NetBurner Development Kit

Mod5213 Programming Guide

Revision 1.2
September 14, 2009

Table of Contents

1 Introduction

The idea behind the Mod5213 development kit is to provide an embedded developer everything he or she needs to develop 32-bit embedded applications. The Mod5213 is pre-programmed with a factory demo application you can run right out of the box. The tools installation is quick and easy; just follow the prompts and the IDE, compiler and software will be installed. No configuration is required.

For those developers that are not familiar with the µC/OS real-time operating system (RTOS), do not worry. Use of the RTOS is not required. However, once you see firsthand how easy it is to use in the NetBurner environment, you may see things differently.

1.1 Development Kit Contents

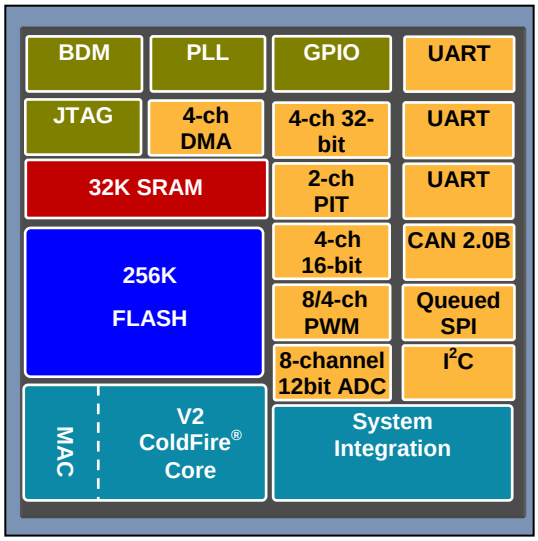
The NetBurner Mod5213 development kit (NDK) includes:

- A Freescale 5213 microprocessor-based module (NetBurner Mod5213)
- A development carrier board for the Mod5213 that includes a power regulator, 4 LEDs, reset switch, RS-232 level shifter, DB9 connectors, CAN transceiver and real-time clock
- Integrated development environment (IDE)
- Real-time operating system (RTOS)
- C/C++ compiler and linker
- Serial cable
- 12 VDC power supply

1.2 The Mod5213

1.2.1 ColdFire 5213 Processor Block Diagram

The NetBurner Mod5213 is based on the Freescale ColdFire 5213 microcontroller. A block diagram of the 5213 is shown below. The signal pins exposed on the Mod5213 come directly from the processor.



1.2.2 Mod5213 Features

The Mod5213 is based on a 32-bit 66 MHz Freescale ColdFire 5213 processor. Features of this chip include:

- ColdFire® V2 Core
- Temperature range: -40ºC to 85ºC
- 63 MIPS @ 66 MHz
- MAC module and HW divide
- Low-power optimization
- Standard 40-pin DIP
- 32 kB SRAM
- 256 kB flash
- CAN 2.0B controller with 16 message buffers
- Three UARTs with DMA capability
- Queued serial peripheral interface (QSPI)
- Inter-integrated circuit (I^2C) bus controller
- Four 32-bit timer channels with DMA capability
- Four 16-bit timer channels with capture/compare/PWM

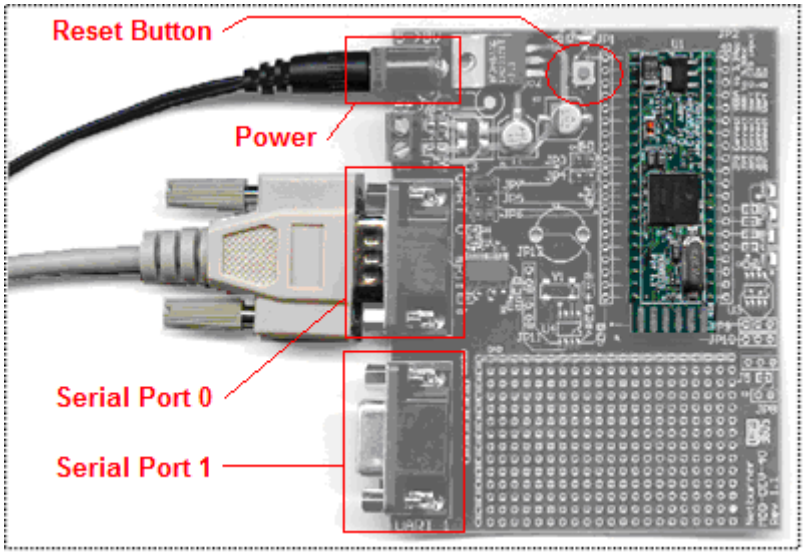
- 4-channel/16-bit or 8-channel/8-bit pulse-width modulation (PWM) generator
- Two programmable interrupt timers (PIT)
- 4-channel DMA controller
- 8-channel/12-bit ADC
- Up to 33 general-purpose I/O
- System integration [phase-locked loop (PLL), software watchdog]
- PIN Dimensions: 1.9" x 0.6"
- PCB Dimensions: 2.3" x 0.7"
- 4.5V to 7.5V input to integrated 3.3V regulator
- 3.3V I/O (not 5V tolerant)

1.3 Applying Power to the Mod5213

The Mod5213 has two power pins: one is a regulated 3.3 VDC input, and the other is a 4.5–7.5 VDC input. You can power the Mod5213 using either pin, but do not connect power to both at the same time. The development kit carrier board has its own voltage regulator that supplies regulated 3.3 VDC to the 3.3 VDC input power pin.

2 Development Kit Setup

This programming guide uses examples based on the Mod5213 and the development kit carrier board. To run the examples, you will need to set up your development hardware by connecting power, the serial port and Mod5213 module, if it isn't mounted already. The diagram below shows the proper connections. The serial cable is connected to serial port 0. The other DB9 is serial port 1. The Mod5213 must be inserted into the socket with the card edge connector facing the center of the board. There is a reset button at the top of the carrier board to the left of the Mod5213.

