Freescale Semiconductor
Application Note

Document Number: AN3888
Rev. 0, 08/2009

Advanced Development with Microsoft®.NET Micro Framework 2.0

by *Multimedia Applications Division*
Freescale Semiconductor, Inc.
Austin, TX

This document describes how to perform advanced development tasks to create applications using an i.MX platform and the Microsoft® .NET Micro Framework.

These tasks (with examples) are as follows:

- GPIO pin configuration
- Working with threads, events, and persistent data
- Working with GUI elements
- Working with Windows® SideShow® applications

Before reading this document, it is suggested that users read the *Development with Microsoft® .NET Micro Framework 2.0* (AN3887) Application Note, which provides information and examples for the keyboard, certain interrupts, and the graphical user interface.

Contents

1. GPIO Pin Configuration 2
1.1. Identify CPU Pins 2
1.2. Configure Platform Pins 2
1.3. Example: Serial Communications with GPIO 6
2. Threads 7
2.1. Multithreading 7
2.2. Implement a Thread 7
3. Events 8
3.1. Define the Delegate and Variable 9
3.2. Implement and Assign the Function 9
3.3. Example: Events, Threads, and GUI Usage 10
4. Persistent Data 14
4.1. Create Storable Data 15
4.2. Example: Persistent Data 15
4.3. Create a Flash Reference 16
4.4. FlashReference Class 16
5. GUI Applications 17
5.1. Standard UI Elements 18
5.2. Using UI Elements 18
5.3. Use the Screen as a Bitmap 24
6. SideShow Applications 28
6.1. SideShow Enhanced Display Types 28
6.2. Example: SideShow Applications 28

1 GPIO Pin Configuration

General Purpose Input/Output (GPIO) ports are sets of single-bit I/O pins that are used for input or output. Embedded applications use GPIOs for controlling items such as switches, LEDs, as well as for handling button presses. This section describes how to program the GPIOs and provides an example for each concept.

1.1 Identify CPU Pins

Pin configuration is performed using the **InputPort**, **InterruptPort** and **OutputPort** classes, located in the **Microsoft.SPOT.Hardware** namespace. Each class is specific to its required pin configuration. Pin information is provided by the i.MX board manufacturer through a static class called **Pins**, which contains multiple **Cpu.Pin** constant definitions that are used to associate specific pins with a unique pin ID.

Pin identification numbers (IDs) are consecutive, where:

- GPIO Port A Pin 0–31 numbers are 0–31
- GPIO Port B Pin 0–31 numbers are 32–63
- GPIO Port C Pin 0–31 numbers are 64–95
- and so on

If pin number is needed that is not defined in the **Pins** class, update the **Pins** class by adding the definition of the pin and associating its corresponding ID as defined above.

1.2 Configure Platform Pins

Freescale recommends configuring the pins for a particular platform in a separate <**Platform**>**Pins** class where <**Platform**> refers to the name of the platform. For example, the MXSDVK platform contains a class called **MxsdvkPins**, which is defined as follows:

```
public static class MxsdvkPins {
        // Buttons
        public const Cpu.Pin leftButton = Pins.GPIO_PORT_B_17;
        public const Cpu.Pin rightButton = Pins.GPIO_PORT_B_15;
        public const Cpu.Pin upButton = Pins.GPIO_PORT_B_14;
        public const Cpu.Pin downButton = Pins.GPIO_PORT_B_16;
        public const Cpu.Pin selectButton = Pins.GPIO_PORT_B_18;
        public const Cpu.Pin rewindButton = Pins.GPIO_PORT_A_12;
        public const Cpu.Pin fastforwardButton = Pins.GPIO_PORT_A_13;

        // LEDs
        public const Cpu.Pin greenKeypadLed = Pins.GPIO_PORT_D_9;
}
```

The **MxsdvkPins** class substitutes user-provided names for the available GPIO pins on the MXSDVK platform. This enables, for example, users to refer to the Left Keypad Button pin as **MxsdvkPins.leftButton**, rather than by the cryptic term **Pins.GPIO_PORT_B_17**.

1.2.1 Configure an Input Pin

The **InputPort** class in the **Microsoft.SPOT.Hardware** namespace configures a pin as an input pin with an optional glitch filter and pull up/down resistor. The glitch filter is used to avoid the bounce effect caused when the pin is connected through a path with mechanical state changes. Examples of mechanical switches include keyboards, push buttons, and relays.

Use these steps to configure an input pin:

1. Identify the pin. Examine the board schematic and select a pin to configure as an input. Refer to the platform schematic to select an available (non-protected) GPIO.
2. Confirm the number to associate with the pin. See Section 1.1, "Identify CPU Pins," to identify the number to be associated with the pin. In the following example, PORTC pin 5 is associated with pin ID 69 on the device:
   ```
   public const Cpu.Pin GPIO_PORT_C_5 = (Cpu.Pin)69;
   ```
3. Update the <**Platform**>**Pins** class and add an intuitive name for the pin:
   ```
   public const Cpu.Pin sensorInput = Pins.GPIO_PORT_C_5
   ```
4. Declare the object that references the **InputPort**. It is typically defined as private for the class in which they are used:
   ```
   private InputPort sensorInputPin;
   ```
5. Define the object and configure the pin. Initialization options vary depending on the pin requirements:
   ```
   sensorInputPin = new InputPort(MxswdvkPins.sensorInput, true, Port.ResistorMode.PullUp);
   ```

   NOTE

   The glitch filter can be enabled if the input pin is connected to a mechanical switch.

   The resistor mode can be specified based on the normal state required for the pin. However, if the i.MX does not support the specified resistor mode, then the resistor mode has no effect on the pin. Define the resistor mode that minimizes current draw to the circuitry around the pin. Select the resistor mode carefully, because the read value on the pin may be inaccurate if the pin is connected to isolated or high impedance circuitry.

   Attempting to configure a pin that is already configured causes an exception. To reconfigure a pin, first call the **Dispose()** method.
6. Read the pin using the **Read()** method of the **InputPin** object:
   ```
   if (sensorInputPin.Read()) runInputAction();
   ```
7. Deallocate resources to disable the pin for a port and mark it as available for reuse:
   ```
   sensorInputPin.Dispose();
   ```

1.2.2 Configure an Interrupt Pin

The **InterruptPin** class is derived from the **InputPort** class. In addition to the characteristics and behaviors inherited from the **InputPort** class, the **InterruptPort** class publishes an event that can be configured to trigger on a change of the level or edge of the pin.

Use these steps to configure an interrupt pin:

1. Identify the pin. Examine the board schematic and select a pin to configure as an input. Refer to the platform schematic to determine an available (non-protected) GPIO.
2. Confirm the number to associate with the pin. See Section 1.1, "Identify CPU Pins," to identify the number to be associated with the pin. In the following example, PORTC pin 6 is associated with pin ID 70 of the device:

   ```
   public const Cpu.Pin GPIO_PORT_C_6 = (Cpu.Pin)70;
   ```
3. Update the <**Platform**>**Pins** class and add an intuitive name for the pin:

   ```
   public const Cpu.Pin leftButton = Pins.GPIO_PORT_C_6
   ```
4. Declare the object and define an object where the pin is referenced in the code:

   ```
   InterruptPort leftButtonPin;
   ```
5. Define the object and configure the pin. Initialization options vary depending on the pin requirements:

   ```
   leftButtonPin = new InterruptPort(MxswdvkPins.leftButton, true,
   Port.ResistorMode.PullUp, Port.InterruptMode.InterruptEdgeHigh);
   ```

   See Section 1.2.1, "Configure an Input Pin," for important information about the glitch filter, resistance mode, and accidental pin re-configuration. Select the interrupt mode carefully. Take into account the resistor mode in order to avoid false interrupts.
6. Configure the interrupt handler. Create a function of type **GPIOInterruptEventHandler**, and then add this function to the event **OnInterrupt** of the **leftButtonPin** object.

