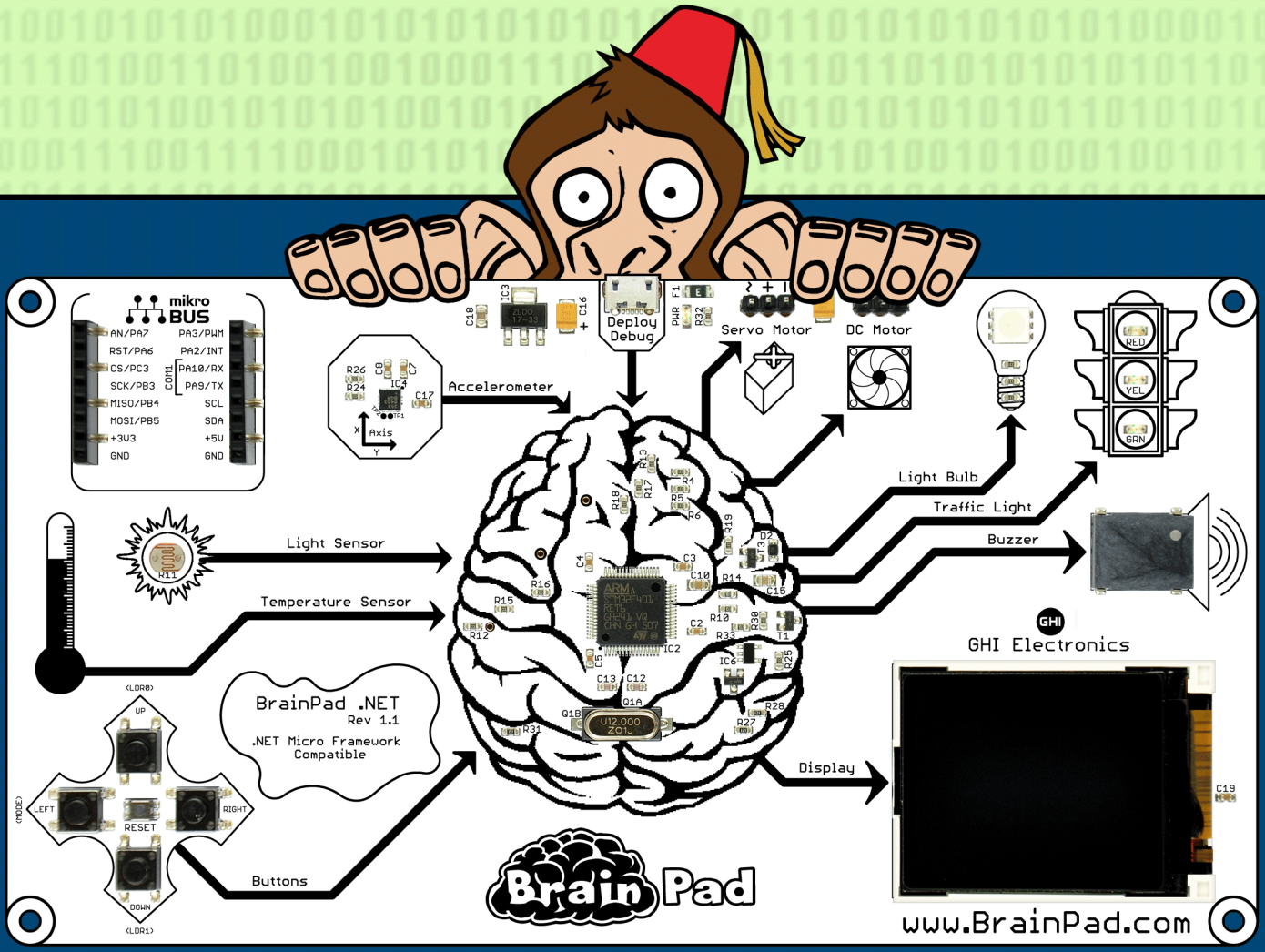
GHI electronics

Brain Pad

C#

THREADING AND EVENTS



Contents

Introduction

The BrainPad circuit board is designed as a powerful educational tool that can be used to teach everyone from kids, to college students and professionals. Kids will start to learn programming using Visual Studio, one of the most widely used professional tools. College students and professionals that already know programming can use the BrainPad circuit board to learn about digital electronics and the connection between computing and the physical world.