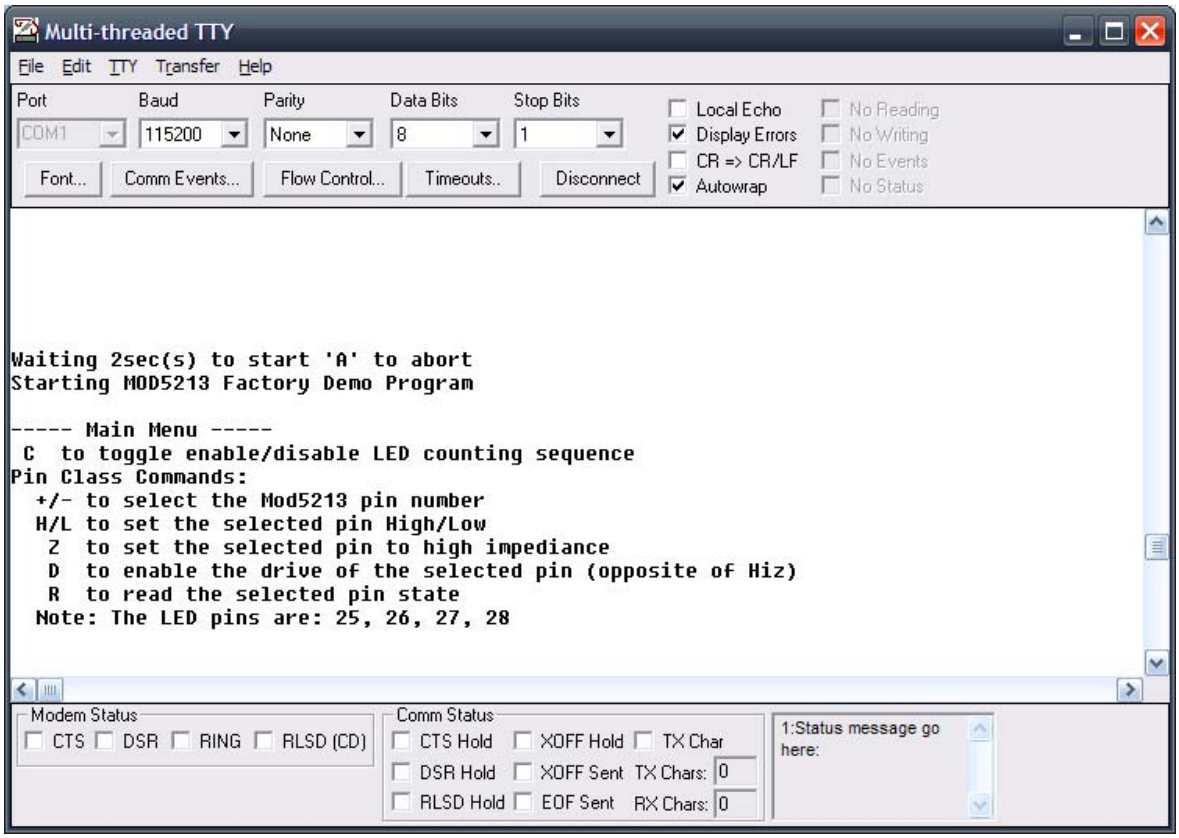


3 Running the Factory Demo

3.1 Setup

The Mod5213 factory demo is pre-programmed into your Mod5213. The demo uses serial port 0 and the LED display. To run the demo:

- Connect power and the serial port as described in the previous section
- Connect the other end of the serial cable to a serial port on your computer
- Run the NetBurner MTTTY program, which can be started from Start → All Programs → NetBurner NNDK → Mttty Serial Terminal, or from NBEclipse → Mttty of the NBEclipse IDE.
- Click on the "Flow Control..." button and verify that the checkboxes for "XON/XOFF Output Control" and "XON/XOFF Input Control" are checked.
- Select the COM port, 115,200 baud, no parity, 8 data bits and 1 stop bit, then click on the "Connect" button in MTTTY
- Press the reset button on the carrier board and verify that you see the Mod5213 boot message in the MTTTY window. An example of the message is shown below.
- If you do not see the boot message, then check the setup and repeat until the boot message appears.



3.2 Description of the Factory Demo

The factory demo is an example of a simple application that uses the RTOS and GPIO. The GPIO functions are based on the NetBurner Pin Class described later in this document. The application will run two tasks at the same time: one for the carrier board LED display, and the other to process user commands via the serial port. When the application runs, it will display the command menu and the LEDs will begin their display sequence. The source code is well documented, and is an excellent example to illustrate programming of the Mod5213.

3.3 Factory Demo Commands

The main menu in the MTTTY terminal window shows the valid commands:

```
 C  to toggle enable/disable LED counting sequence
+/- to select the Mod5213 pin number
H/L to set the selected pin High/Low
 Z  to set the selected pin to high impedance
 D  to enable the drive of the selected pin (opposite of Hiz)
 R  to read the selected pin state
```

The ‘C’ command tells the LED display task whether or not to update the LEDs. Pressing the ‘C’ key will toggle between enabling and disabling LED writes.

The remaining commands let you experiment with the GPIO functions of the Mod5213. Use the '+' / '-' keys to select a pin, then set the pin state to high, low, high impedance (disable drive), enable drive or read the state of the pin as an input. If you disable LED writes with the ‘C’ command, you can write the GPIO states of pins 25, 26, 27 and 28 high and low, which will turn the LEDs on and off.

