Future Technology Devices International Ltd.

Application Note

AN_142

AN_142 Vinculum-II Tool Chain Getting Started Guide

Document Reference No.: FT_000269

Version 1.2.0

Issue Date: 2010-08-18

Future Technology Devices International Limited (FTDI)

Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777   Fax: + 44 (0) 141 429 2758
E-Mail (Support): support1@ftdichip.com Web: http://www.ftdichip.com

Table of Contents

1 Introduction

The scope of this document is to provide an introduction to using the VNC2 toolchain. This document is intended for people who have successfully installed the VNC2 toolchain and is provided as a getting started guide for first time users.

FTDI provide a number of sample applications with the toolchain installation, these samples are designed to familiarise users with the supplied FTDI drivers, libraries and development environment. It is recommend that to follow this tutorial more easily all files installed during the installation are kept in their default location as per the installation wizard.

All code samples in this document are provided for illustration purposes only. They are not guaranteed or supported by FTDI.

2 Overview

The intention of this document is to give novice users of the Vinculum-II software development toolchain the knowledge to build and run their first sample application and to then use this knowledge to go onto write and build a first application from scratch. It does this using a short tutorial.
The tutorial firstly focuses on the Kitt sample, provided along with the Vinculum-II toolchain, to demonstrate: the opening of projects; building firmware for the VNC2; loading this firmware onto the device; running the firmware on the VNC2 and finally using the debugger to step through code. Secondly, it introduces writing an application from scratch based on the Hello World sample. This demonstrates how to use FTDI supplied device drivers and outlines the general structure applications may take.

3 Building Your First Application

Installing the VNC2 toolchain (using the default settings) results in the toolchain being installed within the `Program Files/FTDI/Vinculum II Toolchain` directory on the PC's local hard disk; the installer creates a start menu shortcut, again under the `FTDI/Vinculum II Toolchain` folder heading. The VNC2 IDE is located within either of these two folders; to launch the application double click on Vinculum II IDE icon.

Opening the Sample Project

This is an overview of the IDE GUI, the layout may not match exactly Figure 1, however, this can be easily modified using the built-in docking manager.

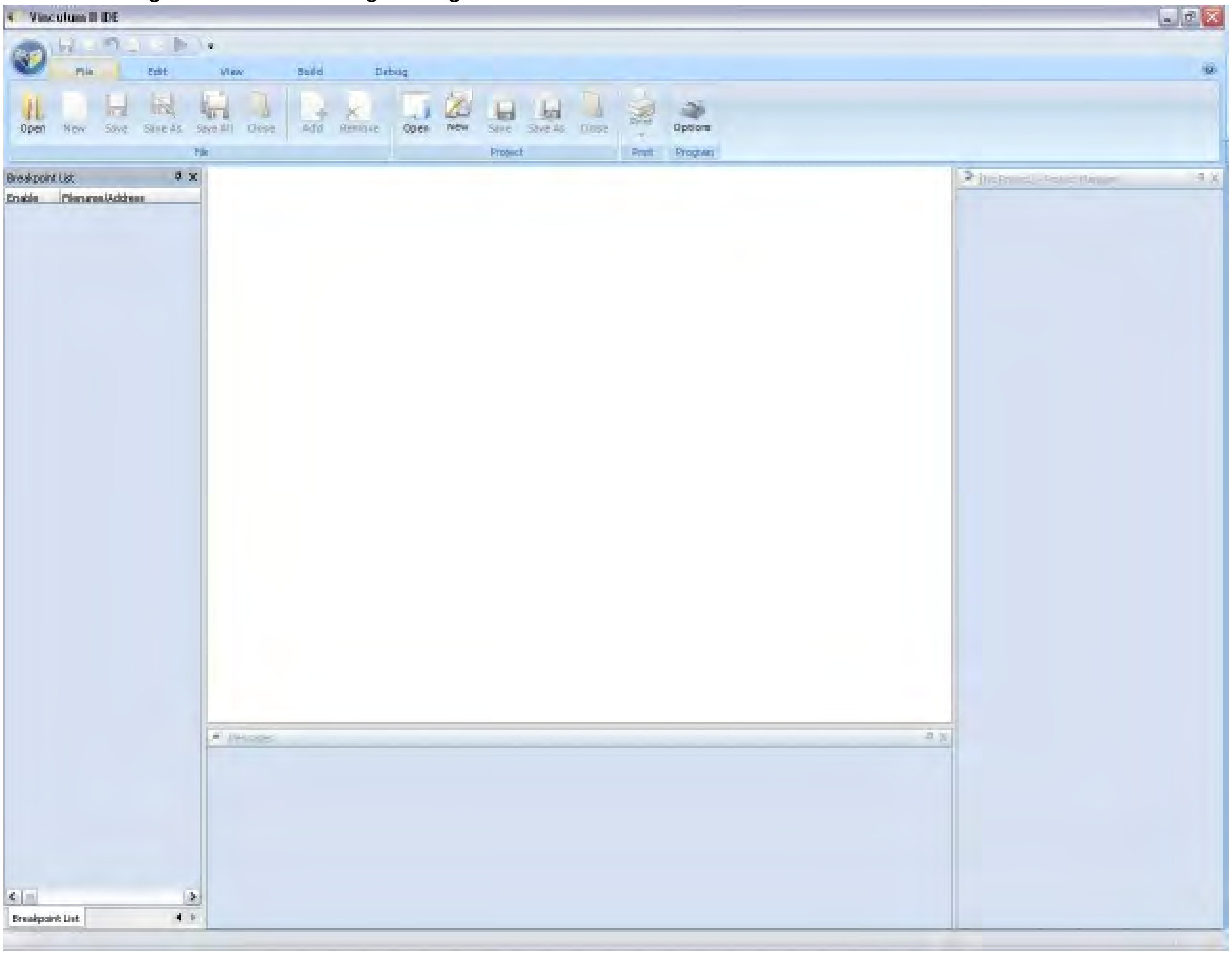


Figure 1

The tabbed tool strip running along the top of the screen gives access to the menu and sub menu items within the IDE.

Within the File tab, as above, notice the Project subcategory, click on the Open button. By following the default settings within the installation wizard the FTDI provided samples are saved within the `My Documents` folder. Using the project dialog box, browse to `My Documents` and find the folder `FTDI/Firmware/Samples/`*ReleaseVersion*`/General`. Within this is a folder called `Kitt` containing a file `Kitt.vproj` (vproj is the file extension used by all VNC2 project files) double click this file to launch it within the IDE.

Building the Application

Notice that when an application is opened within the IDE the Project Manager window now contains all the files relevant to that project. If the Project Manager window is not visible go back to the tabbed tool strip, along to view and make sure that the Project Manager box is checked (Figure 2). IDE panels can be dragged and docked anywhere on the screen using the built-in docking manager, simply click and hold the title bar of a panel to free it and then drag it to the desired area of the screen.

Files within this project can now be opened within the editor window by double clicking them; the editor window allows multiple files to be open concurrently. The archive files under the Libraries folder contain FTDI supplied drivers and VOS Kernel Services, these files cannot be opened or edited.

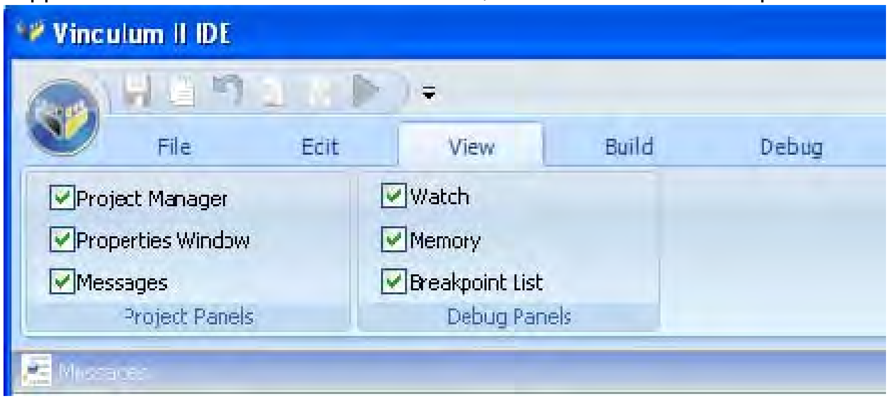
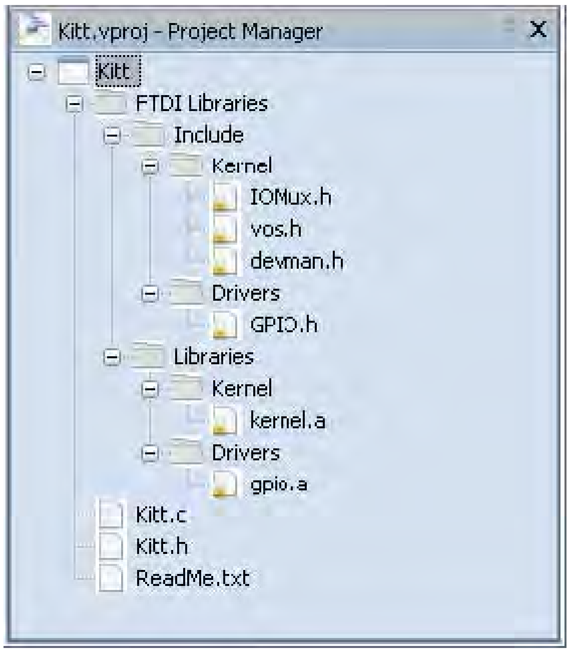


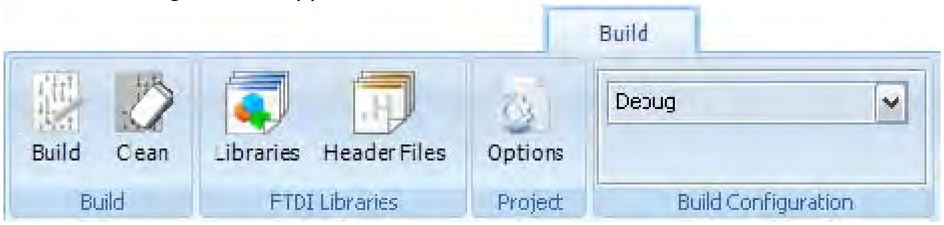
Figure 2

To build the sample Kitt application: go to the tabbed tool strip and along to the Build tab. In the left hand side of the panel is a button called Build, this will generate the ROM file (firmware) that can be programmed into the VNC2 IC. Note that under the Build Configuration sub-category the project is set in Debug mode; this is important at this stage as it will allow debugging of source code after the ROM image has been loaded into the VNC2 device.



```
Kitt.vproj - Project Manager
Kitt
  FTDI Libraries
    Include
      Kernel
        IOMux.h
        vos.h
        devman.h
      Drivers
        GPIO.h
    Libraries
      Kernel
        kernel.a
      Drivers
        gpio.a
  Kitt.c
  Kitt.h
  ReadMe.txt
```

After clicking Build the IDE will attempt to compile, assemble and link the source code into a format that can be loaded into the VNC2. If the source code within kitt.c hasn't been altered there should be no compilation errors, meaning the Kitt application should build first time.



