# Unlocking .NET Micro Framework

**By John R. Malin and Sean D. Liming**
**SJJ Embedded Micro Solutions / www.sjjmicro.com**

Many of you have heard about Microsoft's new embedded offering, the Microsoft® .NET Micro Framework (.NET MF). Very simply, .NET Micro Framework allows you to develop C# application for small footprint devices. If you buy a system with .NET Micro Framework already running on a microcontroller board, than all you have to do is focus on the application part of an embedded system. Using the familiar Microsoft® Visual Studio development suite, C# applications can be developed to directly control hardware I/O. Robotics, security systems, tracking systems, industrial controls, temperature systems, and custom test equipment are just a few of the applications for the .NET Micro Framework. The tedious task of porting the low level device drivers, bootloaders, and other firmware is already been done for you. Some of you familiar with STAMP products may see some similarities.

With any new technology, education is critical to its success, which was the main goal behind our Embedded Development Kit (EDK) for the .NET Micro Framework. The EDK comes with a step-by-step guide that looks at each manage code class that provides access to hardware devices such as SPI access, GPIO, interrupts, storage, etc.

To help demonstrate .NET MF application development, we chose a combination lock example for this article. Robotic sail boats or industrial I/O examples were also thought of, but we wanted to discuss something simple and something that a wide audience already knows about. Of course, a similar project could be done with a PIC or STAMP, but we thought it might be interesting to show something written in C# where we can take advantage of feature like inheritance and reusable classes. In practice, a powerful platform such as .NET MF running on the iPAC-9302 would be used in significantly more complex applications, such as a robot controller.

Instead of a simple enter-the-combination-to-unlock-the-door solution, we decided to create a more feature rich example to demonstrate some of the .NET Micro Framework capabilities and help spawn other project ideas. In this example, we will have a menu driven user interface, different accounts with specific lock out codes, temporary accounts, store usage statistics, and have future support for network individual systems together. With this level of complexity, taking advantage of C# will be a little more appealing than writing all of the linear code in a primitive programming language.

**Feature of the Combination Door Lock Application**
The basic feature set is as follows:

1. Two lock code levels: user and supervisor.
2. Number of digits for each lock code is variable up to a predefined maximum number.
3. Number of allowed lock code retries is variable and set by supervisor.
4. The lock will enter a lockout period if the maximum number of illegal code retries is reached.
5. The lockout time after maximum retires is variable and set by supervisor.
6. The maximum number of lock codes will be predefined.
7. A special limited user lock codes will also be supported for user level lock codes and the number of code activations before retiring the lock code is set by the supervisor up to a predefined maximum.
8. The lock should record usage statistics:
    a. Record the number of times each lock code is used.
    b. Record the number of times the lock is accessed with an improper lock code.
    c. Record the number of times the lock goes into lock-out mode.

As we will see when we get into the code implementation, this seemingly simple set of features will still require a considerable amount of code to provide these features and also provide a reasonably intuitive user interface. It should, also, give you some ideas for enhancing it on your own, once you get this basic design functioning.

### Hardware need for Electronic Combination Lock

Obviously, we are going to use our EDK with EMAC's iPac-9302 hardware as the ARM9 processor board. To address the display and keypad requirements, we took advantage of the versatile SPI interface and nice little device that is a multi-line LCD display and controller with integrated keypad scan circuitry from Modtronix. In order to have a rich user interface experience, we will use the 4-line x 20-character display (Fig 2), which comes in 3 flavors, blue, amber, or green, all with backlights. We did start out using a 2 line display, but as we will discuss later, it wasn't right a fit for this application. .NET Micro Framework has support for graphical LCD panels, but the iPac-9302 is intended for headless or multiline display applications.