3.4 Application Source Code

```
/*************************************************************************
   Mod5213 Factory Demo Program
   This program will illustrate how to implement multiple RTOS tasks, use the
   NetBurner Pin Class to control GPIO pins, initialize serial ports, and
   control the LEDs on the Mod5213 development kit carrier board.
*************************************************************************/

#include "predef.h"
#include <basictypes.h>                 // Include for variable types
#include <bsp.h>                        // 5213 board support package interface
#include <..\MOD5213\system\sim5213.h>  // 5213 structure
#include <ucos.h>                       // Include for RTOS functions
#include <smarttrap.h>                  // NetBurner Smart Trap utility
#include <serialirq.h>                  // Use serial interrupt driver
#include <utils.h>                      // Include for LED writes on carrier board
#include <SerialUpdate.h>               // Update flash via serial port
#include <constants.h>                  // Include for constants like MAIN_PRIO
#include <system.h>                     // Include for system functions
#include <stdio.h>
#include <pins.h>                       // NetBurner Pin Class
#include <gdbstub.h>


BOOL bLedSequenceEnable = TRUE;   // Enable LED display sequence by default


/*
   This declaration will tell the C++ compiler not to "mangle" the function name
   so it can be used in a C program.  We recommend you make all your file
   extensions .cpp to take advantage of better error and type checking, even if
   you write only C code.
*/
extern "C"
{
   void UserMain( void *pd );
}


/*-------------------------------------------------------------------------
   This will make the LEDs on the carrier board scan in a back-and-forth
   motion.
-------------------------------------------------------------------------*/
void LedScan()
{
   static unsigned char position = 0;
   const unsigned char pattern_array[] =  { 0x01, 0x02, 0x04, 0x08, 0x04,
                                            0x02 };
   if ( position > 5 )
      position = 0;
   putleds( pattern_array[position++] );
}
```

```
/*-------------------------------------------------------------------------
   This will make the LEDs on the carrier board count in binary.
-------------------------------------------------------------------------*/
void LedCount()
{
   static int n = 0;   // Init count value to 0
   putleds( n++ );     // Write new value to LEDs
}


/*-------------------------------------------------------------------------
   This is a RTOS task that writes to the LEDs on the carrier board. It will
   switch between two different light sequences.
-------------------------------------------------------------------------*/
void LedTask( void *p )
{
   static int SequenceCount = 0;      // Counts iterations for each LED seq
   static BOOL CountSequence = TRUE;  // Selects Count or Scan sequence

   while ( 1 )   // Loop forever
   {
      /*
         Use the RTOS OSTimeDly() to delay between LED writes. This function
         is a "blocking function", which means it lets lower priority
         tasks run while it is delaying.  This is extremely important in a
         preemptive OS.  Since this task is higher priority than UserMain(),
         it must block, otherwise UserMain() would never run.
      */
      OSTimeDly( TICKS_PER_SECOND / 8 );   // There are 20 ticks per second
      if ( bLedSequenceEnable )            // Check enable flag
      {
         if ( SequenceCount > 128 )  // Switch to other seq after 128 iterations
         {
            SequenceCount = 0;
            CountSequence = !CountSequence;
         }

         SequenceCount++;
         if ( CountSequence )
            LedCount();
         else
            LedScan();
      }
   }
}


/*-------------------------------------------------------------------------
   Display the command menu for user commands.
-------------------------------------------------------------------------*/
void DisplayCommandMenu()
{
   iprintf( "\r\n----- Main Menu -----\r\n" );
   iprintf( "   C  to toggle enable/disable LED counting sequence\r\n" );
   iprintf( "Pin Class Commands:\r\n" );
   iprintf( "  +/- to select the Mod5213 pin number\r\n" );
   iprintf( "  H/L to set the selected pin High/Low\r\n" );
```

```
   iprintf( "   Z  to set the selected pin to high impediance\r\n" );
   iprintf( "   D  to enable the drive of the selected pin " );
   iprintf( "(opposite of Hiz)\r\n" );
   iprintf( "   R  to read the selected pin state\r\n" );
   iprintf( "Note:  The LED pins are 25, 26, 27 and 28\r\n\r\n" );
}


/*-------------------------------------------------------------------------
   Process user serial command input
   The command processor has the following functions:

    C  = Toggle LED counting task enable/disable. You may want to disable LED
         counting so you can toggle pins 25, 26, 27 and 28 with the GPIO output
         commands and see the LEDs change state.

   +/- = Increment/decrement the selected pin number.  The selected pin number
         will respond to the other Pin Class commands such as High, Low, Hiz
         and Drive.

    D  = Enable selected pin's output drive.  The pin will output the state
         previously selected by High, Low or Read.

    H  = Set the selected pin's output to High.

    L  = Set the selected pin's output to Low.

    R  = Configure the selected pin to be an input and return the value (high
         or low).

    Z  = Put the selected pin in high impedance mode by disabling its output
         drive.
-------------------------------------------------------------------------*/
void ProcessCommand( char c )
{
   static int pinn = 4;           // Set initial selected pin value at 4

   iprintf( "Pin[%d]>", pinn );   // Display selected pin
   switch( c )
   {
      case '+':    // Increment the selected pin number
         pinn++;
         if( pinn > 38 ) pinn = 4;
         iprintf( "pin# = %d\r\n", pinn );
         break;
      case '-':    // Decrement the selected pin number
         pinn--;
         if( pinn < 4 ) pinn = 38;
         iprintf( "pin# = %d\r\n", pinn );
         break;
      case 'C':
      case 'c':
         bLedSequenceEnable = !bLedSequenceEnable;
         if ( bLedSequenceEnable )
            iprintf( "\r\nLED sequence display enabled\r\n" );
         else
            iprintf( "\r\nLED sequence display disabled\r\n" );
```

```
            break;
         case 'D':
         case 'd':
            Pins[pinn].function( pinx_GPIO );
            Pins[pinn].drive();
            iprintf( "Pin[%d] = Drive Enabled\r\n", pinn );
            break;
         case 'H':
         case 'h':
            Pins[pinn].function( pinx_GPIO );
            Pins[pinn] = 1;
            iprintf( "Pin[%d] = Hi\r\n", pinn );
            break;
         case 'L':
         case 'l':
            Pins[pinn].function( pinx_GPIO );
            Pins[pinn] = 0;
            iprintf( "Pin[%d] = Low\r\n", pinn );
            break;
         case 'R':
         case 'r':
            {
            Pins[pinn].function( pinx_GPIO );
            BOOL b = Pins[pinn];
            if ( b )
               iprintf( "Pin[%d] = reads Hi\r\n", pinn );
            else
               iprintf( "Pin[%d] = reads Low\r\n", pinn );
            }
            break;
         case 'Z':
         case 'z':
            Pins[pinn].function( pinx_GPIO );
            Pins[pinn].hiz();
            iprintf( "Pin[%d] = Hiz\r\n", pinn );
            break;
         default:
            DisplayCommandMenu();

      }
   }


/*-------------------------------------------------------------------------
   This is the RTOS main task called UserMain().  If you do not want to use
   the RTOS, then you could just write all your code in UserMain(), and treat
   it just like a standard C main().
-------------------------------------------------------------------------*/
void UserMain( void *pd )
{
   /*
      The following function calls will initialize two of the three UARTs to
      a default baud rate of 115,200 bps, 8 data bits, 1 stop bit and no
      parity.  There are other serial functions to call to specify additional
      parameters.  Serial ports are numbered 0, 1 and 2.
   */
   SimpleUart( 0, SystemBaud );
```

```
EnableSmartTraps();   // Enable NetBurner Smart Traps utility

/*
   This section of code will check for a DEBUG build and initialize
   the GDB stub for serial debugging.  The default serial port number on
   the module is 1, and the baud rate is 115,200.  You will need to set
   the baud rate in the debugger to match this value.
*/
#ifdef _DEBUG
InitGDBStubNoBreak( 1, 115200 );
#endif

/*
   When UserMain() starts, it is a very high priority.  Once running,
   it is standard practice to reduce it to something lower.  MAIN_PRIO
   is equal to a priority of 50.  This will enable you to add tasks at
   higher and lower priorities if you wish.
*/
OSChangePrio( MAIN_PRIO );

/*
   Create and start the LED counting task.  A task is basically just a
   function with a priority.  In this case, the function/task name is
   LedTask, and its priority is set to one level higher than UserMain().
   Note that a lower number is a higher priority.  By making the LEDs a
   higher priority than UserMain(), the LEDs will blink at a constant
   rate even during user input and character echo in UserMain().
*/
OSSimpleTaskCreate( LedTask, MAIN_PRIO - 1 );

/*
   Calling this function enables the flash memory updates via the serial
   port.  Serial updates will work at any time when you are using the
   serial interrupt driver, but in polled mode updates can only occur if
   the application is reading from the serial port [e.g., getchar(),
   read()].
*/
EnableSerialUpdate();

// Assign UART 0 to stdio so printf() and getchar() are routed there
assign_stdio( 0 );

// Write boot message to stdio, which is serial port 0
iprintf( "Starting MOD5213 Factory Demo Program\r\n" );

/*
   Write boot message to serial port 1 using writestring() function,
   since this port is not assigned to stdio.
*/
writestring( 1, "Greetings from serial port 1!\r\n" );

/*
   Loop forever. This is like a C main loop. You do not ever want to
   return from UserMain().
*/
DisplayCommandMenu();
```

```
   while ( 1 )
   {
      char c = sgetchar( 0 );
      ProcessCommand( c );
   }
}
```

4 Compiling, Downloading and Running an Example Program

Please refer to section four, "Creating a Project for a NetBurner Example Application" and section five, "Downloading Applications to NetBurner Hardware – Creating 'Run' and 'Debug' Configurations" in the "NBEclipse Getting Started Guide" on how to successfully run a Mod5213 example program. This PDF document can be accessed from the Windows Start menu (assuming a Windows XP environment): Start → All Programs → NetBurner NNDK → NBEclipse → NBEclipse Getting Started, or by directly accessing the file in \Nburn\docs\Eclipse.