   ```
   leftButtonPin.OnInterrupt += new GPIOInterruptEventHandler(leftButtonPin_onInterrupt);
   ```

   NOTE

   Multiple event handlers can be assigned to a particular event. The += and –= operators are used to subscribe and unsubscribe events, respectively.
7. Define the event handler:

   ```
   void leftButtonPin_onInterrupt(Cpu.Pin port, Boolean state, TimeSpan time) {
         Debug.Print("leftButtonPin interrupt fired");
   }
   ```
8. Clear the interrupt. If a level interrupt event is configured in Step 4, call the **ClearInterrupt()** method of the object:

   ```
   leftButtonPin.ClearInterrupt();
   ```

NOTE

The **ClearInterrupt()** method is typically called at the beginning of the event handler code in order to enable the interruption immediately. For a level interrupt configuration, the execution of the **ClearInterrupt()** can be delayed to create more complex event handling mechanisms. For example, when a key is pressed, the event handler code could process the key immediately and then start an asynchronous timer to delay the execution of the **ClearInterrupt()** function. This allows users to set an interval for key press repetition.

9. Read the pin using the **Read()** method of the **inputPin** object:

   ```
   if (sensorInputPin.Read()) runInputAction();
   ```

10. Deallocate resources of the pin to disable the pin for a port and mark it as available for reuse:

    ```
    sensorInputPin.Dispose();
    ```

1.2.3 Configure an Output Pin

The **OutputPort** class of the **Microsoft.SPOT.Hardware** namespace sets the value of a GPIO pin by specifying the initial state of the pin. Configure the state of the pin in runtime by calling the **Write()** function of the pin object.

Use these steps to configure an output pin:

1. Identify the pin. Examine the board schematic and select a pin to configure as an input. Refer to the platform schematic to select an available (non-protected) GPIO. Select the pin based on ease of identification, accessibility, and value of the current. The i.MX microprocessors limit the current provided to the pins.
2. Confirm the number to associate with the pin. See Section 1.1, “Identify CPU Pins,” to identify the number to be associated with the pin. In the example below, PORTC pin 7 is associated with pin ID 71 on the device:

   ```
   public const Cpu.Pin GPIO_PORT_C_7 = (Cpu.Pin)71;
   ```

3. Define an object where the pin is referenced in the code:

   ```
   OutputPort outputPin;
   ```

4. Initialize the object and configure the initial value for the pin. Initial states should be **true** for logic 1 and **false** for logic 0:

   ```
   outputPin = new OutputPort(Pins.GPIO_PORT_C_7, false);
   ```

NOTE

> The **OutputPort** class provides a constructor for additional pin configuration options, such as the glitch filter and resistor mode. Configuration is similar to the configuration of **InputPort**. The recommended initial state is false logic 0, to prevent lack of current at the start up and in extreme situations damage of the i.MX due to the current demand. Attempting to configure a pin that is already configured causes an exception. Call the **Dispose()** method prior to reconfiguring a pin.

5. Read the pin using the **Read()** method of the **outputPin** object:

   ```
   if (outputPin.Read()) runOutputAction();
   ```

6. Write the pin. To force a value on the pin, use the **Write()** method of the **outputPin** object, where **true** represents a logical 1 and **false** represents a logical 0.

   ```
   outputPin.Write(true);
   ```

7. Deallocate resources of the pin to disable the pin for a port and mark it as available for reuse:

   ```
   sensorInputPin.Dispose();
   ```

1.3 Example: Serial Communications with GPIO

This example describes the initialization, configuration, and use of three GPIO pins. The first pin is configured as an input, the second as an interrupt input, and the third as an output.

1. Add a reference to the namespace that contains the classes to be used:

   ```
   using Microsoft.SPOT.Hardware;
   ```

2. Define the objects that are used in this example:

   ```
   InputPort inputPin;
   InterruptPort leftButtonPin;
   OutputPort outputPin;
   ```

3. Initialize and configure the pin objects:

   ```
   inputPin = new InputPort(Pins.GPIO_PORT_C_5, true, Port.ResistorMode.PullUp);
   leftButtonPin = new InterruptPort(Pins.GPIO_PORT_C_6,
         true, Port.ResistorMode.PullUp, Port.InterruptMode.InterruptEdgeHigh);
   outputPin = new OutputPort(Pins.GPIO_PORT_C_7, false);
   leftButtonPin.OnInterrupt +=
         new GPIOInterruptEventHandler(inputPinInterrupt_onInterrupt);
   ```

   The inputPinInterrupt_onInterrupt function is defined as follows:

   ```
   private void inputPinInterrupt_onInterrupt(Cpu.Pin port, Boolean state, TimeSpan time) {
         Debug.Print("inputPinInterrupt interruption");
         //leftButtonPin.ClearInterrupt();
   }
   ```

NOTE

> The **ClearInterrupt()** method is not needed in this case, because the **leftButtonPin** is configured as an edge interrupt.

4. Use the objects. Use **Read** mode for the three pin objects of this example and **Write** mode (only) for the outputPin object. For the **leftButtonPin** object, use the interrupt(s) assigned to the **leftButtonPin** object:

   ```
   if (inputPin.Read()) ...
   if (!leftButtonPin.Read()) ...
   outputPin.Write(!outputPin.Read());
   ```

5. Dispose of the objects:

   ```
   String legend = "Data saved no: " + deviceLog.Log.Count.ToString();
   deviceLog.AddToLog(new Device((byte)deviceLog.Log.Count, legend));
   ```

6. Save the object in Flash:

   ```
   deviceLogFlash.save(deviceLog);
   ```

2 Threads

In development, the term thread means a thread of execution. Threads are a way for a program to fork (or split) itself into two or more, simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another but, in general, a thread is contained inside a process, and different threads in the same process share some resources, while different processes do not.

2.1 Multithreading

Although the .NET Micro Framework can execute only one application at a time, it provides a multithreading functionality that allows execution of multiple threads in parallel. The .NET Micro Framework assigns CPU time to the threads depending on the priorities specified for the threads themselves, and for the priorities of the threads that execute the call. Additionally .NET Micro Framework provides ways to suspend, resume, and sleep the thread.

2.2 Implement a Thread

Threads are managed through the **Thread** class in the **System.Threading** namespace. The **Thread** class creates and controls a thread, sets its priority, and gets its status. This section explains how to create, prioritize, start, sleep, suspend or stop, and verify the state of threads.

2.2.1 Create a Thread

To create a thread, assign a function (without parameters or return data) to the creator of the thread object. The created function contains the code for the thread. The thread code can call other functions and reference shared data, such as global variables, registers, GPIOs, or serial ports.

```
Thread t1 = Thread(new ThreadStart(thread1));
```

The thread object is managed through **t1** and the thread code is in the **thread1()** function.

For shared data in threads, implement a synchronization process (such as flags, semaphores, mutex, queues) in order to read/write valid data.

2.2.2 Prioritize a Thread

Assign a priority to the thread in order to assign more MCU time to one thread than another:

```
t1.Priority = ThreadPriority.Highest;
```

It is recommended to assign higher priority to threads in critical processes, or to threads which have a quick process requirement.

2.2.3 Start a Thread

After defining a thread object for the function to run as a thread, execute the thread using the **Start()** function:

```
t1.Start();
```

2.2.4 Sleep a Thread

The **Thread.Sleep** method suspends the current thread for a specified time. A running thread can typically sleep for a period of time because it has no processes to perform. To suspend the thread, use the **sleep()** static function. A static sleep function affects only the running thread. Use sleep to suspend the main application. For example:

```
Thread.Sleep(5000);                // Sleep current thread for 5 seconds
Thread.Sleep(0);                   // Exit from the current thread until next MCU time assigned
Thread.Sleep(Timeout.Infinite);    // Sleep current thread forever
```

2.2.5 Suspend a Thread

To suspend a thread for an indeterminate amount of time, use the suspend and resume functions of the thread object:

```
t1.Suspend();
t1.Resume();
```

The **Suspend** function takes effect in the next sequential execution of the thread. That is, if the **Suspend** function is called, the thread code continues executing until the MCU time dedicated to that thread ends. To exit the thread, execute the **Thread.Sleep(0)** instruction after the **Suspend()** instruction.

2.2.6 Verify the State of a Thread

Verify the state of a thread through the **ThreadState** property of the thread object. For more information about available states and invocations, see the .NET Micro Framework Help.

```
if ((t1.ThreadState & ThreadState.Suspended) == ThreadState.Suspended)
// do something
```

3 Events

An event is an interruption to the system. The source of the interruption varies: key press, timeout, data received, network found, information request, and so forth. An event is captured and processed by the system to take a corresponding action. To use events, specify the functions to initiate when an action occurs. For example, assign a function at the application level to be called when serial data is received and decoded.