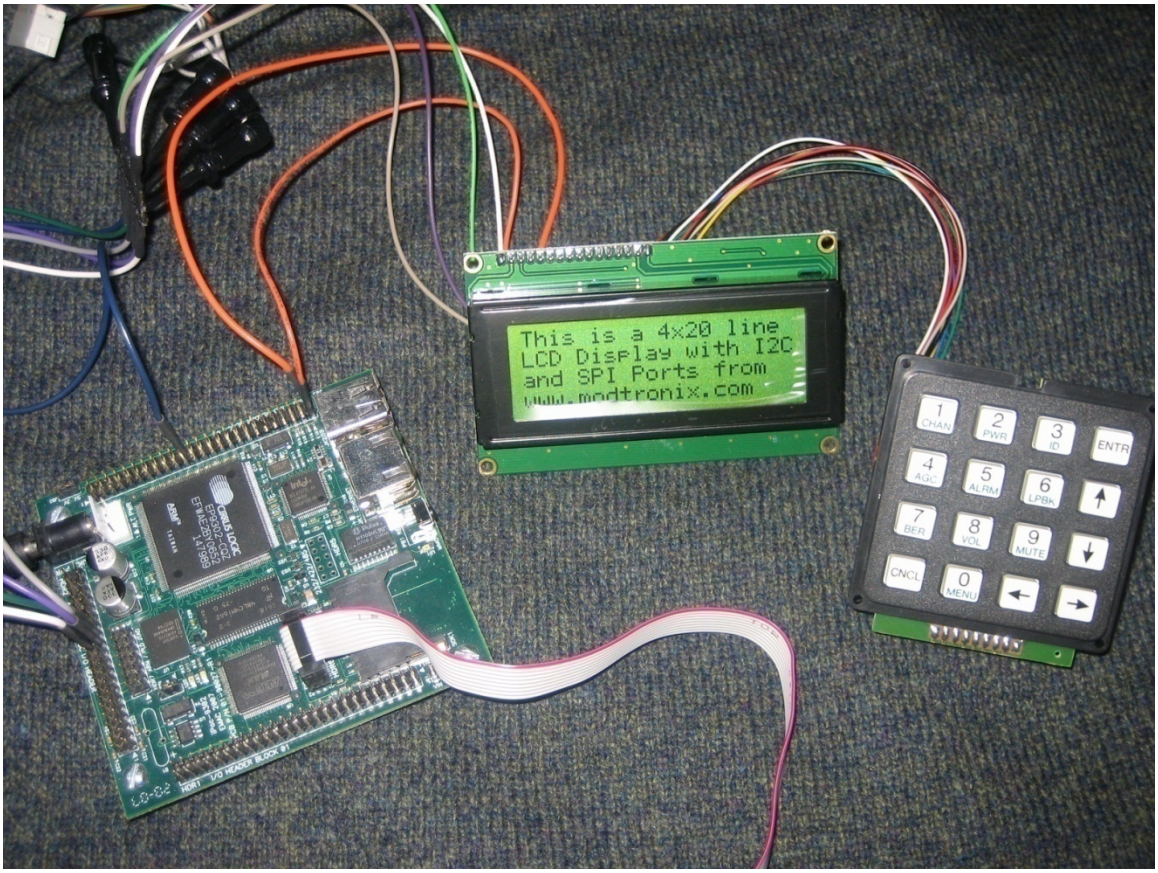


**Fig 1 – Modtronix 4 x 20 Display with Keypad**

We will also use the 16-button, 4 x 4 Modtronix KPAD16 keypad, **Error! Reference source not found.**which has keys for the digits 0 through 9, cancel and enter, and cursor up, down, right, and left keys.

Lastly, we need a solenoid locking mechanism that ideally can be run from a typical 5V logic power supply. One such mechanism is the Edwards 180 series, an electromagnetic mortise door locking mechanism, Fig 2, that will active using 4-6 VDC at 1.3-2.7 amps.

**Fig 2 – Edwards 180 Series Mortise Type Electromagnetic Door Release**

The circuit setup is simple. The SPI port is connected to the SPI-LCD controller. The 4x4 keypad is connected to the SPI-LCD keyboard pins.

