NetBurner Networking in 1 Day!

NetBurner Development Kit

Mod5213 Programming Guide

Table of Contents

1 Introduction

The idea behind the Mod5213 development kit is to provide an embedded developer everything he/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 uC/OS real-time operating system (RTOS), don't 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

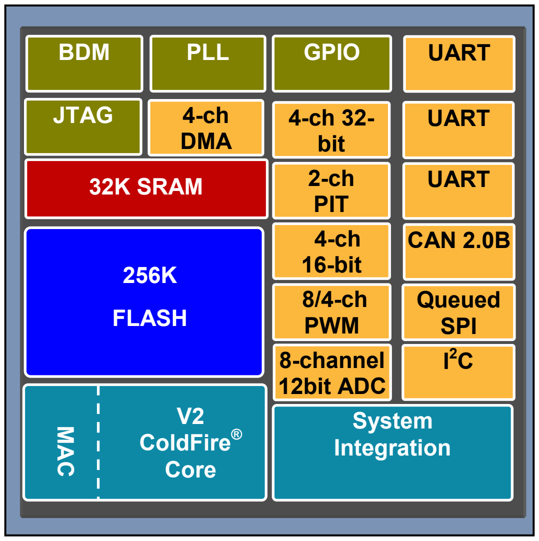
The NetBurner Mod5213 development kit (NDK) includes:

- A Freescale 5213 microprocessor based module (NetBurner Mod5213)
- A development carrier board for the Mod5213 that includes a power regulator, 4 LED's, reset switch, RS-232 level shifters and DB9 connectors. There are pad locations for optional CAN transceiver and real-time clock.
- Integrated Development Environment (IDE)
- Real-time operating system (RTOS)
- C/C++ Compiler and Linker
- Serial cable
- 12VDC 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 direct from the processor.



1.2.2 Mod5213 Features

The Mod5213 is based on a 32-bit 66Mhz 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 (I2C) bus controller
- Four 32-bit timer channels with DMA capability
- Four 16-bit timer channels with capture/compare/PWM
- 4-channel 16-bit/8-channel 8-bit PWM generator
- Two periodic interrupt timers (PITs)
- 4-channel DMA controller

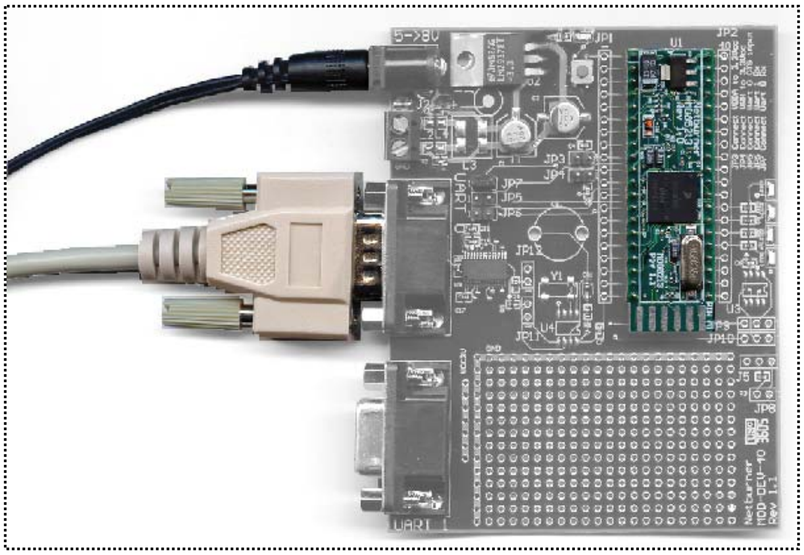
- 8-channel 12-bit ADC
- Up to 33 general-purpose I/O
- System integration (PLL, SW watchdog)
- PIN Dimensions: 1.9" x 0.6"
- PCB Dimensions: 2.3" x 0.7"
- 4.5V to 7.5V input to integrated 3.3V regulator
- 3.3V I/O (not 5V tolerant)

1.3 Applying Power to the Mod5213

The Mod5213 has 2 power pins; one is a regulated 3.3VDC 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 it's own voltage regulator, and supplies regulated 3.3VDC to the 3.3VDC 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 need to set up your development hardware by connecting power, the serial port and Mod5213 module. 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.