A common example of event implementation is found in the serial communications process of encoding and decoding data into frames at different layers. While many solutions are possible, events and threads are useful in allowing the main loop of the program to focus on the application, while delegating the serial communications to a communications module. Thus, the module interrupts the main loop only when required, such as when data is received and decoded.

The .NET Micro Framework functionality makes it easy to use events and threads to develop a system-independent module for serial communications and frame specifications. Libraries (DLLs) are used to include events and threads in an application.

In event communication, a delegate (comparable to a function pointer type) identifies the object or method that receives the event. The delegate is a class that can hold a reference to a method, and specifies the event parameters. Event usage is a two-part process: define the delegate and variable, and then implement and assign the event function.

3.1 Define the Delegate and Variable

Use these steps to define the delegate and variable:

1. Define the delegate. Delegate definitions accept common .NET Micro Framework delegates or custom delegates. For custom delegates, specify the parameter types and return values. For example:

   ```
   delegate void InterruptDelegate(Object sender, String description);
   ```

2. Define the event variable using the event word. Event variable definitions accept common .NET Micro Framework delegates or custom delegates. For example:

   ```
   event InterruptDelegate OnLoad;
   ```

3. Apply the event variable to execute the assigned event code. Apply the event variable in a thread to execute the event code as a separate task. For example:

   ```
   if (OnLoad != null) OnLoad(this, "Loaded all the numbers");
   ```

3.2 Implement and Assign the Function

Use these steps to implement and assign the event function:

1. Define the event function with the same parameter and return types specified in the delegate (the parameter names may differ). For example:

   ```
   void OnLoadEvent(Object sender, String description) {
         Debug.Print(description);
   }
   ```

2. Assign the event. Use the += operand described in Section 1.2.2, “Configure an Interrupt Pin.” For example:

   ```
   OnLoad += new InterruptDelegate(OnLoadEvent);
   ```

3.3 Example: Events, Threads, and GUI Usage

This example, which includes threads, events, and GUI standard usage, uses the following classes:

- **Program** class starts a hard mathematical order process in the **HardMathEvaluation** class
- **HardMathEvaluation** class uses events to notify the **Program** class at the end of each part of the process

3.3.1 HardMathEvaluation Class

The **HardMathEvaluation** class defines one delegate and two events. The objective is to create and order a buffer of semi-random numbers. The class generates the numbers, exits the corresponding event, orders the numbers, and again exits the corresponding event.

The following steps show how the class is built:

1. Include namespaces for the class:

```
using System;
using Microsoft.SPOT;
using System.Threading;
```

2. Define data types and variables to use:

```
public delegate void InterruptDelegate(Object sender, String description);
public event InterruptDelegate OnGenerated;
public event InterruptDelegate OnOrdered;
private int[] buffer;
private Thread thread;
private int ordered;
public int Ordered {get { return ordered;}}
```

— **InterruptDelegate** defines the structure of the functions for the events **OnLoad** and **OnFinish**.

— The **buffer** variable stores the array of numbers (elements) and the ordered variable (accessible as property through **Ordered**) contains the number of elements already ordered.

— The **thread** variable is used for a process that takes a long time to run as another task.

3. Use a constructor to define the number of elements to order and the size of the buffer variable:

```
public HardMathEvaluation(int numElements) {
      buffer = new int[numElements];
}
```

4. Generate the numbers. The **generateNumbers** method creates a thread to work into the process as another task:

```
public void generateNumbers() {
      ordered = 0;
      thread = new Thread(new ThreadStart(generateNumbers_process));
      thread.Start();
}
private void generateNumbers_process() {
      for (int i = 0; i < buffer.Length; i++)
      buffer[i] = Microsoft.SPOT.Math.Random(Int32.MaxValue);
      if (OnGenerated != null) OnGenerated(this, "Generated all the numbers");
}
```

NOTE

Define the **Random** method in the **Math** class with the namespace, because there is another class with the same name in the **System** namespace. Specify which class to use.

5. Order the numbers. The **orderNumbers** method creates a thread to work into the process as another task:

```
public void orderNumbers() {
        thread = new Thread(new ThreadStart(orderNumbers_process));
        thread.Start();
}
private void orderNumbers_process() {
        int temp, j;
        for (int i = 1; i < (buffer.Length); i++) {
                j = i - 1;
                while ((j >= 0) && (buffer[j] > buffer[i])) j--;
                if (buffer[j+1] > buffer[i]) {
                        j = j + 1;
                        temp = buffer[i];
                        for (int k = i; k > j; k--) buffer[k] = buffer[k - 1];
                        buffer[j] = temp;
                }
                ordered = i + 1;
        }
        if (OnOrdered != null) OnOrdered(this, "Done! all the numbers are ordered");
}
```

3.3.2 Program Class

The **Program** class is based on the sample **Program** class provided by the Microsoft Visual Studio® 2005 development system, which is used to create a new project in the following folder:

```
C#/.Net Microframework/Window Application
```

The following steps show the changes made to the provided **Program** class in order to run this example:

1. Include the needed namespaces:

```
using System.Threading;
```

2. Define the types of data and variables to use:

```
private HardMathEvaluation orderNum;
private Text text1;
private Text text2;
private delegate void UpdateTextDelegate(Text IUElement, String text);
private Thread t1;
```

— **orderNum** object is the reference to the **HardMathEvaluation** class that defines and applies the events.

— **text1** and **text2** are the objects that display text in the screen.

— **UpdateTextDelegate** defines the structure of the asynchronous function to be executed for correct screen updating. For more information, see Section 5.1, “Standard UI Elements.”

— The variable **t1** is used to run the update screen process as a separate task.

The **Main()** method remains unchanged.

3. Initialize the variables. This step:
   - Modifies the **CreateWindow** method to create a **panel** object of type **Panel**
   - Initializes the **Text** objects
   - Adds the **Text** objects to the panel
   - Assigns the panel as child of the **mainWindow** object
   - Initializes the **orderNum** object
   - Assigns the events functions

   With these changes, the new **CreateWindow** method contains the following code (some comments are removed to save space):

```
public Window CreateWindow() {
       mainWindow = new Window();
       mainWindow.Height = SystemMetrics.ScreenHeight;
       mainWindow.Width = SystemMetrics.ScreenWidth;

       Panel panel = new Panel(); // Create a panel control.
       text1 = new Text();
       text1.Font = Resources.GetFont(Resources.FontResources.small);
       text1.TextContent = Resources.GetString(Resources.StringResources.String1);
       text1.HorizontalAlignment =
              Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
       text1.VerticalAlignment = Microsoft.SPOT.Presentation.VerticalAlignment.Center;
       text2 = new Text();
       text2.Font = Resources.GetFont(Resources.FontResources.small);
       text2.HorizontalAlignment =
              Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
       text2.VerticalAlignment = Microsoft.SPOT.Presentation.VerticalAlignment.Top;

       panel.Children.Add(text1);
       panel.Children.Add(text2);
       mainWindow.Child = panel;

       mainWindow.AddHandler
              (Buttons.ButtonUpEvent,new ButtonEventHandler(OnButtonUp), false);
       mainWindow.Visibility = Visibility.Visible;
       Buttons.Focus(mainWindow);

       // Create the HardMathEvaluation control and assign its events.
       orderNum = new HardMathEvaluation(1000);
       orderNum.OnGenerated +=
              new HardMathEvaluation.InterruptDelegate(OnGeneratedNumEvent);
       orderNum.OnOrdered +=
              new HardMathEvaluation.InterruptDelegate(OnOrderedNumEvent);
       return mainWindow;
}
```

NOTE

Compiling the code at this point produces errors because the methods with the **OnGeneratedNumEvent** and **OnOrderedNumEvent** events code are not yet defined.

