



8Bit Project: Calculator

With 8BitCADE

Compatible with:

8BitCADE^{XL}

AND

8BitCADE

Contents

8BitCADE Project: Calculator.....	4
File Breakdown.....	4
Controller.h.....	4
Controller.h Code.....	5
Controller.....	6
Controller Method Creation Code.....	6
Button Debounce Code.....	7
Sprites.h.....	9
Calculator Mechanism.....	10
Class Math Code.....	11
Void Setup Code.....	12
Calculator Core Loop.....	13
Laying out the Framework for X Movement.....	15
Collision.....	15
Adding a X Boundary/Collision.....	16
Adding Sprites.....	16
Moving Sprites along the X axis.....	17
Adding Y Axis Movement.....	18
Animating the Buttons.....	21
Final Code.....	22
Calculator.ino.....	22
Controller.h.....	26
Controller.ino.....	27
Sprites.h.....	27

Written by the 8BitCADE Team

Support@8bitcade.com

Version 3

© 2020 8BitCADE Limited

CC BY-NC-SA

Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Note that:

This text showcases a task/challenge that you should attempt – all levels of coders should try and attempt these without seeing the answer.

This text showcases something that you should be doing, regardless of your coding ability. This is usually ensuring that your program is exactly like the one presented.

For beginners we recommend ensuring your software is exactly like the one we present. Your main focus should be on understanding the coding functions, getting use to the coding syntax and understanding why we use specific functions.

For intermediate to advanced coders, we recommend that you do this tutorial first, then try writing your own program with the challenges used as roadmaps/guidelines.

The way this booklet is written is:

1. The brief of what we want the code in this section to achieve.
2. Task: Can you code it by yourself? Here are the functions and what the functions mean
3. Code Explanation
4. Final Code, copy this to get a program just like the one we present!

This allows for the beginners to get a grasp on what each function and coding statement means and dip the deep end by coding some sections by themselves. For intermediate coders it gives a challenge and for advanced coders it allows the code to be planned out ready for you to write it up using your own logic – then all levels can check it with the presented code and adjust it accordingly.

Have fun, if you have any errors with the code. Check your code with the final code, in the final pages of the booklet.

Bring out the learner in you – with 8BitCADE!

-8BitCADE Team

8BitCADE Project: Calculator

In this tutorial, we are going to be learning how to use Arduino to program a calculator for our 8BitCADE/XL.

To fully understand this tutorial, you need to be able to understand basic Arduino syntax and Arduino classes. We advise that the following tutorials are completed before starting this project:

- Arduino Basic's: Classes
- Arduino Basic's: Library & Board Setup

File Breakdown



Calculator: The main Arduino file that contains the setup/loop. (when you open Arduino, save the blank document and name its "Calculator" this will be your main file (the default extension is .ino hence this would be your Calculator.ino file))

Controller.h: Defines the Class Controller, that will be used to read and process the controls

Controller: Creates all of the class methods of the class Controller

Sprites.h: Used to store all of the sprites used in this program.

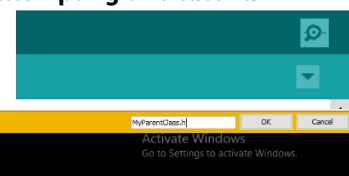
Controller.h

The controller class deals with the buttons controls of the 8BitCADE and has methods inside the class to deal with button debounce. In this header file, we define all attributes and methods for the class. We use header and .ino files to help organize our code. Header files are for defining attributes of a class or library and .ino files are for running code or in our case, writing what does inside the actual methods we define.

Your task is to create a new header file called "Controller.h"

A snippet from the "Classes 101 Booklet", be sure to look at that before attempting this tutorial.

To create a new file in Arduino, you need to press the toggle menu on the right of the files bar (under serial monitor) and click "NewTab"



Here we write the file name and then the file extension: .ino is for Arduino based programs and will be used for the file that will contain all of the class

When you create this file, it's important to add the file extension (.h) so the



program knows it isn't a default .ino file but rather a header file.

Explanation: Upon file creation, you'll notice that Arduino has not produced any void loops or setups. This is as there can only be one of each in the program. Also, as this is a header file, we will only be defining variables and classes.

Note that all attributes and methods need to be defined before we can use them, if you reference a variable that hasn't been created, you will get an error. Below are some helpful definitions before you copy the final code:

point [Variable name] To create a structure with int X and int Y. We defined the structure in the main section of the code.

bool data type is for true and false (or 1/0)

long data types are like integers but have a larger range.

To create a constructor method, simple write **[nameOfClass]();**

```
5 typedef struct point {
6     int x;
7     int y;
8 };
```

A point will be defined in the main file and is a structure. A structure is a defined datatype that holds groups of data, aka int x and int y in this case. We can access these through using point.x or point.y and it will return the integer value of x or y

Controller.h Code

```
1 class Controller
2 {
3     public:
4         //CONSTRUCTOR
5         Controller();
6         //Create from the point structure a position variable. This will be used to move the cursor
7         point position;
8         //Define the method UPDATE used to update the controls every loop
9         void update();
10
11        //BOOLEAN VARIABLES THAT STORES WHETHER BUTTON PRESSED IS A/B
12        bool AButtonPressed;
13        bool BButtonPressed;
14
15        //BOOLEAN VARIABLE THAT STORES WHETHER THE PREVIOUS BUTTON WAS PRESSED
16        bool previousUPButtonPressed;
17        bool previousDOWNButtonPressed;
18        bool previousLEFTButtonPressed;
19        bool previousRIGHTButtonPressed;
20        bool previousAButtonPressed;
21        bool previousBButtonPressed;
22
23        //BOOLEAN VARIABLES THAT CHECKS IF THE BUTTON PRESS IS UPDATED (A/B PRESSED)
24        bool AButtonWasPressed;
25        bool BButtonWasPressed;
26
27
28        //BOOLEAN VARIABLES THAT CHECKS IF THE BUTTON CAN BE PRESSED. USED FOR COLLISION/RESTRICTION
29        bool UPButtonCanBePress = 1;
30        bool DOWNButtonCanBePress = 1;
31        bool LEFTButtonCanBePress = 1;
32        bool RIGHTButtonCanBePress = 1;
33
34        //HOLDS POSITION OF 'CURSOR' OR CHARACTER
35        int pos;
36
37        long debounceDelay = 75; //TIME WAITED UNTILL NEXT BUTTON IS INPUTTED
38        long currenttime = 0; //CURRENT TIME
39        long lastHoldTime = 100; //TIME, IN MILLIS, OF LAST BUTTON PRESS
40
41 };
```

Here we can see, on line 5, we need a constructor, we can leave this blank if we don't have any values to pass through upon creating an object. To do this, use the [classname]() and leave the parameters blank. See more about constructors in the "Classes 101" booklet.

Line 7: **Point position;** defines an attribute from the structure point which we formed in the main Calculator file.

Also note that when defining any variable holding a millis value, ensure you use a data type that can deal with large amounts of data – as recording the time can amount to a lot of data quickly if not updated. Here we used long as it has a large plus and minus range. (see lines 37 to 39).

Please copy this code to ensure you have the correct program – this is important as it defines all the variables we will use. A typo here is usually a major culprit for errors! Copy the code above.