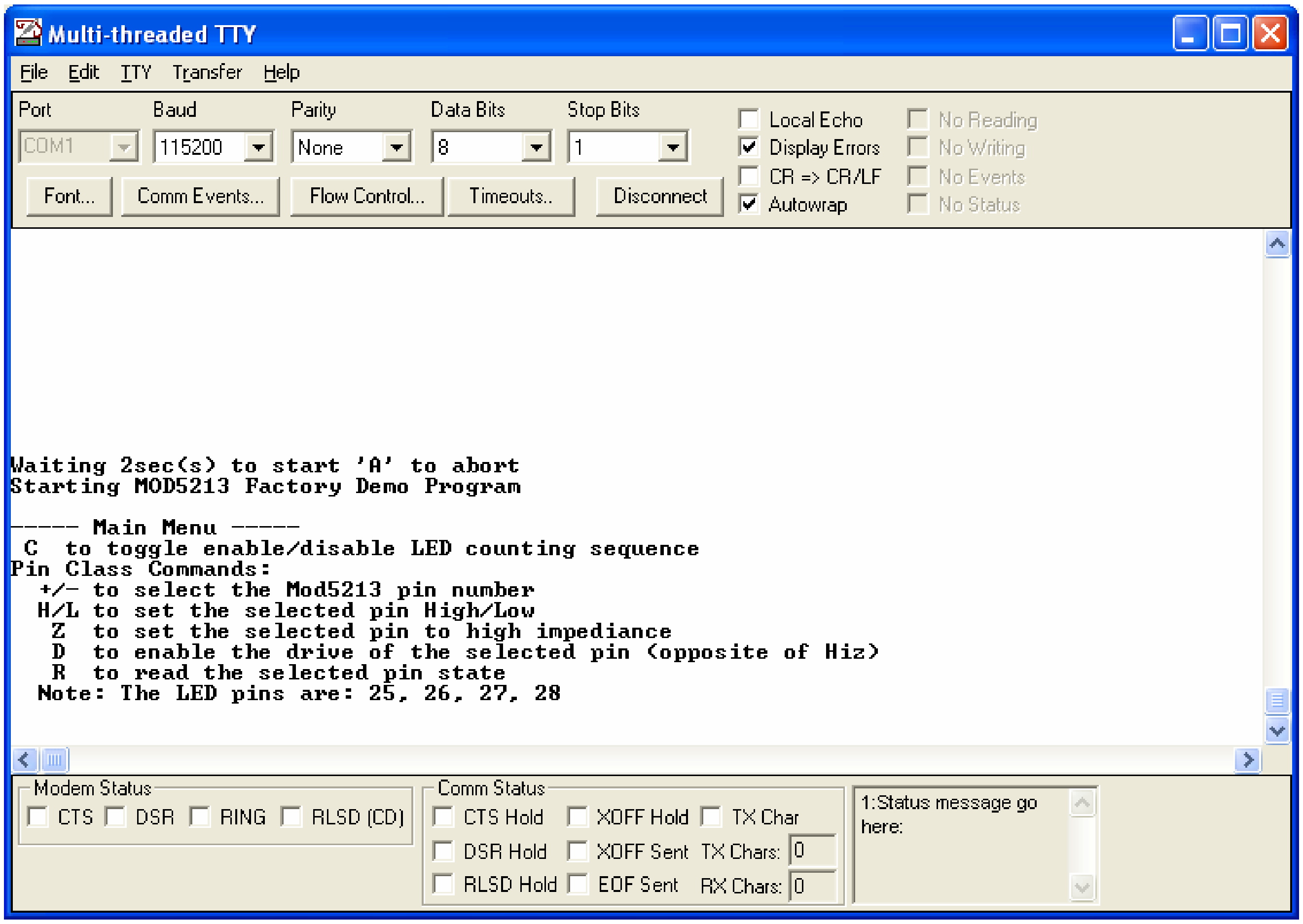


3 Running the Factory Demo

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 -> Programs -> NetBurner NNDK -> MTTTY, or from the Tools menu in DevC++.
- Click on the Flow Control button, and verify the checkboxes for XON/XOFF Output Control, and XON/XOFF Input Control are checked.
- Select the comm. port, 115,200k baud, and click on the Connect button in MTTTY
- Press the reset button on the carrier board and verify 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.



Port: COM1
Baud: 115200
Parity: None
Data Bits: 8
Stop Bits: 1
☐ Local Echo
☑ Display Errors
☐ CR => CR/LF
☑ Autowrap

```
Waiting 2sec(s) to start 'A' to abort
Starting MOD5213 Factory Demo Program

----- Main Menu ------
 C  to toggle enable/disable LED counting sequence
Pin Class Commands:
  +/- to select the Mod5213 pin number
  H/L to set the selected pin High/Low
   Z  to set the selected pin to high impediance
   D  to enable the drive of the selected pin (opposite of Hiz)
   R  to read the selected pin state
  Note: The LED pins are: 25, 26, 27, 28
```

☐ CTS ☐ DSR ☐ RING ☐ RLSD (CD)
☐ CTS Hold ☐ XOFF Hold ☐ TX Char
☐ DSR Hold ☐ XOFF Sent TX Chars: 0
☐ RLSD Hold ☐ EOF Sent RX Chars: 0

1:Status message go here:

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 starts it displays 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 the Mod5213.

3.3 Factory Demo Commands

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

```
Starting MOD5213 Factory Demo Program

----- Main Menu -----
 C  to toggle enable/disable LED counting sequence
Pin Class Commands:
 +/- to select the Mod5213 pin number
 H/L to set the selected pin High/Low
  Z  to set the selected pin to high impediance
  D  to enable the drive of the selected pin (opposite of Hiz)
  R  to read the selected pin state
  Note: The LED pins are: 25, 26, 27, 28
```

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 constands like MAIN_PRIO
#include <system.h>             // Include for system functions
#include <stdio.h>
#include <pins.h>               // NetBurner Pin Class


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
         preemtive 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 )   // After 128 iterations, switch to other seq
         {
            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 (opposite of
Hiz)\r\n");
    iprintf("   R  to read the selected pin state\r\n");
    iprintf("  Note: The LED pins are: 25, 26, 27, 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, 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 baud, 8 data bits, 1
      stop bit, no parity. There are other serial functions to call
      to specify additional parameters. Serial ports are numbered
      0, 1, 2.
   */
   SimpleUart( 0, SystemBaud );
   SimpleUart( 1, SystemBaud );

   /* Enable NetBurner Smart Traps Utility */
   EnableSmartTraps();

   /*
      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 (eg getchar(), read().
   */
   EnableSerialUpdate();