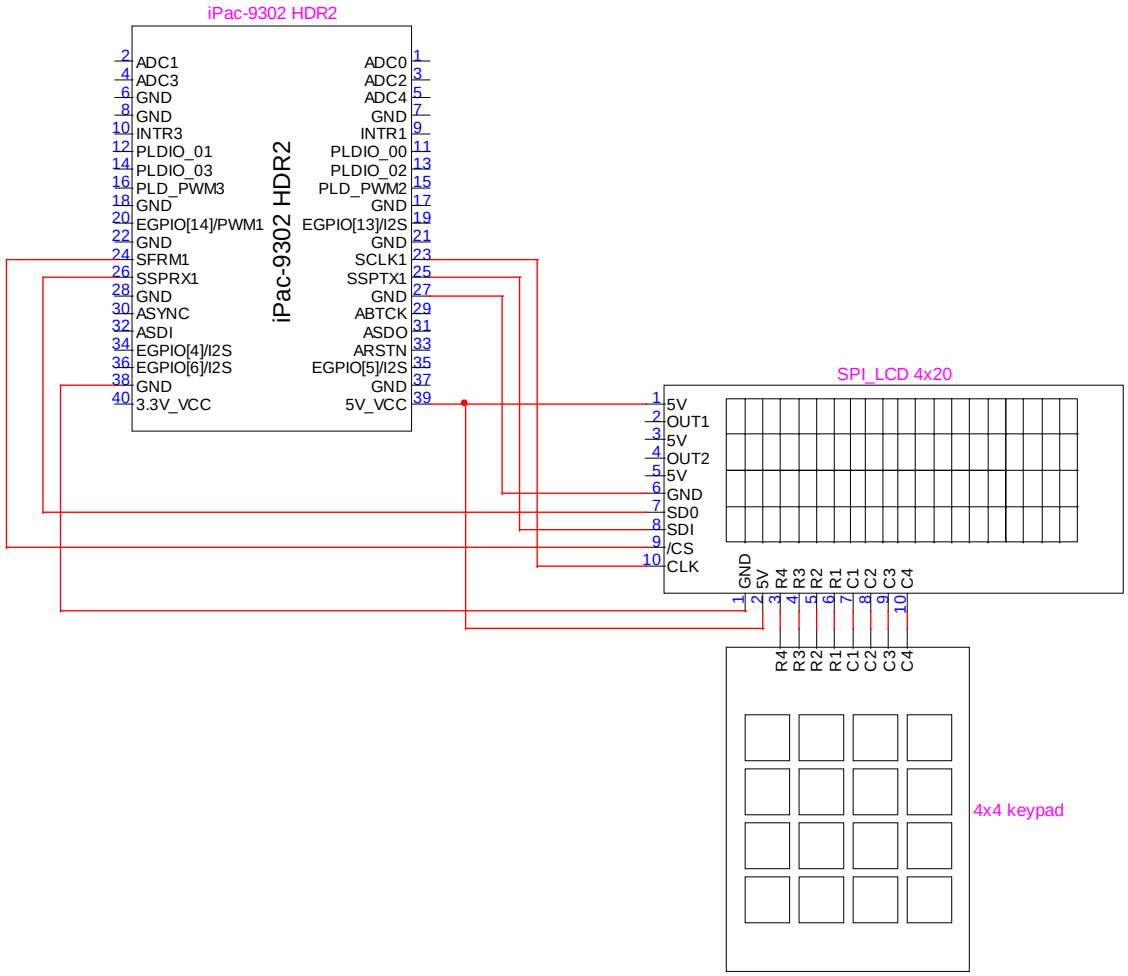


**Fig 3 Connection to the SPI port is simple for this circuit. Since no other SPI devices will be connected, an OR gate and GPIO are not required to generate the /CS signal**

The door lock mechanism itself requires a fair amount of current. Even though the iPac-9302 comes with high current GPIO, the amount of current to drive the lock exceeds the maximum output of 500mA. Knowing that the high current GPIO is active low, A Darlington PNP transistor circuit can be used to drive the solenoid. A diode is added to clamp the inductive kick from the solenoid.

