```
/*------------------------------------------------*\
 * Author    : Salvi Cyril
 * Date      : 7th juny 2017
 * Diploma   : RaspiHome
 * Classroom : T.IS-E2B
 *
 * Description:
 *      RaspiHomePiFaceDigital2 is a program who use
 *   a PiFace Digital 2, it's an electronic card who
 *   can be use to plug electronic component. This
 *   program use the PiFace Digital 2 to activate
 *   light and store.
\*------------------------------------------------*/

using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.Devices.Enumeration;
using Windows.Devices.Spi;

namespace RaspiHomePiFaceDigital2
{
    public class MCP23S17
    {

        private const byte IODIRA = 0x00;     // I/O Direction Register
        private const byte IODIRB = 0x01;      // 1 = Input (default), 0 =
          Output
        private const byte IPOLA = 0x02;     // MCP23x17 Input Polarity
          Register
        private const byte IPOLB = 0x03;     // 0 = Normal (default)(low reads
           as 0), 1 = Inverted (low reads as 1)
        private const byte GPINTENA = 0x04;     // MCP23x17 Interrupt on
          Change Pin Assignements
        private const byte GPINTENB = 0x05;     // 0 = No Interrupt on Change
          (default), 1 = Interrupt on Change
        private const byte DEFVALA = 0x06;    // MCP23x17 Default Compare
          Register for Interrupt on Change
        private const byte DEFVALB = 0x07;     // Opposite of what is here
          will trigger an interrupt (default = 0)
        private const byte INTCONA = 0x08;     // MCP23x17 Interrupt on Change
           Control Register
        private const byte INTCONB = 0x09;     // 1 = pin is compared to
          DEFVAL, 0 = pin is compared to previous state (default)
        private const byte IOCONA = 0x0A;     // MCP23x17 Configuration
          Register
        private const byte IOCONB = 0x0B;    //     Also Configuration
          Register
        private const byte GPPUA = 0x0C;     // MCP23x17 Weak Pull-Up Resistor
           Register
        private const byte GPPUB = 0x0D;     // INPUT ONLY: 0 = No Internal
          100k Pull-Up (default) 1 = Internal 100k Pull-Up
        private const byte INTFA = 0x0E;     // MCP23x17 Interrupt Flag
          Register
        private const byte INTFB = 0x0F;    // READ ONLY: 1 = This Pin
          Triggered the Interrupt
```

```
42          private const byte INTCAPA = 0x10;    // MCP23x17 Interrupt Captured
              Value for Port Register
43          private const byte INTCAPB = 0x11;    // READ ONLY: State of the Pin
              at the Time the Interrupt Occurred
44          private const byte GPIOA = 0x12;      // MCP23x17 GPIO Port Register
45          private const byte GPIOB = 0x13;      // Value on the Port - Writing
              Sets Bits in the Output Latch
46          private const byte OLATA = 0x14;      // MCP23x17 Output Latch
              Register
47          private const byte OLATB = 0x15;      // 1 = Latch High, 0 = Latch Low
               (default) Reading Returns Latch State, Not Port Value!