Overview

Students will learn to use threads for multitasking and how events provide better system handling.

Guidelines

- Prerequisites: CS102
- Ages 12 and up
- PC setup with Visual Studio, .NET Micro Framework and GHI Electronics' software.
- Supplies: BrainPad

Methods

The `BrainPad` object (or `class`) includes methods to control many aspects of the BrainPad's hardware. A method is a set of instructions grouped together. If a student is asked to speak, the command may look like `Student.Say("Hello")`. The `Say` method is simple, but speaking requires many things like taking in air and moving your vocal cords. In the same sense, activating the green light on the traffic light is a simple request but internally it does many small tasks to reach the final goal.

Methods can also take arguments. For example, you could have a method called `Student.Run()` to order a student to run or `Student.Run(slow)` to order them to run slow. Methods can also return a value, like `Student.GetAge()` which returns the student's age.

```
public class Program
{
    public void BrainPadSetup()
    {
        int total = Add(5, 2);
        BrainPad.WriteDebugMessage(total);
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

Example 1 – Shows how a function called `Add` can add 5 + 2 and print the total integer to the Output Window.

Example 1 creates a simple method that takes two integer arguments and returns an integer. The method will add the two arguments and return the results. Let's assume that we need a method that adds two integers and returns a string.

```
public class Program
{
    public void BrainPadSetup()
    {
        int total = Add(5, 2);
        BrainPad.WriteDebugMessage(total);
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public int Add(int a, int b)
    {
        return (a + b).ToString();
    }
}
```

Example 2 – Shows how a function called `Add` can add 5 + 2 and print the total string to the Output Window.

Note how the return type changed from an `int` to a `string`, and the returned value is converted to string using the `ToString()` method. The `ToString()` method is built into the system and works on almost everything.

Method names are like variable names, only certain things are allowed. Method names cannot start with a number, contain a symbol besides the underscore "_" or have a space in them.

The following examples show proper and improper use of method names.

```
bool AreAll4ButtonsPressed()
string Add(int a, int b)
```

Example 3 – These are examples of proper method names.

```
bool areallbuttonspressed()
int method34from94handler()
```

Example 4 – These are examples of hard to read/understand method names.

Note: Method names should always be easy to read and meaningful. This allows a programmer to easily discern what it does.

```
bool Are All Buttons Pressed()
bool AreAllButtonsPressed?()
```

Example 5 – These are examples of illegal method names.

Methods are not required to return anything. To fill that case of not returning a value the keyword `void` is used.

```
void ActivateAlarm()
```

Example 6 – The keyword `void` is used when a method returns nothing.

Finally, methods can also be private or public and static or non-static. This is beyond the scope of this course and `public static` will always be used.

Overloading Methods

The same method name can have one or more argument types. Depending on the argument passed to the method, the system will determine which method to call as shown in Example 7.

```
public class Program
{
    public void BrainPadSetup()
    {
        Test(5);
        Test(5.0);
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Test(int x)
    {
        BrainPad.WriteDebugMessage("integer");
    }

    public void Test(double x)
    {
        BrainPad.WriteDebugMessage("double");
    }
}
```

Example 7 – This code shows how a method can be overloaded to accept multiple argument configurations. In this case, `Test` accepts both an integer and a double.

The first call to `Test` will result in “integer” being printed in the Output Window and the second call will print “double”.

Exercise

Create a method that makes a beep sound and name it Beep. The method will take one argument that determines the beep length in milliseconds. This argument is optional (use overloading).

Boolean Variables

In programming we use `true` or `false` to represent the truth values of logic. These values are known as Boolean or `bool` when coding. For example, let's say we need to check if the up and down buttons are pressed in multiple spots throughout our program. We could check each button in each spot or we could create a reusable method that returns `true` if both are pressed or `false` otherwise, as shown in Example 8.

