**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Electrical Specifications</td>
<td>3</td>
</tr>
<tr>
<td>PinIO Class</td>
<td>3</td>
</tr>
<tr>
<td>Pins Class Constants</td>
<td>3</td>
</tr>
<tr>
<td>Pins Class Member Functions</td>
<td>5</td>
</tr>
<tr>
<td>Program Examples</td>
<td>6</td>
</tr>
</tbody>
</table>
**Introduction**

The PinIO class provides an easy way to configure and operate the Freescale MCF5270 microprocessor GPIO signals. Each signal pin on the processor can have multiple functions. You can use the PinIO class to control GPIO signals without having to explicitly configure the processor registers. Configuration of the processor registers are done in the member functions of the PinIO class. There are fifteen pins on the SB70LC that are made available for GPIO. This document will list the pins that can be used for GPIO and how to use them. Note that the terms "PinIO class" and "Pins class" may be used interchangeably in this document.

If you do wish to access these registers directly, we recommend you use the register structure defined in the "sim5270.h" header file and use the Freescale MCF5271 reference manual to learn the operation of each register.

**Electrical Specifications**

The current drive capabilities of the GPIO pins are the same for all pins. The instantaneous maximum current for a single pin is 25 mA. The sustained current drive is 5 mA. Please see the "MCF5271 Integrated Microprocessor Hardware Specification" PDF document for more information, which can be found at the Freescale web site.

**PinIO Class**

This class is defined in the header file "pins.h", which is located in the \Nburn\include directory. With this class, the pins can be configured for GPIO or some other function. If the pins are set for GPIO, then you can set, clear, read the state of the pins, drive the pins, or set them for high impedance by simply using the appropriate member function.

Since the number and type of pins are unique to each NetBurner module, the definition of the pins (\Nburn\<platform>\include\pinconstant.h) and the functions to use those pins (\Nburn\<platform>\system\pins.cpp) are located within each applicable platform directory.

**Pins Class Constants**

The table below lists the fifteen pins available for GPIO on the SB70LC, as well as their primary and alternate functions, if any:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Definition</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>PIN3_SPI_CS0</td>
<td>1: SPI Chip Select 0</td>
</tr>
<tr>
<td></td>
<td>PIN3_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>4</td>
<td>PIN4_SPI_DOUT</td>
<td>1: SPI Data Out</td>
</tr>
<tr>
<td></td>
<td>PIN4_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>Pin</td>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>------</td>
<td>----------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>5</td>
<td>PIN5_URT0_RTS</td>
<td>1: UART 0 – Request to Send</td>
</tr>
<tr>
<td></td>
<td>PIN5_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>6</td>
<td>PIN6_SDI_DIN</td>
<td>3: SPI Data In</td>
</tr>
<tr>
<td></td>
<td>PIN6_SDA</td>
<td>2: I2C Serial Data</td>
</tr>
<tr>
<td></td>
<td>PIN6_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>7</td>
<td>PIN7_SPI_CLK</td>
<td>3: SPI Clock</td>
</tr>
<tr>
<td></td>
<td>PIN7_SCL</td>
<td>2: I2C Serial Clock</td>
</tr>
<tr>
<td></td>
<td>PIN7_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>8</td>
<td>PIN8_UCTS1</td>
<td>3: UART 1 – Clear to Send</td>
</tr>
<tr>
<td></td>
<td>PIN8_UCTS2</td>
<td>2: UART 2 – Clear to Send</td>
</tr>
<tr>
<td></td>
<td>PIN8_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>9</td>
<td>PIN9_URT1_RTS</td>
<td>3: UART 1 – Request to Send</td>
</tr>
<tr>
<td></td>
<td>PIN9_URT2_RTS</td>
<td>2: UART 2 – Request to Send</td>
</tr>
<tr>
<td></td>
<td>PIN9_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>10</td>
<td>PIN10_UTXD0</td>
<td>1: UART 0 – Transmit</td>
</tr>
<tr>
<td></td>
<td>PIN10_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>11</td>
<td>PIN11_URXD0</td>
<td>1: UART 0 – Receive</td>
</tr>
<tr>
<td></td>
<td>PIN11_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>12</td>
<td>PIN12_UTXD1</td>
<td>3: UART 1 – Transmit</td>
</tr>
<tr>
<td></td>
<td>PIN12_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>13</td>
<td>PIN13_URXD1</td>
<td>3: UART 1 – Receive</td>
</tr>
<tr>
<td></td>
<td>PIN13_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>14</td>
<td>PIN14_URT0_CTS</td>
<td>1: UART 0 – Clear to Send</td>
</tr>
<tr>
<td></td>
<td>PIN14_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>15</td>
<td>PIN15_TIN3</td>
<td>3: DMA Timer Input 3</td>
</tr>
<tr>
<td></td>
<td>PIN15_SPI_CS2</td>
<td>1: SPI Chip Select 2</td>
</tr>
<tr>
<td></td>
<td>PIN15_URT2_CTS</td>
<td>2: UART 2 – Clear to Send</td>
</tr>
<tr>
<td></td>
<td>PIN15_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>16</td>
<td>PIN16_SDA</td>
<td>3: I2C Serial Data</td>
</tr>
<tr>
<td></td>
<td>PIN16_GPIO</td>
<td>0: GPIO</td>
</tr>
<tr>
<td>17</td>
<td>PIN17_SCL</td>
<td>3: I2C Serial Clock</td>
</tr>
<tr>
<td></td>
<td>PIN17_GPIO</td>
<td>0: GPIO</td>
</tr>
</tbody>
</table>