4. Start the process. The process is defined to start when any button is pressed. For this purpose, additional lines are added to the **OnButtonUp** method. The changes are moved into an IF condition, to avoid the case in which a process is running, a key is pressed, and the process tries to run again without having finished the initial run. This IF condition verifies the value of the thread variable, and is used for the screen update.

```
if (t1 == null)
```

The added lines start the hard mathematical order process, which continues through the **generateNumbers** method, and then starts the screen update thread content in the **t1** variable:

```
orderNum.generateNumbers();
t1 = new Thread(new ThreadStart(updateWindow));
t1.Start();
```

With the changes, the **OnButtonUp** method contains the following code (some comments are removed to save space):

```
private void OnButtonUp(object sender, ButtonEventArgs e) {
        Debug.Print(e.Button.ToString());
        if (t1 == null) {
                orderNum.generateNumbers();
                t1 = new Thread(new ThreadStart(updateWindow));
                t1.Start();
        }
}
```

NOTE

Compiling the code at this point produces errors because the thread **updateWindow** method is not yet defined.

5. Define the window update code. To correctly update the window content and view the screen changes, the instructions for visual changes in functions must be called asynchronously. Therefore, **UpdateTextDelegate** is used as a parameter in the asynchronous calls, and is defined with two parameters. The first parameter identifies the user interface text object to update, and the second represents the text to update.
6. Define the **UpdateText** method as follows, respecting the parameters and returning types defined by the delegate **UpdateTextDelegate**:

```
private void UpdateText(Text IUElement, String text) {
        IUElement.TextContent = text;
}
```

NOTE

Any function that respects the parameters and returns the types defined by the **UpdateTextDelegate** can be a function of that type.

7. Implement the **updateWindow** thread method. The **updateWindow** thread method is implemented to update the **text2** object with the number of elements already ordered. The call to the **UpdateText** method is made through the **Dispatcher.Invoke** method. Because this thread method must be running forever to update the text, the code is in a `while(true)` loop.

```
private void updateWindow() {
        while (true) {
```

```
                        text2.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateTextDelegate(UpdateText),
                                new object[] { text2, orderNum.Ordered.ToString() + " ordered" });
                }
        }
```

— The **TimeSpan** parameter represents the .NET Micro Framework delay between the execution of the **Invoke** instruction and the **UpdateText** method. In this case it is one second.

— The **object[]** parameter represents the parameters of the **UpdateText** method in array form.

8. Define the events code. There are two events in this example. One event is exited when the buffer is filled with semi-random number values. The second event is exited when the ordering process is complete.

   The **OnGeneratedNumEvent** event method is assigned to the **orderNum** object for the **OnGenerated** event in Step 3. Implement the function that starts the ordering process and updates a message on the screen. The function looks like this:

```
private void OnGeneratedNumEvent(Object sender, String description) {
        Debug.Print(description);
        orderNum.orderNumbers();
        text1.Dispatcher.Invoke(
                        new TimeSpan(0, 0, 1),
                        new UpdateTextDelegate(UpdateText),
                        new object[] {text1,Resources.GetString
                               (Resources.StringResources.String1)+":"+description});
}
```

   The **OnOrderedNumEvent** event method is assigned to the **orderNum** object for the **OnOrdered** event in Step 3. Implement the function that updates a message on the screen and suspends the **t1** thread, which updates the window with the numbers ordered. The function looks like this:

```
private void OnOrderedNumEvent(Object sender, String description) {
        Debug.Print(description);
        text1.Dispatcher.Invoke(
                new TimeSpan(0, 0, 1),
                new UpdateTextDelegate(UpdateText),
                new object[]{text1,Resources.GetString
                        (Resources.StringResources.String1)+":"+description});
        t1.Suspend();
        t1 = null;
}
```

   After suspending the **t1** thread, the variable is set to null to allow the system to run again by pressing any button. This is also done in Section 2.2.3, “Start a Thread.”

4 Persistent Data

Persistent data is that data stored in Flash memory for restoration as needed, such as configuration options, profiles, and control variables. This section explains how to maintain persistent data in .Net Micro Framework at the managed layer.

4.1 Create Storable Data

To be storable, data must be defined as serializable. Such storable data definitions include **String**, **int**, **byte**, **ArrayList**, and so on. A serializable class can also be created in order to enclose the storable data. Flash memory can be erased and saved a finite number of times. It is recommended to create a serializable class with the storable data, and save the memory only when necessary.

4.2 Example: Persistent Data

In this example, the **Device** class is defined as a serializable class with two variables, **name** and **id**. The application stores the data of these variables in Flash. If the application needs to store the information for more than one Flash device, create a log for the data using an **ArrayList** object, located in the **System.Collections** namespace, as the manager of the data.

```
[Serializable]
public class Device {
         private String name;
         private byte id;
         public String Name {
                  set { name = value; }
                  get { return name; }
         }
         public byte Id {
                  set { id = value; }
                  get { return id; }
         }
         public Device(byte Id, String Name) {
                  id = Id; name = Name;
         }
}
```

The **DeviceLog** class showed below, can be used to store a stack of data. The **DeviceLog** class can be updated with search functions to allow searching for a device by name or ID.

```
[Serializable]
class DeviceLog {
         private ArrayList log = new ArrayList();
         private Device lastDeviceUsed;
         public ArrayList Log {
                  get { return log; }
         }
         public Device LastDeviceUsed {
                  get { return lastDeviceUsed; }
                  set { lastDeviceUsed = value; }
         }
         public void AddToLog(Device device) {
                  log.Insert(0, device);     // this to have the last element added at the start
         }
         public void RemoveFromLog(Device device) {
                  log.Remove(device);
         }
         public void ClearLog() {
                  log.Clear();
                  lastDeviceUsed = null;
         }
}
```

4.3 Create a Flash Reference

The **ExtendedWeakReference** class in the **Microsoft.SPOT** namespace is used as reference to the data to be stored in and recovered from Flash.

To save and recover data from Flash, use these steps:

1. Add a reference to the namespaces where the classes to use are located:

   ```
   using Microsoft.SPOT;
   ```

2. Define the object that is used for the process:

   ```
   ExtendedWeakReference flashReference;
   ```

3. Load the objects to recover the data from Flash, or create the object in Flash:

   ```
   flashReference = ExtendedWeakReference.RecoverOrCreate(
          typeof(Program),                                     // marker class
          id,                                        // id number in the marker class
          ExtendedWeakReference.c_SurvivePowerdown);                     // flags
   flashReference.Priority = (Int32)ExtendedWeakReference.PriorityLevel.Important;
   Object data = flashReference.Target;                           // recovering data
   ```

   Data is null if this is the first time it is used, or if the data is damaged.

4. Save the data in Flash:

   ```
   flashReference.Target = data;
   ```

   To clear the saved data, use null instead of using the data.

4.4 FlashReference Class

The **FlashReference** class encloses the basic functionality of the **ExtendedWeakReference** class. The **FlashReference** class only needs a unique identifier for the object to load/save Flash:

```
public class FlashReference {
        private ExtendedWeakReference flashReference;
        private uint id;
        public FlashReference(uint Id) {
              id = Id;
        }
        public Object load() {
              flashReference = ExtendedWeakReference.RecoverOrCreate(
                          typeof(Program),                     // marker class
                          id,                        // id number in the marker class
                          ExtendedWeakReference.c_SurvivePowerdown);          // flags
              flashReference.Priority=(Int32)ExtendedWeakReference.PriorityLevel.Important;
              Object data = flashReference.Target;                // recovering data
              return data;
        }
        public void save(Object data) {
              flashReference.Target = data;
        }
}
```

4.5 Example: FlashReference Class

The following example uses the **FlashReference**, **DeviceLog** and **Device** classes used in Section 4.2, "Example: Persistent Data," and Section 4.4, "FlashReference Class." This example illustrates the use of the **FlashReference** class for the following tasks:

- Store serializable data in the Flash memory for the first time
- Recover the data from Flash
- Update the data
- Save data in the Flash again

To perform this example, use these steps:

1. Add a reference to the namespaces where the classes to use are located:

```
DeviceLog deviceLog;
FlashReference deviceLogFlash;
```

2. Define the object that is used for the process:

```
DeviceLog deviceLog;
FlashReference deviceLogFlash;
```

3. Load the object to recover the data from Flash. The number 0 (zero) is the identification of the deviceLog object in Flash. Select a number as required, to identify the storable data.

```
deviceLogFlash = new FlashReference(0);
deviceLog = deviceLogFlash.load() as DeviceLog;
```

4. Initialize the object, if needed. Initialize the object for the first read time of the Flash object, or if the Flash object is damaged:

```
if (deviceLog == null) deviceLog = new DeviceLog();
```

5. Update the object. Update/change/delete data in the object:

```
String legend = "Data saved no: " + deviceLog.Log.Count.ToString();
deviceLog.AddToLog(new Device((byte)deviceLog.Log.Count, legend));
```

6. Save the object in Flash:

```
deviceLogFlash.save(deviceLog);
```

5 GUI Applications

This section describes two ways to create graphical user interface elements:

- Use the standard User Interface (UI) objects provided by .NET Micro Framework (which are limited in number, properties, and scope)
- Use the **Bitmap** class

5.1 Standard UI Elements

Visual Studio 2005 and .Net Micro Framework provide standard UI elements that allow quick design of the look and feel of screen components. Table 1 describes the most commonly used standard UI elements.