   /* Assign UART 0 to stdio, so printf(), 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 = getchar();
      ProcessCommand( c );
   }
}
```

4 Compile, Download and Run an Example Program

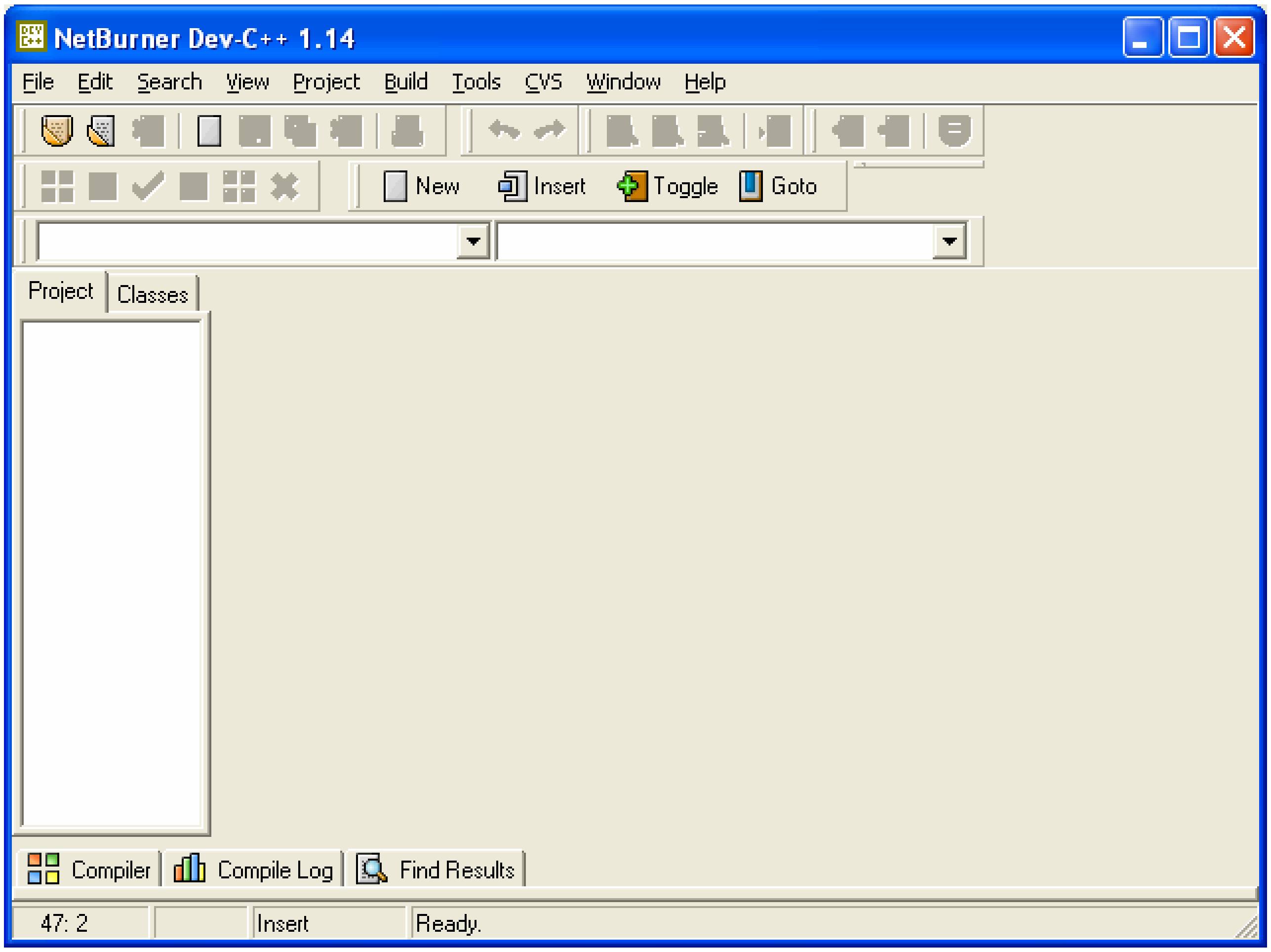
In this section we will compile, download and run a simple General Purpose I/O program that blinks one of the LED's on the carrier board.

4.1 Hardware Setup

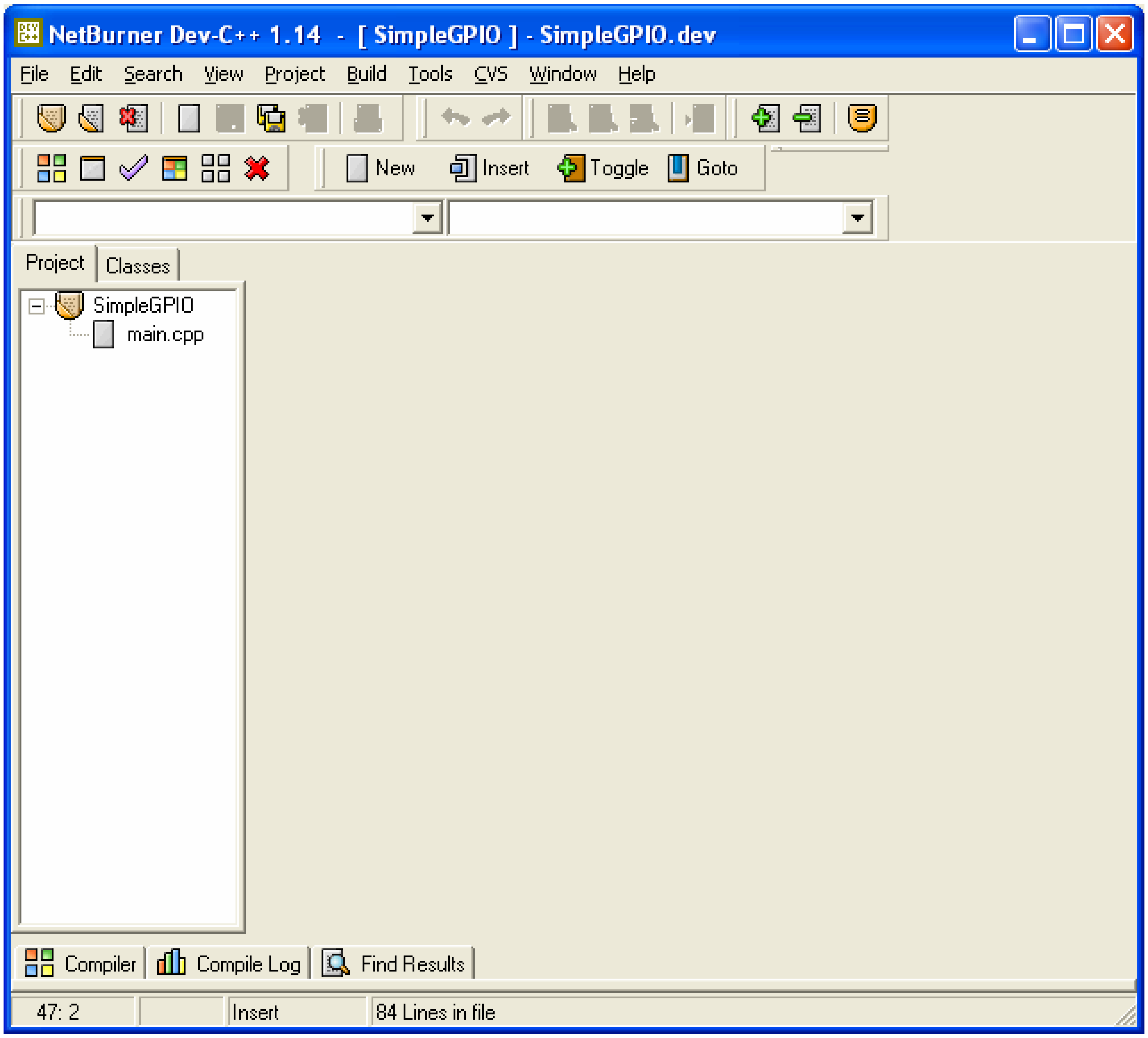
Before working through this exercise, you must make sure that the Mod5213 is connected properly to your computer and you have RS-232 serial communications. For example, you can run MTTTY and verify that you see the boot message when you press the reset button on the Mod5213 carrier board. To start the MTTTY serial terminal program, select Start -> Programs -> NetBurner NNDK -> MTTTY. You can also start MTTTY from the Tools menu of DevC++.

4.2 Open the SimpleGPIO Project

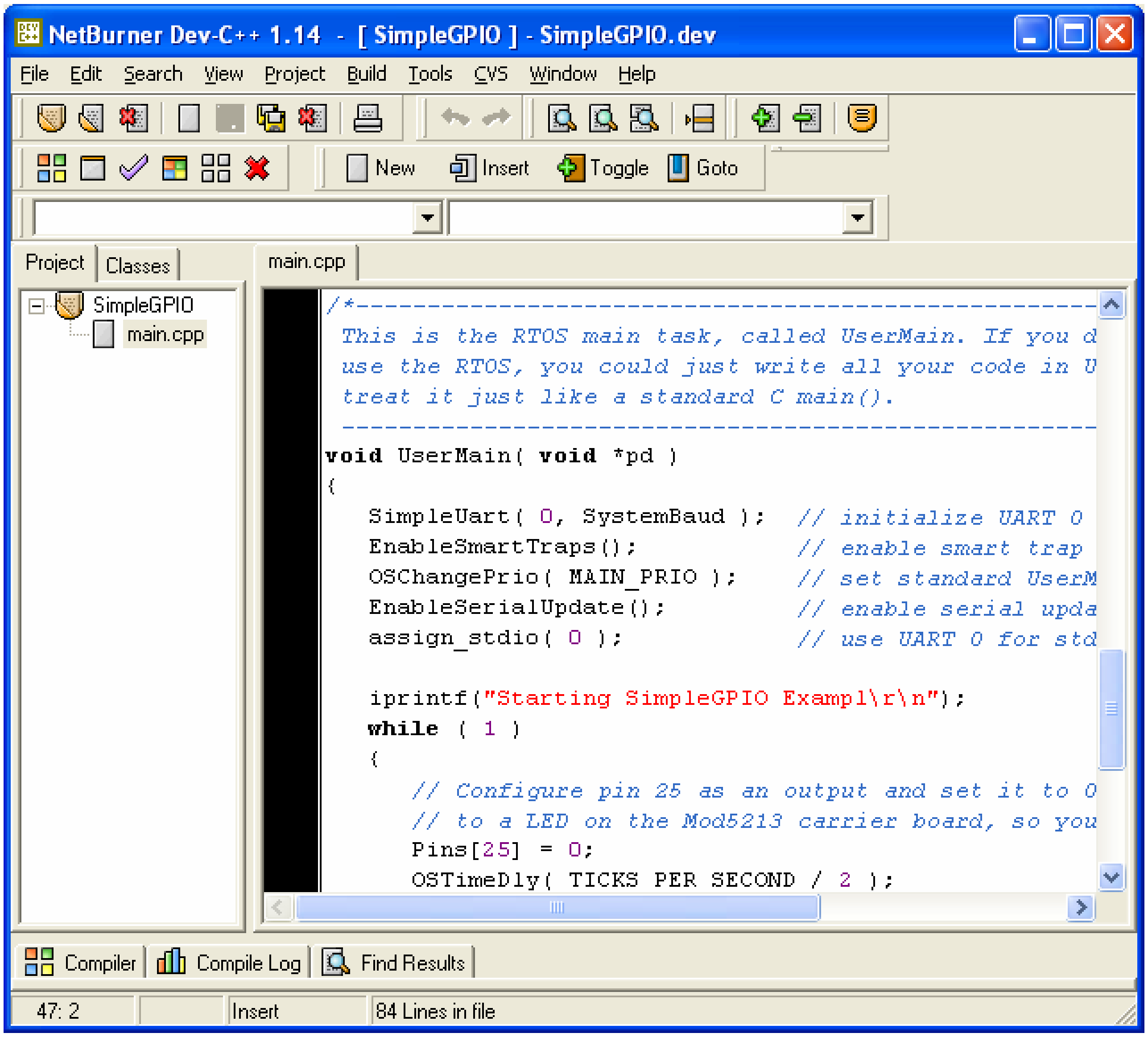
To begin, lets start DevC++ and load the application. Go to the Windows Start Menu and select Start-> Programs -> NetBurner NNDK -> DevC++. You should see the program start as shown below:


NetBurner Dev-C++ 1.14
File Edit Search View Project Build Tools CVS Window Help
New Insert Toggle Goto
Project Classes
Compiler Compile Log Find Results
47: 2 Insert Ready.

There is a project file already created for this example. In DevC++, click on File -> Open Project or File, and open the file named \nburn\examples\Mod5213\SimpleGPIO.dev. File names that end in .dev are DevC++ project files. Once you open the project, you should see the SimpleGPIO project in the Project Pane on the left as shown below.



Now click on “main.cpp” in the Project pane to open the main.cpp source code file in the edit window.


NetBurner Dev-C++ 1.14 - [SimpleGPIO] - SimpleGPIO.dev
File Edit Search View Project Build Tools CVS Window Help
New Insert Toggle Goto
Project Classes
SimpleGPIO
main.cpp
Compiler Compile Log Find Results
47: 2 Insert 84 Lines in file