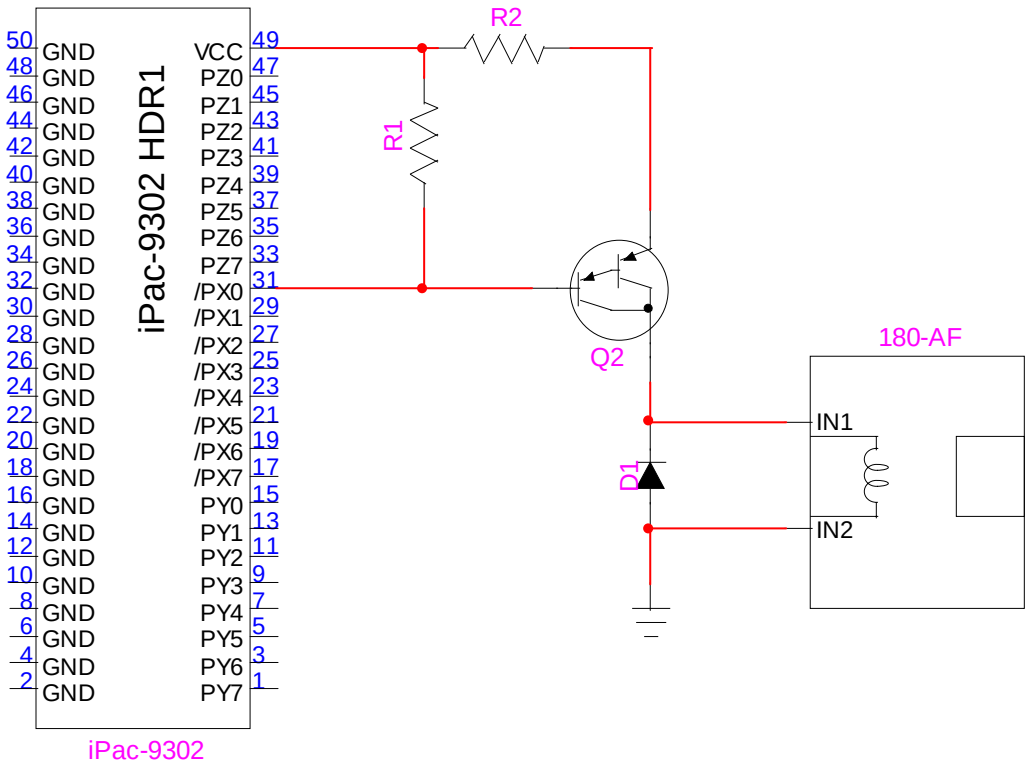


**Fig 4 Darlington Transistor provides the required high current drive to open the lock.**

If you find an electronic lock that requires higher input voltage or you need to use AC to drive the lock, then an optocoupler, solid state relay (SSR), or mechanical relay could be used. Fig 5, shows and example of using an optocoupler.

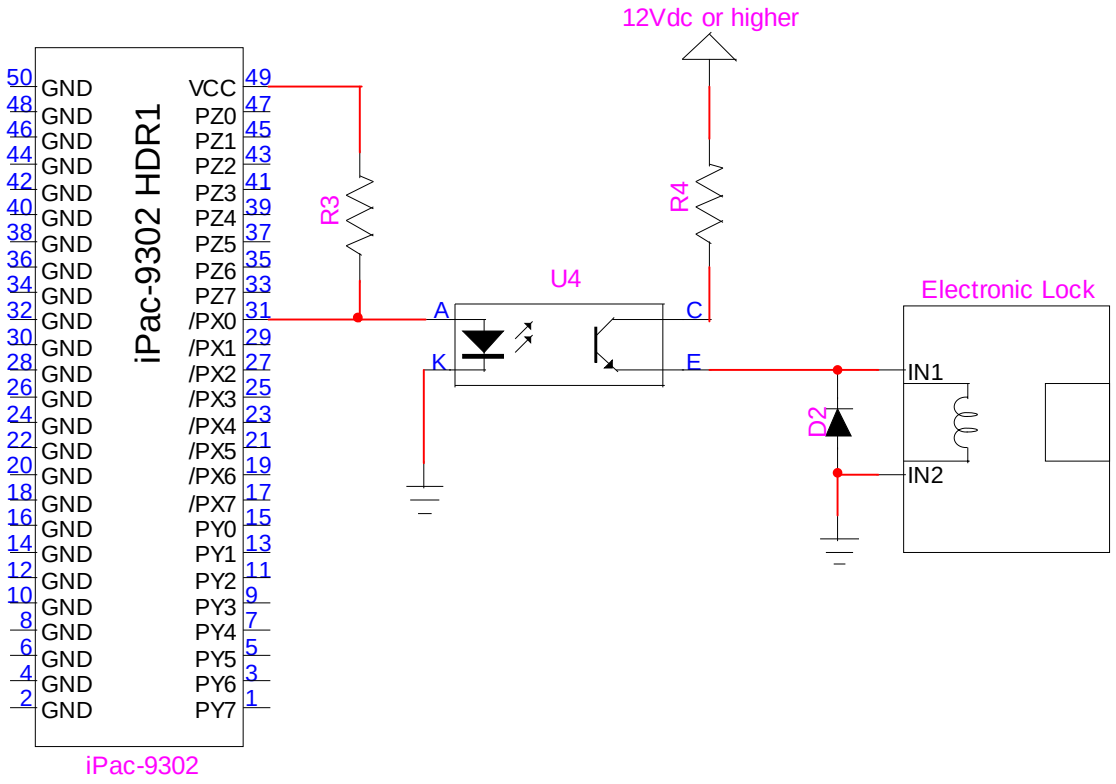


**Fig 5 If you need higher input voltage to activate the lock mechanism, an optocoupler can be used to connect the iPac-9302 to the lock.**

There are many solutions to connect the lock to the circuit. The main thing is that the application very simply toggles the GPIO to open and close the lock. The real coding is in the display menus and storage of the codes.