The outcome from a build attempt is displayed within the Messages Window at the bottom of the screen. The last line within the Messages Window indicating that there have been 0 errors from the build shows that the IDE has successfully created the ROM file.

```
[VinL.exe] : 0 errors, 0 warnings and 0 informational messages
```

Flashing VNC2

The next step is to program the ROM file from the above build process into the VNC2 flash memory, but first connect the VNC2 to the host PC (Figure 3). This demonstration uses the V2EVAL with the 64 pin QFN daughter card installed. The debugger port is connected to the host PC via the blue USB cable (shown at the top of figure 3). The power switch for the device is located just below the black power supply socket in the top left corner of the screen; in this mode, this device is self powered and therefore does not require an external power supply to operate. When the V2EVAL board is connected, the host PC may attempt to install FTDI drivers for the FT4232H connected to the debugger port.

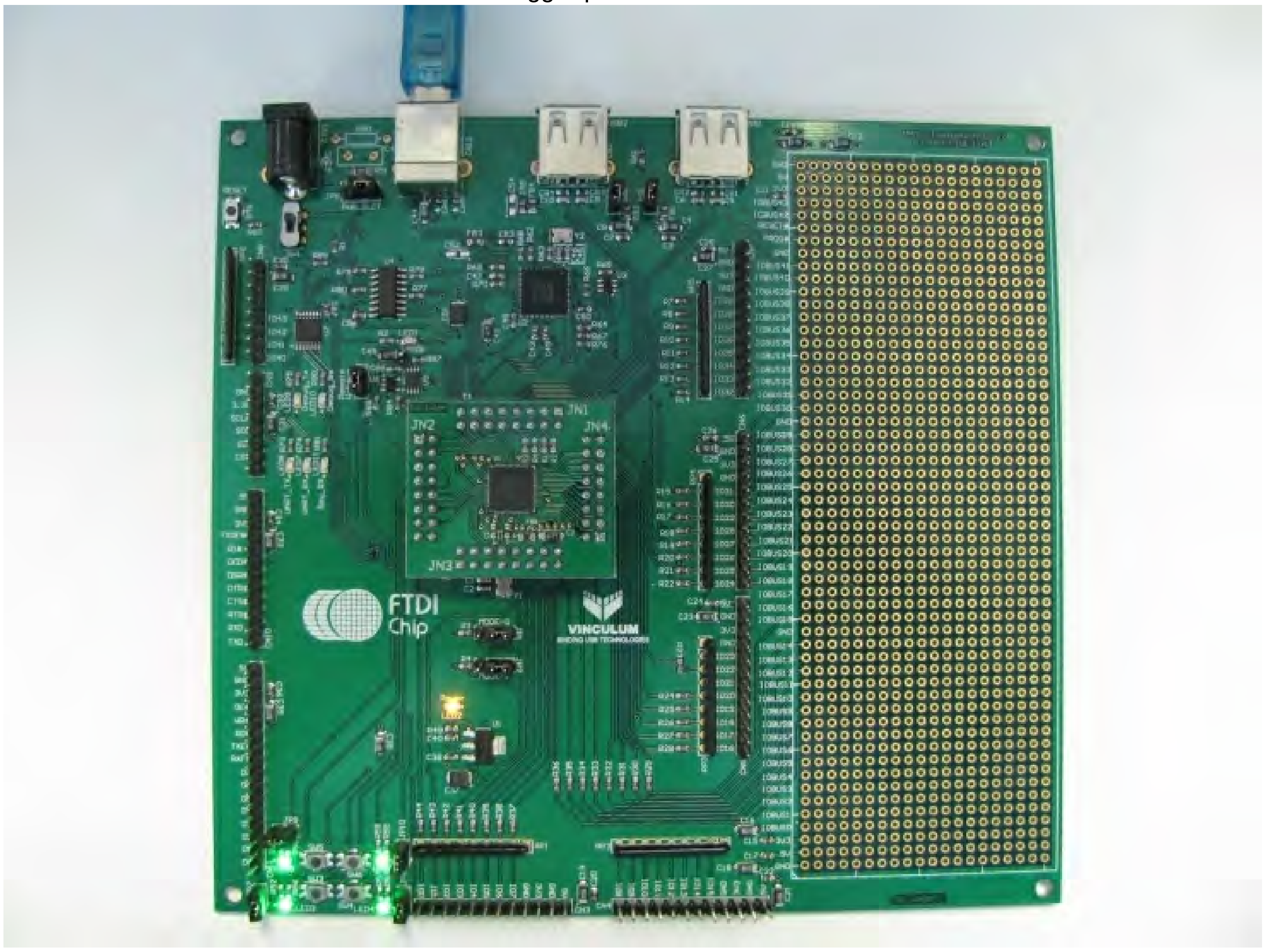
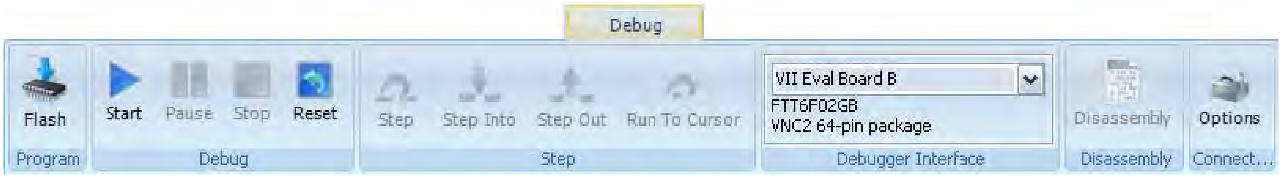


Figure 3

After connecting the V2EVAL board as shown in Figure 3 ,open up the IDE and select the Debug tab within the tool strip Selecting the drop-down menu within the Debugger Interface subcategory initiates the IDE to search for connected devices and, as can be seen below, automatically selects the debugger interface of the V2EVAL board. When a debugger interface is selected the Flash, Start and Reset buttons also become active. To program the flash memory of the VNC2 select Flash from the Debug tab, a dialog box appears showing the Kitt.rom image file that was built earlier, select this file and press open.



The IDE attempts to program the VNC2 flash, all relevant information will be shown in the Message Window at the bottom of the screen.

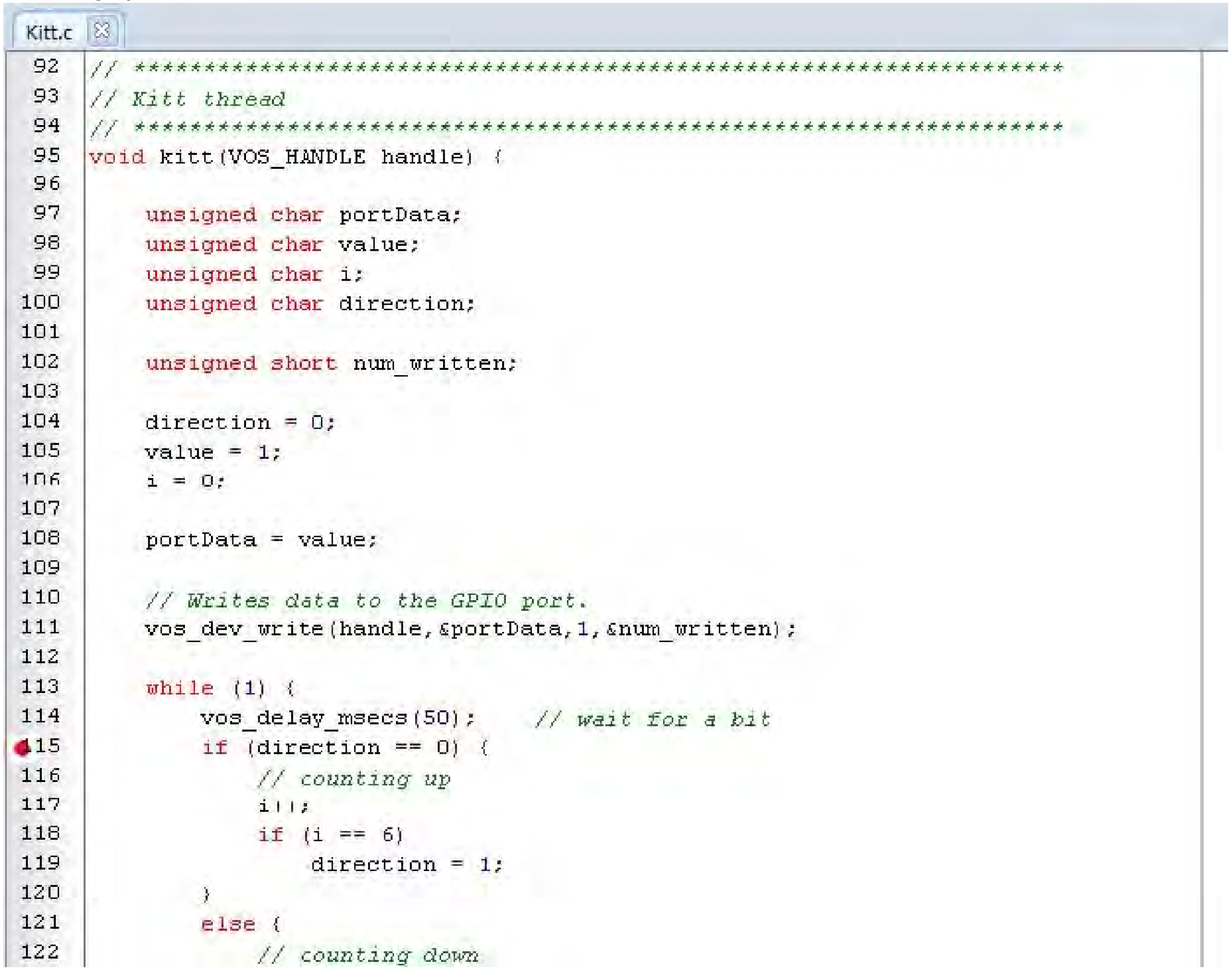
Running the Application

To run the Kitt sample, press the start button in the Debug tab. The four LEDs located in the bottom left hand corner of the V2EVAL board should now be sequentially flashing signifying that the device is running correctly. The Pause and Stop buttons become active only if the ROM file loaded into the VNC2 has been built in debug mode. Pause can be used to suspend execution at the current line within the disassembly file. Stop can be used to halt executing and stop the firmware on the VNC2 executing.