```
/*------------------------------------------------------------
 This is the RTOS main task, called UserMain. If you d
 use the RTOS, you could just write all your code in U
 treat it just like a standard C main().
 ------------------------------------------------------------
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // initialize UART 0
   EnableSmartTraps();           // enable smart trap
   OSChangePrio( MAIN_PRIO );    // set standard UserM
   EnableSerialUpdate();         // enable serial upda
   assign_stdio( 0 );            // use UART 0 for std

   iprintf("Starting SimpleGPIO Exampl\r\n");
   while ( 1 )
   {
      // Configure pin 25 as an output and set it to 0
      // to a LED on the Mod5213 carrier board, so you
      Pins[25] = 0;
      OSTimeDly( TICKS PER SECOND / 2 );
```

4.3 Compile and Download the SimpleGPIO Application

Before reviewing the source code, let’s compile and run the example program. This can be done with a single command. From the DevC++ main menu, select Build -> Compile & Load. As long as the application running on the Mod5213 has the EnableSerialUpdate( ) function, the NetBurner tools will automatically download the new application and program the flash memory in the Mod5213. After executing the Compile & Load command, you should see a quick progress bar as the file is downloaded into the device, then the boot message should appear on MTTTY as the Mod5213 reboots with the new application. One of the LED’s should now be blinking.

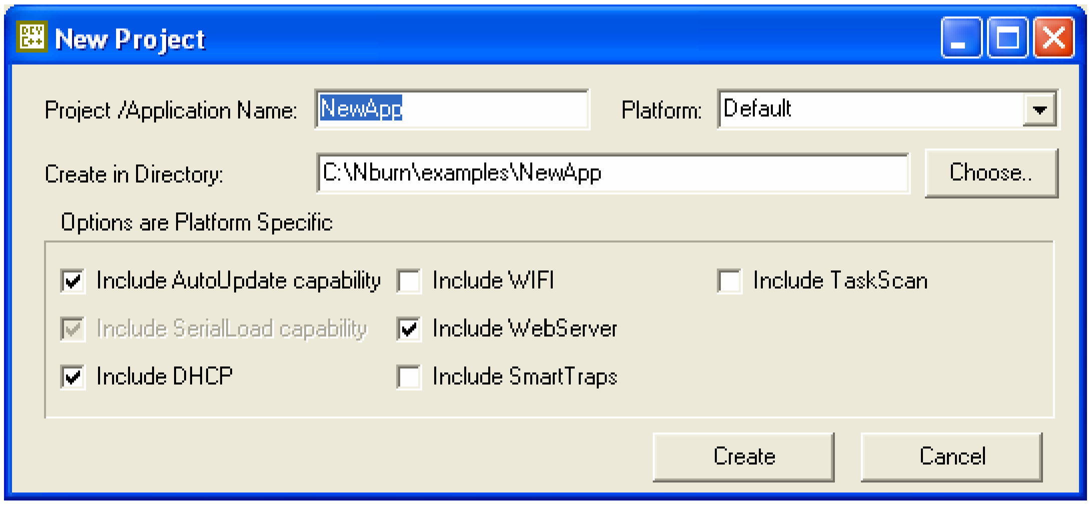
4.4 SimpleGPIO Source Code

```
/******************************************************************************
 Simple GPIO program using the Pin Class

 *****************************************************************************/
#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 <serialirq.h>          // Use UART interrupts instead of polling
#include <utils.h>              // Include for LED writes on carrier board
#include <SerialUpdate.h>       // Update flash via serial port
#include <smarttrap.h>          // NetBurner Smart Trap ability
#include <constants.h>          // Include for constands like MAIN_PRIO
#include <system.h>             // Include for system functions
#include <pins.h>               // Include for Pin Class API


extern "C"
{
   void UserMain( void *pd );   // prevent C++ name mangling
}


/*-------------------------------------------------------------------
 This is the RTOS main task, called UserMain. If you do not want to
 use the RTOS, you could just write all your code in UserMain(), and
 treat it just like a standard C main().
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // initialize UART 0
   EnableSmartTraps();           // enable smart trap utility
   OSChangePrio( MAIN_PRIO );    // set standard UserMain task priority
   EnableSerialUpdate();         // enable serial updates
   assign_stdio( 0 );            // use UART 0 for stdio

   iprintf("Starting SimpleGPIO Exampl\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 );