Controller

The controller file deals with the methods for the controller class. It's important when writing in this file to call files using the "outside class" calling method, as discussed in the "classes 101" booklet:

"void [Classname] :: [Method Name]()"

This file mainly focuses on dealing with button debounce. *Your task is to create a new tab file, name it "Controller" – it will default to .ino*

Before we start implementing debounce algorithms, we have to first create the constructor method, we can utilize this to set the controller.position.x and y position values to 0 (or if we needed the player to start in the middle, we could, therefore, set these values to the centre x and centre y position)

Note that when referencing class attributes in a class, we do not need to write "classname.attribute" we can just write "attribute". It's only outside of classes where we need to specify the classname.

The way the controls will work is that we will take in button inputs, and alter the position.x and position.y values accordingly. Remember that "position" is a structure that contains two integer variables "x" and "y"

We also use a Serial.print to check that the object was created correctly.

Your task is to write the constructor method for the class Controller() and create an empty function called "update". Be sure to use "outside class" method declaration. The below functions might help:

position.x will access the x position of the controller and replacing the x with a y will access the y position for the controller.

Serial.print("Hello World"); Will print to the console the string "Hello World"

Void [Class name] :: [function name]() { [code] } will create a function for the class specified.

Controller Method Creation Code

On the right, is the code you should type in.

What is debounce in a button?

Debounce, in buttons, is when the button input is read multiple times in a short amount of time. Without debouncing, the input could be up to 4 times and therefore would alter the position value too much!

```
1 Controller::Controller()
2 {
3     Serial.print("BUTTON TEST BUTTON TEST");
4     position.x = 0;
5     position.y = 0;
6 }
7
8 void Controller::update()
9 {
10
11 }
```

Your task is to write a simple algorithm that takes one button (`UP_BUTTON`) and ensures that the button is registered as pressed with a debouncing algorithm. The below functions will come in handy:

`Aboy.pressed(UP_BUTTON)` will return either `true` or `false` depending on if the up button is pressed or not (can put any button in `LEFT_BUTTON` etc)

`Millis()` will return the time in milliseconds since the program started running

`currentTime = Millis();` Will capture that time

Once the button is registered as press, we update the position, use `position.y` to access the global `y` position of the class controller. For up we would add 1, for down we would -1.

Button Debounce Code

Explanation: To stop this from happening, we use the below algorithm (repeated for each button) to combat this:

```
27 // A and B Buttons
28 if (aboy.pressed(A_BUTTON) and previousAButtonPressed == 1 and (millis() - lastHoldTime) > 250) {
29     lastHoldTime = millis();
30     AButtonPressed = true;
31 }
```

Here, if the button is currently being pressed (`aboy.pressed(UP_BUTTON)`) and it was previously pressed (`previousUPButtonPressed` would be 1 if it is turned on and 0 if it is off) and the time from the last recorded button press is greater than 150 mili seconds, and the button can be pressed, then change the position accordingly, in this case, add one to the position structure `y`. (structures are accessed using [DOT] `StructureName.variableName`).

`aboy.pressed(UP_BUTTON)` = gets the current button status of the "UP_BUTTON".

`PreviousUPButtonPressed` = Is the status of the button UP, in the last cycle of code (aka the last reading)

```
26 if (aboy.pressed(A_BUTTON) and previousAButtonPressed == 1 and (millis() - A_lastHoldTime) > 250) {
27     A_lastHoldTime = millis();
28     AButtonPressed = true;
29 }
30 else {
31     AButtonPressed = false;
32 }
33 if ((aboy.pressed(B_BUTTON) == 1 and previousBButtonPressed == 1 and (millis() - B_lastHoldTime) > 250) ) {
34     B_lastHoldTime = millis();
35     BButtonPressed = true;
36 }
37 else {
38     BButtonPressed = false;
39 }
```

`(millis() - A_lastHoldTime) > 150` = This allows us to calculate how much time has passed since we last took a reading. We can see that as `A_lastHoldTime`, in the if statement is equal to `millis()`; therefore we record the time it was pressed. If it is less than 150 then we know it is the button bouncing and we know not to record the button.

Note that we have to use separate lastholdtime variables to allow both buttons to be read at the same time, if we used the same variable, when one is read, the other button cannot be read until the time limit is over. Hence using `A_lastholdtime` and `B_lastholdtime`.

While we do not use the A and B buttons, it's important to note the above code as this "Controller" class is used for a lot of 8BitCADE projects.

aboy.pressed(A_BUTTON) = Checks if the button is currently pressed

previousAButtonPressed = Checks the status of the button, in the last cycle of code (aka the last reading)

(millis() - A_lastHoldTime) > 250) = only takes the reading if there has been more then 250 milliseconds from now to the last time we took a button reading.

UPButtonCanBePress is our variable used to deal with collisions and boundaries. We can turn on and off each button allowing us to have full control over where the sprite is allowed to go. I'll go over the function that controls this aspect after this, but for now know that when the player reaches a boundary, we can turn these on and off to stop movement in one direction and to allow movement in the opposite direction.

```
42 //Set all previous button pressed variables
43 previousUPButtonPressed = aboy.pressed(UP_BUTTON);
44 previousDOWNButtonPressed = aboy.pressed(DOWN_BUTTON);
45 previousLEFTButtonPressed = aboy.pressed(LEFT_BUTTON);
46 previousRIGHTButtonPressed = aboy.pressed(RIGHT_BUTTON);
47 previousAButtonPressed = aboy.pressed(A_BUTTON);
48 previousBButtonPressed = aboy.pressed(B_BUTTON);
```

Here we set the previous button pressed variables to the current button press status for the next loop. This is part of the above algorithm. See the "full code" below to check if your file is correct.

The full code for the methods of the class in file "Controller" is below:

```
1 Controller::Controller()
2 {
3   Serial.print("BUTTON TEST BUTTON TEST");
4   position.x = 0;
5   position.y = 0;
6 }
7
8 void Controller::update()
9 {
10  if (aboy.pressed(UP_BUTTON) and previousUPButtonPressed == 1 and (millis() - lastHoldTime) > 150 and UPButtonCanBePress) {
11    lastHoldTime = millis();
12    position.y += 1;
13  }
14  if ((aboy.pressed(DOWN_BUTTON) == 1 and previousDOWNButtonPressed == 1 and (millis() - lastHoldTime) > 150) and DOWNButtonCanBePress) {
15    lastHoldTime = millis();
16    position.y -= 1;
17  }
18  if (aboy.pressed(LEFT_BUTTON) and previousLEFTButtonPressed == 1 and (millis() - lastHoldTime) > 150 and LEFTButtonCanBePress) {
19    lastHoldTime = millis();
20    position.x -= 1;
21  }
22  if ((aboy.pressed(RIGHT_BUTTON) == 1 and previousRIGHTButtonPressed == 1 and (millis() - lastHoldTime) > 150) and RIGHTButtonCanBePress) {
23    lastHoldTime = millis();
24    position.x += 1;
25  }
26  if (aboy.pressed(A_BUTTON) and previousAButtonPressed == 1 and (millis() - A_lastHoldTime) > 250) {
27    A_lastHoldTime = millis();
28    AButtonPressed = true;
29  }
30  else {
31    AButtonPressed = false;
32  }
33  if ((aboy.pressed(B_BUTTON) == 1 and previousBButtonPressed == 1 and (millis() - B_lastHoldTime) > 250) ) {
34    B_lastHoldTime = millis();
35    BButtonPressed = true;
36  }
37  else {
38    BButtonPressed = false;
39  }
40  previousUPButtonPressed = aboy.pressed(UP_BUTTON);
41  previousDOWNButtonPressed = aboy.pressed(DOWN_BUTTON);
42  previousLEFTButtonPressed = aboy.pressed(LEFT_BUTTON);
43  previousRIGHTButtonPressed = aboy.pressed(RIGHT_BUTTON);
44  previousAButtonPressed = aboy.pressed(A_BUTTON);
45  previousBButtonPressed = aboy.pressed(B_BUTTON);
46 }
```

Please copy this code to ensure you have the correct program.