**Pin Constants Table**

The "Definition" column in the table above describes the values available for each pin when used with the PinIO class member function "function". For example, if pin JP1-17 needs to be configured for GPIO, then it would be written as:

```c
Pins[17].function( PIN17_GPIO );
```

Or, if I²C serial clock signal functionality is needed, then it would be written as:

```c
Pins[17].function( PIN17_SCL );
```

The "Function" column in the table describes the primary, alternate, and GPIO functions for each pin. The numbers to the left represent the following:

- 0 = GPIO
- 3 = Primary Function
- 1 = Alternate Function 1 (primary for some pins with dual configurations)
- 2 = Alternate Function 2
**Pins Class Member Functions**

Using the Pins class member functions to configure and use the GPIO pins eliminates the time and complexity of having to look up the proper documentation and use the right register and bits for a desired pin or set of pins. For example, if one were to use pin JP1-12 (UART 1 – Transmit) for GPIO and set it high without the PinIO class, then it would be written like this:

```c
#include <sim5270.h>

sim.gpio.par_uart &= ~0x0300; // Configure pin JP1-12 for GPIO
sim.gpio.ppdsdr_uartl |= 0x20; // Set bit to be driven out on pin
sim.gpio.pddr_uartl |= 0x20; // Set signal direction as output
```

Knowing the right register and bits are not required with the PinIO class, thus making it more convenient:

```c
#include <pins.h>
Pins[12].function( PIN12_GPIO ); // Configure pin JP1-12 for GPIO
Pins[12] = 1; // Set pin as output high
```

The following lists the member functions that can be used with the PinIO class:

<table>
<thead>
<tr>
<th>Member Function Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| **void** set()       | Set output high                      | Pins[15].set();
|                      |                                      | Pins[15] = 1;                                    |
| **void** clr()       | Set output low                       | Pins[4].clr();
|                      |                                      | Pins[4] = 0;                                     |
| BOOL read()          | Read pin high/low state              | BOOL bpinstate = Pins[5];
|                      |                                      | if ( !Pins[5] )
|                      |                                      | printf ( "The pin is low" );                     |
| **void** hiz()       | Set output to tri-state (high impedance input) | Pins[9].hz();                                   |
| **void** drive       | Turn output on (opposite of tri-state) | Pins[16].drive();                               |
| **void** function()  | Set pin to special function or GPIO  | Pins[8].function( PIN8_GPIO );
|                      |                                      | Pins[8].function( PIN8_UCTS2 );                  |
Program Examples

/****************************
* SIMPLE ALTERNATING HIGH/LOW OUTPUT PIN:
* This program configures pin JP1-17 as GPIO output. In an infinite
* loop, alternating high and low signals are driven out on the pin
* every second. The change in state of the pin can be confirmed by
* using a multimeter, oscilloscope, or connecting an LED between
* JP1-17 and ground. Another purpose for this example is to
* demonstrate the usage of the set() and clr() functions. In the
* next example, assigning '1' and '0' in place of set() and clr()
* are used respectively, but basically performs the same function.
****************************/

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <pins.h>

extern "C"
{
  void UserMain( void *pd );
}

const char *AppName = "SB70LC-PinsClassExample";

void UserMain( void *pd )
{
  InitializeStack();
  if ( EthernetIP == 0 ) GetDHCPAddress();
  OSChangePrio( MAIN_PRIO );
  EnableAutoUpdate();

  iprintf( "Application started\r\n" );

  Pins[17].function( PIN17_GPIO ); // Configure JP1-17 for GPIO

  while ( 1 )
  {
    OSTimeDly( 1 * TICKS_PER_SECOND );
    Pins[17].set(); // Set pin high
    OSTimeDly( 1 * TICKS_PER_SECOND );
    Pins[17].clr(); // Set pin low
  }
}
/************************************************** ********************
* SENDING SIGNALS FROM AN OUTPUT PIN TO AN INPUT PIN:
* 
* This program configures pins JP1-4 and JP1-16 as GPIO output and
* GPIO input, respectively. In order for this program to properly
* work, a jumper wire is needed to connect JP1-4 and JP1-16 together
* on the SB70LC module.
* 
* In an infinite loop, alternating high and low signals are driven
* out on JP1-4, where JP1-16 will then be read. If the signal read
* from JP1-16 is high, then the message "Hit!" will be outputted
* through the serial port to MTIYY. If the signal read from JP1-16
* is low, then the message "Miss!" will be outputted. After each
* send/read, there is a one-second delay.
************************************************** *********************/

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcppclient.h>
#include <pins.h>

 extern "C"
{
  void UserMain( void *pd );
}

const char *AppName = "SB70LC-PinsClassExample2";

void UserMain( void *pd )
{
  InitializeStack();
  if ( EthernetIP == 0 ) GetDHCPAddress();
  OSChangePrio( MAIN_Prio );
  EnableAutoUpdate();
  iprintf( "Application started\n\n" );
  Pins[4].function( PIN4_GPIO ); // Configure JP1-4 for GPIO
  Pins[16].function( PIN16_GPIO ); // Configure JP1-16 for GPIO
  while ( 1 )
  {
    OSTimeDly( 1 * TICKS_PER_SECOND );
    Pins[4] = 1; // Set JP1-4 output high
    if ( Pins[16] ) // Read JP1-16 input pin state
      iprintf( "Hit!\n\n" );
    else
      iprintf( "Miss!\n\n" );
    OSTimeDly( 1 * TICKS_PER_SECOND );
    Pins[4] = 0; // Set JP1-4 output low
if ( Pins[16] )                // Read JP1-16 input pin state
    iprintf( "Hit!\r\n" );
else
    iprintf( "Miss!\r\n" );
}