      // set pin 25 to a 1
      Pins[25] = 1;
      OSTimeDly( TICKS_PER_SECOND / 2 );

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

5 Creating Your First Custom Program

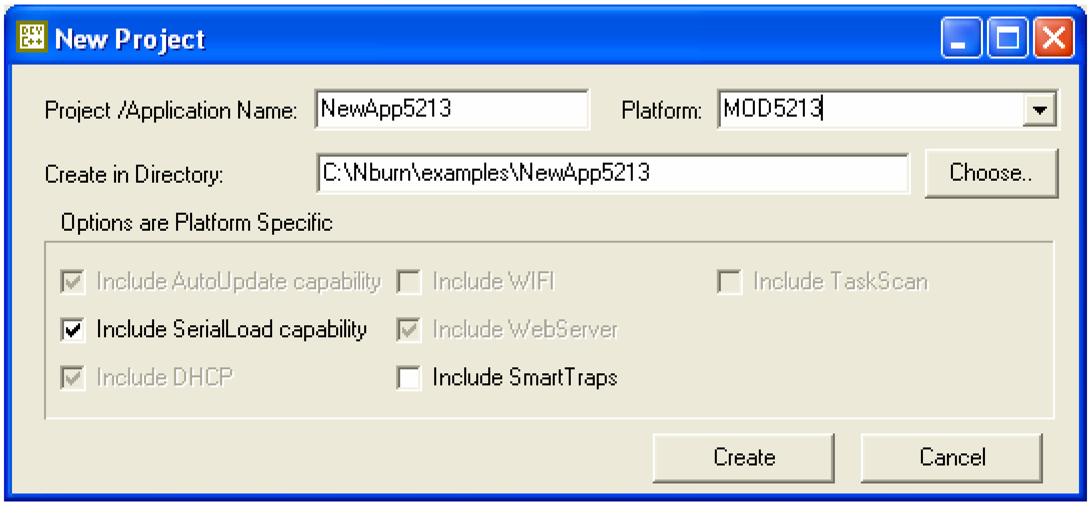
5.1 Using the AppWizard

A very fast way to create a minimal application is to use the DevC++ AppWizard. From the DevC++ main menu, select File -> New -> AppWizard. A dialog box with options will appear:



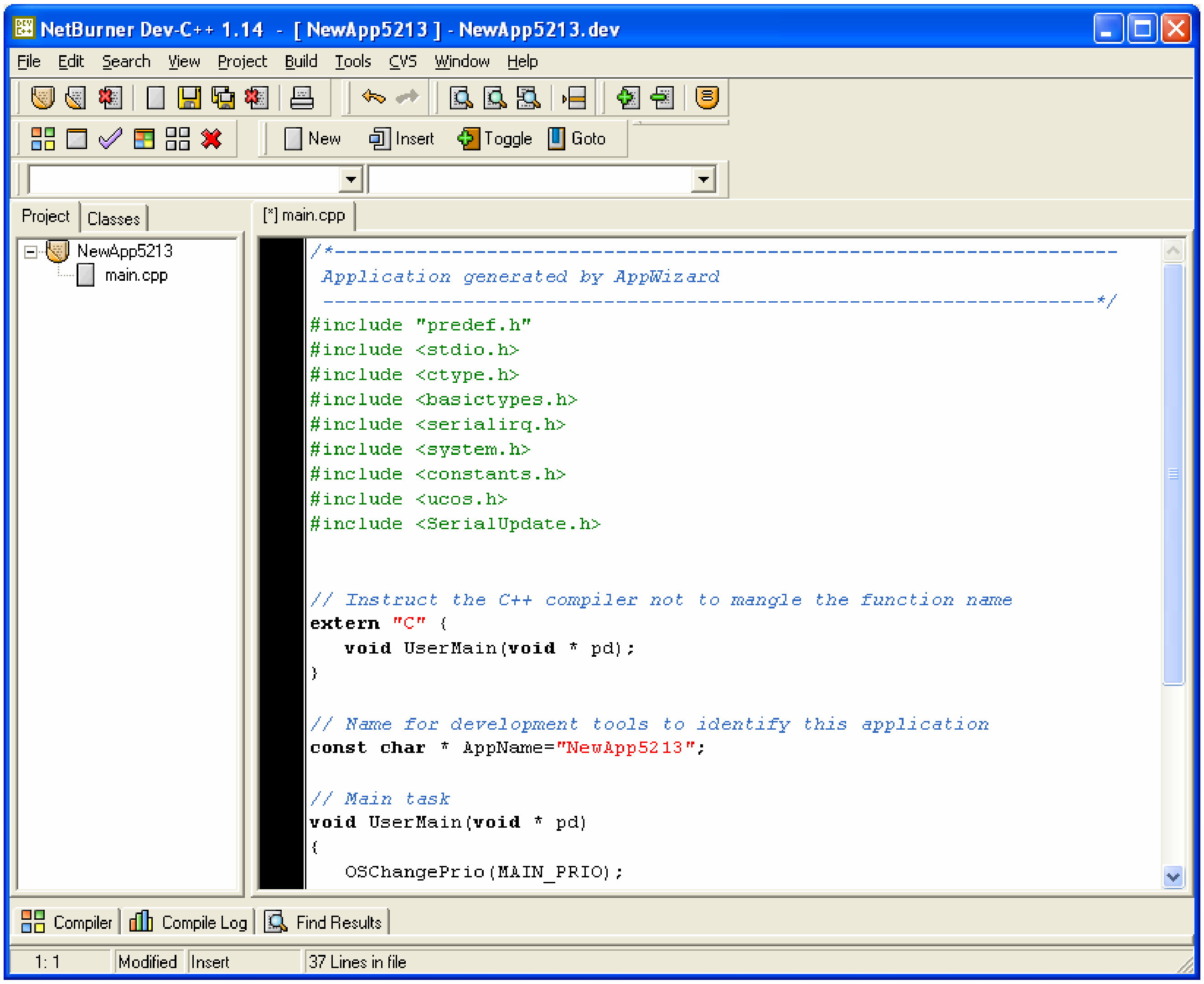
Project /Application Name: NewApp
Platform: Default
Create in Directory: C:\Nburn\examples\NewApp
Options are Platform Specific
[✓] Include AutoUpdate capability
[✓] Include SerialLoad capability
[✓] Include DHCP
[ ] Include WIFI
[✓] Include WebServer
[ ] Include SmartTraps
[ ] Include TaskScan

Depending on your installed platform, a number of options can appear. In the dialog box above, multiple NetBurner platforms are installed, and the default platform supports networking. In our case we will rename the application name to NewApp5213, and change the platform to Mod5213. These changes will result in the dialog box below:



Project /Application Name: NewApp5213
Platform: MOD5213
Create in Directory: C:\Nburn\examples\NewApp5213
Options are Platform Specific
[✓] Include AutoUpdate capability
[✓] Include SerialLoad capability
[✓] Include DHCP
[ ] Include WIFI
[✓] Include WebServer
[ ] Include SmartTraps
[ ] Include TaskScan