```
public class Program
{
    public void BrainPadSetup()
    {
        // Declared but not used
    }

    public void BrainPadLoop()
    {
        if (UpAndDownPressed())
        {
            BrainPad.TrafficLight.TurnGreenLightOn();
        }
        else
        {
            BrainPad.TrafficLight.TurnGreenLightOff();
        }
    }

    public bool UpAndDownPressed()
    {
        if (BrainPad.Button.IsUpPressed() && BrainPad.Button.IsDownPressed())
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Example 8 – This code calls a function that returns whether up and down are pressed and if so it turns the green light on.

Exercise

Create a `bool` variable named `MyBool` and then read the down button state into this variable. Use the `ToString()` method to print the `MyBool` value in the Output window. Run this in a `while` loop that repeats five times every second.

The new Keyword

In the examples used so far, the `BrainPad` object has been used directly. This will not work for all object types. Remember the `Student.Say("Hello")` example? This statement is not completely valid because we don't know which student is going to say "Hello". To access a specific student, you need to create a variable to hold the `Student` object as shown in Example 9.

```
Student mike = new Student();
```

Example 9 – Here we create a variable called `mike` which holds the `Student` object.

The difference between the `BrainPad` object and the `Student` object is that there is one, and only one, `BrainPad` object. The `Student` object is just a type and we need to construct (create a variable) to access each one.

Exercise

Write hypothetical code that constructs `Planet` object named `earth` and read its `DayLength()` method.

Threading

Threading in the programming world is a way to describe multitasking. Each task is a thread that runs separately. The threading support in .NET Micro Framework on the BrainPad is easy to work with. First, we need to inform the system that the threading library (Example 10) needs to be imported.

```
using System.Threading;
```

Example 10 – This code imports the threading library.

Before adding a thread, we need a method for it to use as shown in Example 11.

```
public class Program
{
    public void BrainPadSetup()
    {
        Thread blinkerThread = new Thread(Blinker);
        blinkerThread.Start();

        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnRedLightOn();
            BrainPad.Wait.Seconds(0.1);
            BrainPad.TrafficLight.TurnRedLightOff();
            BrainPad.Wait.Seconds(1);
        }
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Blinker()
    {
        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnGreenLightOn();
            BrainPad.Wait.Seconds(0.2);
            BrainPad.TrafficLight.TurnGreenLightOff();
            BrainPad.Wait.Seconds(0.2);
        }
    }
}
```

Example 11 – Here we create a method called Blinker that will be used for threading.

The previous program will blink the green light. Stepping through code, we can easily see how the Blinker method never returns execution to `BrainPadSetup()`. The program

keeps looping infinitely inside the Blinker method. But most programs would probably need to blink the light while doing something else. This is where threads come in very handy.

First, we need to construct a `Thread` object (Example 12). This object has special internal control over the program flow.

```
Thread blinkerThread = new Thread(Blinker);
```

Example 12 – This code constructs a `Thread` object that uses the `Blinker` method.

Note how the names easily identify what they represent. The `blinkerThread` is a thread that handles the `Blinker` method. All we need to do is `Start()` the thread and the `Blinker` method will be executed. However, there is still an issue. The `BrainPadSetup()` method will reach the end, which will cause the program and all its threads to terminate. A temporary solution is to make the `BrainPadSetup()` method wait indefinitely is by using -1 milliseconds as shown in Example 13.

```
public class Program
{
    public void BrainPadSetup()
    {
        Thread blinkerThread = new Thread(Blinker);
        blinkerThread.Start();

        BrainPad.Wait.Milliseconds(-1);
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Blinker()
    {
        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnGreenLightOn();
            BrainPad.Wait.Seconds(0.2);
            BrainPad.TrafficLight.TurnGreenLightOff();
            BrainPad.Wait.Seconds(0.2);
        }
    }
}
```

Example 13 – This code runs the `Blinker` method in a thread.