Copy the code above

Sprites.h

This file is used as an easy way to store the sprites that we use for the up and down arrows of our calculator. We store them in a header file as they are defined arrays.

Your Task: is to simply create the header file, the code is presented below and should be copied.

```
1 const unsigned char PROGMEM downButtonFill[] =
2 {
3 // width, height,
4 10, 7,
5 0x06, 0x0f, 0x1f, 0x3f, 0x7f, 0x7f, 0x3f, 0x1f, 0x0f, 0x06,
6 };
7
8 const unsigned char PROGMEM downButtonNoFill[] =
9 {
10 // width, height,
11 10, 7,
12 0x06, 0x09, 0x11, 0x21, 0x41, 0x41, 0x21, 0x11, 0x09, 0x06,
13 };
14
15 const unsigned char PROGMEM upButtonFill[] =
16 {
17 // width, height,
18 10, 7,
19 0x30, 0x78, 0x7c, 0x7e, 0x7f, 0x7f, 0x7e, 0x7c, 0x78, 0x30,
20 };
21 const unsigned char PROGMEM upButtonNoFill[] =
22 {
23 // width, height,
24 10, 7,
25 0x30, 0x48, 0x44, 0x42, 0x41, 0x41, 0x42, 0x44, 0x48, 0x30,
26 };
```

Copy the code above. Explanation: of **const unsigned char PROGMEM upButtonNoFill[]**

PROGMEM = Means the data is stored in the ROM program memory, not the RAM that is the default storage area. We store spirits in ROM/program memory as they tend to be larger and we have a small amount of RAM.

Const = Means it cannot change

Unsigned = means it must be positive (cannot be negative)

Char = means it is a character that is of 8 bit (1 byte) size range from -127 to 128

const unsigned char = A character that is an 8bit value that can only hold numbers that cannot be negative.

upButtonNoFill[] = we are declaring an array of unknown size, however as we are putting the values inside the array but default, it does not matter. (unlike C++ where you must put the number of elements that are present inside of that array)

The first two values of the array are the width and height; the rest are the pixels.

The way the above code was produced was through two websites:

<https://www.pixilart.com/> - To create pixel art. This was exported as a png.

<https://teamarg.github.io/arduboy-image-converter/> - To then convert the png file to the code you see above.

*I use two kinds of arrows: one with just the arrow outline and one that is filled. This is so then when the user selects the arrow, I can flash the filled arrow so they know it has been pressed. Try doing something similar. **You must, however, create the file with a width of 7 pixels and a height of 10 pixels. NOTE that this is not compulsory, you should be using the code presented above.** When you have finished this program, you can then personalize.*

Now that we have finished the other files, it's time to piece the puzzle together.

Calculator Mechanism

To create the calculator mechanism, we use a class, within this class, we have a method that calculates the equation depending on the sign. This will be written in your main "Calculator.ino" file (containing the void setup and loop) and should be typed above the "void setup{}" code.

Your task is to create the class and see if you can create the method that does this. Note that we utilize the constructor to get the parameters of the equation. (covered in Classes 101). The below functions might be useful:

*Switch + case statement: Similar to a bunch of If statements, a switch and case statement compare a value with the case value, aka if ValueToBeCompared was 2, then the 3rd case down would run as that is case 2; and ValueToBeCompared == 2, therefore a value would be returned and the loop would break. If no value is matched, then the default block is run. If the breaks are not used, then any case statement that matches will run and so will the default block of code. **See the example on the right.***

```
1 switch (ValueToBeCompared) {
2 case 0:
3   //Do something like return a value
4   return [aValue];
5   break;
6 case 1:
7   //Do something like return a value
8   return [aValue];
9   break;
10 case 2:
11   //Do something like return a value
12   return [aValue];
13   break;
14 case 3:
15   //Do something like return a value
16   return [aValue];
17   break;
18 default: // if none of the cases match
19   //Do something like return a value
20   return 0;
21 }
```

Class Math Code

Explanation: Here we can see, the final class, called math. In this class, we have two constructor variables, one for when we create an object with no parameters (we always need this one) and one for when there are parameters present. The parameters required are each part of the equation, value 1, the sign position and value 2. Whatever this function returns is the final answer. Hence using return in the switch statement. The **double calculate()** is a method to calculate the final answer depends on the sign position. We use double as it has a large range and deals with decimals. When we declare a function we must define the data type of the return values, therefore writing double means that any value we return will be double. In the comments, you can see what each position represents. The default value is 0 ensuring that something is printed to the screen no matter what and that we do not get an error. **Hopefully, your code looks similar to this format, if not**

```
35 class math {
36   public:
37     math();
38     double a;
39     double b;
40     int sign;
41     math(int temp_a, int temp_sign, int temp_b) {
42       a = temp_a;
43       b = temp_b;
44       sign = temp_sign;
45     }
46     double calculate() {
47       switch (sign) {
48         case 0: //"+"
49           return a + b;
50           break;
51         case 1: //"-"
52           return a - b;
53           break;
54         case 2: //"*"
55           return a * b;
56           break;
57         case 3: "/"
58           return a / b;
59           break;
60         default:
61           return 0;
62       }
63     }
64 };
```

then type in the above code. If not then adjust it accordingly, updating the variables with the ones presented above. Copy the code above into the Calculator.ino file.

Before we get started on creating the calculator mechanism inside the void loop, we first need to boot up Arduoy, set pins and setup serial.

Your task is to see if you can write the void setup for this program. The things we need to do is boot the Arduoy (to initialize the library), set the pin modes as OUTPUTs of the 3 LEDs (10 = Yellow, 3 = Green, 9 = Red), clear the Arduoy screen and then begin serial. The below functions might come in handy:

Void setup() { [CODE] } = code that runs once as soon as the program starts running

aboy.boot() = will boot the Arduoy without the Arduoy splash screen, it will also initialize the library

pinMode([PinNumber],[INPUT/OUTPUT]); = Will define if a pin is an input or output if it will be receiving data or sending out data

aboy.clear(); = In simple terms, this function (clear) will clear the screen. It will clear the screen buffer and display it. This means it will clear the screen and whatever is on it. Every time we print something, **aboy.print("Hello World");**, it gets stored in the screen buffer. When we use the **aboy.display()** it will display the screen buffer, aka the "Hello World".

`Serial.begin([Frequency]);` = will begin the serial and set the rate of transmission (frequency) we will use 9600.

Void Setup Code

```
57 void setup() {
58   aboy.boot();
59   //Allows us to prepare the arduboy library without the arduboy splashscreen
60   pinMode(10, OUTPUT); //Yellow
61   pinMode(3, OUTPUT); //Green
62   pinMode(9, OUTPUT); //Red
63
64   aboy.clear(); // Clear screen
65   Serial.begin(9600); // begin serial
66 }
```

Hopefully, your code looks similar to this format, if not then type in the above code. If not then adjust it accordingly, updating the variables with the ones presented above. Copy the code above.

Now we can move onto the loop.

Your task is to write the code to be able to do the following:

- Prepare the Arduboy for printing/print to screen. This involves clearing/displaying the screen, setting the cursor and text size.
- Calculate and print the equation and result on the screen
 - A challenge: If the sum is an integer, then remove the '.00' on the end of the double, else keep it.

