SB70LC PinIO Class

Application Note

Revision 1.2
October 28, 2009
Document Status: Initial Release

Table of Contents

Introduction 3

Electrical Specifications 3

PinIO Class 3

Pins Class Constants 3

Pins Class Member Functions 5

Program Examples 6

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:

Pin	**Definition**	**Function**
3	PIN3_SPI_CS0 PIN3_GPIO	1: SPI Chip Select 0 0: GPIO
4	PIN4_SPI_DOUT PIN4_GPIO	1: SPI Data Out 0: GPIO

5	PIN5_URT0_RTS PIN5_GPIO	1: UART 0 – Request to Send 0: GPIO
6	PIN6_SDI_DIN PIN6_SDA PIN6_GPIO	3: SPI Data In 2: I2C Serial Data 0: GPIO
7	PIN7_SPI_CLK PIN7_SCL PIN7_GPIO	3: SPI Clock 2: I2C Serial Clock 0: GPIO
8	PIN8_UCTS1 PIN8_UCTS2 PIN8_GPIO	3: UART 1 – Clear to Send 2: UART 2 – Clear to Send 0: GPIO
9	PIN9_URT1_RTS PIN9_URT2_RTS PIN9_GPIO	3: UART 1 – Request to Send 2: UART 2 – Request to Send 0: GPIO
10	PIN10_UTXD0 PIN10_GPIO	1: UART 0 - Transmit 0: GPIO
11	PIN11_URXD0 PIN11_GPIO	1: UART 0 - Receive 0: GPIO
12	PIN12_UTXD1 PIN12_GPIO	3: UART 1 - Transmit 0: GPIO
13	PIN13_URXD1 PIN13_GPIO	3: UART 1 - Receive 0: GPIO
14	PIN14_URT0_CTS PIN14_GPIO	1: UART 0 – Clear to Send 0: GPIO
15	PIN15_TIN3 PIN15_SPI_CS2 PIN15_URT2_CTS PIN15_GPIO	3: DMA Timer Input 3 1: SPI Chip Select 2 2: UART 2 – Clear to Send 0: GPIO
16	PIN16_SDA PIN16_GPIO	3: I2C Serial Data 0: GPIO
17	PIN17_SCL PIN17_GPIO	3: I2C Serial Clock 0: GPIO

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:

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

Or, if I^{2}C serial clock signal functionality is needed, then it would be written as:

```
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:

```
#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:

```
#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:

Member Function Name	**Description**	**Example**

`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] )`
		 `   iprintf ( "The pin is low" );`

`void hiz()`	Set output to tri-state (high impedance input)	`Pins[9].hiz();`
`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 MTTTY. 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 <dhcpclient.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\r\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!\r\n" );
      else
         iprintf( "Miss!\r\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" );

   }
}
```