**Application State Machine Diagram**

Our first step was to put together a state machine diagram to layout the menu driven system and logic the application will execute.

Since our keypad has cursor keys for up, down, left, and right navigation, we can use a menu system and the cursor keys to scroll among options and the enter key to select the desired option. Each of these screens or menus can represent an underlying state of the lock and we can implement a state machine that will correlate the current state of the lock, make sure that the corresponding menu screen is displayed, and provide a simple means to navigate from screen to screen, and hence, state to state. For multiple functions, we can use a hierarchy of menus that we can move forward and back through to interact with the lock and a state machine is a simple and clean way to manage this. The other nice thing about using a state machine is that it compartmentalizes the operations of each screen/state and makes it easy to extend this design, later, and add new functionality and screens without worrying about breaking the existing code. Let's determine how many basic menu screens we need and how to organize them, see Fig 6.

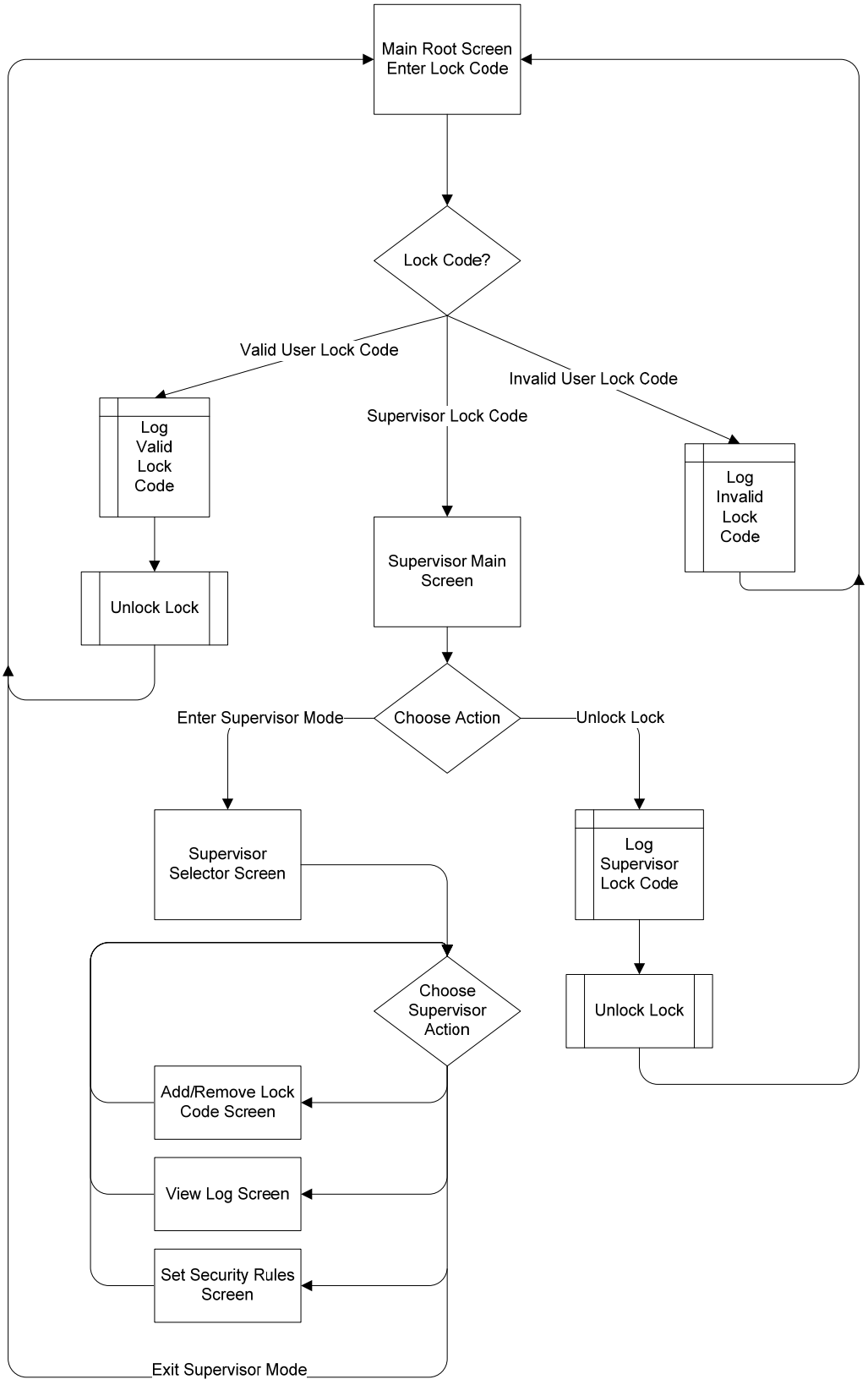


**Fig 6 - We can use a menu system and the cursor keys to scroll among options. The flowchart above indicates the structure of the lock software.**