Debugging the Application

The debugger interface supplied by FTDI allows for source code debugging at C and assembly level; this tutorial illustrates using the C level debugging.

The IDE allows breakpoints to be added to the C source, in the below diagram (Figure 4) a breakpoint has been added to the source code at line 115 A breakpoint is added by clicking the desired line number in the left hand side gutter of the screen. Breakpoints can be placed on lines with no code, for example lines with comments, but these are grayed out when the debugger starts and will not be hit. VNC2 supports 3 breakpoints being set concurrently; any extra breakpoints are deselected within the Breakpoint List window and are grayed out within the editor.



Kitt.c

```
 92  // *******************************************************************
 93  // Kitt thread
 94  // *******************************************************************
 95  void kitt(VOS_HANDLE handle) {
 96
 97      unsigned char portData;
 98      unsigned char value;
 99      unsigned char i;
100      unsigned char direction;
101
102      unsigned short num_written;
103
104      direction = 0;
105      value = 1;
106      i = 0;
107
108      portData = value;
109
110      // Writes data to the GPIO port.
111      vos_dev_write(handle,&portData,1,&num_written);
112
113      while (1) {
114          vos_delay_msecs(50);    // wait for a bit
115          if (direction == 0) {
116              // counting up
117              i++;
118              if (i == 6)
119                  direction = 1;
120          }
121          else {
122              // counting down
```

Figure 4

To hit a breakpoint press Start within the debug menu. The application runs to this breakpoint and execution from the VNC2 stops; this allows for lines of code to be single stepped using the Step control panel within the Debug menu.

Individual variables within the source code can also be added to a watch list; this allows for the value of certain variables to be monitored during execution of the source code. To add a watch bring up the Watch window from the View tab within the tool strip. Right click within the Watch List and select Add Watch; enter the name of the variable to be monitored and press Add Watch. Figure 5 illustrates the variable value added to a watch list. All watches that have been added are displayed within the Watch List; a watch that has a value of undefined is either outside the current scope of execution or is not defined within the current application. During single stepping of code it is now possible to monitor changes within the value field of each variable to aid with debugging.

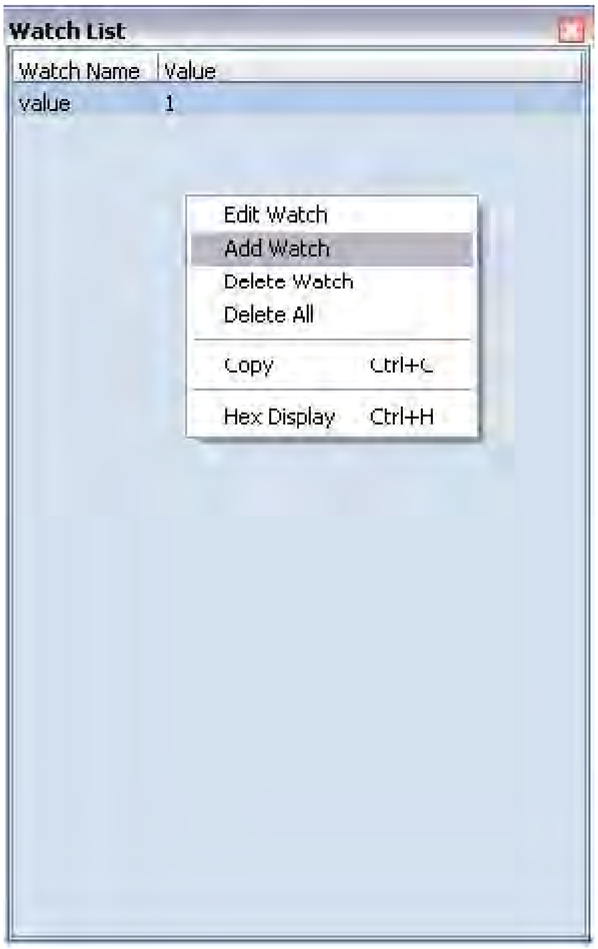

Watch List
Watch Name
Value
value
1
Edit Watch
Add Watch
Delete Watch
Delete All
Copy Ctrl+C
Hex Display Ctrl+H

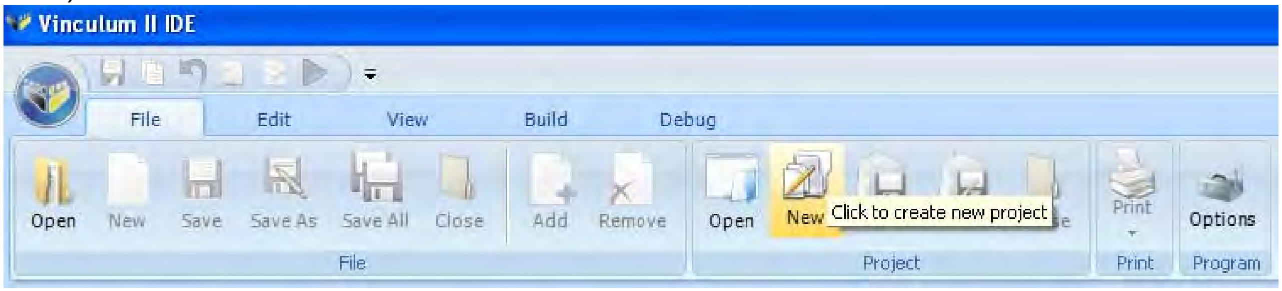
Figure 5

4 Writing 'Hello World' Application

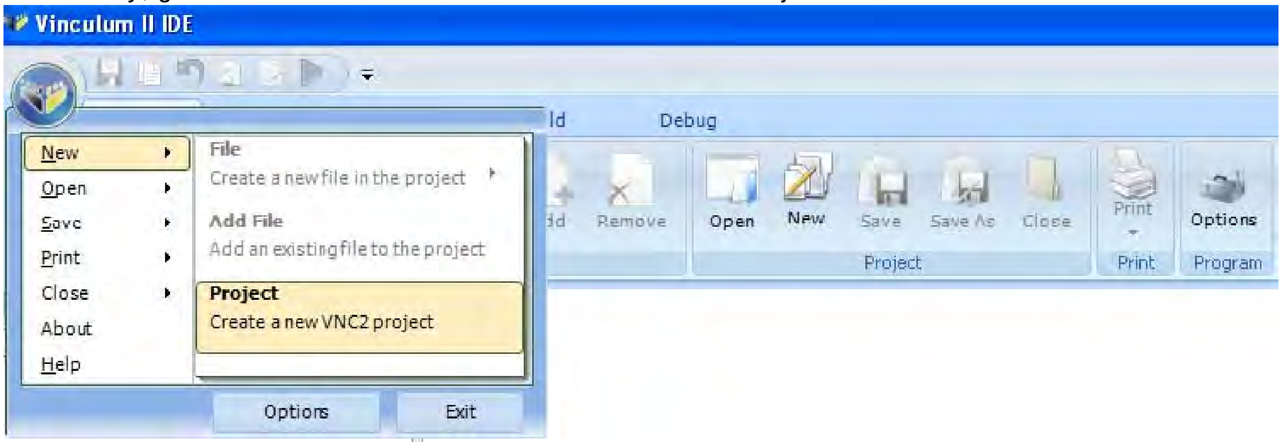
This section illustrates the creation of an application from scratch based upon the Hello World sample provided by FTDI. Hello World is a simple application that connects to a USB flash drive, creates a new text file on the drive and writes the string “Hello World” to this file. This project demonstrates the main components of writing an application and how to use a selection of the provided drivers, Kernel services and libraries.

Creating a new Project

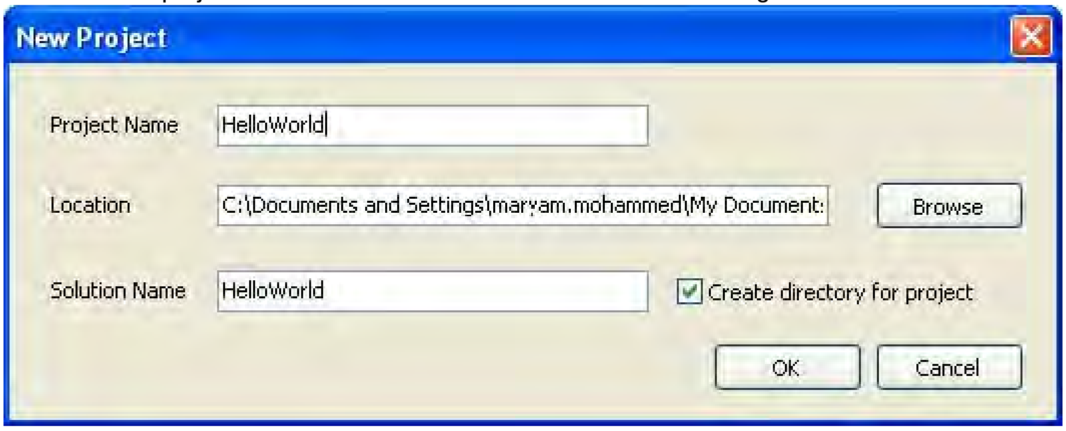
To create a new project, go to the File tab within the toolbar and click New under the Project tab (see Figure below).



Alternatively, go to the circular Vinculum button and click New->Project



This pops up a New Project dialog box which allows for browsing to the project location and renaming of the project and solution. It is necessary to complete the text boxes before clicking OK.
Create a new project called HelloWorld as demonstrated within Figure 6.



Project Name: HelloWorld
Location: C:\Documents and Settings\maryam.mohammed\My Document:
Solution Name: HelloWorld
[✓] Create directory for project

Figure 6

Note: The project and solution names do not need to match. It is recommended that the Create directory for project check box is selected; this creates a project directory containing the project and any subsequent files.
The result of creating a new project in the Project Manager panel is illustrated in Figure 7 which shows a new project called HelloWorld which has been created.