As you can see, the options are now Mod5213 specific. For this example, let's just use the SerialLoad capability to enable flash updates through the serial port. Once the option is selected, click on the Create button to create the source code and project. You should now see a project and source window like the one below:


NetBurner Dev-C++ 1.14 - [NewApp5213] - NewApp5213.dev
File Edit Search View Project Build Tools CVS Window Help
New Insert Toggle Goto
Project Classes
NewApp5213
main.cpp
[*] main.cpp
Compiler Compile Log Find Results
1: 1 Modified Insert 37 Lines in 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);
```

5.2 AppWizard Source Code Listing

The source code generated by the AppWizard is shown on the following pages. It creates an application shell consisting of a UserMain( ) task, the ability to handle a serial flash update, and serial communications on UART 0. The while(1) loop is where we will add our custom application code. Any system initialization or creation of other tasks usually takes place before the while(1) loop, such as serial port initialization, task priority and enabling serial updates.

If you do not want to do any RTOS programming, you can simply write all your application code inside UserMain( ). The preemptive RTOS will not waste any CPU cycles if you do not have other tasks, and you will be able to take advantage of the serial port drivers and serial port flash updates.

In the example you may notice that iprintf( ) is used instead of printf( ). The ‘i’ stands for “integer”. By using iprintf( ) you can avoid linking in the floating point support that printf( ) requires, and save 30k – 40k bytes of code space.

```
/*-------------------------------------------------------------------
 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 );
   }
}
```

5.3 Using the .cpp Source Code File Extension

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

5.4 Modifying the AppWizard Example

Now lets modify the AppWizard example and run it. The changes are highlighted in bold. To access the LEDs on the carrier board we need to add #include <utils.h>. Variable initialization and changes to the while (1) loop are shown below.

```
#include <utils.h>


int n = 0;
while ( 1 )
{
   iprintf( “I am a Mod5213!\r\n” );
   putleds( n++ );
   OSTimeDly( TICKS_PER_SECOND );
}
```

Once you have made the changes, select Compile & Load (from the main menu, the icon under the main menu, or by pressing the F9 key). When the application runs you should see the text scrolling by and LED’s counting once per second.

6 Determining Flash and SRAM Usage

Each time you compile an application, the amount of flash memory used is displayed in the output window. If your application attempts to use more flash or SRAM than is physically available, a warning will be generated.

An example output of an application that uses 38k of flash is shown below:

```
Block starts at ffc04010
Block ends at ffc0db40
Block size = 38k (39728 bytes)
```

To determine SRAM usage you will need to look at the .map file generated in your project directory. The map file will show the SRAM based variables by name and address. An example map file with the intermediate variables removed is shown below:

```
*(.data)
 .data          0x20000400        0x4 ../m68k-elf/lib/m5206e/crt0.o

<< intermediate variables omitted in this example >>

                0x200025d0               __end = _end
```

SRAM usage includes the .data and .bss sections. A simple way to determine the memory used is to locate the *(.data) label, and the __end = _end label. The amount of memory used is the difference between the two hexadecimal numbers. The above example is from an application that uses the RTOS, Pin Class, and interrupt driven serial I/O:

```
  0x200025d0 - 0x20000400 = 0x21D0 = 8,656 bytes.
```

Typical application sizes can range from 9k bytes of flash space for minimal implementations that do not use the RTOS or library calls like printf( ), to 40k bytes in applications that use the RTOS, stdio, and library calls like iprintf( ). Since the Mod5213 has 256k bytes of flash space, you have plenty of room! Typical SRAM usage can range from 1.5k bytes to 8k bytes for a full featured application with interrupt driven serial I/O with associated memory buffers.

7 How Serial Flash Downloads Work

The serial flash download to the Mod5213 is a very useful tool. To enable this capability, your application must include the header file #include <SerialUpdate.h>, and call the function EnableSerialUpdate( ). The NetBurner application code will listen on all serial ports for incoming updates, and process the update if the proper command sequence is sent. This is a preemptive feature, and you application can freely use this serial port for any purpose you wish.

When executing a serial update, DevC++ will attempt to gain access to a serial port on your PC. If no applications are running that are using a serial port the update will open the serial port, do the update, and close the serial port. MTTTY has been written to be cooperative with the serial update utility. If MTTTY is running on a serial port, the serial update utility will run the update through MTTTY. However, if you have any other program or serial terminal (eg Windows HyperTerminal) using a serial port, the serial update utility will not be able to use that port.

For those who want to use polled serial I/O, 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 monitor program also support serial downloads, so if you are developing an application that is repeatedly crashing, you can enter the monitor at the prompt by typing an 'A', and then proceed with a serial update in the normal manor (e.g. Compile & Load in DevC++).

8 Polled and Interrupt Driven Serial Port Drivers

8.1 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 because it is busy elsewhere. Unless you are constrained on SRAM space, interrupt driven serial I/O is recommended.

8.2 Example Serial Polling and Interrupt Driven Programs

If we look back at the AppWizard generated application, we can see that it uses interrupt driven serial I/O, shown in bold below:

```
/*-------------------------------------------------------------------
 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 covert 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 be looking for (polling) any incoming characters. The code to do this has been added to UserMain( ) and highlighted in bold:

```
/*-------------------------------------------------------------------
 Application generated by AppWizard
 ------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <basictypes.h>
#include <serialpoll.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 )
   {
      if ( charavail( 0 ) )   // check for I/O on UART 0
         char c = getchar();
      OSTimeDly( TICKS_PER_SECOND );
   }
}
```

8.3 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. I have included all the definitions in the header file so you can see other system parameters as well.

```
#define TICKS_PER_SECOND (20)   /* System clock tick */
#define OS_MAX_TASKS    20    /* Max number of system tasks */

/* IDLE task is set at lowest priority, 63 */
#define MAIN_PRIO (50)        /* used for UserMain() */
#define IDLE_STK_SIZE (256)
#define USER_TASK_STK_SIZE (512)

#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)
```

As you can see, the default buffer sizes for each of the three UARTs is 256 bytes for transmit and receive. The total memory used is 256 * 6 = 1,536 bytes of SRAM. You can change these value to increase or reduce buffer size. After making a change you need to recompile the system library. This is easily done in DevC++ by selecting Build -> Rebuild All, then compiling your project with Compile or Compile & Load.

8.4 The NetBurner Serial API

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

8.4.1 Open a Serial Port

The following function calls are used to initialize a serial port. The "simple" version assumes default values for the most common parameters.

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

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

SimpleUart( ) will use 1 stop bit, 8 data bits and no parity.

Return Values:
- 0 on Success
- SERIAL_ERR_NOSUCH_PORT
- SERIAL_ERR_PORT_ALREADYOPEN
- SERIAL_ERR_PARAM_ERROR

Polled version: Calls InitPolledUart( ).
Interrupt version: Calls InitIRQUart( ).

When using interrupt driven I/O, the serial buffer sizes are defined in \nburn\include_nn\constants.h.

8.4.2 Check if a Character is Available to be Read

```
BOOL charavail( int portnum );      // 0, 1 or 2
```

Returns true if a char is available to be read.
Polled version: Calls Polled_charaval( ).
Interrupt version: Calls IRQ_charavail( ).