In Example 14, while the green light is blinking in its own thread, the system can now go do other things like flash the red light really quickly once a second.

```
public class Program
{
    public void BrainPadSetup()
    {
        Thread blinkerThread = new Thread(Blinker);
        blinkerThread.Start();

        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnRedLightOn();
            BrainPad.Wait.Seconds(0.1);
            BrainPad.TrafficLight.TurnRedLightOff();
            BrainPad.Wait.Seconds(1);
        }
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Blinker()
    {
        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnGreenLightOn();
            BrainPad.Wait.Seconds(0.2);
            BrainPad.TrafficLight.TurnGreenLightOff();
            BrainPad.Wait.Seconds(0.2);
        }
    }
}
```

Example 14 – This code blinks the green and red lights at different speeds using threading.

Exercise

Write a program that blinks the yellow light once a second and sounds the buzzer shortly if the down button is pressed.

Events

If a program needs to turn a light on via a button press, that program will need to check the button's state indefinitely. How often should we check the button? What if the button was pressed and released before the check? If we check too fast the system cannot enter low power mode.

This is important for battery operated devices like circuit boards or mobile phones. If the phone was always fully on, the battery would not last more than a few minutes. The only way a mobile phone can last an entire day on a charged battery is by shutting off unneeded components (like turning the screen off).

The proper way to handle the button is to subscribe to an event that is fired when the button is pressed or released. The BrainPad's `BrainPad.Button.ButtonChanged` event allows us to subscribe using the `+=` symbols. Now every time a button is pressed the `Button_ButtonChanged` method is called as shown in Example 15.

```
public class Program
{
    public void BrainPadSetup()
    {
        BrainPad.Button.ButtonChanged += Button_ButtonChanged;
        BrainPad.Wait.Milliseconds(-1);
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Button_ButtonChanged(BrainPad.Button.DPad d, BrainPad.Button.State
state)
    {
        if (d == BrainPad.Button.DPad.Down)
        {
            if (state == BrainPad.Button.State.Pressed)
            {
                BrainPad.TrafficLight.TurnGreenLightOn();
            }
            else
            {
                BrainPad.TrafficLight.TurnGreenLightOff();
            }
        }
    }
}
```

Example 15 – This code shows how a button event can be used to turn lights on and off.

When typing, after you enter the `+=` symbols, Visual Studio will instruct you to press TAB to insert a pre-named event handler. After doing so, you'll want to press TAB again to generate the actual handler inside the class.

Activating a light on a button press can be done in a loop but then the system is always running. Using events in this example, the system is mostly sleeping (in low power mode). As the program shows in Example 16, the first thing it does is subscribe to the button event. The system sleeps until one of the buttons is pressed or released, at which point it wakes up and runs the `Button_ButtonChanged` method.

```
public class Program
{
    public void BrainPadSetup()
    {
        BrainPad.Button.ButtonChanged += Button_ButtonChanged;

        while (BrainPad.Looping)
        {
            BrainPad.TrafficLight.TurnRedLightOn();
            BrainPad.Wait.Seconds(0.2);
            BrainPad.TrafficLight.TurnRedLightOff();
            BrainPad.Wait.Seconds(0.2);
        }
    }

    public void BrainPadLoop()
    {
        // Declared but not used
    }

    public void Button_ButtonChanged(BrainPad.Button.DPad d, BrainPad.Button.State
state)
    {
        if (d == BrainPad.Button.DPad.Down)
        {
            if (state == BrainPad.Button.State.Pressed)
            {
                BrainPad.TrafficLight.TurnGreenLightOn();
            }
            else
            {
                BrainPad.TrafficLight.TurnGreenLightOff();
            }
        }
    }
}
```