From the main, root screen, if a user lock code is entered and recognized, then a success message will be displayed and the lock will be opened. If a supervisor lock code is entered, the base supervisor screen will be displayed. This screen will have two choices: unlock the lock or enter supervisor mode. If unlock is chosen, the lock will be unlocked as it was with a user lock code. If the supervisor mode is chosen, then the supervisor selector screen will be displayed, allowing the supervisor modes to be chosen. There will be a selection for adding or removing lock codes, both user and supervisor. There will be a selection to view the logged data, there will be a selection to set the security rules, and, finally, there will be a selection to exit the supervisor mode.

**Starting Point: Basic C# Application**

No matter what the hardware, the core application development is very much the same. Visual Studio is used to create a C# .NET Micro Framework application. The using statement is used to make coding with the namespaces easier. The application has a Main method that calls and runs a dynamic App.Run method. Figure 7 shows what he basic application would look like

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT.Hardware.SJJ;
using Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver;
using System.Threading;

namespace CombinationLock
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(
                Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();
        }

    }

    public class App
    {

        public App()
        {
        }

        public void Run()
        {
        }
    }
}
```

**Fig 7 - Program.cs ready to create the application code**

The only difference between our solution and someone else hardware is that we added our own names spaces for the EDK hardware and custom SPI LCD driver. More on the SPI LCD driver is coming up.

**Organizing the Code**

Once the basic application is setup, now we can take advantage of classes to organize our code. There are 4 main classes the site outside of the core App class:

- LockStateMachine – This is the heart of the control program, so we can encapsulate the definitions of the various lock states that are needed and the managing of the current lock state into a class. There is not much to this code.

- DispScreen - As we go from state to state, we have screens that must be displayed and interacted with the keypad. We can encapsulate the screen management and actual painting of the screens in another this class. The advantage in creating a Display screen class is that if we want to add new features we can easily add new screens to the class.

- LockAccount - The most important feature is to be able to create lock accounts, each with its own valid lock code, monitor statistic, and store all this the data in flash.

- SPI_LCD_4LDriver – Finally, there is the interaction with the LCD screen and the keypad, which is through the SPI interface. We could work directly with the SPI interface class that the .NET Micro Framework provides, but it would make the code simpler and more intuitive to abstract that into a class driver that would provide basic methods to write strings to the LCD, manipulate the cursor, and read keypad input and encapsulate the raw SPI commands required to do this in a driver class.

**Inheriting a SPI LCD Driver**
When we were creating the shell for this application, we added an LCD driver resource, recall the line:

```
using Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver;
```

This driver comes straight from the EDK and it supports a 2 line x16 character display. As we looked into have menus and user interaction, this display was a bit limiting. The 4line x 20 character display was selected since it was using the same controller as the 2 x 16. This was a big advantage in developing this application. Instead of recreating the special driver for 4x20, we took advantage of C# and the basic concept of class inheritance to support 4x20 display. We created a new abstracted driver class for the 4x20 display but instead of creating an entire new class, we based the new driver class on the original 2x16 driver class taking advantage of inheritance in .NET. This way we can take advantage of the driver code already developed and then add the functionality needed to support the 4x20 display. The new derived driver class will be called: SPI_LCD_4LDriver, and it will be based on the original SPI_LCDDriver.

There are some changes that are required. We have methods in the base class driver that can write any length of string to the display whether the result will wrap to the following line(s) or not. We also have methods that trim the write string and insure that it will not overflow the line and wrap to the next line, but the truncation of the string is hard coded for the 16 character line. We also have the ability to write to 2 lines, not 4 lines. What we needed to add to the base class is a new string truncation routine that will truncate to the 20 character line and then provide a method that will allow us to specify any of the 4 lines to write to. What we used from the base driver class is the method to write non-truncated strings to the first or second lines. We also used the display clear and cursor manipulation resources of the base driver class. If you look at the source code with the article, the SPI_LCD_4LDriver at line 756 shows code to access the 4 line LCD.

**Creating the Screens: DispScreens**
Although we are calling this the display screen class, it really manages both the LCD display and the keypad. The LCD display and the keypad are managed by the same controller and interfaced through the same SPI connection. The DispScreen class has a collection of constants that are strings and characters that will be used to create the text displayed on each of the screens and to create special characters like the right arrow prompt, asterisk, exclamation and question marks, etc. The DispScreen class has a series of methods that will manage input and output and synchronize cursor positioning. Also, there are other methods included for display and cursor manipulation. You can see the whole list of constants and methods starting at line 867 in the source code.

**Creating the Lock Account Class**
Because we want to have different types of lock account codes, unlimited user accounts, limit user accounts, and supervisor accounts; and because we want to capture some statistics for

each individual account, we can create a general account class that will define and capture these account qualities and provide the methods to manipulate each account so we can create, delete, and examine each valid lock code. The account class will be called: LockAccount. This class will also demonstrate the flexibility of C# in that it allows nesting of classes. Within the account class we created a more specialized class that just deals with each of the lock code specifically. This class with be called: ValidLockCode, and it will encapsulate the properties of each valid lock code and provide special access methods for the accounts. The LockAccount class starts at line 1974 in the source code.

For this combination lock program to work properly, we need to be able to add and remove valid lock codes and keep some statistics about each lock code's properties and usage. The LockAccount Class with its nested ValidLockCode Class gives us the means to do this. Not only do we need to keep track of lock codes when the lock program is running, but we also need to persist all this information through a power cycle, i.e., we need to have the lock code data store in some non-volatile storage somewhere and it needs to be retrievable when the program boots up, again, after power has been switched off and then back on. It would not be very handy if every time we powered the system down, we had to enter in all the lock codes and lock code properties, again, when we powered the system back on.