The NBEclipse guide provides a full introduction to the NBEclipse IDE, and describes the steps of creating a project and importing existing example program source files to configuring your Run settings before finally downloading and running an application on the module. The NBEclipse guide is tailored to help users on how to program any NetBurner device, especially the non-networked Mod5213 module (the Mod5270 prime example in the guide section is used as a reference point for any device). Since the Mod5213 has no support for Ethernet networks, pay attention to parts of the section that draw attention to different options for the Mod5213.

4.1 Hardware Setup

Before following the NBEclipse guide, you must make sure that the Mod5213 is connected properly to your computer and have RS-232 serial communications. For example, you can run MTTTY and verify that you see the boot message when you press the reset button on the Mod5213 carrier board. To start the MTTTY serial terminal program, select Start → All Programs → NetBurner NNDK → Mttty Serial Terminal.

4.2 Using the .cpp Source Code File Extension

Even if you do not intend to use any C++ source code, it is recommended that you use the .cpp file extensions to take advantage of the enhanced compiler error and type checking. It will also enable you to use NetBurner APIs that do rely on C++.

5 How Serial Flash Downloads Work

The serial flash download to the Mod5213 is a very useful tool. Your application must include the following line:

```
#include <SerialUpdate.h>
```

and call the function `EnableSerialUpdate()` in order to enable this capability. The NetBurner application code will listen on any initialized serial ports for incoming updates, and process the update if the proper command sequence is sent. This is a preemptive feature, so a custom application is free to use the serial ports for any purpose.

The SerialLoad application tool or NBEclipse IDE (which in turn calls SerialLoad) will attempt to gain access to a serial port on your PC when executing a serial update. If no running applications are actively utilizing the serial ports, then the serial update process will perform an application download. MTTTY has been written to work with the serial update utility, which is also known as "Serial Update Tool" or "SerialLoad". If MTTTY is running on a serial port, then the serial update utility will run the update through MTTTY, provided that software flow control is enabled. However, if you are using another program or serial terminal (e.g., Microsoft Windows HyperTerminal) to interface with the serial port, then the serial update utility will not be able to use that port.

If polled serial I/O is being used in the custom application, then you need to remember that the serial update will only work if there is a blocking call to a serial input, such as `getchar()` or `read()`. If you are using interrupt-driven I/O, then this is not an issue.

The Mod5213's boot monitor also supports serial downloads, so if you are developing an application that is repeatedly crashing, then you can abort to the monitor prompt after reset by typing <Shift> + 'A' (or just 'A' if caps lock is enabled), followed by performing a serial update in the normal manner (by using the "Serial Update Tool" standalone application or by using the NBEclipse IDE).

6 Polled and Interrupt-Driven Serial Port Drivers

6.1 Polled vs. Interrupt-Driven

The NetBurner API provides two types of serial interfaces for the Mod5213 onboard UARTs: polled and interrupt-driven. You can switch between either mode easily just by changing an include file in your application; the application function calls are identical.

Polling means that any time your application attempts a serial read or write, the underlying code will block until a character can be read or written. This is accomplished by polling a status bit in the UART registers. The advantage of polling is that it takes up less SRAM resources than an interrupt-driven scheme since the serial I/O is not buffered.

Interrupt-driven means that the serial I/O is buffered so your application does not have to wait for the actual I/O to occur. It also means the application will not miss any incoming characters when it is busy elsewhere. Unless you are constrained on SRAM space, interrupt-driven serial I/O is recommended.

6.2 Serial Polling and Interrupt-Driven Example Programs

In the AppWizard-generated application below, we can see that it uses interrupt-driven serial I/O, as indicated by the included serialirq.h header file:

```
/*****************************************************************************
   Application generated by AppWizard
*****************************************************************************/

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <basictypes.h>
#include <serialirq.h>
#include <system.h>
#include <constants.h>
#include <ucos.h>
#include <SerialUpdate.h>


// Instruct the C++ compiler not to mangle the function name
extern "C"
{
   void UserMain( void *pd );
}


// Name for development tools to identify this application
```

```
const char *AppName = "NewApp5213";


// Main task
void UserMain( void *pd )
{
   OSChangePrio( MAIN_PRIO );
   EnableSerialUpdate();
   SimpleUart( 0, SystemBaud );
   assign_stdio( 0 );
   iprintf( "Application started\r\n" );

   while ( 1 )
   {
      OSTimeDly( TICKS_PER_SECOND );
   }
}
```

We can convert this to polled serial I/O just by changing the include file. However, if we want the serial flash update utility to work, we need to look (poll) for any incoming characters. The code to do this has been added to the `while` loop in `UserMain()` and highlighted in italics:

```
/*****************************************************************************
   Application generated by AppWizard
*****************************************************************************/

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <basictypes.h>
#include <serialirq.h>
#include <system.h>
#include <constants.h>
#include <ucos.h>
#include <SerialUpdate.h>


// Instruct the C++ compiler not to mangle the function name
extern "C"
{
   void UserMain( void *pd );
}


// Name for development tools to identify this application
const char *AppName = "NewApp5213";


// Main task
void UserMain( void *pd )
{
   OSChangePrio( MAIN_PRIO );
   EnableSerialUpdate();
   SimpleUart( 0, SystemBaud );
   assign_stdio( 0 );
   iprintf( "Application started\r\n" );
```

```
   while ( 1 )
   {
      // Check for I/O on UART 0
      if ( charavail( 0 ) )
         char c = getchar();

      OSTimeDly( TICKS_PER_SECOND );
   }
}
```

6.3 Modifying Interrupt Serial Buffer Values

The serial I/O buffer sizes for the Mod5213 are located in \Nburn\include_nn\constants.h, and are shown below (in bytes):

```
/**
 * Maximum UART buffer sizes.
 */
#define SERIAL0_RX_BUFFER_SIZE   (  256 )
#define SERIAL0_TX_BUFFER_SIZE   (  256 )
#define SERIAL1_RX_BUFFER_SIZE   (  256 )
#define SERIAL1_TX_BUFFER_SIZE   (  256 )
#define SERIAL2_RX_BUFFER_SIZE   (  256 )
#define SERIAL2_TX_BUFFER_SIZE   (  256 )
```

The default transmit and receive buffer size for each of the three UARTs is 256 bytes. The total memory used is: 256 bytes * 6 = 1536 bytes of SRAM. You can change these values to increase or decrease the overall memory usage. After making the modifications, the system library needs to be recompiled in order for the new changes to take effect. This can easily be done in the NBEclipse IDE by going to the top main menu and selecting: NBEclipse → Rebuild system files, followed by recompiling your custom application.