48
49          public const byte On = 1;
50          public const byte Off = 0;
51          public const byte Output = 0;
52          public const byte Input = 1;
53
54          private const byte Address = 0x00;   // offset address if hardware
              addressing is on and is 0 - 7 (A0 - A2)
55          private const byte BaseAddW = 0x40;  // MCP23S17 Write base address
56          private const byte BaseAddR = 0x41;  // MCP23S17 Read Base Address
57          private const byte HAEN = 0x08;  // IOCON register for MCP23S17, x08
              enables hardware address so sent address must match hardware pins
              A0-A2
58
59
60          private static UInt16 PinMode = 0XFFFF;     // default Pinmode for the
               MXP23S17 set to inputs
61          private static UInt16 PullUpMode = 0XFFFF;     // default pullups for
              the MXP23S17 set to weak pullup
62          private static UInt16 InversionMode = 0X0000;     // default invert to
               normal
63          private static UInt16 PinState = 0X0000;     // default pinstate to
              all 0's
64
65          /*RaspBerry Pi2  Parameters*/
66          private const string SPI_CONTROLLER_NAME = "SPI0";  /* For Raspberry
              Pi 2, use SPI0                       */
67          private const Int32 SPI_CHIP_SELECT_LINE = 0;       /* Line 0 maps to
              physical pin number 24 on the Rpi2, line 1 to pin 26        */
68
69          private static byte[] readBuffer3 = new byte[3]; /*this is defined to
              hold the output data*/
70          private static byte[] readBuffer4 = new byte[4]; /*this is defined to
              hold the output data*/
71          private static byte[] writeBuffer3 = new byte[3];//register, then 16
              bit value
72          private static byte[] writeBuffer4 = new byte[4];//register, then 16
              bit value
73
74          private static SpiDevice SpiGPIO;
75          public static async Task InitilizeSPI()
76          {
77              try
78              {
79                  var settings = new SpiConnectionSettings
```

```
                    (SPI_CHIP_SELECT_LINE);
80                  settings.ClockFrequency = 1000000;// 10000000;
81                  settings.Mode = SpiMode.Mode0; //Mode0,1,2,3;  MCP23S17 needs
                      mode 0
82
83                  string spiAqs = SpiDevice.GetDeviceSelector
                      (SPI_CONTROLLER_NAME);
84                  var deviceInfo = await DeviceInformation.FindAllAsync(spiAqs);
85                  SpiGPIO = await SpiDevice.FromIdAsync(deviceInfo[0].Id,
                      settings);
86              }
87
88              /* If initialization fails, display the exception and stop running
                   */
89              catch (Exception ex)
90              {
91                  Debug.WriteLine(ex.Message);
92                  //statusText.Text = "\nSPI Initialization Failed";
93              }
94          }
95
96          public static void InitializeMCP23S17()
97          {
98              WriteRegister8(IOCONA, HAEN);                    // enable the
                    hardware address incase there is more than one chip
99              WriteRegister16(IODIRA, PinMode);                 // Set the
                    default or current pin mode
100
101         }
102         public static void WriteRegister8(byte register, byte value)
103         {
104             // Direct port manipulation speeds taking Slave Select LOW before
                    SPI action
105             writeBuffer3[0] = (BaseAddW | (Address << 1));
106             writeBuffer3[1] = register;
107             writeBuffer3[2] = value;
108             try
109             {
110                 SpiGPIO.Write(writeBuffer3);
111             }
112
113             /* If initialization fails, display the exception and stop running
                   */
114             catch (Exception ex)
115             {
116                 Debug.WriteLine(ex.Message);
117                 //statusText.Text = "\nFailed to Wrie to DAC";
118             }// Send the byte
119         }
120         public static void WriteRegister16(byte register, UInt16 value)
121         {
122             writeBuffer4[0] = (BaseAddW | (Address << 1));
123             writeBuffer4[1] = register;
124             writeBuffer4[2] = (byte)(value >> 8);
125             writeBuffer4[3] = (byte)(value & 0XFF);
126             try
```

```
127             {
128                 SpiGPIO.Write(writeBuffer4);
129             }
130 
131             /* If initialization fails, display the exception and stop running */
132             catch (Exception ex)
133             {
134                 Debug.WriteLine(ex.Message);
135                 //statusText.Text = "\nFailed to Wrie to DAC";
136             }
137         }
138 
139         // Set the pin mode a pin at a time or all 16 in one go
140         // any value other then Input is taken as output
141         public static void setPinMode(byte pin, byte mode)
142         {