Table 1. Standard UI Elements

Element	Description
Canvas	Defines an area, or canvas, within which child elements can be explicitly positioned by using coordinates that are relative to the upper-left corner of the canvas
Image	Displays a bitmap image
ListBox	Implements a list of selectable items
ListBoxItem	Implements a selectable item inside a ListBox object
Panel	Constitutes a base class for all panel elements
StackPanel	Arranges child elements (child objects) in a single line that can be oriented either horizontally or vertically
Text	Displays a block of text
TextFlow	Provides members that control how text flows on the display device (screen)
TextRun	Provides members used to create and work with a text run, which is a string of characters that share a single property set
Shape	Represents a line or a two-dimensional shape displayed on a hardware display device; the implemented shape objects are: ellipse, line, polygon, and rectangle

5.2 Using UI Elements

To interact with the interface elements, use functions that are called in the queue of messages of any UI element. This avoid issues with the interaction of visual UI elements.

The user interface elements (**UIElement**) are an implementation of the dispatcher abstract object (**DispatcherObject**). This dispatcher object adds the queue messaging service to the UI, which must be used for any visual change in the object to take effect. Use any dispatcher of any object in the window, including the window object, to update any visual property of the UI elements.

To update the elements and view the effects in the screen, use these steps:

1. Configure the delegate for the function that updates the user interface element:

```
private delegate void UpdateScreenDelegate(UIElement newValue,bool setFocus);
private delegate
       void AddCanvasChildDelegate(UIElement newValue, int top, int left, bool setFocus);
private delegate void UpdateHintDelegate(String hint);
```

2. Implement the functions. These functions must be defined with the same parameters, using the same types of data. The content of the function updates the properties of the user interface elements. The user interface element to update can be a parameter for the function, or the user interface element can be referenced by a global variable.

```
private void UpdateScreen(UIElement newValue, bool setFocus) {
       _mainWindow.Child = newValue;
       if (setFocus) Buttons.Focus(newValue);
}
```

```
private void AddCanvasChild(UIElement newValue, int top, int left, bool setFocus) {
       Canvas.SetTop(newValue, top);
       Canvas.SetLeft(newValue, left);
       if (setFocus) Buttons.Focus(newValue);
       _canvas.Children.Add(newValue);
}
private void UpdateHint(String text) {
       if (_hint != null) _hint.TextContent = text;
}
```

The **Buttons.Focus** method moves the button focus to a specified display element, and in this case, is used to catch keyboard events. This method is used to focus (using **setFocus**) on any element that is associated with the keyboard, such as **ListBox**. At any specific point in time, there can be only one element on a button device display that has the button's focus. For example, if the **Buttons.Focus** method is applied to the **ListBox** element, and then the **ListBox** element cleared, then the **Buttons.Focus** method must be applied to other user interface elements in the window, including the window itself, in order to continue catching the keyboard events.

3. Add functions to the queue. Put the function call in the queue for the correct update of the user interface elements:

```
_mainWindow.Dispatcher.Invoke(
       new TimeSpan(0, 0, 1),
       new UpdateScreenDelegate(UpdateScreen),
       new object[] { _panel, true });
_mainWindow.Dispatcher.Invoke(
       new TimeSpan(0, 0, 1),
       new AddCanvasChildDelegate(AddCanvasChild),
       new object[] { image, 40, 10, false });
_mainWindow.Dispatcher.Invoke(
       new TimeSpan(0, 0, 1),
       new UpdateHintDelegate(UpdateHint),
       new object[] { "Actual State: Draw Image" });
```

5.2.1 Example: Graphical User Interface

The following example illustrates architecture development of an application using .Net Micro Framework user interface elements. This example uses the convention of the underscore character (_) to indicate global private variables.

1. Add references to the namespaces where the classes are located:

```
using System.Threading;
using Microsoft.SPOT.Presentation.Media;
using Microsoft.SPOT.Presentation.Shapes;
```

2. Define the objects that are used for this example:

```
/* Delegates */
private delegate void UpdateScreenDelegate(UIElement newValue, bool setFocus);
private delegate void AddCanvasChildDelegate
       (UIElement newValue, int top, int left, bool setFocus);
private delegate void UpdateHintDelegate(String hint);
/* Global variables for reference of user interface elements */
private Window _mainWindow;
private StackPanel _panel;
private Canvas _canvas;
private Thread _stateT;
private ListBox _listbox;
private Text _hint;
/* States of the main states machine */
private enum statesMain {initialState, drawImage, drawListBox, idle, undefined, end, }
/* States control variable */
private struct stateType {
       public statesMain actual;
       public statesMain prior;
       public statesMain next;
}
private stateType _state;
/* Additional enumeration control */
private enum shapeTypes { rectangle, circle, ellipse, line, }
```

3. Initialize and configure the objects through the **CreateWindow** and the **stateMachine** with the state of **initialState**. With these changes, the **CreateWindow** method looks like this:

```
public Window CreateWindow() {
       _mainWindow = new Window();
       _mainWindow.Height = SystemMetrics.ScreenHeight;
       _mainWindow.Width = SystemMetrics.ScreenWidth;
       Text text = new Text();
       text.Font = Resources.GetFont(Resources.FontResources.small);
       text.TextContent = Resources.GetString(Resources.StringResources.String1);
       text.HorizontalAlignment =
              Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
       text.VerticalAlignment = Microsoft.SPOT.Presentation.VerticalAlignment.Center;
       _mainWindow.Child = text;
       _mainWindow.AddHandler
              (Buttons.ButtonUpEvent, new ButtonEventHandler(OnButtonUp), false);
       _mainWindow.Visibility = Visibility.Visible; Buttons.Focus(_mainWindow);
       /* Initializing the states */
       _state = new stateType();
       _state.prior = statesMain.undefined;
       _state.actual = statesMain.idle;
       _state.next = statesMain.initialState;
       /* Creating and running the thread for the state machine */
       _stateT = new Thread(new ThreadStart(stateMachine));
       _stateT.Start();
       return _mainWindow;
}
```

4. Use the objects when a key is pressed while in the state machine function. In this example the use of complex objects is done through detached functions. With these changes, the **OnButtonUp** method looks like this:

```
private void OnButtonUp(object sender, ButtonEventArgs e) {
       if (e.Button == Button.Right) {
              /* Update the states after right button is pressed */
              _state.prior = _state.actual;
              _state.actual = _state.next;
              _state.next = statesMain.undefined;
       }
       if ((_state.prior == statesMain.drawListBox)
              && (e.Button == Button.Select)) {
       /* In case of the state "drawListBox" the button select is pressed
       * draw a picture depending of the list box element selected */
```

```
                switch (_listbox.SelectedIndex) {
                        case 0: {createShape(shapeTypes.rectangle,150,10,140,220); break; }
                        case 1: {createShape(shapeTypes.circle,150,10,140,220); break; }
                        case 2: {createShape(shapeTypes.ellipse,150,10,140,220); break;}
                        case 3: {createShape(shapeTypes.line,150,10,140,220); break; }
                }
        }
}
```