8.4.3 Get a Character

```
char sgetchar( int portnum );       // 0, 1 or 2
```

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

Polled version: Calls Polled_getchar( ).
Interrupt version: Calls IRQ_getchar( ).

Important: 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 char is available.

8.4.4 Write a Character

```
void writechar( int portnum,          // 0, 1 or 2
                char c);              // character to write


void writestring( int portnum,             // 0, 1 or 2
                  const char * s );        // pointer to a string to write
```

Where **portnum** is the port number 0, 1 or 2. The variable **s** is a pointer to a constant string to be sent.

Polled version: Calls Polled_write( )
Interrupt version: Calls IRQ_write( )

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

8.4.5 Close a Serial Port

```
void close( int portnum );            // 0, 1 or 2
```

Polled version: Calls Polled_close( ).
Interrupt version: Calls IRQ_close( ).

8.4.6 Assign a Serial Port as Stdio

```
void assign_stdio( int portnum );     // 0, 1 or 2
```

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

Polled version: Calls Polled_assign_stdio( ).
Interrupt version: Calls IRQ_assign_stdio( ).

8.4.7 Assign a Serial Port as Stderr

```
void assign_sterr( int portnum );   // 0, 1 or 2
```

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

Polled version: Calls Polled_assign_sterr( ).
Interrupt version: Calls IRQ_assign_sterr( ).

8.4.8 Create a Serial File Pointer

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

Creates a pointer of type FILE that can be used to read and write to serial port with functions that take file pointers as parameters, such as: fprintf( ), fscanf( ), etc...

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);
```

Polled version: Calls Polled_create_file( ).
Interrupt version: Calls IRQ_create_file( ).

9 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, QSPI 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 set high, low or high impedance. The table below summarizes the functions of each pin. Each NetBurner platform that supports the Pin Class will have its own definition file located in \nburn\<platform>\include\pincostant.h.

Connector Pin	Can this be used as 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_SCL	PIN4_CANTX	PIN4_UART2_TX
5	Yes	PIN5_GPIO	PIN5_SDA	PIN5_CANRX	PIN5_UART2_RX
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_SYNCA	PIN32_UART2_RX
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 descriptions in each of the fields above. They represent the actual definitions 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\pinconstants.h.

9.1 Which Pins Can Be Used as GPIO?

The signal pin description chart in the previous section specifies which pins can be used as GPIO. In addition to the pins specified as "No" in the GPIO column, you should not use pin 2 or pin 3, since they are the UART 0 receive and transmit signals. You will need these for the Mod5213 monitor interface.

9.2 What is the NetBurner Pin Class?

As you can see from the pin function table, each pin can be used for a number of purposes. To configure a pin for a specific purpose, an application must know what registers need to be programmed in the 5213 processor. Rather than make everyone read the extensive ColdFire 5213 manual, we have created the Pin Class to make configuration and operation much easier. Although the Pin Class is written in C++, you do not need to know any 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, you are free to do so. If you do not include any Pin Class function calls, 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 so simple to use, you may want to give it a try for a quick benchmark before writing your own configuration code.

9.3 Pin Class API Summary

The following sections will describe the Pin Class API functions.

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 (tristate)
     Pins[x].drive();   // turn on GPIO output

     int n = Pins[x];   // read GPIO input and return integer
     BOOL b = Pins[x];  // read GPIO input and return boolean

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

The value of "function" in the function( ) call above is the definition in the previous Primary and Alternate function table. For example, to configure pins 4 and 5 as the CAN interface:

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

9.4 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 output pin 25 is used since it is connected to a LED on the carrier board. This way we can see the state of the pin as the LED turns on and off. For an input we use pin 4. We cannot automatically change the state of the input pin, so it will always read 0. The program runs in a loop that toggles the output state each ½ second, and reads the input once per second.

```
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // initialize UART 0
   EnableSmartTraps();           // enable smart trap utility
   OSChangePrio( MAIN_PRIO );    // set standard UserMain task priority
   EnableSerialUpdate();         // enable serial updates
   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 );

      // set pin 25 to a 1, LED should light
      Pins[25] = 1;
      OSTimeDly( TICKS_PER_SECOND / 2 );

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

9.5 A Simple Pin Class Special Function Example

The following code illustrates how to configure a pin for a special function, I2C. This handles the pin configuration. The application would still need to implement the I2C driver.

```
void UserMain( void *pd )
{
   SimpleUart( 0, SystemBaud );  // initialize UART 0
   EnableSmartTraps();           // enable smart trap utility
   OSChangePrio( MAIN_PRIO );    // set standard UserMain task priority
   EnableSerialUpdate();         // enable serial updates
   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

   }
}
```

10 Analog to Digital Functions

10.1 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 4 multiplexed analog inputs, providing 8 channels of analog input.

The ColdFire 5213 processor on-board A/D features include:

- 12-bit resolution
- Maximum ADC clock frequency of 5.33MHz, 187.5ns period
- Sampling rate up to 1.78 million samples per second (see footnote 1).
- Single conversion time of 8.5 ADC clock cycles (8.5 × 187.5ns = 1.595us)
- Additional conversion time of 6 ADC clock cycles (6 × 187.5ns = 1.126us)
- Eight conversions in 26.5 ADC clocks (26.5×187.5ns = 4.972us) 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 savings 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 pre-programmed 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 1.126 µs, providing an overall sample rate of 1,776,667 samples per second.

10.2 The NetBurner Mod5213 A/D API

The Mod5213 API supports automatic continuous sampling of all 8 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, you will need to create an A/D driver specific to your application. The Freescale 5213 Users Manual is a good reference on how to configure the A/D system to meet your application requirements.

10.3 Default Sample Rate

The A/D sample rate is 33,177,600 / (clock_div * 6). If all 8 A/D's are running, then the sample rate for any one channel must be divided by the number of active channels - in this case 8. For the default value of clock_div = 7, the sample rate is: 33,177,600 / (7*6*8) = 98,742 Samples per second.

10.4 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, then call ReadA2DResult( ) whenever you wish to get the last sample. The functions are shown below:

```
void EnableAD( BYTE clock_div = 7 ); // default divider value is 7

WORD ReadA2DResult( int ch );        // ch = channel number 0 - 7
```

Note that the 5213 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.