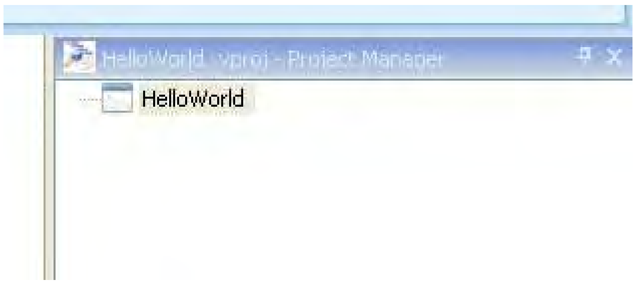
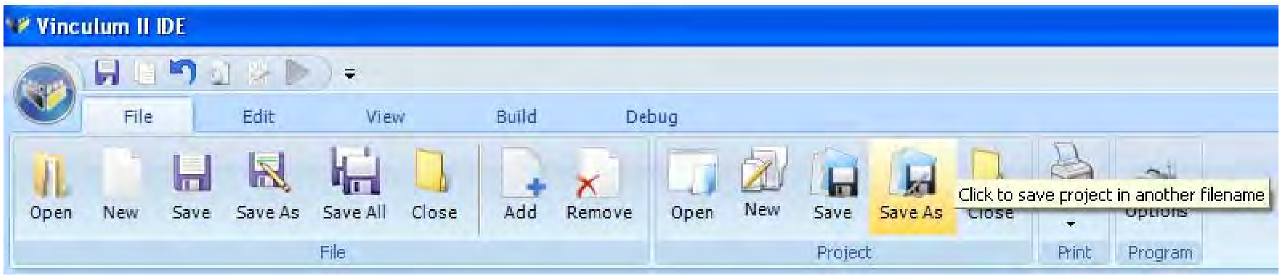


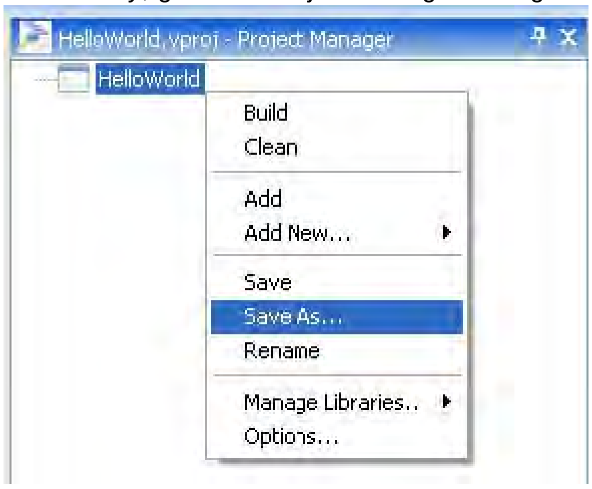
Figure 7

Saving a Project

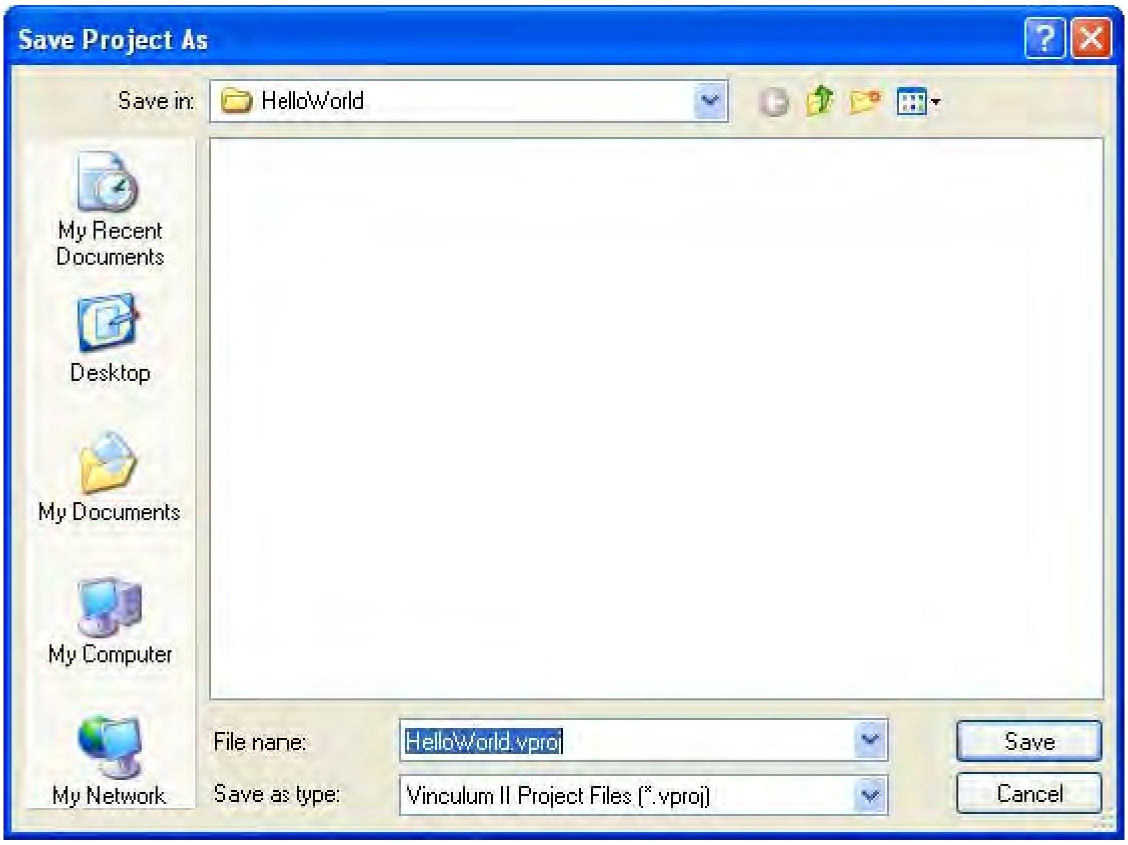
To save a project, use the Save As button within the File tab.



Alternatively, go to the Project Manager and right click "HelloWorld" ->Save As.



This will result in the Save Project As dialog box (Figure 8) appearing.



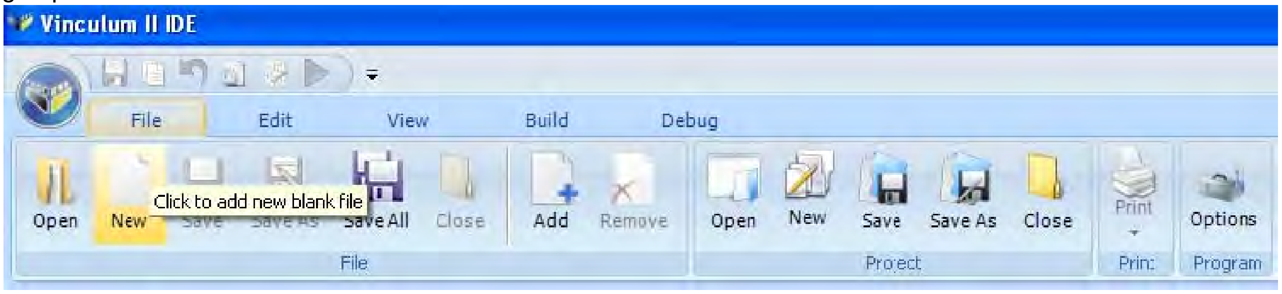
File name: HelloWorld.vproj

Save as type: Vinculum II Project Files (*.vproj)

Figure 8
Select a location and filename for saving the project.

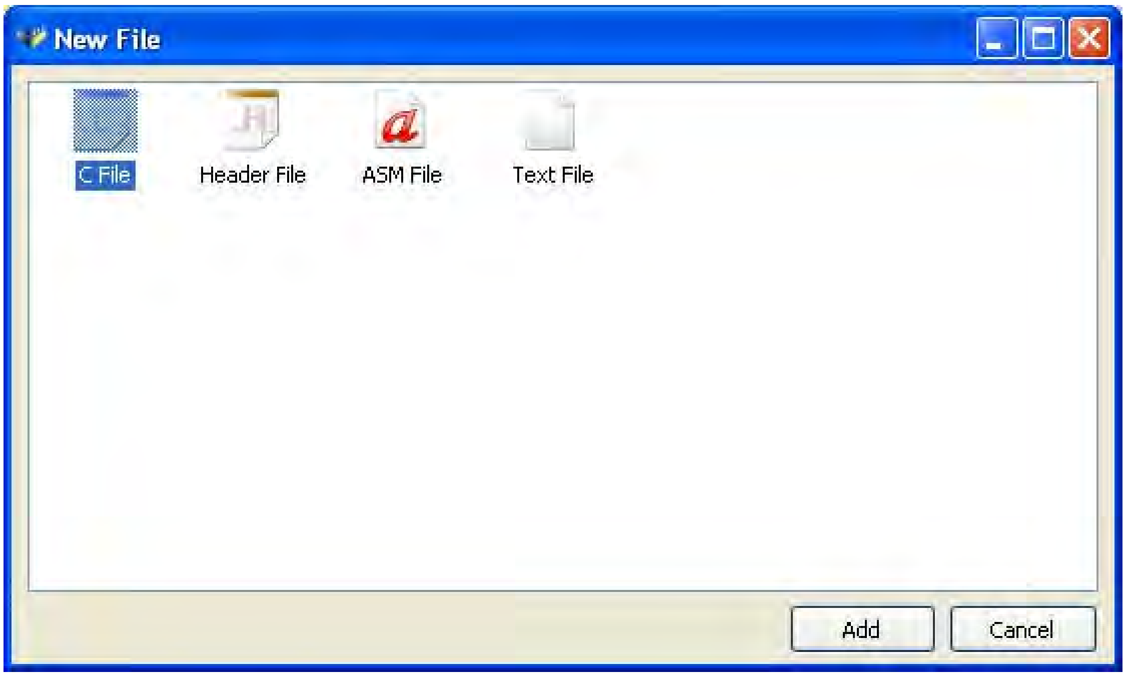
Adding New Files to a Project

To add a new file to the HelloWorld project, go to the File tab within the toolbar and click New in the File group.



Click to add new blank file

The New File pop-up allows different types of file to be added to a project. Firstly add a new C File which will contain the main body of code for the application. Select C File in the New File pop-up and press Add. Repeat this to add another new file to the project, this time a Header File.



Notice that within the Project Manager window, as shown in Figure 9, there are now two new files under the project heading. To rename both files: right click on File.c within the Project Manager and select Rename. The IDE prompts to save this file first before renaming it, click Yes to confirm this. Within the save dialog box rename this file as HelloWorld.c and click OK. Repeat these steps for the header file in the project, calling it HelloWorld.h.

