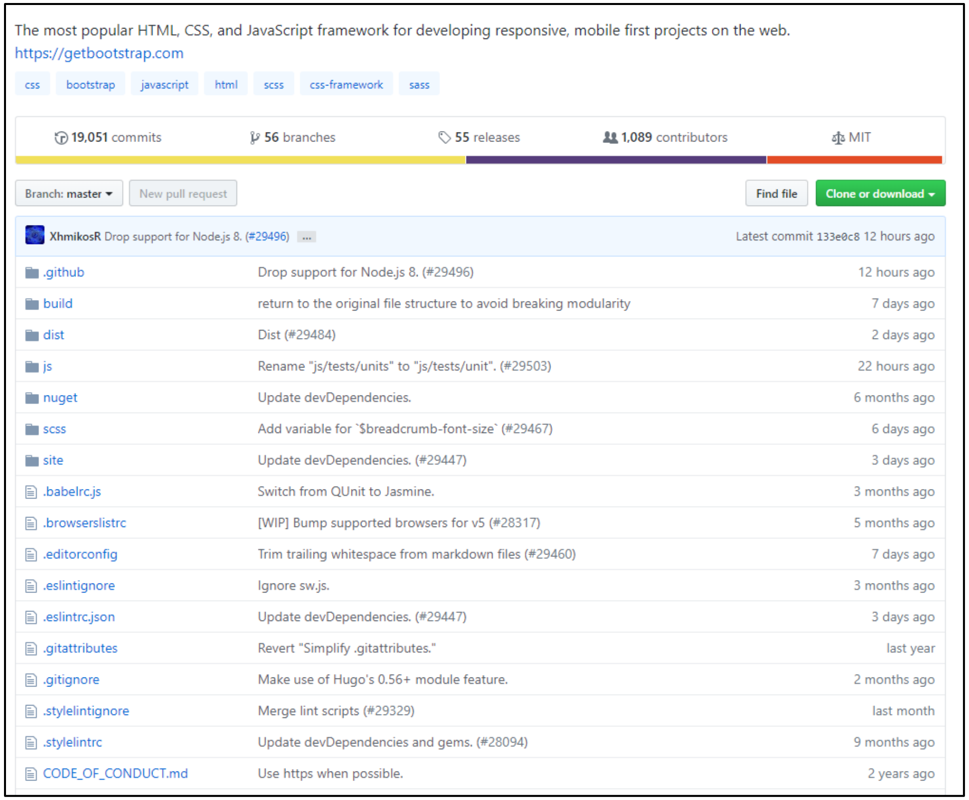
Version Control

- While developing software, it is important to track the changes made within your code. This is accomplished by version control.
- **Version Control** is the practice of managing and recording changes to software or other frequently changed documents
- Without version control, changes are more frequently lost, miscommunicated, or duplicated.
- Version control helps facilitate effective communication in development teams.

Version Control Systems

In large software projects, version control is often handled by a version control system developed by a third party. A **version control system** (VCS) is a software program that creates and tracks multiple versions of a codebase on a server.

Some examples of VCS software include GitHub, Subversion, and BitKeeper


The most popular HTML, CSS, and JavaScript framework for developing responsive, mobile first projects on the web.
https://getbootstrap.com

XhmikosR Drop support for Node.js 8. (#29496) … Latest commit 133e0c8 12 hours ago

.github	Drop support for Node.js 8. (#29496)	12 hours ago
build	return to the original file structure to avoid breaking modularity	7 days ago
dist	Dist (#29484)	2 days ago
js	Rename "js/tests/units" to "js/tests/unit". (#29503)	22 hours ago
nuget	Update devDependencies.	6 months ago
scss	Add variable for `$breadcrumb-font-size` (#29467)	6 days ago
site	Update devDependencies. (#29447)	3 days ago
.babelrc.js	Switch from QUnit to Jasmine.	3 months ago
.browserslistrc	[WIP] Bump supported browsers for v5 (#28317)	5 months ago
.editorconfig	Trim trailing whitespace from markdown files (#29460)	7 days ago
.eslintignore	Ignore sw.js.	3 months ago
.eslintrc.json	Update devDependencies. (#29447)	3 days ago
.gitattributes	Revert "Simplify .gitattributes."	last year
.gitignore	Make use of Hugo's 0.56+ module feature.	2 months ago
.stylelintignore	Merge lint scripts (#29329)	last month
.stylelintrc	Update devDependencies and gems. (#28094)	9 months ago
CODE_OF_CONDUCT.md	Use https when possible.	2 years ago

Figure 14: Example of a GitHub repository for a large software program

Working Copies and Branches

In development, it is often useful for multiple programmers to edit software at the same time.