Since this is an embedded system there is no hard drive or file system support available. There is local flash storage through a unique .NET Micro Framework feature called ExtendedWeakReferences (EWR). The feature is based on a little know WeakReference solution found in .NET Framework. As you can guess, the documentation for EWR is a limited, and it would take some time to discuss the EWR topic in detail, which we have done in our step-by-step guide.

The first line defines a EWR as myAppData, which is the structure to be saved. The next line defines a static class called TypeAppDataFlag. Think of this as the file name for the data. The next two classes, the theNonVolatileData Class and the ValidLockCode Class have the Serializable designation. theNonVolatileData Class is a special class that we created to hold the data that we want to store in non-volatile, flash, memory. Any objects that are referenced by objects in a serializable object must also be serializable, so we must add the `[Serializable]` field to the ValidLockCode Class as well.

Finally, there are two methods to save the data, SaveData(), and retrieve the data, GetData(). If you look at the GetData() method you will see that this method attempts to recover the saved data from flash, but if it can't, it creates default settings that will be used for a first-time boot of the system.

One item to note: the iPac-9302 comes with support for MMC/SD, but at the time of this writing support was not available. Writing to the MMC/SD could offer the ability to write a duplicate set of data to removable media.

The rest of the methods in the LockAccount class deal with account access.

**App Run Method: Main body of the application**

The entire program runs in an infinite While-loop under the App class' Run method. In the While-loop are a couple of Switch-case statements that monitor the user input. When a change in state occurs the

## Visual Studio and .NET Micro Framework SDK

Having worked with various processors, operating systems, and debug/development tools, it's nice to have .NET Micro Framework application development integrated into Visual Studio. There is no need to learn a different tool chain.

Developing a .NET MF application in Visual Studio is no different than any other application development. Best of all, the application can be remotely debugged while it is running on the hardware. Multiple break points can be set, variables can be set in the watch, and you can step

line by line through the application. This was important when stepping through the code and finding logical errors in the state machine and in the classes.

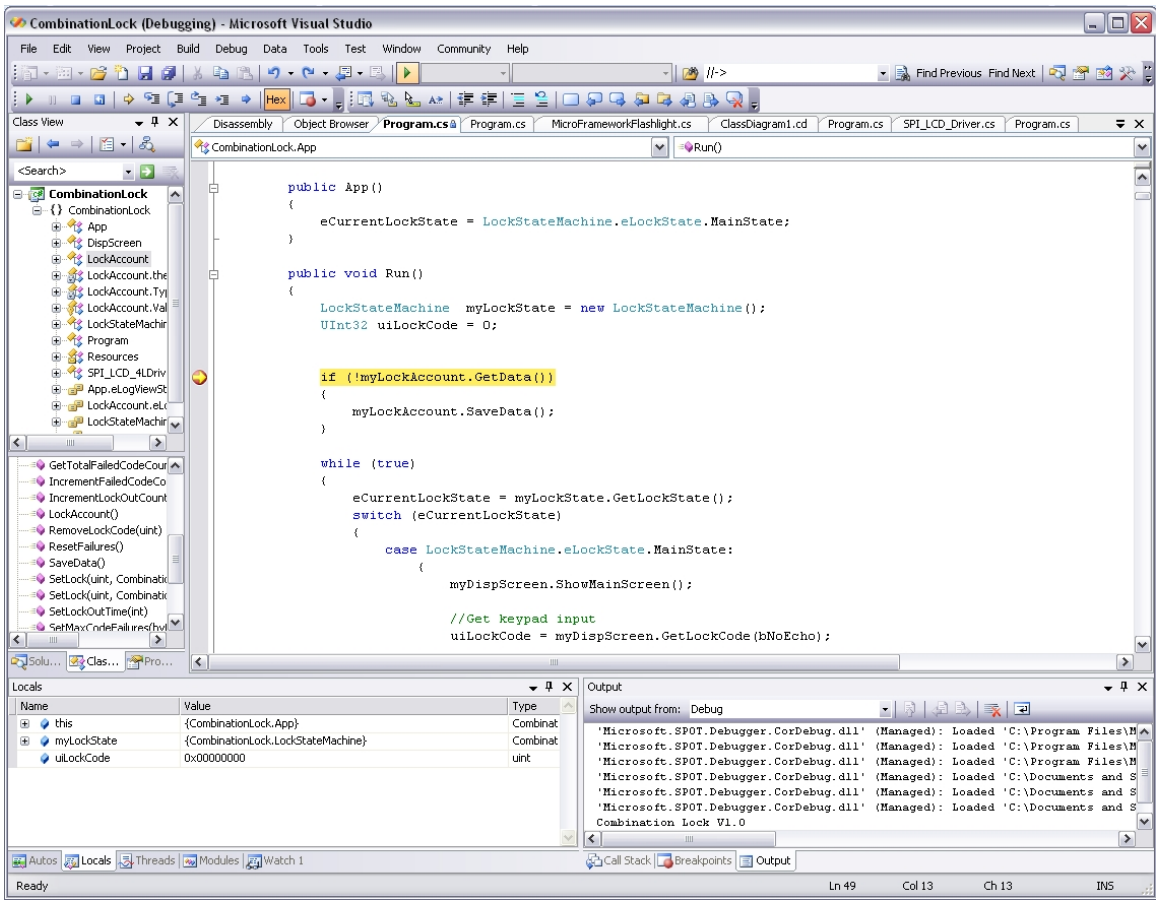


**Fig 8 - Visual Studio Debug Session with Output Window Showing**

In order to write .NET Micro Framework applications, you will need

- Visual Studio 2005 with Service Pack 1 (SP1)
- the .NET Micro Framework SDK. The .NET MF Software Development Kit (SDK) is a free download from MSDN

The .NET MF SDK contains assembly files, documentation, example applications, and development tools.

**Running the application**
Once everything is working, the menu system lets the user easily access the lock and other menu items.


**Fig 9 - Different states of the application**

Although we used this for a combination lock, the menu system and data storage can be applied to any type of embedded system or research project.

**Even More Capability**

We could add more capability to the lock system. For example, the lock could send the data to a back room server via networking or wireless connectivity. For auditing purposes, security personnel could audit the local logs with the logs on the server, thus adding a more sophisticated level of security. We could add sensors or cameras to monitor if anyone is in the room or if the lights are on or off. .NET MF also supports graphical LCD where applications can be built with the Windows Presentation Foundation classes. The multiline display can be replaced with something with a richer interface.

**Summary:**
The popular .NET programming environment is now available for those wanting to build small footprint devices. In this article, we put together an application demonstrating the different features of .NET Micro Framework and the advantages of writing in manage code using C#. With general purpose platforms that have .NET Micro Framework, like our EDK featuring the iPac-9302 from EMAC, Inc., it is easy to write and test applications just like writing applications for the desktop or mobile devices. With direct access to hardware, you can prototype or build a variety of different types of projects.