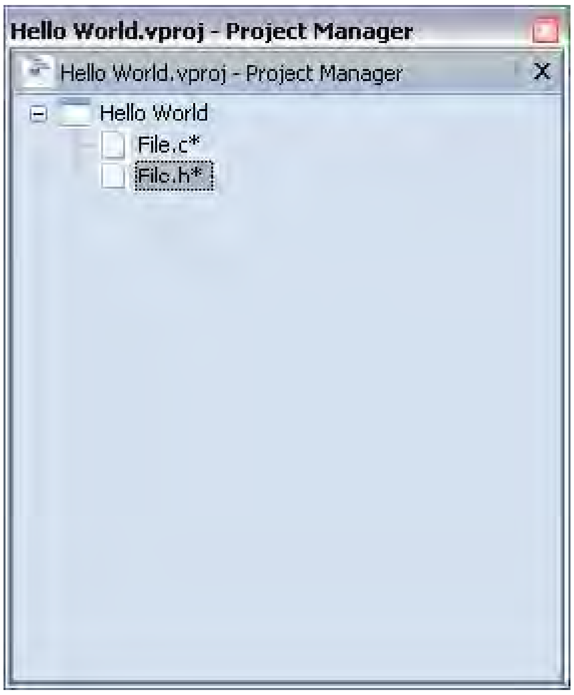
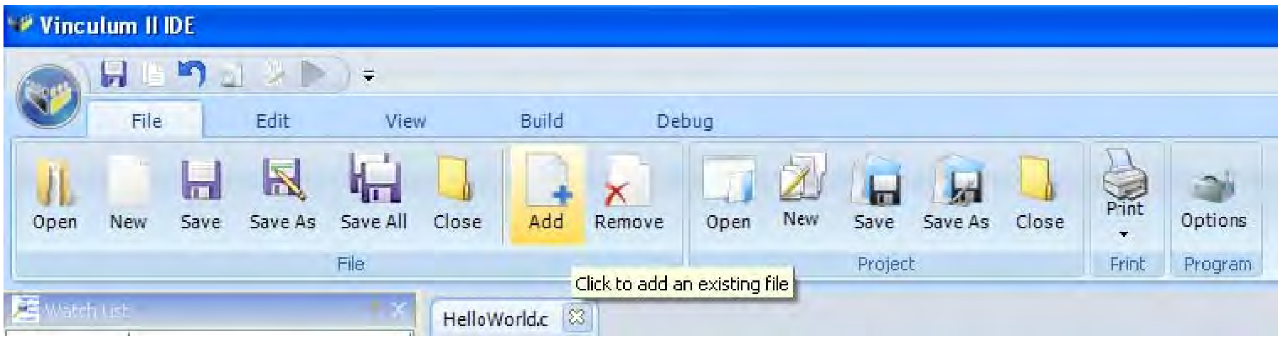

Hello World.vproj - Project Manager
Hello World.vproj - Project Manager
Hello World
File.c*
File.h*

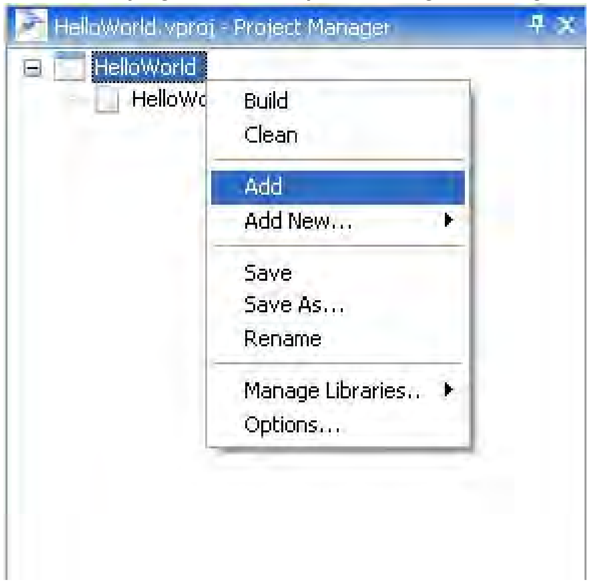
Figure 9

Adding Existing Files to a Project

To add an existing file to the “HelloWorld” project, go to the File tab of the toolbar and click Add in File tab group. The open dialog (Figure 10) allows for different file types(C,ASM, Header or Text) to be added from this project or any other project. It is also possible to add multiple files by holding the CTRL key while clicking on each of the files required to be added.



Alternatively, go to the Project Manager and right click "HelloWorld" project ->Add



To add a file click the Open button. All added files are added to the Project Manager window as illustrated in Figure 11.

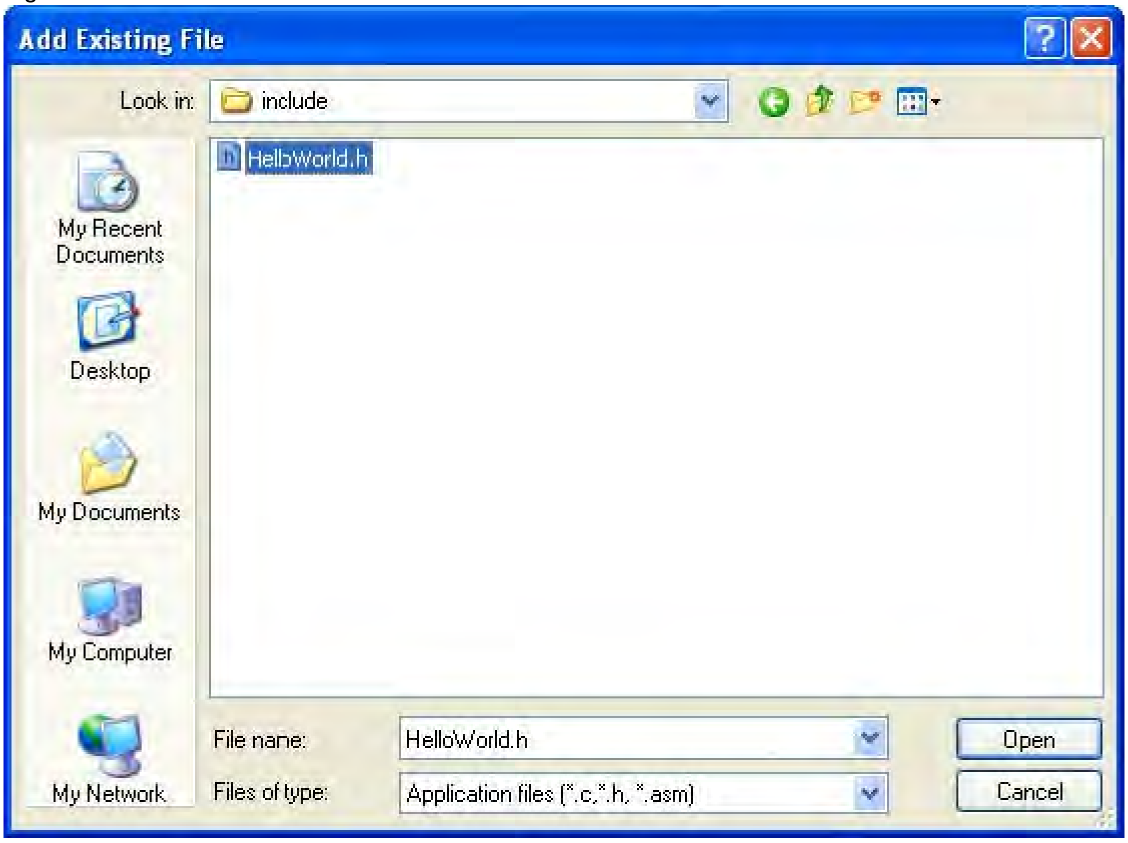
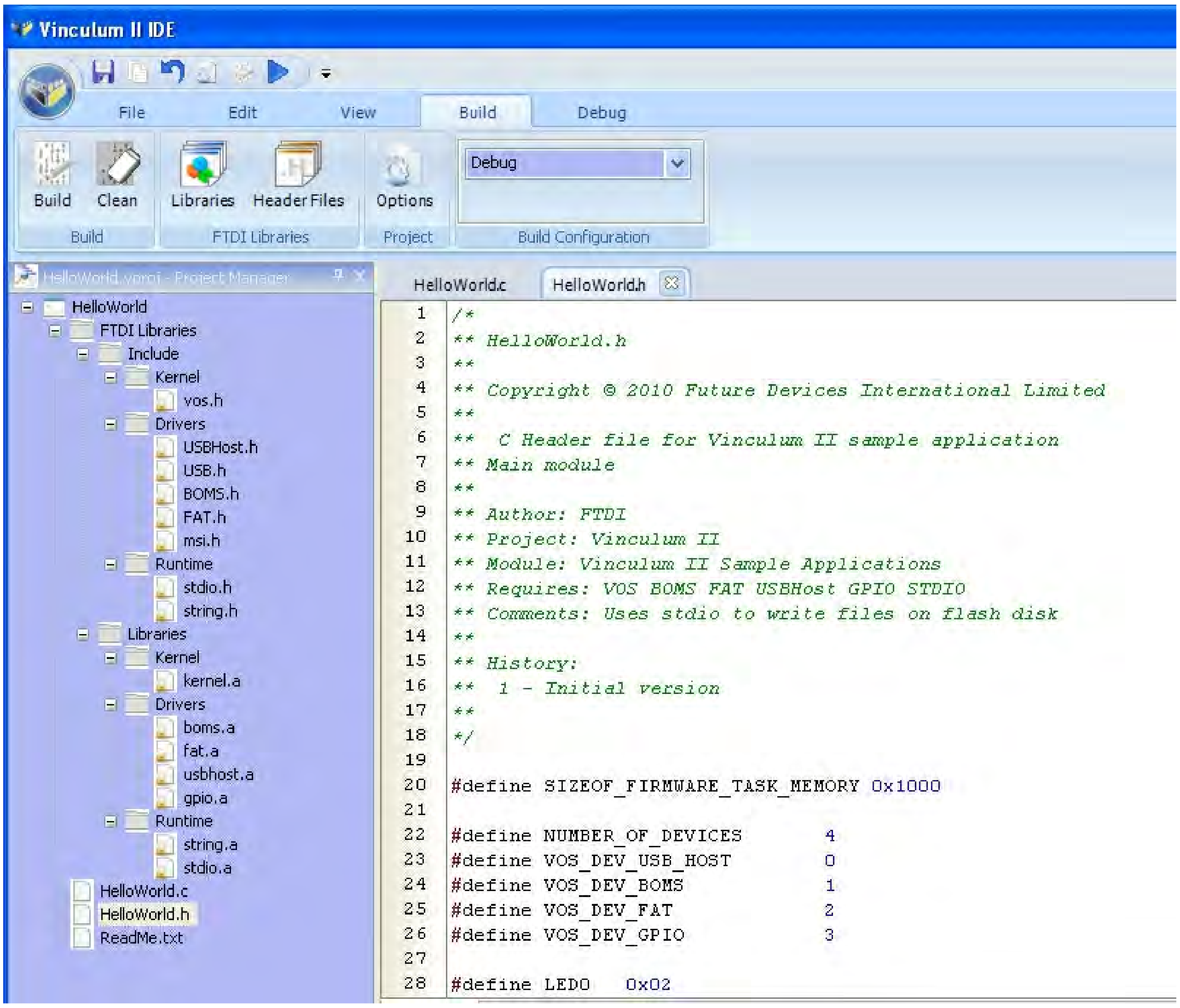


Figure 10



```
/*
** HelloWorld.h
**
** Copyright © 2010 Future Devices International Limited
**
**  C Header file for Vinculum II sample application
** Main module
**
** Author: FTDI
** Project: Vinculum II
** Module: Vinculum II Sample Applications
** Requires: VOS BOMS FAT USBHost GPIO STDIO
** Comments: Uses stdio to write files on flash disk
**
** History:
**  1 - Initial version
**
*/

#define SIZEOF_FIRMWARE_TASK_MEMORY 0x1000

#define NUMBER_OF_DEVICES       4
#define VOS_DEV_USB_HOST        0
#define VOS_DEV_BOMS            1
#define VOS_DEV_FAT             2
#define VOS_DEV_GPIO            3

#define LEDO    0x02
```

Figure 11

5 Writing an Application

In this section, an application is written which writes a line of text to a file on a disk. The listing is available in the Getting Started Code Listing topic and it is in the samples directory under `General/HelloWorld`.