In the state machine, each state is divided into three parts. The first part is the process to execute in the state, the second is the update of the hint message, and the third is the update of the states. The states change the actual state to idle, and the next state to the next decision state after an event occurs (key pressed). The stateMachine method looks like this:

```
private void stateMachine() {
        while(true) switch (_state.actual) {
                case statesMain.idle: break;
                case statesMain.initialState: {
                        _panel = new StackPanel();
                        _panel.Height = _mainWindow.ActualHeight;
                        _panel.Width= _mainWindow.ActualWidth;
                        Text textTitle = new Text();
                        textTitle.Font = Resources.GetFont(Resources.FontResources.small);
                        textTitle.TextContent = "GUI Standard Interface";
                        textTitle.HorizontalAlignment =
                                Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
                        textTitle.ForeColor =
                                (Microsoft.SPOT.Presentation.Media.Color)0xFF0000;
                        _hint = new Text();
                        _hint.Font = Resources.GetFont(Resources.FontResources.small);
                        _hint.TextContent = "Actual State: Initial State";
                        _hint.HorizontalAlignment =
                                Microsoft.SPOT.Presentation.HorizontalAlignment.Left;
                        _hint.ForeColor =
                                (Microsoft.SPOT.Presentation.Media.Color)0x000000;
                        _canvas = new Canvas();
                        _canvas.Height = _panel.Height - 30;
                        _panel.Children.Add(textTitle);
                        _panel.Children.Add(_canvas);
                        _panel.Children.Add(_hint);
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateScreenDelegate(UpdateScreen),
                                new object[] { new Panel(), true });
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateScreenDelegate(UpdateScreen),
                                new object[] { _panel, true });
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.drawImage;
                        break; }
                case statesMain.drawImage: {
                        Image image = new
                        Image(Resources.GetBitmap(Resources.BitmapResources.logo));
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new AddCanvasChildDelegate(AddCanvasChild),
                                new object[] { image, 40, 10, false });
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateHintDelegate(UpdateHint),
                                new object[] { "Actual State: Draw Image" });
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.drawListBox;
                        break; }
                case statesMain.drawListBox: {
                        _listbox = createListbox();
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new AddCanvasChildDelegate(AddCanvasChild),
                                new object[] { _listbox, 40, 60, true });
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateHintDelegate(UpdateHint),
                                new object[] { "Actual State: Draw List Box" });
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.end;
                        break; }
                case statesMain.end: {
                        _mainWindow.Dispatcher.Invoke(
                                new TimeSpan(0, 0, 1),
                                new UpdateHintDelegate(UpdateHint),
                                new object[] { "Actual State: Done!" });
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.initialState;
                        break; }
                default: break;
        }
}
```

The additional methods look like this:

```
private ListBox createListbox() {
        ListBox listbox = new ListBox();
        listbox.Child.Height = 100;
        listbox.Child.Width = 170;
        ((Control)listbox.Child).Background = new LinearGradientBrush
                (Colors.White, Colors.Blue, 0, 0, listbox.Child.Height, 1);
        // Create TextListBoxItems
        Font font = Resources.GetFont(Resources.FontResources.small);
        TextListBoxItem drawRectangleLbi = new TextListBoxItem("Draw Rectangle", font);
        TextListBoxItem drawCircleLbi = new TextListBoxItem("Draw Circle", font);
        TextListBoxItem drawEllipseLbi = new TextListBoxItem("Draw Ellipse", font);
        TextListBoxItem drawLineLbi = new TextListBoxItem("Draw Line", font);
        // Add TextListBoxItems to listbox
        listbox.Items.Add(drawRectangleLbi);
        listbox.Items.Add(drawCircleLbi);
        listbox.Items.Add(drawEllipseLbi);
        listbox.Items.Add(drawLineLbi);
        listbox.SelectedItem = drawRectangleLbi;
        return listbox;
```

```
}
private void UpdateScreen(UIElement newValue, bool setFocus) {
       _mainWindow.Child = newValue;
       if (setFocus) Buttons.Focus(newValue);
}
private void AddCanvasChild(UIElement newValue, int top, int left, bool setFocus) {
       Canvas.SetTop(newValue, top);
       Canvas.SetLeft(newValue, left);
       if (setFocus) Buttons.Focus(newValue);
       _canvas.Children.Add(newValue);
}
private void UpdateHint(String text) {
       if (_hint != null) _hint.TextContent = text;
}
private void createShape(shapeTypes shapeType, int top, int left, int maxHeight, int
maxWidth) {
       // Get random numbers
       int topRelaive = Microsoft.SPOT.Math.Random(maxHeight);
       int leftRelaive = Microsoft.SPOT.Math.Random(maxWidth);
       int topPos = topRelaive + top;
       int leftPos = leftRelaive + left;
       maxHeight -= topRelaive;
       maxWidth -= leftRelaive;
       if (maxHeight < 2) maxHeight = 2;
       if (maxWidth < 2) maxWidth = 2;
       Color color = (Color)Microsoft.SPOT.Math.Random(0xFFFFFF);
       int maxRadius = 0;
       if (maxWidth < maxHeight) maxRadius = maxWidth/2;
       else maxRadius = maxHeight / 2;
       Shape shape = null;
       switch (shapeType) {
              case shapeTypes.rectangle:
                     shape = new Rectangle();
                     shape.Width = Microsoft.SPOT.Math.Random(maxWidth);
                     shape.Height = Microsoft.SPOT.Math.Random(maxHeight);
                     shape.Stroke = new Pen(color);
// Note: The .NET Micro Framework SDK does not support a non-filled rectangle at this time
//** Fill the rectangle with the same color as the pen stroke
                     shape.Fill = new SolidColorBrush(color);
                     break;
              case shapeTypes.circle:
                     int radius = Microsoft.SPOT.Math.Random(maxRadius);
                     shape = new Ellipse(radius, radius);
                     shape.Stroke = new Pen(color);
                     break;
              case shapeTypes.ellipse:
                     int xRadius = Microsoft.SPOT.Math.Random(maxWidth/2);
                     int yRadius = Microsoft.SPOT.Math.Random(maxHeight/2);
                     shape = new Ellipse(xRadius, yRadius);
                     shape.Stroke = new Pen(color);
                     break;
              case shapeTypes.line:
                     int xDis = Microsoft.SPOT.Math.Random(maxRadius);
                     int yDis = Microsoft.SPOT.Math.Random(maxRadius);
                     shape = new Line(xDis, yDis);
                     shape.Stroke = new Pen(color);
                     break;
```

```
                default: break;
            }
            if (shape != null)
                _mainWindow.Dispatcher.Invoke(
                    new TimeSpan(0, 0, 1),
                    new AddCanvasChildDelegate(AddCanvasChild),
                    new object[] { shape, topPos, leftPos, false });
        }
```

For this example, a class is created for a specific functionality for the list box items, as follows:

```
class TextListBoxItem : ListBoxItem {
    public TextListBoxItem(string str, Font font) : base() {
        Text text = new Text(font, str);
        text.HorizontalAlignment = HorizontalAlignment.Center;
        this.Child = text;
        this.Background = null; // Set the background to transparent
    }
    protected override void OnIsSelectedChanged(bool isSelected) {
        Text text = this.Child as Text;
        if (isSelected) {
            text.ForeColor = Colors.Blue;
            this.Background = new SolidColorBrush(Colors.Gray);
        } else {
            text.ForeColor = Colors.Black;
            this.Background = null;
        }
    }
}
```

5.3 Use the Screen as a Bitmap

To use the screen as a bitmap: create the bitmap, update the pixels in the bitmap, and place the bitmap in the screen.

5.3.1 Bitmap Class

The **Bitmap** class encapsulates a GDI+ bitmap, which consists of the pixel data for a graphics image and its attributes. A Bitmap is an object used to work with images defined by pixel data. The **Bitmap** class provides a complete set of methods that allows drawing text, lines, circles, ellipses, and rectangles in the bitmap, merging other bitmaps, and modifying the bitmap by pixel. The **Flush** method of the **Bitmap** class places the bitmap in the screen. Table 2 describes the common methods for drawing within the bitmap.

Table 2. Bitmap Drawing Methods

Method	Description
Clear	Clears the entire drawing surface
DrawEllipse	Draws a filled ellipse on the display device
DrawImage	Draws a rectangular block of pixels on the display device
DrawLine	Draws a line on the display device
DrawRectangel	Draws a rectangle on the display device

Table 2. Bitmap Drawing Methods (continued)

Method	Description
DrawText	Draws text on the display device
DrawTextInRect	Draws text in a specified rectangle
Flush	Flushes the current bitmap to the display device
SexPixel	Turns a specified pixel on or off

5.3.2 Example: Graphical User Interface

This example shows how to develop the architecture of an application using .Net Micro Framework with a Bitmap as the screen. This example uses the convention of the underscore character (_) to indicate global private variables.