Useful functions that might come in handy:

`aboy.setCursor([X coord] , [Y coord]);` = This will set the cursor ready to print text. It requires two parameters.

`Aboy.setTextSize();` = Will multiply the current text size by a scale factor. The value must be greater than or equal to 1. Current text size is 6x8 pixels. A multiplier of 2 would mean the characters become

`([Data Type]) [Variable]` = will attempt to turn the variable to the specified data type, this is called Cast. E.g. `(int) DecimalNumber` would turn the decimal number into an integer.

`Aboy.print();` = similar to `Serial.print();` will display the content at the set cursor position, if its text it will display it at the text size specified before (default 1).

`Aboy.display();` = Displays the screen buffer, aka if you have a print statement without the display, nothing will be shown on the screen – this is used to display the content onto the screen.

Calculator Core Loop

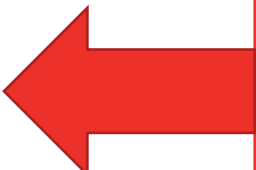
In this section, the final code is presented on this page, the explanation for this code is on the next page. Ensure you understand this "unit" before moving on.

Copy the code below:

Uploading the below code should give you a simple 0+0=0 on your screen.

These are the new define variables. Typed in before the void setup()

```
1 #include <Arduboy2.h>
2 Arduboy2 aboy;
3 #include "sprites.h"
4
5 typedef struct point {
6     int x;
7     int y;
8 };
9
10 #include "controller.h"
11 Controller Controller;
12
13 //Values for Equation
14 int FirstDigit = 0;
15 int signOption = 0;
16 String Sign[4] = {"+", "-", "x", "/"};
17 int SecondDigit = 0;
18 double Sum;
19 int int_sum;
20
21 int Option = 0;
22
23 String mystring;
24 String equation[5] = {"0","+","0","=","0"};
25
```



Its important to declare the "struct point" before we include the controller.h and create the object as the struct point is referenced inside of the file controller.h – if we were to swap these around then we would get a declaration error. This is the same for any other variables that are referenced in another file – we must first declare the variable and then include the file.

Update your void loop()

```
68 void loop() {
69     aboy.clear();
70
71     aboy.setCursor(9, 24);
72     //set cursor position
73     aboy.setTextSize(2);
74     //set Text Size
75
76     equation[0] = FirstDigit; //update the array with current First Digit
77     equation[1] = Sign[signOption]; //Update Equation with the Sign symbol
78     equation[2] = SecondDigit; //update the array with current Second Digit
79     math math(FirstDigit, signOption, SecondDigit); //create an object of math with the parameters
80     Sum = math.calculate(); //Calculate the sum of the current values + sign
81
82     //the below if statement checks if the value is a decimal
83     if ((int_sum = (int)Sum) == Sum) {
84         int_sum = (int)Sum;
85         equation[4] = int_sum;
86     }
87     else {
88         equation[4] = Sum;
89     }
90
91     mystring = equation[0] + equation[1] + equation[2] + equation[3] + equation[4];
92     //convert all digits to strings ready to be printed to the screen
93     aboy.print(mystring);
94
95     aboy.display();
96 }
```

Explanation:

```
68 void loop() {
69   aboy.clear();
70
71   aboy.setCursor(9, 24);
72   //set cursor position
73   aboy.setTextSize(2);
74   //set Text Size
75
76   equation[0] = FirstDigit; //update the array with current First Digit
77   equation[1] = Sign[signOption]; //Update Equation with the Sign symbol
78   equation[2] = SecondDigit; //update the array with current Second Digit
79   math math(FirstDigit, signOption, SecondDigit); //create an object of math with the parameters
80   Sum = math.calculate(); //Calculate the sum of the current values + sign
```

Lines 76 to 78 prepare the equation array (I'll show you the newly defined arrays that we need to update after this) with the new digits. We store the whole equation in an array for easy storage. Line 77 also demonstrates how we are going to use and present math symbols. Below shows the array mentioned. We will have an array that will store all of the signs

(For beginners, to create an array, we must first DefineDataType ArrayName [Array size] = {"Element 1","Element 2"}; E.g. `16 String Sign[4] = {"+", "-", "x", "/"};`

We then use a variable called "signOption" that will be updated with the user input and change the printed sign.

Line 79 creates an object called math with the parameters of the equation. We then call the method to calculate(); on line 80 and, as we used returns, can make Sum equal to the result of the equation.

```
82 //the below if statement checks if the value is a decimal
83 if ((int_sum = (int)Sum) == Sum) {
84   int_sum = (int)Sum;
85   equation[4] = int_sum;
86 }
87 else {
88   equation[4] = Sum;
89 }
```

The challenge: Here we check the sum, if **int_sum** is equal to **Sum** then it is an integer as they are equal (35 is the same as 35.00). In the If statement, we make **int_sum** equal to the converted value of the sum using a cast function "**(int) Sum**". Therefore, if the answer is 35.5, **int_sum** would equal 35 but **Sum** would still equal 35.5, therefore the else part of the statement runs and, as we don't want to lose data and want to display the decimal points, we append (add to the array) the **Sum**. If they are equal, we make **int_sum = (int)Sum**, if it isn't already, and append the integer version of the **Sum** (without the .00) to the equation to be printed.

```
91 mystring = equation[0] + equation[1] + equation[2] + equation[3] + equation[4];
92 //convert all digits to strings ready to be printed to the screen
93 aboy.print(mystring);
94
95 aboy.display();
96 }
```

We then get each element in the array equation, and add them together (create a single string from the array) using the plus icon (simply "one-string" + "two-string" adds the strings together "one stringtwo string"). Allowing us to print the whole equation in one print statement and then display it with `aboy.display()`;

Next, we need to add the special arrows and allow the user to move the "cursor"/arrows around and change the characters.

Laying out the Framework for X Movement

The movement utilizes the controller.position.x and .y. In this unit we lay the framework for the movement. The x movement will move between each digit in the equation: our equation will look like "0+0=0" with the 3 digits, 0+0, being altered to the users need.

Your task is to create the structure for the x movement. When the position is 0, something needs to happen. For now, write a comment saying the name of each digit. Aka, for 0, simply write // the First digit. For 1, //Sign. For 2, //Second Digit. Useful functions that might help are below:

Switch Case: See explanation from before.

Controller.position.x = will store the x value, and is updated when the user changes it (with the LEFT and RIGHT buttons).

Explanation: Type this switch case after printing "mystring" in the void loop

```
99 | switch (Controller.position.x) { //X SHIFT
100 |     case 0: //First Digit
101 |     case 1: //Sign
102 |     case 2: //Second Digit
103 | }
```

The Controller.position.x value will control the movement with the LEFT and RIGHT buttons, aka movement along the x-axis. For us, that is cycling between 3 options, the first value, sign and second value. To achieve this, we use a basic switch statement that compares the Controller.position.x that will change when the user clicks the LEFT or RIGHT button. We can then put the code for each element in the equation based on if it is currently selected or not

We also need to update the position each loop, therefore we need to call the Controller method update. **Write this after clearing the aboy screen (aboy.clear());**

```
72 | Controller.update();
```

Collision

Your Task: However, the current code means that we can move the controller.position.x, yes between 0 and 2, but also doesn't stop us when we reach the boundary (aka move too far), meaning the player can go off-screen. Write code that will restrict the player from exceeding controller.position.x between 0 and 2 (3 values, for each digit). The functions below might be useful:

Void [FunctionName] (int ArgOne, int ArgTwo){ [Code] } = Allows us to create a function with parameters, in this case, we created a function that has no returns (as void means there is no return, therefore, no need to specify a data type). Here we have two parameters that must be met upon using this function.