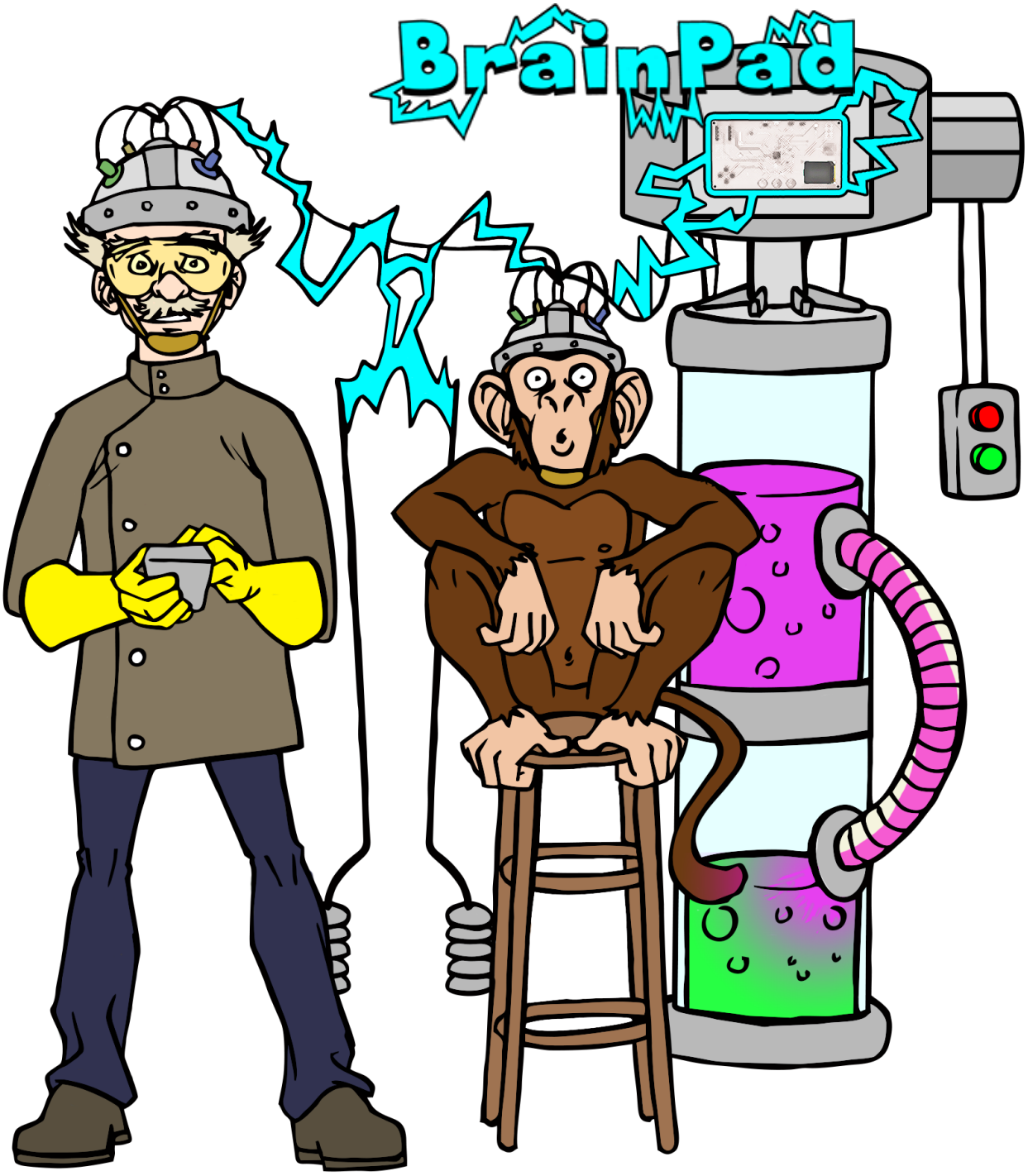
Example 16 – This code allows the system to sleep mostly. It only wakes up when a button is pressed to see if it needs to turn the green light on.

The arguments passed to `Button_ButtonChanged` show which button caused the event and whether it was pressed or released.

The previous program can now execute other tasks, like blinking the red light, while an event will handle the button press to turn the green light on whenever the down button is pressed.

Exercise

Blink the red light in a thread while an event checks if the down button is pressed to turn on the green light. The same event will check if the up button is pressed to make a short beep on the buzzer.



GHIElectronics.com/support/brainpad