10.5 Mod5213 Hardware Configuration

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

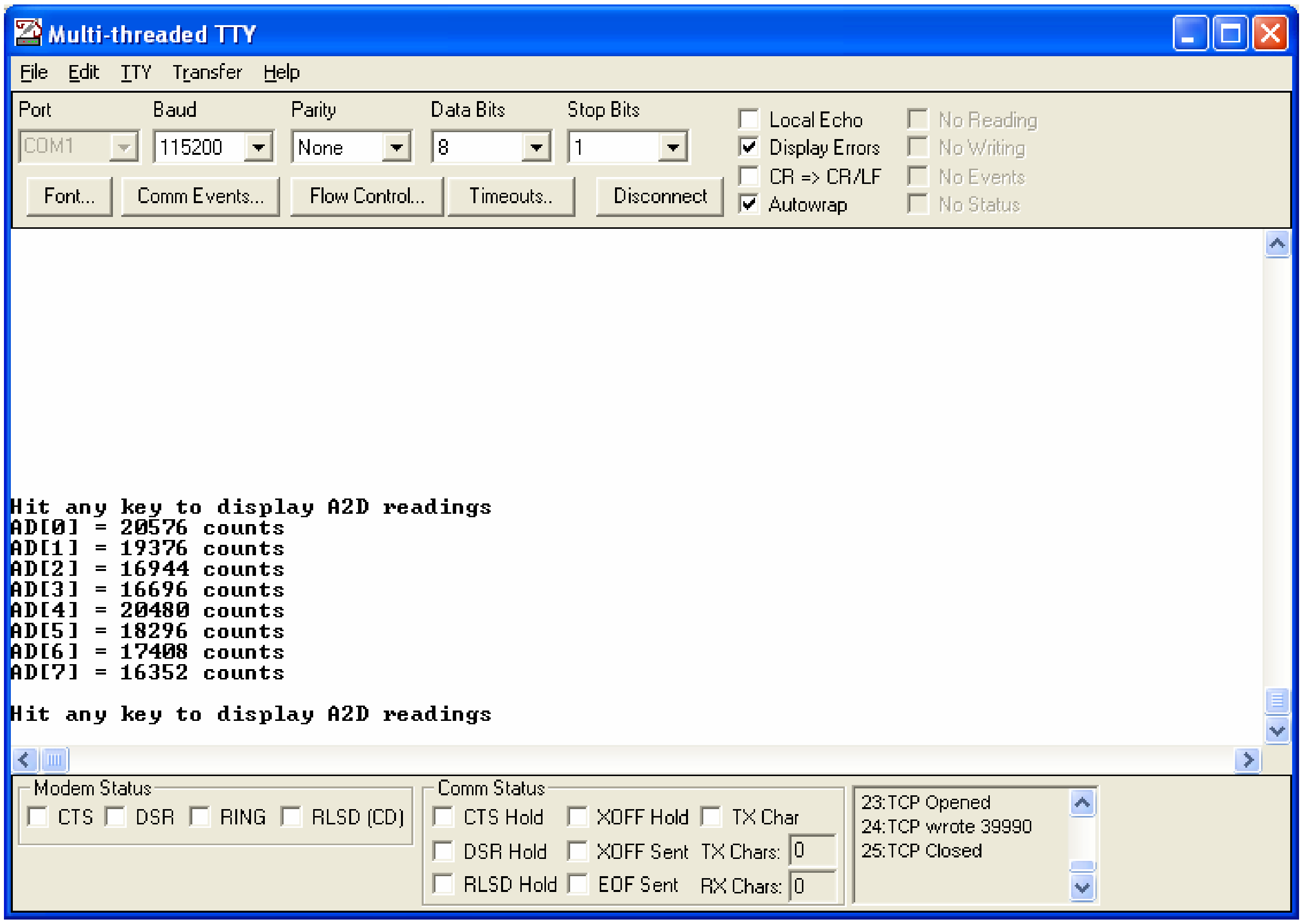
VDDA	A/D voltage input
VRH	A/D voltage reference high
VRL/VSSA	A/D voltage reference low and voltage power ground.

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 3.3VDC
JP4	Connect VRH to 3.3VDC

10.6 A/D Example

The Mod5213 A/D example is located in \nburn\examples\Mod5213\a2d. When you run the example you can view the A/D readings via the serial port and MTTTY. A MTTTY screen shot is shown below:



Port: COM1
Baud: 115200
Parity: None
Data Bits: 8
Stop Bits: 1
Local Echo [ ]
Display Errors [✓]
CR => CR/LF [ ]
Autowrap [✓]
TX Chars: 0
RX Chars: 0

```
Hit any key to display A2D readings
AD[0] = 20576 counts
AD[1] = 19376 counts
AD[2] = 16944 counts
AD[3] = 16696 counts
AD[4] = 20480 counts
AD[5] = 18296 counts
AD[6] = 17408 counts
AD[7] = 16352 counts

Hit any key to display A2D readings
```

```
23:TCP Opened
24:TCP wrote 39990
25:TCP Closed
```

10.7 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 98Khz.  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 accommodate the various 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" );
   }
```

11 NetBurner uC/OS RTOS

The uC/OS is a very stable, fast and reliable operating system. The NetBurner implementation takes up very 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 factory demo program. Even if you only need a single UserMain( ) task, some of the RTOS functions such as OSTimeDly( ) and interrupt driven serial I/O can come in handy. Just code the application like UserMain( ) is the equivalent of a standard C type main( ) function.

If that is all your applications require, you may not even need to dig any deeper. But if you do need advanced features, uC/OS has plenty to offer. For more in-depth information on advanced features, please refer to the NetBurner RTOS programming guide.

11.1 RTOS System Resource Usage

Use of the RTOS will have the following impact on system resources. The default values for these resources are located in \nburn\include_nn\ucos.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 ucos.h. Whenever you make a change to a system file in \nburn\include_nn or \nburn\system_nn, be sure to "Rebuild All" so the system library is rebuilt and the changes take effect.

11.1.1 Task Stack Size

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

```
   OSSimpleTaskCreate( TaskName, priority );
```

the default stack size will be used. The UserMain( ) task and system idle task are created at boot and will use 512 bytes and 256 bytes respectively. There values are defined as:

```
#define IDLE_STK_SIZE (256)
#define USER_TASK_STK_SIZE (512)
```

11.1.2 System Clock Tick

The RTOS system clock uses the 5213 Periodic Interrupt Timer (PIT) 0, which uses interrupt request 1 at priority level 3. The default number of ticks per second is defined as:

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

If you want to change the ticks per second you can; the recommended values are between 20 and 200. For high resolution timing, such as microseconds, the best method is to use another hardware timer in the 5213.

11.1.3 Maximum Number of Tasks

The maximum number of tasks is set at 20 by default. Allowing for reserved tasks, this number can be increased to 56.

```
#define OS_MAX_TASKS    20 /* Max number of system tasks */
```

11.1.4 Task Priorities

The convention in uC/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 nop instructions.

The recommended priority for UserMain( ) is 50. This is just a number selected so applications can easily pick priorities higher and lower for other tasks. This number originated on NetBurner network enabled platforms, which occupy some numbers between 20 and 30.

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

11.1.5 Interrupt Driven Serial Ports

The interrupt driven serial port drivers use the RTOS. Interrupts on the Mod5213 are beyond the scope of this document. Basically you can have interrupt requests numbered from 0 – 7, and interrupt priority levels from 0 – 7 for each interrupt request. The serial port interrupt driver uses interrupt request level 3 for all three UARTs, with priorities set to 1 for UART 0 , 2 for UART 1 and 3 for UART 2. This configuration is located in \nburn\Mod5213\system\ irq_serial_init.cpp.

The interrupt driver also uses buffers to store serial data. The default sizes are defined below:

```
#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)
```

11.1.6 Building Applications Without the RTOS

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

```
#include "predef.h"
#include <basictypes.h>         // Include for variable types
#include <serialpoll.h>         // Use serial polled driver
#include <SerialUpdate.h>       // Update flash via serial port

/*
   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);
}
```