Controller.LEFTButtonCanBePress; = was a Boolean variable we defined earlier that switches the movement in a certain direction on/off.

Adding an X Boundary/Collision

Explanation: For this specific program, we only have one boundary, that is the X boundary of only being able to move between updating the 3 digits of the calculator. To achieve this, we use the below function that can easily be duplicated for the Y coordinates in other programs however in this program, we want the user to be able to type any number and therefore do not have a boundary,

```
232 void Check_Range_X(int lowrange, int highrange)
233 {
234     if (Controller.position.x <= lowrange) {
235         Controller.LEFTButtonCanBePress = 0;
236         Controller.RIGHTButtonCanBePress = 1;
237     }
238     else if (Controller.position.x >= highrange) {
239         Controller.LEFTButtonCanBePress = 1;
240         Controller.RIGHTButtonCanBePress = 0;
241     }
242     else {
243         Controller.LEFTButtonCanBePress = 1;
244         Controller.RIGHTButtonCanBePress = 1;
245     }
246 }
```

Hopefully, your code looks similar to this format, if not then type in the above code. It should be placed after the void loop. Copy the code.

Here you can see, when we define the function we require two numbers, this help makes the function dynamic and be able to be used in many different situations. Here we require that we get the range parameters – aka lowest range and highest range. This is then used in the if statement.

If the current position is lower than the range, then we turn off movement going even lower (left button) as we want the player to stop moving, stopping the movement also stops the player if the player is holding down the right button and moving into a boundary, this function checks it and therefore stops ALL movement. We only allow movement on the X-axis to be done via the right button, aka to get back in range. The next statement uses a similar format, however, it's the higher range, therefore, we turn off the movement for the right button, and only allow the position to be updated when the user moves back into range (aka only allow the user to press the left button).

```
144 Check_Range_X(0, 2);
145 Controller.update();
```

Be sure to write this section of code inside the void loop!

Here you can see we run it in the main loop before we update the position values (aka before we check the button controls). This is so we can check if the player is in range, and therefore only allow the correct buttons to be read (I say read because the user can still press the button, however as we set the "ButtonCanBePress" value to false/0, we choose to ignore it regardless.

This ensures that the user can never exceed the values 0, 1 or 2. There are simpler ways to achieve this, however, by simply writing an if statement as shown below:

```
If (Controller.position.x == 0 or Controller.position.x == 1 or Controller.position.x == 3) { [CODE] }
```

However, this introduces us to algorithmic design and is a common way to deal with larger boundaries. As, if the boundary was between 0 and 100, we could easily change this.

Adding Sprites

While we can currently cycle between each digit, we cannot see this change (you can always add a `Serial.print(Controller.position.x);` to demonstrate this). To show this, we will use the sprites we added in the `sprites.h` header file. These will also come in handy when we want to show the y movement.

Your task is to show movement, display the arrow sprites above each digit when it is selected. We will later animate these to fill when we press up and down - however, for now, simply move the arrows between each digit. The below functions might be useful:

`Sprites::drawOverwrite(Xpos, Ypos, SpriteName, 0);` = This will display the sprite at the specified x and y position. The 0 simply means what frame to show, for this project, we only use one frame animation, therefore this will always be 0.

`break;` = will exit out of the loop when it is executed.

Tip, the up arrow should be printed with a y value of 15 and the down arrow should be printed with an x value of 42.

Challenge: When the calculator has 2 digits, we need to relocate the arrow to be centred. Write a simple math statement in the X Position of the `drawOverwrite` function that relocates the arrow accordingly based on the number of digits present. Functions that might come in handy:

`String.length();` = will return the amount of characters in a string. E.g. "10" has 2 characters while "hello" has 6.

(string) Variable = is a cast function that converts the variable to a string.

Moving Sprites along the X-axis

```
99 switch (Controller.position.x) { //X SHIFT
100   case 0: //First Digit
101     Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 15, upButtonNoFill, 0);
102     Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 42, downButtonNoFill, 0);
103     break;
104   case 1: //Sign
105     Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 15, upButtonNoFill, 0);
106     Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 42, downButtonNoFill, 0);
107     break;
108   case 2: //Second Digit
109     Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 15, upButtonNoFill, 0);
110     Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 42, downButtonNoFill, 0);
111     break;
112 }
113 aboy.display();
114 }
```

Hopefully, your code looks similar to this format, if not then type in the above code. This should update your current switch statement. Copy the code above.

Explanation: The main focus is what's inside the Position X position value. Note that I introduce a new function I coded called '`numOfDigits();`'

```
132 int numOfDigits(int integer) {
133   String string = (String)integer;
134   return string.length();
135 }
```

This function takes an input integer and returns the number of digits that integer has. Aka input 10, return 2. We do this by utilizing the `[StringVar].length()` function that returns the number of characters in a

string. To do this, the line before we simply declare a new string variable called `string` and make it equal to the integer but in string form, therefore 10 becomes "10". Allowing us to use the `.length()` function

```
99 switch (Controller.position.x) { //X SHIFT
100   case 0: //First Digit
101     Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 15, upButtonNoFill, 0);
102     Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 42, downButtonNoFill, 0);
103     break;
```

Why do we use this? As you can see, we use this to calculate the number of digits and therefore adjust the x position to centre the arrow correctly. Below is each x position equation for each

case section. The other parameters are all the same: all upButtons are on the same 15 y value and down buttons on the same 42 y value.

Case 0:

X_Constant + (6 * numOfDigits(FirstDigit)) - 5

X_Constant = 9 and is simply the offset from the screen (when we print the text we print is at an x coord of 9. This is constant for all case position X values.

(6 * numOfDigits(FirstDigit)) = This finds the midpoint of the digit (each character has a width of 12 pixels), therefore by multiplying by 6 we can add on the centre of each digit present.

- 5 = as the sprites are 10 pixels wide, we minus 5 from the new centre point as the sprites are drawn from the top left corner. This is constant for all case position X values, as we always calculate the centre point and then have to shift it over by -5 to print the sprite.

Case 1:

X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5

(12 * numOfDigits(FirstDigit)) = Adds the number of digits present in the first digit to the position, here we multiply it by 12 as we are using a text size of 2, which has a width of 12. This ensures that when we add the centre point of the signOption digits, it is at the correct offset – regardless of how many digits come before it.

(6 * numOfDigits(signOption)) = calculates the midpoint of the signOption. This allows it to be dynamic

We then -5 to ensure it prints correctly

Case 2

X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5

The case 3 must take into consideration all leading digits. Therefore, we add all of the digits leading up to it **(+ (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)))** and then find the midpoint of the current digits, **(6 * numOfDigits(SecondDigit))**, then we -5 to ensure it prints correctly.

This allows the printing of the arrows to be dynamic and always be centred.

```
13 | int X_Constant = 9;| Be sure to write this value before the void setup.
```

Adding Y Axis Movement

Now that we have each case statement for each digit. We can start allowing the user to set the numbers for the sum. The first and second digit will have the same movement system, while the signOption case will need to be restricted to be move between only the 4 signs.