Header File

To write some example code for this application starting with the header file: double click HelloWorld.h within the Project Manager to open this file in the IDE editor. Header files contain forward declarations of functions, constant values and any other global variable declarations that are shared throughout the application. Although it is not strictly necessary to use a header file within this project it is good programming practice to get into the habit of using them, especially when dealing with more complicated projects than the HelloWorld application.
The first thing to define within the header is the size of stack memory that the application thread is going to need. Details of this will be explained further when it comes to creating a thread within the application.
Paste the following code fragment into the header file:

```
#define SIZEOF_FIRMWARE_TASK_MEMORY 0x1000
```

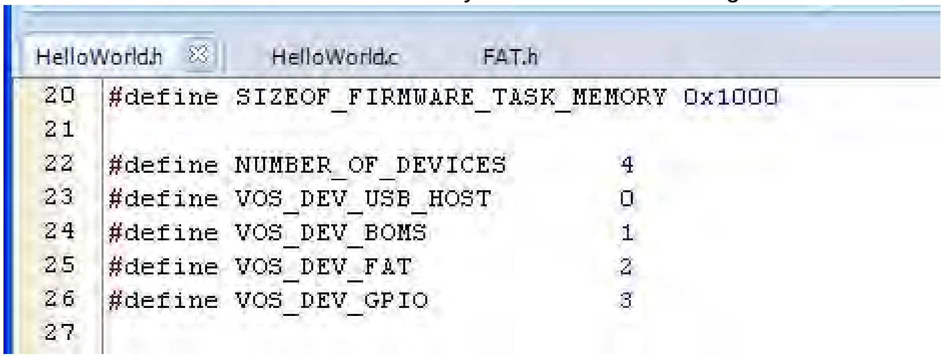
Next decide the number of device interfaces that are used within the application. The HelloWorld app requires: a USB Host driver to connect to the USB flash drive; a BOMS driver and FAT file system driver to allow communication to the flash disc and also a GPIO driver allowing for visual feedback to the user using the LEDs on the V2EVAL board. When initializing each device they must have a unique identifier that is used within the Kernel's device manager. As well as this the number of devices used within the application must be explicitly specified.
Again, copy and paste the following code fragment into the .h file.

```
#define NUMBER_OF_DEVICES 4
#define VOS_DEV_USB_HOST 0
#define VOS_DEV_BOMS 1
#define VOS_DEV_GPIO 2
#define VOS_DEV_FAT 3
```

Lastly, add a forward declaration for the application thread:

```
void firmware(void);
```

The header file should now match closely the code shown in Figure 12.



```
#define SIZEOF_FIRMWARE_TASK_MEMORY 0x1000

#define NUMBER_OF_DEVICES        4
#define VOS_DEV_USB_HOST         0
#define VOS_DEV_BOMS             1
#define VOS_DEV_FAT              2
#define VOS_DEV_GPIO             3
```

Figure 12

Other definitions of LED combinations are found in the listing of the header file in the Getting Started Code Listing topic.

FTDI Libraries

With VNC2, all applications integrate with FTDI provided libraries that contain VOS Kernel Services, device drivers and runtime libraries. Kernel Services provide all the data structures and primitives that an application uses, as well as providing control throughout the lifetime of the application.
The Device Manager defines a standard API for device drivers. All devices are accessed using this standard API to make application development easier. Device Manager is the interface between user applications and Kernel Services. Runtime libraries contain functions which are common to most C language implementations, for example string and standard IO.
The Hello World application requires a selection of Runtime Libraries, FTDI drivers as well as Kernel Services to run. These are provided in the form of archive files which come with the VNC2 toolchain installation. To utilize the provided libraries they must be included in the application. Each archive file has a corresponding

header file that defines its API, providing information on functions and data structures that are contained within the archive files.
The Hello World application requires the following device drivers: USBHost acting as an interface to the USB drive; the BOMS driver to communicate with a mass storage device; the FAT driver to communicate with the device file structure and the GPIO driver which allows for visual feedback using the LEDs. The string runtime library which contains string manipulation functions and stdio to provide file I/O functions also needs inclusion. Finally Kernel Services, which provide overall control of the device drivers, need to be added. As well as adding the archive files the corresponding header files require inclusion.

3.4.2.1 Adding Library Files

To add Library Archive Files to the HelloWorld project, go to the Build tab of the toolbar and click Libraries in the FTDI Libraries tab group.

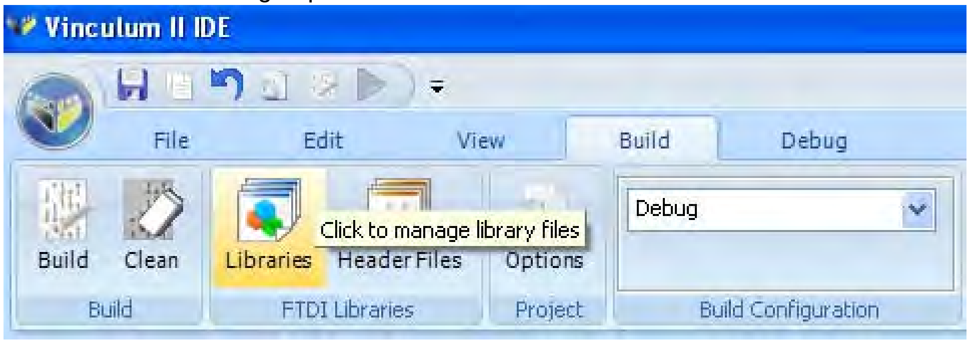
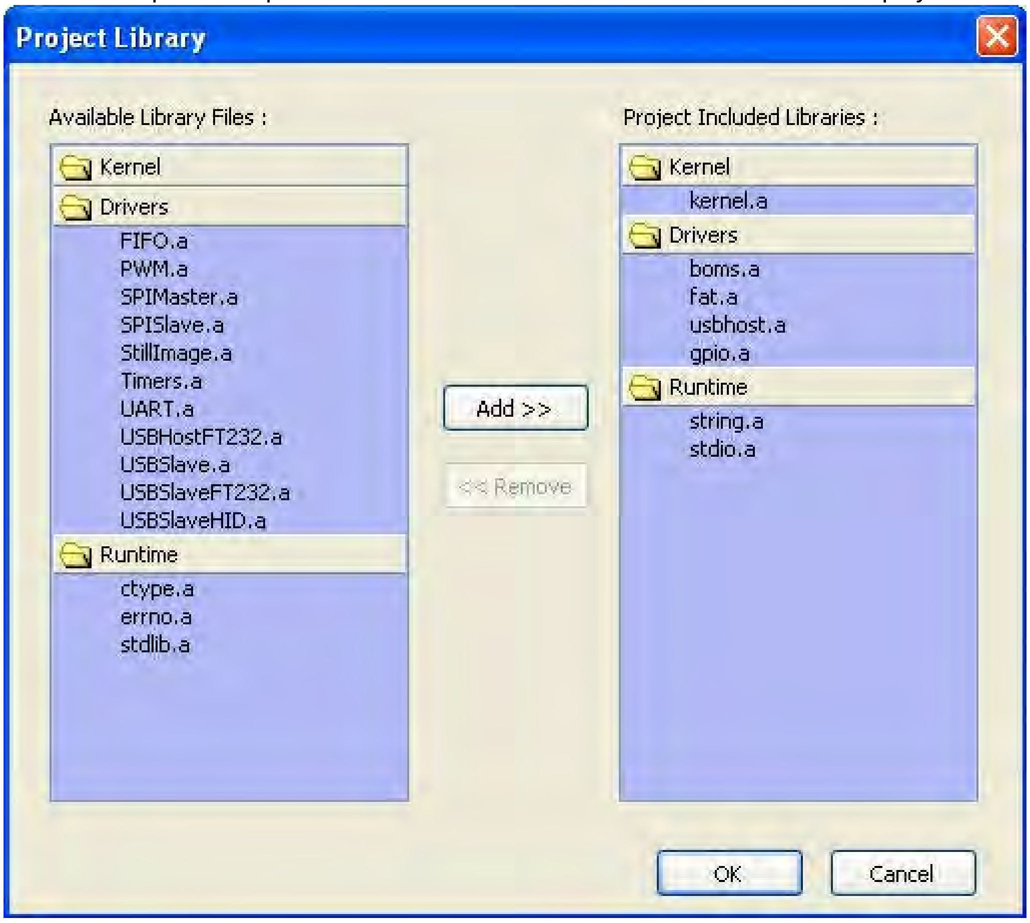


Figure 13 shows the Project Library dialog box which appears. To add a library, click the desired archive file in the left hand pane and press the Add button. The list of added libraries is displayed within the right hand pane.



Available Library Files :

- Kernel
- Drivers
  - FIFO.a
  - PWM.a
  - SPIMaster.a
  - SPISlave.a
  - StillImage.a
  - Timers.a
  - UART.a
  - USBHostFT232.a
  - USBSlave.a
  - USBSlaveFT232.a
  - USBSlaveHID.a
- Runtime
  - ctype.a
  - errno.a
  - stdlib.a

Project Included Libraries :

- Kernel
  - kernel.a
- Drivers
  - boms.a
  - fat.a
  - usbhost.a
  - gpio.a
- Runtime
  - string.a
  - stdio.a

Figure 13

The list of archive files that are required for the HelloWorld project are: Kernel.a, BOMS.a, fat.a, usbhost.a, gpio.a, stdio.a and string.a.
The corresponding header files must also be added to the project. This is achieved by going to the Build tab and selecting Header Files. Adding header files is done in the same manner as library files. The header files

required for HelloWorld are: vos.h, USBHost.h, USB.h, BOMS.h, Fat.h, GPIO.h, stdio.h and string.h.