6.4 The NetBurner Serial API

The following sections describe the NetBurner serial API function calls. All the functions can be run in polled or interrupt-driven mode by changing the include file as described earlier in this chapter. Each API function call has the underlying polled and interrupt-driven functions defined.

6.4.1 Open a Serial Port

The following function calls are used to initialize a serial port. The `SimpleUart()` call assumes default values for the most common parameters (used 1 stop bit, 8 data bits and no parity).

```
int InitUart( int portnum,               // UART port number:  0, 1 or 2
              unsigned int baudrate,     // Baud rate:  1200-115,200 bps
              int stop_bits,             // Valid values:  1 or 2
              int data_bits,             // Valid values:  5, 6, 7 or 8
              parity_mode parity );      // Valid values:  eParityNone,
                                         //    eParityOdd, eParityEven or
                                         //    eParityMulti

int SimpleUart( int portnum,             // UART port number:  0, 1 or 2
                unsigned int baudrate);  // Baud rate:  1200-115,200 bps
```

Return Values:

```
Success                      (  0 )
SERIAL_ERR_NOSUCH_PORT       ( -1 )
SERIAL_ERR_PORT_NOTOPEN      ( -2 )
SERIAL_ERR_PORT_ALREADYOPEN  ( -3 )
SERIAL_ERR_PARAM_ERROR       ( -4 )
```

When serialpoll.h is included: `InitPolledUart()` is called.
When serialirq.h is included: `InitIRQUart()` is called.

6.4.2 Check if a Character is Available to be Read

```
BOOL charavail( int portnum );       // UART port number:  0, 1 or 2
```

When serialpoll.h is included: `polled_charavail()` is called.
When serialirq.h is included: `IRQ_charavail()` is called.

6.4.3 Get a Character

This function will block until a character is available to be read.

```
char sgetchar( int portnum );        // UART port number:  0, 1 or 2
```

When serialpoll.h is included: `polled_getchar()` is called.
When serialirq.h is included: `IRQ_getchar()` is called.

Note: The polled version does not yield to the RTOS, so no lower priority task can run. The IRQ version will yield to the RTOS until a character is available to be read.

6.4.4 Write a Character or String

Both of these functions will block until at least one character can be written.

```
void writechar( int portnum,         // UART port number:  0, 1 or 2
                char c );            // Character to write

void writestring( int portnum,       // UART port number:  0, 1 or 2
                  const char *s );   // Pointer to a constant string to write
```

When serialpoll.h is included: `polled_writechar()` / `polled_writestring()` is called.
When serialirq.h is included: `IRQ_writechar()` / `IRQ_writestring()` is called.

6.4.5 Close a Serial Port

```
void close( int portnum );           // UART port number:  0, 1 or 2
```

When serialpoll.h is included: `polled_close()` is called.
When serialirq.h is included: `IRQ_close()` is called.

6.4.6 Assign a Serial Port as stdio

This function will enable you to use stdio function calls with the specified serial port, such as `iprintf()`, `printf()`, `iscanf()` and `scanf()`.

```
void assign_stdio( int portnum );    // UART port number:  0, 1 or 2
```

When serialpoll.h is included: `polled_assign_stdio()` is called.
When serialirq.h is included: `IRQ_assign_stdio()` is called.

6.4.7 Assign a Serial Port as stderr

This function allows you to use standard error I/O with the specified serial port, such as `fprintf( stderr, ... )`, `fscanf( stderr, ... )`, etcetera.

```
void assign_sterr( int portnum );    // UART port number:  0, 1 or 2
```

When serialpoll.h is included: `polled_assign_err()` is called.
When serialirq.h is included: `IRQ_assign_err()` is called.

6.4.8 Create a Serial File Pointer

This creates a FILE-type pointer that can be used to read and write to a serial port with functions that take file pointers as parameters, such as `fprintf()`, `fscanf()`, etcetera.

```
FILE *fp = create_file( int portnum );   // UART port number:  0, 1 or 2
```

For example:

```
FILE *fp = create_file( 1 );                 // Create a FILE pointer for
                                             //    UART 1
fprintf( fp, "This goes out port 1\r\n" );   // Write string
fclose( fp );
```

When serialpoll.h is included: `polled_create_file()` is called.
When serialirq.h is included: `IRQ_create_file()` is called.

7 General Purpose I/O and the NetBurner Pin Class

The pins on the two 20-pin headers on the Mod5213 consist mostly of signal pins that can be set to a special function or GPIO, two power pins and a ground pin. Each signal pin can be set to a special function or GPIO. For example, pin 4 can be set to GPIO, I^2C Clock, CAN transmit or UART 2 transmit. When using a pin as GPIO, you can configure it as an input, or an output that can be low, high or high impedance. The table below summarizes the functions of each pin.

Connector Pin	Support GPIO?	GPIO Function	Primary Function	1st Alternate Function	2nd Alternate Function
1	No	-	Reset Input	-	-
2	Yes	PIN2_GPIO	PIN2_UART0_RX	-	-
3	Yes	PIN3_GPIO	PIN3_UART0_TX	-	-
4	Yes	PIN4_GPIO	PIN4_SDA	PIN4_CANRX	PIN4_UART2_RX
5	Yes	PIN5_GPIO	PIN5_SCL	PIN5_CANTX	PIN5_UART2_TX
6	Yes	PIN6_GPIO	PIN6_IRQ1	PIN6_SYNCA	PIN6_PWM1
7	Yes	PIN7_GPIO	PIN7_IRQ4	-	-
8	Yes	PIN8_GPIO	PIN8_IRQ7	-	-
9	No	-	VDDA	-	-
10	No	-	VRH	-	-
11	Yes	PIN11_GPIO	PIN11_AN2	-	-
12	Yes	PIN12_GPIO	PIN12_AN1	-	-
13	Yes	PIN13_GPIO	PIN13_AN0	-	-
14	Yes	PIN14_GPIO	PIN14_AN3	-	-
15	Yes	PIN15_GPIO	PIN15_AN7	-	-
16	Yes	PIN16_GPIO	PIN16_AN6	-	-
17	Yes	PIN17_GPIO	PIN17_AN5	-	-
18	Yes	PIN18_GPIO	PIN18_AN4	-	-
19	No	-	VSSA/VRL	-	-
20	No	-	Ground	-	-
21	Yes	PIN21_GPIO	PIN21_DTIN3	PIN21_DTOUT3	PIN21_PWM6
22	Yes	PIN22_GPIO	PIN22_DTIN2	PIN22_DTOUT2	PIN22_PWM4
23	Yes	PIN23_GPIO	PIN23_DTIN1	PIN23_DTOUT1	PIN23_PWM2
24	Yes	PIN24_GPIO	PIN24_DTIN0	PIN24_DTOUT0	PIN24_PWM0
25	Yes	PIN25_GPIO	PIN25_GPT3	-	PIN25_PWM7
26	Yes	PIN26_GPIO	PIN26_GPT2	-	PIN26_PWM5
27	Yes	PIN27_GPIO	PIN27_GPT1	-	PIN27_PWM3