Your Task: In each case statement, write code that will update the FirstDigit, signOption and SecondDigit value according to the users Controller.position.y. Note that we will need to apply a boundary to the signOption as it should only change/update its value if it is between 0 and 3 (4 values, 1 for each sign). The below functions might come in handy:

Switch + Case: See above explanation

`Controller.position.y` = will allow you to access the position value for y that updates with the UP and DOWN buttons.

Explanation:

Case 0 and 2:

```
103 case 0: //First Digit|          190
104     switch (Controller.position.y - LastYPos) {          191
105         case -1: FirstDigit -= 1; break;                192
106         case 0: break;                                  193
107         case 1: FirstDigit += 1; break;                 194
108     }                                                    195
                                                    case 2:
                                                    switch (Controller.position.y - LastYPos) {
                                                    case -1: SecondDigit -= 1; break;
                                                    case 0: break;
                                                    case 1: SecondDigit += 1; break;
                                                    }
```

For the digits, we use the same code. Here we have a switch statement that first checks to see if the `controller.position.y` has changed. We can see if it has changed by comparing it with the last value, aka minus it from the `LastYPos`. We will ONLY get 3 results: -1, 0 & 1. Using a switch statement, we can decide if we need to either increase the first digit or decrease the first digit – this means that each button press will change the `FirstDigit` by one and therefore allow the user to control the equation digits. If the case is 0, then there is no change and we can just break out of the switch loop. This code is used for both cases, changing the appropriate digits.

Case 1:

```
113 case 1: //Sign
114     if ((signOption + (Controller.position.y - LastYPos )) >= 0 and (signOption + (Controller.position.y - LastYPos )) <= 3) {
115         switch (Controller.position.y - LastYPos) {
116             case -1: signOption -= 1; break;
117             case 0: break;
118             case 1: signOption += 1; break;
119         }
120     }
```

The case 1 uses the same switch statement, however, we have an if statement before it to only change the digit if it's between the range of 0 to 3 (as there are only 4 signs).

`(signOption + (Controller.position.y - LastYPos)) >= 0 and (signOption + (Controller.position.y - LastYPos)) <= 3`

Here we almost predict the value, we add the resultant of `Controller.position.y - LastYPos` to the `signOption` and see if it is in range (aka if it is greater than or equal to 0 or less than or equal to 3), if it is within range, then we can go ahead and make that calculation. If it isn't then do not change the value.

The full switch statement is below, copy the code.



Full Switch Statement:

```

102 switch (Controller.position.x) { //X SHIFT
103     case 0: //First Digit
104         switch (Controller.position.y - LastYPos) {
105             case -1: FirstDigit -= 1; break;
106             case 0: break;
107             case 1: FirstDigit += 1; break;
108         }
109
110         Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 15, upButtonNoFill, 0);
111         Sprites::drawOverwrite(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 42, downButtonNoFill, 0);
112         break;
113     case 1: //Sign
114         if ((signOption + (Controller.position.y - LastYPos)) >= 0 and (signOption + (Controller.position.y - LastYPos)) <= 3) {
115             switch (Controller.position.y - LastYPos) {
116                 case -1: signOption -= 1; break;
117                 case 0: break;
118                 case 1: signOption += 1; break;
119             }
120         }
121
122         Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 15, upButtonNoFill, 0);
123         Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 42, downButtonNoFill, 0);
124         break;
125     case 2: //Second Digit
126         switch (Controller.position.y - LastYPos) {
127             case -1: SecondDigit -= 1; break;
128             case 0: break;
129             case 1: SecondDigit += 1; break;
130         }
131
132         Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 15, upButtonNoFill, 0);
133         Sprites::drawOverwrite(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 42, downButtonNoFill, 0);
134         break;
135 }

```

The full code for the switch statement. Please copy to ensure your program is kept up to date with ours. Be sure to also copy the code below.

Don't forget to define the new variables we use before the void setup:

```

24 int LastYPos;
25 int LastXPos;

```

We also need to ensure we update the LastYPos and LastXPos variables each loop, this part of the code goes to the end of the loop, before the display command.

```

136 LastYPos = Controller.position.y;
137 LastXPos = Controller.position.x;
138 |
139 aboy.display();

```

Animating the Buttons

Now that we have the basics of the program, it feels very static. To make it feel more dynamic and responsive, we can animate the buttons to help the user see/feel like they have virtually clicked the button.

Your task is to write an "UpButton" and "DownButton" Function that we can reuse for every button. This function will, when the button is pressed: fill the arrow and move the arrow (e.g. if up arrow pressed, move arrow up by -2 to the y coordinate). The Below functions might come in handy:

`Millis();` = returns current runtime in milliseconds from the Arduino is turned on.

`Sprites::drawOverwrite(xpos, ypos, sprite, 0);` = to draw a sprite, requires the x and y position and then the sprite name. the 0 simply means what frame to print. In our case, this will always remain 0.

```
206 void buttonUp(int xpos, int ypos) {
207   if (Controller.position.y - LastYPos == 1) {
208     UPlastPressedTime = millis();
209     Sprites::drawOverwrite(xpos, ypos - 2, upButtonFill, 0);
210   }
211   else if ((millis() - UPlastPressedTime) < 150) {
212     Sprites::drawOverwrite(xpos, ypos - 2, upButtonFill, 0);
213   }
214   else {
215     Sprites::drawOverwrite(xpos, ypos, upButtonNoFill, 0);
216   }
217 }
218
219 void buttonDown(int xpos, int ypos) {
220   if (Controller.position.y - LastYPos == -1) {
221     DOWNlastPressedTime = millis();
222     Sprites::drawOverwrite(xpos, ypos + 2, downButtonFill, 0);
223   }
224   else if ((millis() - DOWNlastPressedTime) < 150) {
225     Sprites::drawOverwrite(xpos, ypos + 2, downButtonFill, 0);
226   }
227   else {
228     Sprites::drawOverwrite(xpos, ypos, downButtonNoFill, 0);
229   }
230 }
```

Explanation: Here we have two similar functions. `buttonUp`, takes two inputs, the x and y position for the arrow (this will be a copy of what we used before when we printed the first arrow). Then we use 3 if statements to decide if the arrow should be filled or not. The first is if the button has been pressed. We can derive this from a change in the Y position (by minusing the current position by the last Y position, if it equals one, then the button is being pressed up and therefore should be filled. We can see how that the `buttonDown` uses a

`Controller.position.y - LastYPos == -1`. A minus 1 instead as it indicates that the position is decreasing therefore the down button is being pressed. If the y position does change, then we set the `UPlastPressedTime`, ready to compare for the next if statement, and then use the local parameters of the function to draw the arrow in the correct place. Note that we use `ypos - 2` (for the up button) and `ypos + 2` (for the down button). This adds a slight jump to the button. Also, note that we are displaying the `upButtonFill`. However, upon just uploading this, you'll release that it switches too fast and the moving animation does not flow well. To fix this, we use `milis` in the next statement. The next statement is an else if statement, if the button isn't being pressed, then essentially fill the button for 150 milliseconds (as we compare the `LastPressedTime` to the current time). This helps with debounce and makes each press the same length. If both of these are false, then write the `ButtonNoFill` at the `xpos` and `ypos`.