Application Code

This section illustrates writing the main application code. A full listing is in the Getting Started Code Listing topic.
There are three distinct parts to a VNC2 application.
The first of these is the includes section and global definitions; this is where declarations of the Kernel services, runtime libraries and driver header files that are used within the application are.
The second section is the main function; this is the entry point into the application and must only be defined once. Within this function most of the setup and IOMux routines, as well as initializing application threads, are taken care of.
The final component is user threads; these contain the main functionality of the system. An application can have any number of user threads, however in this simple example there is only one. In our application we will only have one thread. When it is created we require to keep a handle to that thread. This is of type vos_tcb_t and defined as a global.

```
vos_tcb_t       *tcbFirmware;
```

Driver Includes and Handles

The head of the HelloWorld.c file must contain include statements for all the header files, Kernel, drivers and runtime libraries that are used. The files are the same as the header files that were added during the FTDI libraries section of this tutorial.

```
#include "vos.h"
#include "USBHost.h"
#include "USB.h"
#include "BOMS.h"
#include "FAT.h"
#include "GPIO.h"
#include "string.h"
#include "helloWorld.h"
```

As well as header files there must also be declarations for any global variables that are to be used throughout the application. When an FTDI driver is opened Device Manager returns a unique handle for that device, each handle is of type `VOS_HANDLE`, declared within devman.h. These handles are used throughout the application to uniquely identify each device so are therefore declared as global variables.

```
VOS_HANDLE      hUsb,
                hBOMS,
                hFAT,
                hGpio;
```

Main Function

The main function is the entry point each time the application is run. Within this routine are most of the initialization routines which are run before starting the application threads.
To begin, declare a context for the USB Host and GPIO drivers, the context is used later to configure the device before opening it.

```
void main(void)
{
usbhost_context_t usb_ctx;
gpio_context_t gpioCtx;
```

Next, initialize the Kernel for the number of devices being used, the time slice for each thread (Quantum) and the interval for timer interrupts (tick). The NUMBER_OF_DEVICES comes from the header file where it was explicitly set to 3; when writing a system that requires more devices it is important to remember to increase this number otherwise any extra devices are not registered with the Kernel and Device Manager. The default clock frequency for the CPU is 48MHz;this has been added for completeness.

```
vos_init(VOS_QUANTUM, VOS_TICK_INTERVAL, NUMBER_OF_DEVICES);
vos_set_clock_frequency(VOS_48MHZ_CLOCK_FREQUENCY);
```

VNC2 features several peripherals, however it is not possible to route all of these signals concurrently. To allow signals to be routed to their required pins VNC2 comes with an I/O Multiplexer (IOMux) which provides a

simple API to allow signals to be routed to specific pins. FTDI provides an IOMux configuration utility as part of the installation, giving a visual representation of the pins to aid with routing signals. The utility will generate C code that can be cut-n-paste straight into any application.
The IOMux code used to routed to connect to a V2EVAL board allows routing to 64, 48 or 32 pin devices. The code here is edited for clarity, refer to Getting Started Code Listing for the full listing.

```
if (vos_get_package_type() == VINCULUM_II_64_PIN)
{
// GPIO port A bit 1 to pin 12
vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
...
vos_iomux_define_input(42,IOMUX_IN_UART_CTS_N); //UART CTS#
}
else if (vos_get_package_type() == VINCULUM_II_48_PIN)
{
// GPIO port A bit 1 to pin 12
vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
...
vos_iomux_define_input(34,IOMUX_IN_UART_CTS_N); //UART CTS#
}
else // VINCULUM_II_32_PIN
{
// GPIO port A bit 1 to pin 12
vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
...
vos_iomux_define_input(26,IOMUX_IN_UART_CTS_N); //UART CTS#
}
```

Next, configure devices and open a handle to each of these devices. VNC2 has two USB Host interfaces available. HelloWorld configures the second USB Host to connect to the flash drive. Within the usbhost_context_t, declare the maximum number of interfaces to be enumerated. When calling the usbhost_init() function, specify the device number to register with Device Manager. In this example it is only necessary to register the second USB Host interface. Therefore pass -1 as the first parameter and the device number for our USB Host (from the header file) as the second parameter. The third parameter is USB host context.

```
// Initialize the USBHost driver and open a handle to the device...
usb_ctx.if_count = 4; // Use a max of 4 USB interfaces
usbhost_init(-1, VOS_DEV_USB_HOST, &usb_ctx);
```

To initialize the GPIO, the port number to be used (A,B,C,D or E) is passed with the device context when calling the gpio_init() function. This is illustrated as follows:

```
// Initialize the GPIO driver and open a handle to the device...
gpioCtx.port_identifier = GPIO_PORT_A;
gpio_init(VOS_DEV_GPIO,&gpioCtx);
```

The BOMS Driver and FAT File System Driver are simpler to call and do not require a context to initialize the device, again they pass the device number to Device Manager to register the driver when boms_init() and fatdrv_init() are called.

```
// Initialize the BOMS driver and open a handle to the device...
boms_init(VOS_DEV_BOMS);
fatdrv_init(VOS_DEV_FAT);
```

All user application threads must be declared within the main routine. When creating a thread, use vos_create_thread() and pass a pointer to the thread function. In this example a forward declaration for a thread called firmware was created in the header file and this is the name that is passed into vos_create_thread().
The first parameter in vos_create_thread() is the thread priority; this value determines the priority of the thread in relation to other application threads. The thread priority must be a value between 1 and 31 with 1 being the lowest priority thread.
The SIZEOF_FIRMWARE_TASK_MEMORY, as defined within the header file, is the amount of stack usage that is allocated to the application thread. The stack size required for a thread depends on its complexity, for this application a stack size of 0x1000 will be more than adequate.
The last parameter is the arg size field; vos_create_thread() allows for any number of extra parameters to be passed into the function. The arg size field must reflect the total size of the arguments passed into the function. In this example the thread has no arguments and therefore arg size is zero.

```
// Create our application thread here...
vos_create_thread(29, SIZEOF_FIRMWARE_TASK_MEMORY, firmware, 0);
```

The last step within the main routine is to call the Kernel Scheduler to start the application threads. The call to vos_start_scheduler() is an indication that setup and initialization is finished, and control passes from main to the application threads.

```
// Start the scheduler to kick off our thread...
vos_start_scheduler();
```

Application Thread

This is the body of the application and contains firmware code to control the VNC2. To start, declare the thread function and local variables

```
void firmware(void)
{
unsigned char *tx_buf = "Hello World! \r\n";
unsigned char connectstate;
unsigned char status;
// USB host variables
usbhost_device_handle *ifDev;
usbhost_ioctl_cb_t hc_iocb;
usbhost_ioctl_cb_class_t hc_iocb_class;
// BOMS device variables
msi_ioctl_cb_t boms_iocb;
boms_ioctl_cb_attach_t boms_att;
// FAT file system variables
fat_ioctl_cb_t fat_ioctl;
fatdrv_ioctl_cb_attach_t fat_att;
FILE *file;
// GPIO variables
gpio_ioctl_cb_t gpio_iocb;
unsigned char leds;
```

Firstly open the USB Host controller driver. The function vos_dev_open() requires the device number of the USB host driver and returns a handle to the instance of the driver.

```
hUsb = vos_dev_open(VOS_DEV_USB_HOST);
```

Next, configure the GPIO driver so that all signals are set to output, this enables feedback to be shown through the LEDs. Control requests to drivers are performed through I/O control calls where the call to be performed is specified and any extra data required by the call is passed in.

```
hGpio = vos_dev_open(VOS_DEV_GPIO);

gpio_iocb.ioctl_code = VOS_IOCTL_GPIO_SET_MASK;
gpio_iocb.value = 0xff; // set all as output
vos_dev_ioctl(hGpio, &gpio_iocb);
```

To determine whether there is a USB device connected to USB Host 2, use the VOS_IOCTL_USBHOST_GET_CONNECT_STATE IOCTL function. When a device is detected, information is relayed back to the user via the LEDs.

```
do
{
//wait for enumeration to complete
hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_GET_CONNECT_STATE;
hc_iocb.get = &connectstate;
vos_dev_ioctl(hUsb, &hc_iocb);
if (connectstate == PORT_STATE_ENUMERATED)
{
    // connected to USB device and enumerated
    leds = 0xAA; vos_dev_write(hGpio,&leds,1,NULL);
```

Now, use the VOS_IOCTL_USBHOST_DEVICE_FIND_HANDLE_BY_CLASS IOCTL to determine if the connected device is a BOMS class device. The class, subclass and protocol of the device must also be passed to this IOCTL. If the driver finds a device matching the BOMS flash disk then a handle is returned in the `get` section of the IOCTL block.

```
// find BOMS class device
hc_iocb_class.dev_class = USB_CLASS_MASS_STORAGE;
hc_iocb_class.dev_subclass = USB_SUBCLASS_MASS_STORAGE_SCSI;
hc_iocb_class.dev_protocol = USB_PROTOCOL_MASS_STORAGE_BOMS;
// user ioctl to find first hub device
hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_DEVICE_FIND_HANDLE_BY_CLASS;
hc_iocb.handle.dif = NULL;
hc_iocb.set = &hc_iocb_class;
hc_iocb.get = &ifDev;
vos_dev_ioctl(hUsb, &hc_iocb);
if(!ifDev)
{
    // We didn't manage to find a device matching the required class.
    break;
}
```

Using the ifDev handle as received from the previous IOCTL, attach the BOMS driver to the flash disc using the BOMS MSI_IOCTL_BOMS_ATTACH IOCTL.

```
hBoms = vos_dev_open(VOS_DEV_BOMS);

// Attach BOMS driver to our USB Flash Disk
boms_att.hc_handle = hUsb;
boms_att.ifDev = ifDev;
boms_iocb.ioctl_code = MSI_IOCTL_BOMS_ATTACH;
boms_iocb.set = &boms_att;
boms_iocb.get = NULL;
status = vos_dev_ioctl(hBoms, &boms_iocb);
```

The FAT driver is required to match the file structure on BOMS devices and allow reading and writing of files. Calling the FAT_IOCTL_FS_ATTACH will cause subsequent file system operations to be sent to the BOMS disk.

```
hFat = vos_dev_open(VOS_DEV_FAT);

// Attach the FAT driver to the BOMS device
fat_ioctl.ioctl_code = FAT_IOCTL_FS_ATTACH;
fat_ioctl.set = &fat_att;
fat_att.boms_handle = hBoms;
fat_att.partition = 0;
status = vos_dev_ioctl(hFAT, &fat_ioctl);
```

Once the FAT file system and BOMS are attached then the stdio library can be initialised with the fsAttach function.

```
fsAttach(hFAT);
```

The stdio library can now be used to access files on the disk.
Notice the use of the strlen function as defined within the string runtime library to calculate the length of the Hello World buffer.

```
file = fopen("TEST.TXT", "a+");
fwrite(tx_buf, strlen(tx_buf), sizeof(char), file);
fclose(file);
```

Follow the instructions in Building Your First Application to build the project and flash the VNC2.