To perform this example, use these steps:

1. Add a reference to the namespaces where the classes to use are located:

```
using System.Threading;
using Microsoft.SPOT.Presentation.Media;
```

2. Define the objects to be used in this example:

```
private Window _mainWindow;                                 // main window element
private Bitmap _screen;                                   // bitmap used for flush
private Bitmap _back;                                // based bitmap to be updated
private Thread _stateT;                             // thread of the state machine
/* States of the main state machine */
private enum statesMain {idle,initialState,drawBackground,drawButtons,end,undefined, }
/* State machine control variable */
       private struct stateType {
       public statesMain actual;
       public statesMain prior;
       public statesMain next;
}
private stateType _state;
/* Definition of the buttons */
private enum buttonNames {first, button1, button2, none, }
/* Buttons control variable */
private struct buttonStates {
       public buttonNames pressed; /* Button pressed */
       public buttonNames selected; /* Button selected */
       public buttonNames prior; /* Last button pressed */
}
private buttonStates _button;
```

3. Initialize and configure the objects. In this example the initialization is done through the **CreateWindow** and the **stateScreen** with the state of **initialState**. After the changes the **CreateWindow** method looks like this:

```
public Window CreateWindow() {
       _mainWindow = new Window();
       _mainWindow.Height = SystemMetrics.ScreenHeight;
       _mainWindow.Width = SystemMetrics.ScreenWidth;
       Text text = new Text();
```

```
        text.Font = Resources.GetFont(Resources.FontResources.small);
        text.TextContent = Resources.GetString(Resources.StringResources.String1);
        text.HorizontalAlignment=Microsoft.SPOT.Presentation.HorizontalAlignment.Center;
        text.VerticalAlignment = Microsoft.SPOT.Presentation.VerticalAlignment.Center;
        _mainWindow.Child = text;
        _mainWindow.AddHandler(Buttons.ButtonUpEvent,
        new ButtonEventHandler(OnButtonUp), false);
        _mainWindow.Visibility = Visibility.Visible;
        Buttons.Focus(_mainWindow);
        /* Initializing the states */
        _state = new stateType();
        _state.prior = statesMain.undefined;
        _state.actual = statesMain.idle;
        _state.next = statesMain.initialState;
        /* Initializing the button state */
        _button = new buttonStates();
        /* Creating and running the thread for the state machine */
        _stateT = new Thread(new ThreadStart(stateScreen));
        _stateT.Start();
        return _mainWindow;
}
```

4. Use the objects. The objects are used when a key is pressed and in the state machine function. With our changes, the **OnButtonUp** method looks like this:

```
private void OnButtonUp(object sender, ButtonEventArgs e) {
        // Print the button code to the Visual Studio output window.
        Debug.Print(e.Button.ToString());
        if (e.Button == Button.Right) {
                /* Update the states after right button is pressed */
                _state.prior = _state.actual;
                _state.actual = _state.next;
                _state.next = statesMain.undefined;
        }
        switch (_state.prior) {
                case statesMain.drawButtons : {
        /* In case of the state "drawButtons" the button select is changed
        * when up or down button is pressed, and the change of states
        * update the screen */
                        if (e.Button == Button.Select) {
                                _button.prior = _button.pressed;
                                _button.pressed = _button.selected;
                        } else {
                                if (e.Button == Button.Up) _button.selected--;
                                else if (e.Button == Button.Down) _button.selected++;
                                if (_button.selected == buttonNames.none)
                                            _button.selected = buttonNames.first + 1;
                                else if (_button.selected == buttonNames.first)
                                             _button.selected = buttonNames.none - 1;
                        }
                        _state.prior = _state.actual;
                        _state.actual = statesMain.drawButtons;
                        _state.next = statesMain.undefined;
                        break;
                }
        }
}
```

In the state machine, each state is divided into three parts. The first part is the process to execute in the state; the second is the update of the hint message, and the third is the update of the states. The states change the actual state to idle, and then change the next state to the next decision state, after an event occurs (key pressed). The state machine **stateScreen** method looks like this:

```
private void stateScreen() {
        Font font = Resources.GetFont(Resources.FontResources.small);
        while (true) switch (_state.actual) {
                case statesMain.idle: break;
                case statesMain.initialState: {
                        _button.prior = buttonNames.none;
                        _button.selected = buttonNames.none;
                        _button.pressed = buttonNames.none;
                        _back = new Bitmap(240, 320);
                        _back.DrawRectangle(Color.White, 10, 0, 0, 240, 320, 2, 2,
                                Color.White, 0, 0, Color.White, 240, 320, 0);
                        _screen = new Bitmap(240, 320);
                        _screen.DrawImage(0,0,_back,0,0,240,320);
                        _screen.Flush();
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.drawBackground;
                        break; }
                case statesMain.drawBackground: {
                        _back.DrawImage(35, 10,
                        Resources.GetBitmap
                                (Resources.BitmapResources.freescale), 0, 0, 170, 57);
                        _back.DrawRectangle(Color.White, 1, 35, 10, 170, 57, 2, 2,
                                Color.White, 0, 0, Color.White, 240, 320, 0);
                        _screen.DrawImage(0, 0, _back, 0, 0, 240, 320);
                        _screen.DrawTextInRect("State: Background", 10, 300, 220, 20,
                                Bitmap.DT_AlignmentCenter |
                                Bitmap.DT_TrimmingCharacterEllipsis, (Color)0xFFFFFF, font);
                        _screen.Flush();
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.drawButtons;
                        break; }
                case statesMain.drawButtons: {
                        if (_button.selected == buttonNames.none)
                                _button.selected = buttonNames.first + 1;
                        _screen.DrawImage(0, 0, _back, 0, 0, 240, 320);
                        String hint = "State: Buttons";
                        if (_button.pressed != buttonNames.none) {
                                if (_button.pressed == buttonNames.button1)
                                        hint += " : Button 1 pressed";
                                else if (_button.pressed == buttonNames.button2)
                                        hint += " : Button 2 pressed";
                        }
                        _screen.DrawTextInRect(hint, 10, 300, 220, 20,
                                Bitmap.DT_AlignmentCenter |
                                Bitmap.DT_TrimmingCharacterEllipsis,(Color)0xFFFFFF, font);
                        _screen.DrawTextInRect("Button 1", 10, 100, 220, 20,
                                Bitmap.DT_AlignmentCenter |
                                Bitmap.DT_TrimmingCharacterEllipsis,(Color)0x0000FF, font);
                        _screen.DrawTextInRect("Button 2", 10, 200, 220, 20,
                                Bitmap.DT_AlignmentCenter |
                                Bitmap.DT_TrimmingCharacterEllipsis,(Color)0x0000FF, font);
                        if (_button.selected == buttonNames.button1)
                                _screen.DrawEllipse((Color)0xFFFFFF, 120, 107, 110, 10);
                        else if (_button.selected == buttonNames.button2)
                                _screen.DrawEllipse((Color)0xFFFFFF, 120, 207, 110, 10);
                                _screen.Flush();
                                _state.prior = _state.actual;
                                _state.actual = statesMain.idle;
                                _state.next = statesMain.end;
                                break; }
                case statesMain.end: {
                        _screen.DrawImage(0, 0, _back, 0, 0, 240, 320);
                        _screen.DrawTextInRect("State: End", 10, 300, 220, 20,
                                Bitmap.DT_AlignmentCenter |
                                Bitmap.DT_TrimmingCharacterEllipsis,(Color)0xFFFFFF, font);
                        _screen.Flush();
                        _state.prior = _state.actual;
                        _state.actual = statesMain.idle;
                        _state.next = statesMain.initialState;
                        break; }
                default: break;
        }
}
```

6 SideShow Applications

Windows SideShow is a technology that enables a Windows PC to drive a variety of auxiliary display devices connected to the main PC. These devices can be separate from or integrated into the main PC (for example, a display embedded on the outside of a laptop lid), enabling access to information and media even when the PC is (mostly) turned off. SideShow can also drive the display of PC data on mobile phones and other devices that are connected by Bluetooth or other wireless network protocols.