28	Yes	PIN28_GPIO	PIN28_GPT0	-	PIN28_PWM1
29	Yes	PIN29_GPIO	PIN29_UART1_RX	-	-
30	Yes	PIN30_GPIO	PIN30_UART1_TX	-	-
31	Yes	PIN31_GPIO	PIN31_UART1_CTS	PIN31_SYNCA	PIN31_UART2_RX
32	Yes	PIN32_GPIO	PIN32_UART1_RTS	PIN32_SYNCB	PIN32_UART2_TX
33	Yes	PIN33_GPIO	PIN33_QSPI_CS2	-	-
34	Yes	PIN34_GPIO	PIN34_QSPI_CS1	-	-
35	Yes	PIN35_GPIO	PIN35_QSPI_CS0	PIN35_SDA	PIN35_UART1_CTS
36	Yes	PIN36_GPIO	PIN36_QSPI_DOUT	PIN36_CANTX	PIN36_UART1_TX
37	Yes	PIN37_GPIO	PIN37_QSPI_DIN	PIN37_CANRX	PIN37_UART1_RX
38	Yes	PIN38_GPIO	PIN38_QSPI_CLK	PIN38_SCL	PIN38_UART1_RTS
39	No	-	VDD (3.3VDC)	-	-
40	No	-	Unregulated Input Power, 4VDC – 7VDC	-	-

Note the supported function names for each pin. They represent the actual definition names that can be used to configure each pin if you choose to use the NetBurner Pin Class. These definitions are located in the header file \Nburn\MOD5213\include\pinconstant.h.

7.1 Which Pins Can Be Used as GPIO?

The signal pin description chart above specifies in the "Support GPIO?" column whether a pin can be used as GPIO. One exception to this case are the UART 0 receive and transmit signals; despite being configurable for GPIO, you will need these to interface with the Mod5213 system monitor.

7.2 What is the NetBurner Pin Class?

As you can see from the signal pin chart, each pin can be used for a variety of functions. However, before it can be made useful, one needs to know what module register needs to be written to and what bits need to be set in the MCF5213 processor in order to configure the signal pin. Rather than make the user read the extensive Freescale ColdFire 5213 reference manual, the Pin Class was created to make configuration and operation much easier. Although the Pin Class is written in C++, you do not need to know much C++ to use it; your application can be written using only C syntax.

Note that you do not need to use the Pin Class. If you prefer to handle the configuration and management yourself, then you are free to do so. If you do not include any Pin Class function calls, then none of the Pin Class code will be linked to your application. The Pin Class is a very efficient implementation with performance that will meet most requirements. It is simple to use – you may want to give it a try for a quick benchmark before writing your own configuration code.

7.3 Pin Class API Summary

To read, write or configure a pin where 'x' is the pin number:

```
Pins[x] = 0;        // Set GPIO output low
Pins[x] = 1;        // Set GPIO output high

Pins[x].hiz();      // Set GPIO to high impedance (tri-state)
Pins[x].drive();    // Turn on GPIO output

int n = Pins[x];    // Read GPIO input and return an int-type value
BOOL b = Pins[x];   // Read GPIO input and return a boolean-type value

Pins[x].function( FUNCTION );   // Configure pin for special function
```

The value of "`FUNCTION`" in the `Pins[x].function()` call above represents any definition name in the primary or alternate function columns of the signal description chart above. For example, to configure pins 4 and 5 for the CAN interface:

```
Pins[4].function( PIN4_CANTX );
Pins[5].function( PIN5_CANRX );
```

7.4 A Simple Pin Class GPIO Example

The following example illustrates how to read and write signal pins that can be used as GPIO. In this example, pin 25 is used as GPIO output since it is connected to a LED on the MOD-DEV-40 development board. This way, we can see the state of the pin as the LED turns on and off. Pin 4 is used as GPIO input – we cannot automatically change the state of the input pin, so it will always read low (0). The program runs in a loop that toggles the output state each ½ second, and reads the input pin once every second.

```
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // Initialize UART 0
   EnableSmartTraps();           // Enable SmartTrap utility
   OSChangePrio( MAIN_PRIO );    // Set standard main task priority
   EnableSerialUpdate();         // Enable Serial Update utility
   assign_stdio( 0 );            // Use UART 0 for stdio

   iprintf( "Starting SimpleGPIO Example\r\n" );

   while ( 1 )
   {
      // Configure pin 25 as an output and set it to 0.  This pin is connected
      // to a LED on the Mod5213 carrier board, so you can watch it blink.
      Pins[25] = 0;
      OSTimeDly( TICKS_PER_SECOND / 2 );

      // Setting pin 25 high (1) lights up the LED
      Pins[25] = 1;
```

```
      OSTimeDly( TICKS_PER_SECOND / 2 );

      int n = Pins[4];   // Read the current value of pin 4 as an input
      iprintf( "Pin[4] input value = %d\r\n", n );
   }
}
```

7.5 A Simple Pin Class Special Function Example

The following code illustrates how to configure a pin for a special function: I^2C. However, this only demonstrates how to handle the pin configuration. The application would still need to implement the I^2C driver.

```
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // Initialize UART 0
   EnableSmartTraps();           // Enable SmartTrap utility
   OSChangePrio( MAIN_PRIO );    // Set standard main task priority
   EnableSerialUpdate();         // Enable Serial Update utility
   assign_stdio( 0 );            // Use UART 0 for stdio

   iprintf( "Starting SimplePinFunction Example\r\n" );
   Pins[4].function( PIN4_SCL );
   Pins[5].function( PIN5_SDA );

   while ( 1 )
   {
      // Application code goes here
   }
}
```

8 Analog-to-Digital Functions

8.1 Mod5213 A/D Capabilities

The Mod5213 has two separate 12-bit A/D converters, each with their own sample-and-hold circuit. Each converter has four multiplexed analog inputs, providing eight channels of analog input.