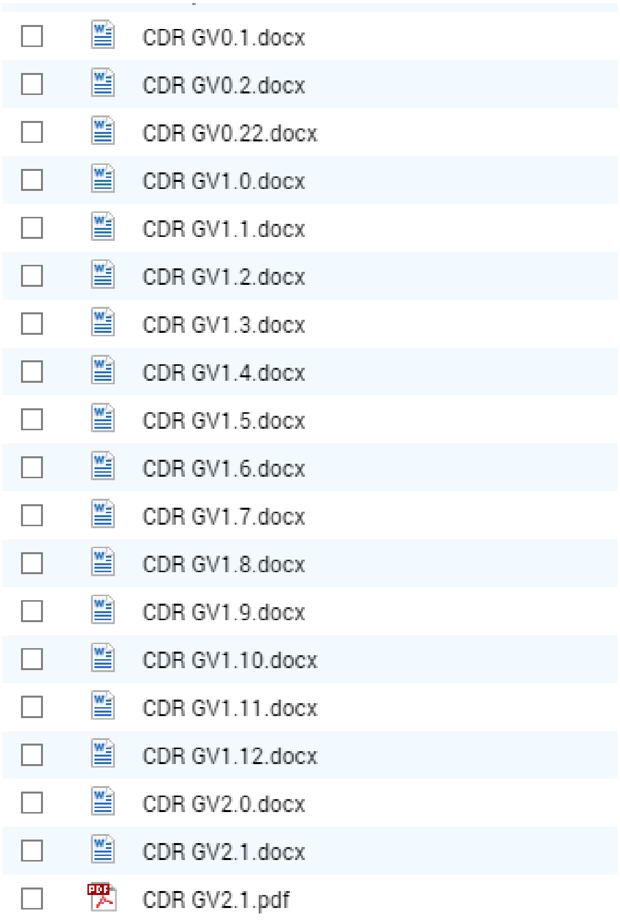
The initial copy of the software that the programmers begin with is called the **working copy** or **baseline**. The edited software that each programmer creates is known as a **branch**. Multiple branches may exist at the same time. A branch may become the working copy when the team agrees to shift to the new branch for further development work.

When a programmer is finished with his or her changes to the branch, they may compile a **change list** which summarizes all changes made to the software.

File Tags

In version control, a **file tag** is a series of numbers or letters that designate the version of an existing document or software. File tags often include a version number that in incremented with any changes or the date the file was modified.

The system for updating the file tag is defined during project creation and is followed throughout the lifetime of the project.



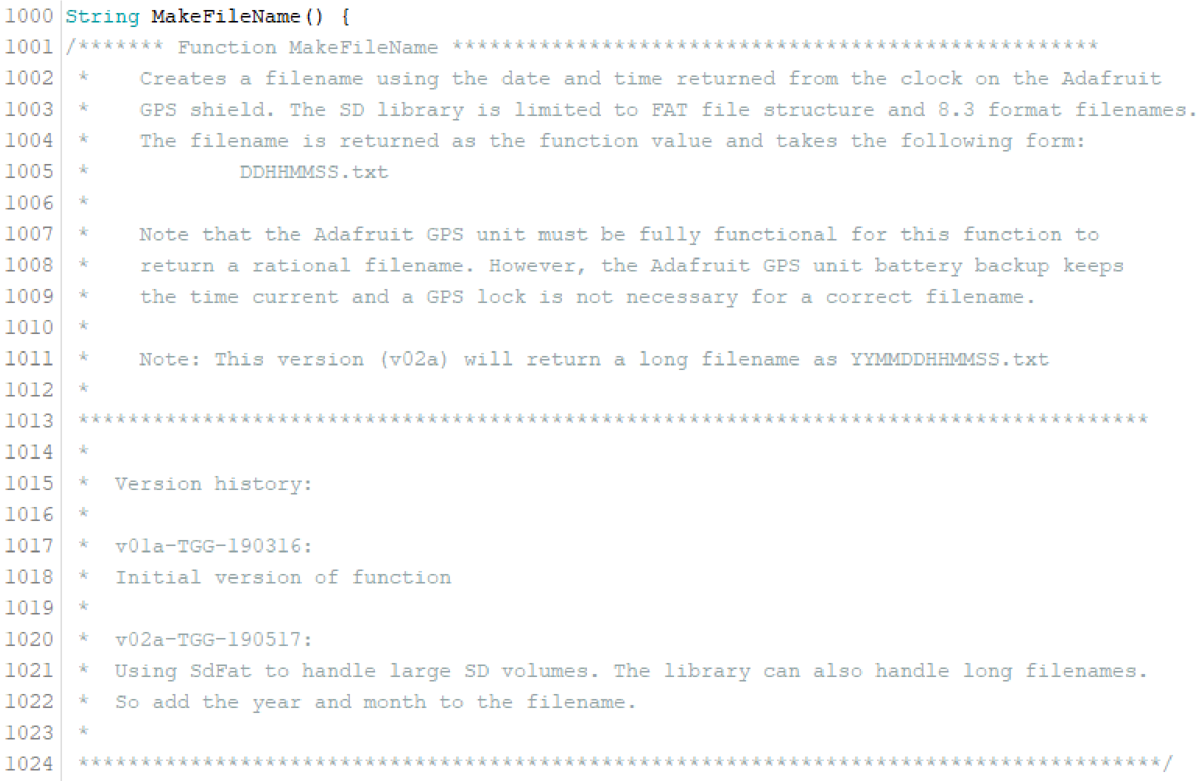
☐ CDR GV0.1.docx
☐ CDR GV0.2.docx
☐ CDR GV0.22.docx
☐ CDR GV1.0.docx
☐ CDR GV1.1.docx
☐ CDR GV1.2.docx
☐ CDR GV1.3.docx
☐ CDR GV1.4.docx
☐ CDR GV1.5.docx
☐ CDR GV1.6.docx
☐ CDR GV1.7.docx
☐ CDR GV1.8.docx
☐ CDR GV1.9.docx
☐ CDR GV1.10.docx
☐ CDR GV1.11.docx
☐ CDR GV1.12.docx
☐ CDR GV2.0.docx
☐ CDR GV2.1.docx
☐ CDR GV2.1.pdf