143             if (pin > 15) return;               // only a 16bit port so do a bounds check, it cant be less than zero as this is a byte value
144             if (mode == Input)
145             {
146                 PinMode |= (UInt16)(1 << (pin));              // update the pinMode register with new direction
147             }
148             else
149             {
150                 PinMode &= (UInt16)(~(1 << (pin)));           // update the pinMode register with new direction
151             }
152             WriteRegister16(IODIRA, PinMode);               // Call the generic word writer with start register and the mode cache
153         }
154         public static void SetPinMode(UInt16 mode)
155         {
156             WriteRegister16(IODIRA, mode);
157             PinMode = mode;
158         }
159 
160         // Set the pullup a pin at a time or all 16 in one go
161         // any value other than On is taken as off
162         public static void pullupMode(byte pin, byte mode)
163         {
164             if (pin > 15) return;
165             if (mode == On)
166             {
167                 PullUpMode |= (UInt16)(1 << (pin));
168             }
169             else
170             {
171                 PullUpMode &= (UInt16)(~(1 << (pin)));
172             }
173             WriteRegister16(GPPUA, PullUpMode);
174         }
175         public static void PullupMode(UInt16 mode)
176         {
177             WriteRegister16(GPPUA, mode);
```

```
178             PullUpMode = mode;
179         }
180
181         // Set the inversion a pin at a time or all 16 in one go
182         public static void InvertMode(byte pin, byte mode)
183         {
184             if (pin > 15) return;
185             if (mode == On)
186             {
187                 InversionMode |= (UInt16)(1 << (pin - 1));
188             }
189             else
190             {
191                 InversionMode &= (UInt16)(~(1 << (pin - 1)));
192             }
193             WriteRegister16(IPOLA, InversionMode);
194         }
195         public static void InvertMode(UInt16 mode)
196         {
197             WriteRegister16(IPOLA, mode);
198             InversionMode = mode;
199         }
200
201         // WRITE FUNCTIONS - BY WORD AND BY PIN
202
203         public static void WritePin(byte pin, byte value)
204         {
205             if (pin > 15) return;
206             if (value > 1) return;
207             if (value == 1)
208             {
209                 PinState |= (UInt16)(1 << pin);
210             }
211             else
212             {
213                 PinState &= (UInt16)(~(1 << pin));
214             }
215             WriteRegister16(GPIOA, PinState);
216         }
217         public static void WriteWord(UInt16 value)
218         {
219             WriteRegister16(GPIOA, value);
220             PinState = value;
221         }
222
223         // READ FUNCTIONS - BY WORD, BYTE AND BY PIN
224         public static UInt16 ReadRegister16()
225         {
226             writeBuffer4[0] = (BaseAddR | (Address << 1));
227             writeBuffer4[1] = GPIOA;
228             writeBuffer4[2] = 0;
229             writeBuffer4[3] = 0;
230             SpiGPIO.TransferFullDuplex(writeBuffer4, readBuffer4);
231             return convertToInt(readBuffer4);                                   //  ↵
                  Return the constructed word, the format is 0x(register value)
232         }
```

```
233         public static byte ReadRegister8(byte register)
234         {        // This function will read a single register, and return it
235             writeBuffer3[0] = (BaseAddR | (Address << 1));  // Send the
                  MCP23S17 opcode, chip address, and read bit
236             writeBuffer3[1] = register;
237             SpiGPIO.TransferFullDuplex(writeBuffer3, readBuffer3);
238             return readBuffer4[2]; // convertToInt
                  (readBuffer);                              // Return the
                  constructed word, the format is 0x(register value)
239         }
240         public static UInt16 ReadPin(byte pin)
241         {
242             if (pin > 15) return 0x00;                  // If the pin value is
                   not valid (1-16) return, do nothing and return
243             UInt16 value = ReadRegister16();                        //
                  Initialize a variable to hold the read values to be returned
244             UInt16 pinmask = (UInt16)(1 << pin);                        //
                  Initialize a variable to hold the read values to be returned
245             return ((value & pinmask) > 0) ? On : Off;  // Call the word
                  reading function, extract HIGH/LOW information from the
                  requested pin
246         }
247
248         private static UInt16 convertToInt(byte[] data)
249         {
250             // byte[0] = command, byte[1] register, byte[2] = data high, byte
                  [3] = data low
251             UInt16 result = (UInt16)(data[2] & 0xFF);
252             result <<= 8;
253             result += data[3];
254             return result;
255         }
256     }
257 }
258
```