The ColdFire 5213 processor's onboard A/D features include:

- 12-bit resolution
- Maximum ADC clock frequency of 5.33 MHz (period of 187.5 ns)
- Sampling rate up to 1.78 million samples per second (see Footnote 1)
- Single conversion time of 8.5 ADC clock cycles (8.5 * 187.5 ns = 1.595 µs)
- Additional conversion time of 6 ADC clock cycles (6 * 187.5 ns = 1.126 µs)
- Eight conversions in 26.5 ADC clocks (26.5 * 187.5 ns = 4.972 µs) using simultaneous mode
- Ability to simultaneously sample and hold two inputs
- Ability to sequentially scan and store up to eight measurements
- Internal multiplex to select two of eight inputs
- Power saving modes allow automatic shutdown/startup of all or part of ADC
- Inputs that are not selected can tolerate injected/sourced current without affecting ADC performance, supporting operation in noisy industrial environments
- Optional interrupts at the end of a scan if an out-of-range limit is exceeded (high or low), or at zero crossing
- Optional sample correction by subtracting a preprogrammed offset value
- Signed or unsigned result
- Single-ended or differential inputs for all input pins with support for an arbitrary mix of input types

Footnote 1: Once in loop mode, the time between each conversion is six ADC Clock cycles (1.125 µs). Using simultaneous conversion, two samples are captured in 1.126µs, providing an overall sample rate of 1,776,667 samples per second.

8.2 The NetBurner Mod5213 A/D API

The Mod5213 API supports automatic continuous sampling of all eight input channels, and a function to read the last sampled value for a particular analog input channel. If you need precise interrupt-driven sampling and control, then you will need to create an A/D driver specific to your application. The Freescale ColdFire 5213 Reference Manual is a good reference on how to configure the A/D system to meet your application requirements.

8.3 Default Sample Rate

The A/D sample rate for a single, active A/D channel is: 33,177,600 Hz / (clock_div * 6). The sample rate for any one channel must be divided by the number of active channels. For example, if all eight A/D channels are active, and the default value of "clock_div = 7" is used, then the sample rate per channel is: 33,177,600 Hz / (clock_div * 6 * 8 active channels) = 98,742 samples per second.

8.4 API Functions

Using the A/D functions with the Mod5213 API is very simple: call the function `EnableAD()` to activate the A/D background sampling, and then call `ReadA2DResult()` to read the latest sample. The functions are shown below:

```
void EnableAD( BYTE clock_div = 7 );    // Default clock divider value is 7
WORD ReadA2DResult( int ch );           // Valid channel numbers:  0 to 7
```

Note that the MCF5213 A/D hardware reports the 12-bit value in a 16-bit format to accommodate the many A/D operating modes. For a single-ended measurement, the count value is stored in the upper 12 bits of a 16-bit word.

8.5 Mod5213 Hardware Configuration

The Mod5213 provides the following connections on the 40-pin header:

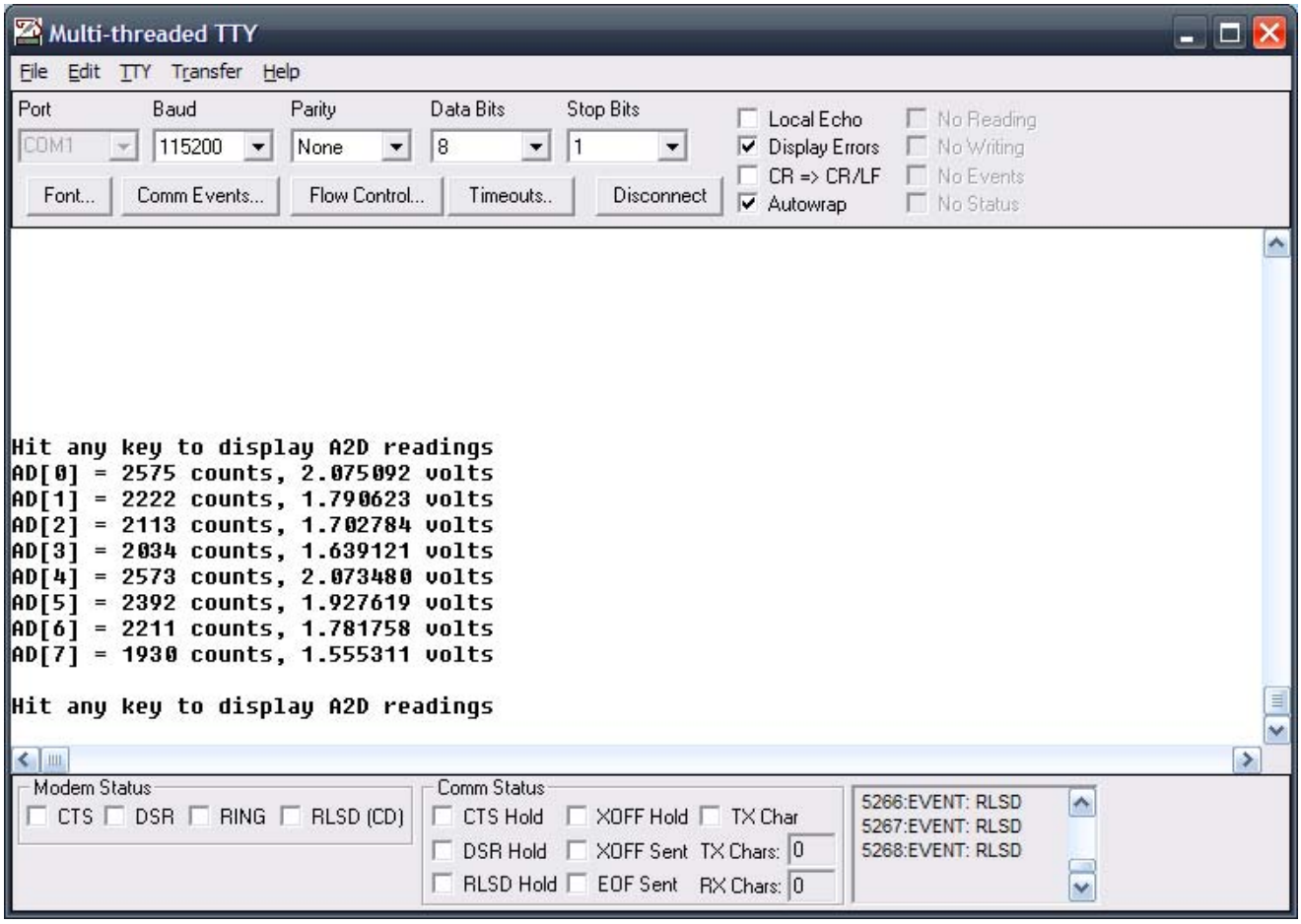
VDDA	:	A/D voltage input
VRH	:	A/D voltage reference high
VRL/VSSA	:	A/D voltage reference low and voltage power ground

The A/D on the Mod5213 has its own power and ground connections, as well as a separate A/D voltage reference input in case you want to use a precision voltage reference. The Mod5213 development kit carrier board has two jumpers that can be used for development purposes:

JP3	:	Connect VDDA to 3.3 VDC
JP4	:	Connect VRH to 3.3 VDC

8.6 A/D Example

The Mod5213 A/D example is located in \Nburn\examples\MOD5213\a2d. You can view the A/D readings via the serial port in the MTTTY terminal window when you run the example. A MTTTY screenshot is shown below:



8.7 A/D Example Source Code Listing

```
#include "predef.h"
#include <basictypes.h>
#include <bsp.h>
#include <..\MOD5213\system\sim5213.h>
#include <ucos.h>
#include <smarttrap.h>
#include <serialirq.h>
#include <utils.h>
#include <SerialUpdate.h>
#include <constants.h>
#include <system.h>
#include <stdio.h>
#include <a2d.h>
#include <Pins.h>


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


void UserMain( void *pd )
{
   OSChangePrio( MAIN_PRIO );
   EnableSmartTraps();
   EnableSerialUpdate();
   SimpleUart( 0, SystemBaud );
   assign_stdio( 0 );

   // Configure the A2D pins as analog inputs
   Pins[11].function( PIN11_AN2 );
   Pins[12].function( PIN12_AN1 );
   Pins[13].function( PIN13_AN0 );
   Pins[14].function( PIN14_AN3 );
   Pins[15].function( PIN15_AN7 );
   Pins[16].function( PIN16_AN6 );
   Pins[17].function( PIN17_AN5 );
   Pins[18].function( PIN18_AN4 );

   /*
      Enable the A2D.  The A2D subsystem will run in the background
      doing samples at 98.742 KHz per channel.  This is all done in
      hardware with no CPU overhead.
   */
   EnableAD();

   while ( 1 )
   {
      iprintf( "Hit any key to display A2D readings\r\n" );
      char c = sgetchar( 0 );   // Direct call to serial driver, not stdio
      for ( int i = 0; i < 8; i++ )
      {
```

```
            /*
               The count value returned by ReadA2DResult() is the value
               stored from the previous sample, at the 98.742KHz sample
               rate. The number of sample counts is stored as a 16-bit
               value to accomodate the varioius configurations of the
               A/D channels. Since we are doing a simple single ended
               measurement between 0 and 3.3V, we will left shift the
               count value so it falls within the 12-bit 4096 count range.
            */
            int counts = ReadA2DResult( i ) >> 3;
            float volts = ( ( float ) counts / ( 4095.0 ) ) * 3.3;
            printf( "AD[%d] = %d counts, %f volts\r\n", i, counts, volts );
         }

         iprintf( "\r\n" );
      }
   }
```

9 NetBurner µC/OS RTOS

The µC/OS is a very stable, fast and reliable operating system. The NetBurner implementation takes up very little memory, and it can make applications much easier to code and maintain. In most cases, applications will simply create a few tasks and pass messages between tasks, as in the case with the Mod5213 factory demo application. Even if you only need a single `UserMain()` task, some of the RTOS functions such as `OSTimeDly()` and the interrupt-driven serial I/O can come in handy.

The `UserMain()` function in the application code is equivalent to a standard C-type `main()` function. If that is all your application requires, then you may not need to use additional RTOS functions. But if you do need advanced features, µC/OS has plenty to offer. For more in-depth information on various RTOS functions, please refer to the "NetBurner's µC/OS RTOS Library" PDF document in \Nburn\docs\NetBurnerRuntimeLibrary\uCOSLibrary.pdf.

9.1 RTOS System Resource Usage

Use of the RTOS will have an impact on system resources. The default values for these resources are located in \Nburn\include_nn\constants.h. You have the option of specifying a non-default value when you call an RTOS function, or you can change the default values in constants.h. Whenever you make a change to a system file in \Nburn\include_nn (.h) or \Nburn\system_nn (.c, .cpp), be sure to rebuild the system libraries – either via NBEclipse or in the Windows command prompt – for the changes to take effect.

9.1.1 Task Stack Size

Each task has its own stack space. If you use the following function:

```
OSSimpleTaskCreate( TaskName, Priority );
```

the default stack size will be used (1024 bytes). The system idle and `UserMain()` tasks are created at boot and will use 256 bytes and 1024 bytes respectively. Their values are defined as:

```
#define IDLE_STK_SIZE            (  256 )   /* Idle task */
#define USER_TASK_STK_SIZE       ( 1024 )   /* User-defined tasks */
```

9.1.2 System Clock Tick

The RTOS system clock uses the MCF5213 Programmable Interrupt Timer (PIT) 0, which is configured with at interrupt request level '1' and priority level '3'. The default number of ticks per second is defined as:

```
#define TICKS_PER_SECOND          (    20 )    /* System clock tick */
```

If you need to change the ticks-per-second value, then the recommended values are between 20 and 200. For high resolution timing such a microsecond intervals, the best method is to use another hardware timer in the MCF5213, such as a separate PIT timer channel, or one of the DMA timer channels.

9.1.3 Maximum Number of Tasks

The maximum number of tasks is set at twenty by default. Allowing for reserved tasks, this number can be increased to fifty-six.

```
#define OS_MAX_TASKS              (    20 )    /* Maximum number of system tasks */
```

9.1.4 Task Priorities

The convention in µC/OS is that the lower the number, the higher the priority. This means a priority number of 50 is lower priority than 49. In a preemptive RTOS, the highest priority task (lowest priority number) will always run unless a blocking function is called in the higher priority task.

Priorities range from 0 to 63, with 0-3 reserved for future system usage, and 63 reserved for the system idle task. The system idle task is the lowest priority since it is designed to run only when nothing else of higher priority is available. It is just a series of assembly "nop" (no operation) instructions.

The recommended priority for `UserMain()` is 50. This number is arbitrarily selected so that applications can easily pick lower or higher priorities for other tasks. It also originated from NetBurner network-enabled platforms, where additional system-created tasks occupy priorities between 20 and 30.

```
#define MAIN_PRIO                 (    50 )    /* UserMain() */
```

9.1.5 Interrupt-Driven Serial Ports

The interrupt-driven serial port drivers also use the RTOS library, specifically the OS semaphore library functions for managing shared resources (i.e., serial receive and transmit buffers): `OSSemInit()`, `OSSemPend()` and `OSSemPost()`. Details on how these functions work can be found in the "NetBurner's µC/OS Library" PDF document. The interrupt-driven serial driver uses the buffers to store receiving and sending data. The default size for both receive and transmit buffers for all UARTs is 256 bytes. This value is defined in \Nburn\include_nn\constants.h, and is also mentioned in section 6.3 of this document.

9.1.6 Building Applications Without the RTOS

You can build applications without using the µC/OS RTOS if you want total control of the hardware. To do this, you create a `main()` function instead of a `UserMain()` function. An example is shown below:

```
#include "predef.h"
#include <basictypes.h>         // Include for user-defined types
#include <serialpoll.h>         // Use polled serial driver
#include <SerialUpdate.h>       // Enable support for serial flash update

/*
   The main() function is normally handled by the operating system.  You can
   declare your own if you want to have complete control over the hardware
   and not use the RTOS.
*/
int main()
{
   SimpleUart( 0, 115200 );
   EnableSerialUpdate();

   writestring( 0, "This application does not use the RTOS\r\n" );

   while( 1 );
}
```