Figure 15: Example of a file tag system for iterations of a document

LaACES
Student
Ballooning
Course

Function Version History

- It is useful to track changes of a function. This can be done by implementing a change log inside the code



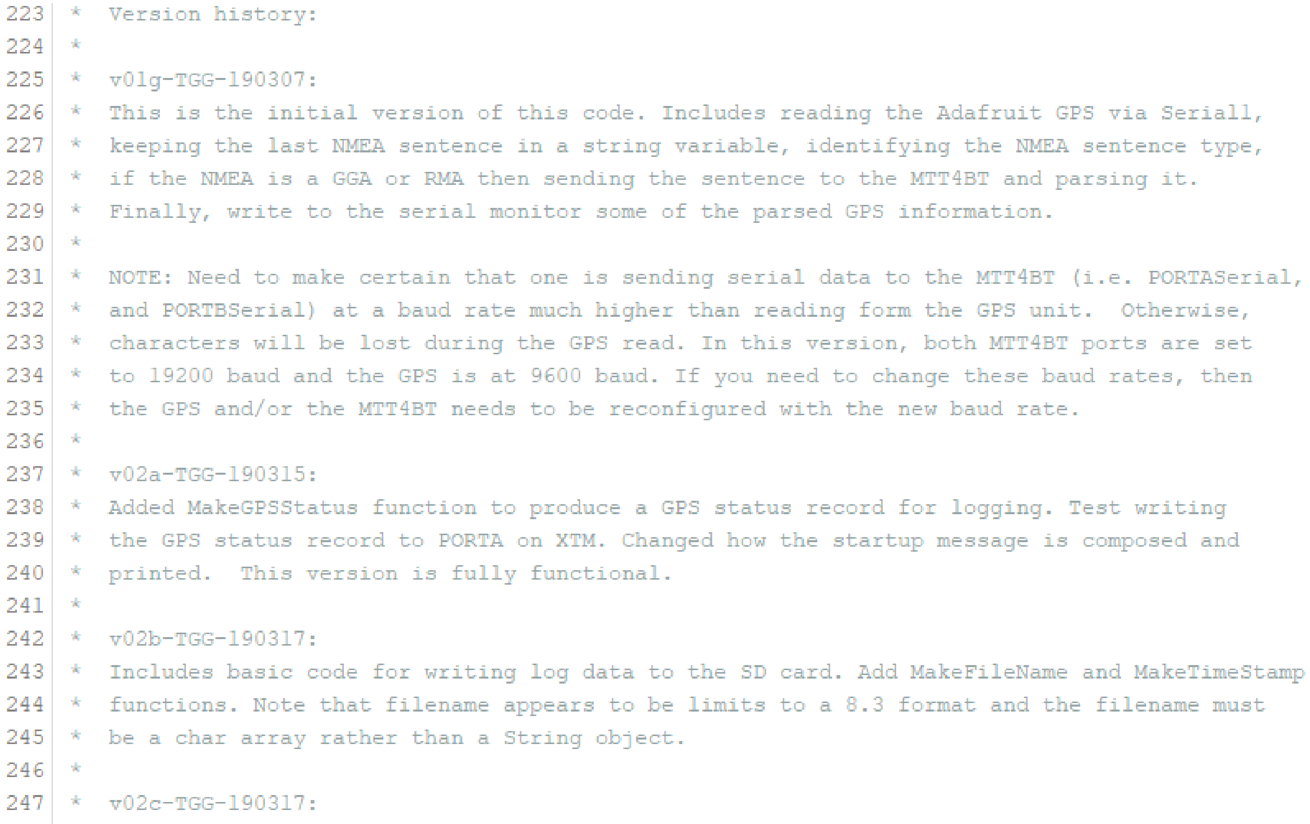
```
String MakeFileName() {
/******* Function MakeFileName *****************************************************
 *    Creates a filename using the date and time returned from the clock on the Adafruit
 *    GPS shield. The SD library is limited to FAT file structure and 8.3 format filenames.
 *    The filename is returned as the function value and takes the following form:
 *            DDHHMMSS.txt
 *
 *    Note that the Adafruit GPS unit must be fully functional for this function to
 *    return a rational filename. However, the Adafruit GPS unit battery backup keeps
 *    the time current and a GPS lock is not necessary for a correct filename.
 *
 *    Note: This version (v02a) will return a long filename as YYMMDDHHMMSS.txt
 *
 ****************************************************************************************
 *
 *  Version history:
 *
 *  v01a-TGG-190316:
 *  Initial version of function
 *
 *  v02a-TGG-190517:
 *  Using SdFat to handle large SD volumes. The library can also handle long filenames.
 *  So add the year and month to the filename.
 *
 ****************************************************************************************/
```