6.1 SideShow Enhanced Display Types

Two types of applications can be loaded into the SideShow enhanced display devices:

- XML applications can be downloaded from the Internet and installed and uninstalled in the device using the SideShow portal in the connected a Windows Vista® PC
- Built-in applications are libraries developed over .Net Micro Framework that can be added to the SideShow device for specific functionality in the device

6.2 Example: SideShow Applications

This section provides an example for creating device-specific, built-in SideShow applications using .Net Micro Framework. Because .Net Micro Framework can run only one application at a time, SideShow is the application that runs and the SideShow applications must be created and added as .Net Micro Framework libraries (dlls).

Two main components are required in the SideShow applications:

- DLL component that must be compliant with the interface **IDeviceApplication**
- Main form executable that must be inherited from the `ApplicationForm` class

The SideShow Application libraries must respect the following structure:

```
//-------------------------------------------------------------------------------------
// SideShow application
// An application must implement the IDeviceApplication interface. The Shell uses this
// interface to control the application.
//-------------------------------------------------------------------------------------
public class SideshowExample : IDeviceApplication
{
        public SideshowExample()
        {
                Debug.Print("Sideshow Example Application, DLL Contructor");
        }

//---------------------------------------------------------------------------------------
// Built-in order number
// This static property identifies this class as a built-in (stand-alone) application. The Shell
// locates built-in applications by using reflection to find all classes that declare this
// property. The Shell creates an instance of each built-in application at system startup time.
//
// The Shell orders the built-in applications according to their order numbers (smallest first).
// The built-in applications are displayed behind the PC gadget endpoint applications.
//--------------------------------------------------------------------------------------
        public static int BuiltInOrder
        {
                get { return 50; }
        }
        //
        // IDeviceApplication methods
        //
//--------------------------------------------------------------------------------------
// Install
// The Shell calls this method right after creating an instance of the application. A built-in
// application does not receive data from a PC gadget, so the cache argument is always null.
//
// The application can do any one-time initialization in this method.
//---------------------------------------------------------------------------------------
        public void Install(IApplicationCache cache)
        {
                Debug.Print("Sideshow Example Application, DLL Installed");
        }
//---------------------------------------------------------------------------------------
// Uninstall
// This method is never called for built-in applications because they are never unistalled.
//---------------------------------------------------------------------------------------
        public void Uninstall()
        {
        }
//---------------------------------------------------------------------------------------
// Set Shell attributes
// The Shell calls this method to get the application name and icons that will be displayes on
// the home page. The method returns these in the attribute class that is passed in. The icons
// should be 32-bit color icons with alpha channel. Each icon resource is a byte array which
// contains a .bmp file image. The icons are returned as byte arrays rather than Bitmap class
// instances, so that the Shell can access the raw pixel data.
//---------------------------------------------------------------------------------------
        public void SetShellAttributes(ShellAttributes attributes)
        {
```

```
                  attributes.Name = "Hello World App";
                  attributes.Icon16Data=Resources.GetBytes(Resources.BinaryResources.ICON_16);
                  attributes.Icon32Data=Resources.GetBytes(Resources.BinaryResources.ICON_32);
                  attributes.Icon48Data=Resources.GetBytes(Resources.BinaryResources.ICON_48);
         }
         //
         // Member variable to hold the applicaiton form
         //
         private HelloWorldForm _form;
//--------------------------------------------------------------------------------------
// Get form
// The Shell calls this method to get the application main window when the user selects this
// application on the Shell home page. An application typically caches the form and returns the
// same one every time.
//--------------------------------------------------------------------------------------
         public ApplicationForm GetForm()
         {
                  if (_form == null)
                  {
                           _form = new HelloWorldForm();
                  }
                  return _form;
         }
//--------------------------------------------------------------------------------------
// Get Glance Data
// The Shell calls this method when the application is highlighted on the Shell home page. The
// method optionally returns one or more lines of text that give a high-level status for the
// application. For example, a calendar application may return the time and place of the next
// appointment as the Primary status and later appointments this day as secondary status. Note
// that the seconday status is an array of strings. A typical device can display up to 15 lines
// of status.
//
// For built-in applications that don't have status to display, the primary glance data can be a
// short description of what the application does.
//
// The Shell calls GetGlanceData when the application gets the focus on the home page or when it
// is told that the glance data has changed. The application must call
// Globals.Shell.UpdateGlanceData(this) to notify the Shell when the glance data has changed.
//--------------------------------------------------------------------------------------
         public GlanceData GetGlanceData()
         {
                  GlanceData glance = new GlanceData();
                  glance.Primary = "(great example!!!)";
                  return glance;
         }
}
```

The SideShow main form must respect the following structure:

```
//---------------------------------------------------------------------------
// HelloWorldForm
// This is the main window of the application. It must be derived from base class
// ApplicationForm. ApplicationForm defines virtual methods for interacting with
// the SideShow Shell and provides a default implementation for each method.
//---------------------------------------------------------------------------
private class HelloWorldForm : ApplicationForm
{
         //-------------------------------------------------------------------------
```

```
// Constructor
// Passes "true" to ApplicationForm base to specify the form should have a
// title bar. Application forms should have a title bar because it identifies
// the application with an icon and name. It also displays the current time.
//-----------------------------------------------------------------------------
public HelloWorldForm()
         : base(true)
{
         Debug.Print("Form construct");
}
//------------------------------------------------------------------------------
// Activate event handler
// This is called when the user selects this application from the Shell home page.
// This is called right after the form is made visible. Prior to calling this
// the Shell sets the title bar icon and name to the icon and name provided
// by the gadget. The application can replace or extend the title text here.
//------------------------------------------------------------------------------
protected override void OnActivate(EventArgs e)
{
         //
         // Set the initial view. It's up to the application whether to display the
         // same view everytime or return to the last visited view. The
         // ApplicationForm sets the focus to the View when it gets the focus from
         // the OS.
         //
         // To conserve device memory an application should create its UI on
         // activation and release it on deactivation.
         //
         Debug.Print("Form activate");
         View = CreateContent();
}
//------------------------------------------------------------------------------
// Deactivate event handler
// This is called when the user leaves the application and returns to the Shell
// home page.The application should release UI objects and stop any processing
// that isn't needed when the application isn't the active one
//------------------------------------------------------------------------------
protected override void OnDeactivate(EventArgs e)
{
         Debug.Print("Form on deactivate");
}
//------------------------------------------------------------------------------
// Handle theme change
// If an application displays text or creates custom UI controls it should
// use the fonts, colors, backgrounds, etc. specified by the currently
// selected theme. The theme properties are exposed by the ThemeResources
// class.
// This method is called when the user switches the theme. The application
// should respond by updating all displays to use the new theme settings. The
// simplest way to update is to just recreate the Views.
//------------------------------------------------------------------------------
public override void OnThemeChange()
{
         base.OnThemeChange();
         if (View != null)
         {
                  View = CreateContent();
```

```
                }
        }
        //----------------------------------------------------------------------------
        // Create content
        // This method creates the view for the form. For this simple example the
        // view is just a centered text control which displays "Hello World".
        //
        // More complex applications often have multiple views and UI to select
        // the current view (often a context menu). The view control is private to
        // the application. The only part the base form plays is to set the focus
        // to the new view whenever it is changed.
        //----------------------------------------------------------------------------
        private UIElement CreateContent()
        {
                Color color = ThemeResources.GetColor(ThemeColor.PageText);
                Font font = ThemeResources.GetFont(ThemeFont.PageBold);
                Text text = new Text(font, "");
                text.TextContent = Resources.GetString(Resources.StringResources.String1);
                text.ForeColor = color;
                text.VerticalAlignment = VerticalAlignment.Center;
                text.HorizontalAlignment = HorizontalAlignment.Center;
                return text;
        }
        //------------------------------------------------------------------------------
        // Button Up event handler
        // This is called when any button is released.
        // The application flow can be linked to the buttons events through this event
        // handler.
        //------------------------------------------------------------------------------
        protected override void OnButtonUp(ButtonEventArgs e)
        {
                switch (e.Button)
                {
                case Button.Up:
                        Debug.Print("button UP pressed");
                        break;
                case Button.Right:
                        Debug.Print("button Right pressed");
                        break;
                case Button.Down:
                        Debug.Print("button Down pressed");
                        break;
                case Button.Left:
                        Debug.Print("button Left pressed");
                        break;
                case Button.Select:
                        Debug.Print("button Select pressed");
                        break;
                }
        }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Document Number: AN3888
Rev. 0
08/2009