Add these functions after the void loop. **Please copy to ensure your program is kept up to date with ours.** Copy the code above

```
20 double UPlastPressedTime;
21 double DOWNlastPressedTime;
```

We, of course, need to *declare the variables* we use in this function before the void setup before we can upload the code. To integrate these functions, we simply replace the `Sprites::drawOverwrite(xpos, ypos, sprite,0);` with the new functions. Copying in the x and y parameters we used on the arrows to the functions. The below code shows how we would implement these new functions:

Implementation of the functions below, replace the `sprite::drawOverwrite` functions with the below functions - Copy the code below

```
104 switch (Controller.position.x) { //X SHIFT
105   case 0: //First Digit
106     switch (Controller.position.y - LastYPos) {
107       case -1: FirstDigit -= 1; break;
108       case 0: break;
109       case 1: FirstDigit += 1; break;
110     }
111     buttonUp(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 15);
112     buttonDown(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 42);
113     break;
114
115   case 1: //Sign
116     if ((signOption + (Controller.position.y - LastYPos)) >= 0 and (signOption + (Controller.position.y - LastYPos)) <= 3) {
117       switch (Controller.position.y - LastYPos) {
118         case -1: signOption -= 1; break;
119         case 0: break;
120         case 1: signOption += 1; break;
121       }
122     }
123
124     buttonUp(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 15);
125     buttonDown(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signOption)) - 5, 42);
126     break;
127
128   case 2: //Second Digit
129     switch (Controller.position.y - LastYPos) {
130       case -1: SecondDigit -= 1; break;
131       case 0: break;
132       case 1: SecondDigit += 1; break;
133     }
134
135     buttonUp(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 15);
136     buttonDown(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(signOption)) + (6 * numOfDigits(SecondDigit)) - 5, 42);
137     break;
138 }
```

Now that we have the core functional aspects of the calculator, the below code shows what you should have so far – go through it and compare what you have written to ensure you have the correct code.

Final Code

Congratulations on completing this course! Now you have a calculator on your 8BitCADE. Now is your chance to play around with it and see what you can add. Why not add a splash screen, your name or see what else you can do! The world is yours!

Calculator.ino

```
1. // BSD 3-Clause License
2. //
3. // Copyright (c) 2020, Jack Daly (@8bitcade)
4. // All rights reserved.
5. //
6. // Redistribution and use in source and binary forms, with or without
7. // modification, are permitted provided that the following conditions are met:
8. //
9. // 1. Redistributions of source code must retain the above copyright notice, this
10. // list of conditions and the following disclaimer.
11. //
12. // 2. Redistributions in binary form must reproduce the above copyright notice,
```

```

13. //      this list of conditions and the following disclaimer in the documentation
14. //      and/or other materials provided with the distribution.
15. //
16. //      3. Neither the name of the copyright holder nor the names of its
17. //      contributors may be used to endorse or promote products derived from
18. //      this software without specific prior written permission.
19. //
20. //      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
21. //      AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
22. //      IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
23. //      DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
24. //      FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
25. //      DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
26. //      SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
27. //      CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
28. //      OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
29. //      OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
30. //
31. #include <Arduboy2.h>
32. Arduboy2 aboy;
33. #include "sprites.h"
34.
35. typedef struct point {
36.     int x;
37.     int y;
38. };
39.
40. #include "controller.h"
41. Controller Controller;
42.
43. int X_Constant = 9;
44.
45. //Values for Equation
46. int FirstDigit = 0;
47. int signOption = 0;
48. String Sign[4] = {"+", "-", "x", "/"};
49. int SecondDigit = 0;
50. double UPlastPressedTime;
51. double DOWNlastPressedTime;
52. double Sum;
53. int int_sum;
54.
55. int Option = 0;
56. int LastYPos;
57. int LastXPos;
58.
59.
60. String mystring;
61. String equation[5] = {"0", "+", "0", "=", "0"};
62.
63. class math {
64.     public:
65.         math();
66.         double a;
67.         double b;
68.         int sign;
69.         math(int temp_a, int temp_sign, int temp_b) {
70.             a = temp_a;
71.             b = temp_b;
72.             sign = temp_sign;
73.         }
74.         double calculate() {
75.             switch (sign) {
76.                 case 0: //"+"
77.                     return a + b;
78.             }

```

```

79.     case 1: //"-"
80.         return a - b;
81.         break;
82.     case 2: //"*"
83.         return a * b;
84.         break;
85.     case 3: "/"
86.         return a / b;
87.         break;
88.     default:
89.         return 0;
90.     }
91. }
92. };
93.
94. void setup() {
95.     aboy.boot();
96.     //Allows us to prepare the arduboy library without the arduboy splashscreen
97.     pinMode(10, OUTPUT); //Yellow
98.     pinMode(3, OUTPUT); //Green
99.     pinMode(9, OUTPUT); //Red
100.
101.     aboy.clear(); // Clear screen
102.     Serial.begin(9600); // begin serial
103. }
104.
105. void loop() {
106.     aboy.clear();
107.
108.     Check_Range_X(0, 2);
109.     Controller.update();
110.
111.     aboy.setCursor(9, 24);
112.     aboy.setTextSize(2);
113.
114.     equation[0] = FirstDigit; //update the array with current First Digit
115.     equation[1] = Sign[signOption]; //Update Equation with the Sign symbol
116.     equation[2] = SecondDigit; //update the array with current Second Digit
117.     math math(FirstDigit, signOption, SecondDigit); //create an object of math with t
    he parameters
118.     Sum = math.calculate(); //Calculate the sum of the current values + sign
119.
120.     //the below if statement checks if the value is a decimal
121.     if ((int_sum = (int)Sum) == Sum) {
122.         int_sum = (int)Sum;
123.         equation[4] = int_sum;
124.     }
125.     else {
126.         equation[4] = Sum;
127.     }
128.
129.     mystring = equation[0] + equation[1] + equation[2] + equation[3] + equation[4];
130.     //convert all digits to strings ready to be printed to the screen
131.     aboy.print(mystring);
132.
133.
134.     switch (Controller.position.x) { //X SHIFT
135.         case 0: //First Digit
136.             switch (Controller.position.y - LastYPos) {
137.                 case -1: FirstDigit -= 1; break;
138.                 case 0: break;
139.                 case 1: FirstDigit += 1; break;
140.             }
141.             buttonUp(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 15);
142.             buttonDown(X_Constant + (6 * numOfDigits(FirstDigit)) - 5, 42);
143.             break;

```



```

144.
145.     case 1: //Sign
146.         if ((signOption + (Controller.position.y - LastYPos )) >= 0 and (signOption +
(Controller.position.y - LastYPos )) <= 3) {
147.             switch (Controller.position.y - LastYPos) {
148.                 case -1: signOption -= 1; break;
149.                 case 0: break;
150.                 case 1: signOption += 1; break;
151.             }
152.         }
153.
154.         buttonUp(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(signO
ption)) - 5, 15);
155.         buttonDown(X_Constant + (12 * numOfDigits(FirstDigit)) + (6 * numOfDigits(sig
nOption)) - 5, 42);
156.         break;
157.
158.     case 2: //Second Digit
159.         switch (Controller.position.y - LastYPos) {
160.             case -1: SecondDigit -= 1; break;
161.             case 0: break;
162.             case 1: SecondDigit += 1; break;
163.         }
164.
165.         buttonUp(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(sign
Option)) + (6 * numOfDigits(SecondDigit)) - 5, 15);
166.         buttonDown(X_Constant + (12 * numOfDigits(FirstDigit)) + (12 * numOfDigits(si
gnOption)) + (6 * numOfDigits(SecondDigit)) - 5, 42);
167.         break;
168.     }
169.     LastYPos = Controller.position.y;
170.     LastXPos = Controller.position.x;
171.
172.     aboy.display();
173. }
174.
175. void buttonUp(int xpos, int ypos) {
176.     if (Controller.position.y - LastYPos == 1) {
177.         UPlastPressedTime = millis();
178.         Sprites::drawOverwrite(xpos, ypos - 2, upButtonFill, 0);
179.     }
180.     else if ((millis() - UPlastPressedTime) < 150) {
181.         Sprites::drawOverwrite(xpos, ypos - 2, upButtonFill, 0);
182.     }
183.     else {
184.         Sprites::drawOverwrite(xpos, ypos, upButtonNoFill, 0);
185.     }
186. }
187.
188. void buttonDown(int xpos, int ypos) {
189.     if (Controller.position.y - LastYPos == -1) {
190.         DOWNlastPressedTime = millis();
191.         Sprites::drawOverwrite(xpos, ypos + 2, downButtonFill, 0);
192.     }
193.     else if ((millis() - DOWNlastPressedTime) < 150) {
194.         Sprites::drawOverwrite(xpos, ypos + 2, downButtonFill, 0);
195.     }
196.     else {
197.         Sprites::drawOverwrite(xpos, ypos, downButtonNoFill, 0);
198.     }
199. }
200.
201.
202.
203. void Check_Range_X(int lowrange, int highrange)
204. {