Figure 16: Example of a change log for a function. After the description of a function, include version history. This will tell a user when any changes were made and what those changes were.

LaACES Student Ballooning Course

Sketch Version History

- Like functions, sketch changes should be documented. This should be done in the beginning of the sketch



```
 *  Version history:
 *
 *  v01g-TGG-190307:
 *  This is the initial version of this code. Includes reading the Adafruit GPS via Seriall,
 *  keeping the last NMEA sentence in a string variable, identifying the NMEA sentence type,
 *  if the NMEA is a GGA or RMA then sending the sentence to the MTT4BT and parsing it.
 *  Finally, write to the serial monitor some of the parsed GPS information.
 *
 *  NOTE: Need to make certain that one is sending serial data to the MTT4BT (i.e. PORTASerial,
 *  and PORTBSerial) at a baud rate much higher than reading form the GPS unit.  Otherwise,
 *  characters will be lost during the GPS read. In this version, both MTT4BT ports are set
 *  to 19200 baud and the GPS is at 9600 baud. If you need to change these baud rates, then
 *  the GPS and/or the MTT4BT needs to be reconfigured with the new baud rate.
 *
 *  v02a-TGG-190315:
 *  Added MakeGPSStatus function to produce a GPS status record for logging. Test writing
 *  the GPS status record to PORTA on XTM. Changed how the startup message is composed and
 *  printed.  This version is fully functional.
 *
 *  v02b-TGG-190317:
 *  Includes basic code for writing log data to the SD card. Add MakeFileName and MakeTimeStamp
 *  functions. Note that filename appears to be limits to a 8.3 format and the filename must
 *  be a char array rather than a String object.
 *
 *  v02c-TGG-190317:
```

Figure 17: Example of version history for a sketch. Every time the sketch is worked on, a new section is added describing the changes that were made.

Troubleshooting Your Code

- Check syntax
- Check punctuation: semicolons, brackets and parenthesis must be placed correctly
- Ensure correct placement of conditional statements and loops
- Use correct data types
- Make sure global and local variables are accessible to the appropriate functions

Good Bookkeeping

- There is typically more than one way of writing a program to accomplish a particular task; as such, programmers tend to have their own styles

- It is good practice to write your code in a manner that is easy for you to navigate through and clear enough for others to understand

- Practice taking advantage of whitespace, utilize control characters, identify variables and functions using descriptive names, and always comment your code
- Work to establish good habits while you are learning

References

- For a list of Arduino keywords, visit https://www.arduino.cc/reference/en/