```

```

205.     if (Controller.position.x <= lowrange) {
206.         Controller.LEFTButtonCanBePress = 0;
207.         Controller.RIGHTButtonCanBePress = 1;
208.     }
209.     else if (Controller.position.x >= highrange) {
210.         Controller.LEFTButtonCanBePress = 1;
211.         Controller.RIGHTButtonCanBePress = 0;
212.     }
213.     else {
214.         Controller.LEFTButtonCanBePress = 1;
215.         Controller.RIGHTButtonCanBePress = 1;
216.     }
217. }
218.
219. int numOfDigits(int integer) {
220.     String string = (String)integer;
221.     return string.length();
222. }

```

Controller.h

```

1. class Controller
2. {
3.     public:
4.         //CONSTRUCTOR
5.         Controller();
6.         //Create from the point structure a position variable. This will be used to move th
7.         e cursor
8.         point position;
9.         //Define the method UPDATE used to update the controls every loop
10.        void update();
11.
12.        //BOOLEAN VARIABLES THAT STORES WHETHER BUTTON PRESSED IS A/B
13.        bool AButtonPressed;
14.        bool BButtonPressed;
15.
16.        //BOOLEAN VARIABLE THAT STORES WHETHER THE PREVIOUS BUTTON WAS PRESSED
17.        bool previousUPButtonPressed;
18.        bool previousDOWNButtonPressed;
19.        bool previousLEFTButtonPressed;
20.        bool previousRIGHTButtonPressed;
21.        bool previousAButtonPressed;
22.        bool previousBButtonPressed;
23.
24.        //BOOLEAN VARIABLES THAT CHECKS IF THE BUTTON PRESS IS UPDATED (A/B PRESSED)
25.        bool AButtonWasPressed;
26.        bool BButtonWasPressed;
27.
28.        //BOOLEAN VARIABLES THAT CHECKS IF THE BUTTON CAN BE PRESSED. USED FOR COLLISION/RE
29.        STRICTION
30.        bool UPButtonCanBePress = 1;
31.        bool DOWNButtonCanBePress = 1;
32.        bool LEFTButtonCanBePress = 1;
33.        bool RIGHTButtonCanBePress = 1;
34.
35.        //HOLDS POSITION OF 'CURSOR' OR CHARACTER
36.        int pos;
37.
38.        long debounceDelay = 75; //TIME WAITED UNTILL NEXT BUTTON IS INPUTTED
39.        long currenttime = 0; //CURRENT TIME
40.        long lastHoldTime = 100; //TIME, IN MILLIS, OF LAST BUTTON PRESS

```

```
41. };
```

Controller.ino

```
1. Controller::Controller()
2. {
3.   Serial.print("BUTTON TEST BUTTON TEST");
4.   position.x = 0;
5.   position.y = 0;
6. }
7.
8. void Controller::update()
9. {
10.  if (aboy.pressed(UP_BUTTON) and previousUPButtonPressed == 1 and (millis() - lastHold
    Time) > 150 and UPButtonCanBePress) {
11.    lastHoldTime = millis();
12.    position.y += 1;
13.  }
14.  if ((aboy.pressed(DOWN_BUTTON) == 1 and previousDOWNButtonPressed == 1 and (millis()
    - lastHoldTime) > 150) and DOWNButtonCanBePress) {
15.    lastHoldTime = millis();
16.    position.y -= 1;
17.  }
18.  if (aboy.pressed(LEFT_BUTTON) and previousLEFTButtonPressed == 1 and (millis() - last
    HoldTime) > 150 and LEFTButtonCanBePress) {
19.    lastHoldTime = millis();
20.    position.x -= 1;
21.  }
22.  if ((aboy.pressed(RIGHT_BUTTON) == 1 and previousRIGHTButtonPressed == 1 and (millis(
    ) - lastHoldTime) > 150) and RIGHTButtonCanBePress) {
23.    lastHoldTime = millis();
24.    position.x += 1;
25.  }
26.  if (aboy.pressed(A_BUTTON) and previousAButtonPressed == 1 and (millis() - lastHoldTi
    me) > 250) {
27.    lastHoldTime = millis();
28.    AButtonPressed = true;
29.  }
30.  else {
31.    AButtonPressed = false;
32.  }
33.  if ((aboy.pressed(B_BUTTON) == 1 and previousBButtonPressed == 1 and (millis() - last
    HoldTime) > 250) ) {
34.    lastHoldTime = millis();
35.    BButtonPressed = true;
36.  }
37.  else {
38.    BButtonPressed = false;
39.  }
40.  previousUPButtonPressed = aboy.pressed(UP_BUTTON);
41.  previousDOWNButtonPressed = aboy.pressed(DOWN_BUTTON);
42.  previousLEFTButtonPressed = aboy.pressed(LEFT_BUTTON);
43.  previousRIGHTButtonPressed = aboy.pressed(RIGHT_BUTTON);
44.  previousAButtonPressed = aboy.pressed(A_BUTTON);
45.  previousBButtonPressed = aboy.pressed(B_BUTTON);
46. }
```

Sprites.h

```
1. const unsigned char PROGMEM downButtonFill[] =
2. {
```

```

3. // width, height,
4. 10, 7,
5. 0x06, 0x0f, 0x1f, 0x3f, 0x7f, 0x7f, 0x3f, 0x1f, 0x0f, 0x06,
6. };
7.
8. const unsigned char PROGMEM downButtonNoFill[] =
9. {
10. // width, height,
11. 10, 7,
12. 0x06, 0x09, 0x11, 0x21, 0x41, 0x41, 0x21, 0x11, 0x09, 0x06,
13. };
14.
15. const unsigned char PROGMEM upButtonFill[] =
16. {
17. // width, height,
18. 10, 7,
19. 0x30, 0x78, 0x7c, 0x7e, 0x7f, 0x7f, 0x7e, 0x7c, 0x78, 0x30,
20. };
21. const unsigned char PROGMEM upButtonNoFill[] =
22. {
23. // width, height,
24. 10, 7,
25. 0x30, 0x48, 0x44, 0x42, 0x41, 0x41, 0x42, 0x44, 0x48, 0x30,
26. };

```

Thank you for following along with this tutorial. If you have any programming questions, we strongly advise that you check out the [Arduboy community](#). If you have any 8BitCADE related issues, please email us at 8BitCADE@support.com.

Check out our other tutorials at 8bitcade.com/learn