6 Code Listing

HelloWorld.h

```
/*
** HelloWorld.h
**
** Copyright © 2010 Future Devices International Limited
**
**  C Header file for Vinculum II sample application
** Main module
**
** Author: FTDI
** Project: Vinculum II
** Module: Vinculum II Sample Applications
** Requires: VOS BOMS FAT USBHost GPIO STDIO
** Comments: Uses stdio to write files on flash disk
**
** History:
**  1 – Initial version
**
*/

#define SIZEOF_FIRMWARE_TASK_MEMORY 0x1000

#define NUMBER_OF_DEVICES       4
#define VOS_DEV_USB_HOST        0
#define VOS_DEV_BOMS            1
#define VOS_DEV_FAT             2
#define VOS_DEV_GPIO            3

#define LED0   0x02
#define LED1   0x04
#define LED2   0x20
#define LED3   0x40
```

HelloWorld.c

```
/*
** HelloWorld.c
**
** Copyright © 2010 Future Devices International Limited
**
**  C Source file for Vinculum II sample application
** Main module
**
** Author: FTDI
** Project: Vinculum II
** Module: Vinculum II Sample Applications
** Requires: VOS BOMS FAT UART USBHost GPIO STDIO
** Comments: Uses stdio to write files on flash disk
**
** History:
**  1 – Initial version
**
*/

#include "vos.h"

#include "USBHost.h"
#include "USB.h"
#include "MSI.h"
#include "BOMS.h"
#include "FAT.h"
#include "GPIO.h"
```

```
#include "stdio.h"
#include "string.h"


#include "HelloWorld.h"

VOS_HANDLE       hUsb,
                 hBoms,
                 hGpio,
                 hFAT;

vos_tcb_t        *tcbFirmware;


char *tx_buf = "Hello World! \n";


void firmware(void);

void main(void)
{
    // USB Host configuration context
    usbhost_context_t usb_ctx;
    // GPIO configuration context
    gpio_context_t gpioCtx;

    vos_init(10, VOS_TICK_INTERVAL, NUMBER_OF_DEVICES);
    vos_set_clock_frequency(VOS_48MHZ_CLOCK_FREQUENCY);

    if (vos_get_package_type() == VINCULUM_II_64_PIN)
    {
        // GPIO port A bit 1 to pin 12
        vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
        // GPIO port A bit 2 to pin 13
        vos_iomux_define_output(13,IOMUX_OUT_GPIO_PORT_A_2); //LED4
        // GPIO port A bit 5 to pin 29
        vos_iomux_define_output(29,IOMUX_OUT_GPIO_PORT_A_5); //LED5
        // GPIO port A bit 6 to pin 31
        vos_iomux_define_output(31,IOMUX_OUT_GPIO_PORT_A_6); //LED6
        // UART to V2EVAL board pins
        vos_iomux_define_output(39,IOMUX_OUT_UART_TXD); //UART Tx
        vos_iomux_define_input(40,IOMUX_IN_UART_RXD); //UART Rx
        vos_iomux_define_output(41,IOMUX_OUT_UART_RTS_N); //UART RTS#
        vos_iomux_define_input(42,IOMUX_IN_UART_CTS_N); //UART CTS#
    }
    else if (vos_get_package_type() == VINCULUM_II_48_PIN)
    {
        // GPIO port A bit 1 to pin 12
        vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
        // GPIO port A bit 2 to pin 13
        vos_iomux_define_output(13,IOMUX_OUT_GPIO_PORT_A_2); //LED4
        // GPIO port A bit 4 to pin 45
        vos_iomux_define_output(45,IOMUX_OUT_GPIO_PORT_A_4); //LED6
        // GPIO port A bit 5 to pin 46
        vos_iomux_define_output(46,IOMUX_OUT_GPIO_PORT_A_5); //LED5
        // UART to V2EVAL board pins
        vos_iomux_define_output(31,IOMUX_OUT_UART_TXD); //UART Tx
        vos_iomux_define_input(32,IOMUX_IN_UART_RXD); //UART Rx
        vos_iomux_define_output(33,IOMUX_OUT_UART_RTS_N); //UART RTS#
        vos_iomux_define_input(34,IOMUX_IN_UART_CTS_N); //UART CTS#
    }
    else // VINCULUM_II_32_PIN
    {
        // GPIO port A bit 1 to pin 12
        vos_iomux_define_output(12,IOMUX_OUT_GPIO_PORT_A_1); //LED3
        // GPIO port A bit 2 to pin 14
        vos_iomux_define_output(14,IOMUX_OUT_GPIO_PORT_A_2); //LED4
        // UART to V2EVAL board pins
        vos_iomux_define_output(23,IOMUX_OUT_UART_TXD); //UART Tx
        vos_iomux_define_input(24,IOMUX_IN_UART_RXD); //UART Rx
        vos_iomux_define_output(25,IOMUX_OUT_UART_RTS_N); //UART RTS#
        vos_iomux_define_input(26,IOMUX_IN_UART_CTS_N); //UART CTS#
    }

    // use a max of 4 USB devices
    usb_ctx.if_count = 4;
    usbhost_init(-1, VOS_DEV_USB_HOST, &usb_ctx);

    boms_init(VOS_DEV_BOMS);
    fatdrv_init(VOS_DEV_FAT);

    gpioCtx.port_identifier = GPIO_PORT_A;
    gpio_init(VOS_DEV_GPIO,&gpioCtx);

    tcbFirmware = vos_create_thread(29, SIZEOF_FIRMWARE_TASK_MEMORY, firmware, 0);

    vos_start_scheduler();

main_loop:
    goto main_loop;
}

void firmware(void)
{
    unsigned char connectstate;
    unsigned char status;

    usbhost_device_handle *ifDev;
    usbhost_ioctl_cb_t hc_iocb;
    usbhost_ioctl_cb_class_t hc_iocb_class;

    msi_ioctl_cb_t boms_iocb;
    boms_ioctl_cb_attach_t boms_att;

    fat_ioctl_cb_t fat_ioctl;
    fatdrv_ioctl_cb_attach_t fat_att;

    gpio_ioctl_cb_t gpio_iocb;
    unsigned char leds;

    FILE *file;


    // open host controller
    hUsb = vos_dev_open(VOS_DEV_USB_HOST);

    // open GPIO device
    hGpio = vos_dev_open(VOS_DEV_GPIO);

    gpio_iocb.ioctl_code = VOS_IOCTL_GPIO_SET_MASK;
    gpio_iocb.value = 0xff;    // set all as output
    vos_dev_ioctl(hGpio, &gpio_iocb);

    do
    {
        //wait for enumeration to complete
        vos_delay_msecs(250);
        leds = LED0;  vos_dev_write(hGpio,&leds,1,NULL);
        vos_delay_msecs(250);
        leds = 0;  vos_dev_write(hGpio,&leds,1,NULL);

        // user ioctl to see if bus available
        hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_GET_CONNECT_STATE;
        hc_iocb.get = &connectstate;
        vos_dev_ioctl(hUsb, &hc_iocb);

        if (connectstate == PORT_STATE_ENUMERATED)
        {
            leds = LED1;  vos_dev_write(hGpio,&leds,1,NULL);

            // find and connect a BOMS device

            // USBHost ioctl to find first BOMS device on host
            hc_iocb.ioctl_code = VOS_IOCTL_USBHOST_DEVICE_FIND_HANDLE_BY_CLASS;
            hc_iocb.handle.dif = NULL;
            hc_iocb.set = &hc_iocb_class;
            hc_iocb.get = &ifDev;
            hc_iocb_class.dev_class = USB_CLASS_MASS_STORAGE;
            hc_iocb_class.dev_subclass = USB_SUBCLASS_MASS_STORAGE_SCSI;
            hc_iocb_class.dev_protocol = USB_PROTOCOL_MASS_STORAGE_BOMS;

            if (vos_dev_ioctl(hUsb, &hc_iocb) != USBHOST_OK)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            // now we have a device, intialise a BOMS driver for it
            hBoms = vos_dev_open(VOS_DEV_BOMS);

            // BOMS ioctl to attach BOMS driver to device on host
            boms_iocb.ioctl_code = MSI_IOCTL_BOMS_ATTACH;
            boms_iocb.set = &boms_att;
            boms_iocb.get = NULL;
            boms_att.hc_handle = hUsb;
            boms_att.ifDev = ifDev;

            status = vos_dev_ioctl(hBoms, &boms_iocb);
            if (status != MSI_OK)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            // now we have the BOMS connected open the FAT driver
            hFAT = vos_dev_open(VOS_DEV_FAT);

            fat_ioctl.ioctl_code = FAT_IOCTL_FS_ATTACH;
            fat_ioctl.set = &fat_att;
            fat_att.boms_handle = hBoms;
            fat_att.partition = 0;

            status = vos_dev_ioctl(hFAT, &fat_ioctl);
            if (status != FAT_OK)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            // lastly attach the stdio file system to the FAT file system
            fsAttach(hFAT);

            // now call the stdio runtime functions
            file = fopen("TEST.TXT", "a+");

            if (file == NULL)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            if (fwrite(tx_buf, strlen(tx_buf), sizeof(char), file) == -1)
            {
                 leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                 vos_delay_msecs(1000);
            }

            if (fclose(file) == -1)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
            }

            leds = LED1;  vos_dev_write(hGpio,&leds,1,NULL);

            fat_ioctl.ioctl_code = FAT_IOCTL_FS_DETACH;
            if (vos_dev_ioctl(hFAT, &fat_ioctl) != FAT_OK)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            vos_dev_close(hFAT);

            boms_iocb.ioctl_code = MSI_IOCTL_BOMS_DETACH;
            if (vos_dev_ioctl(hBoms, &boms_iocb) != MSI_OK)
            {
                leds = LED3;  vos_dev_write(hGpio,&leds,1,NULL);
                vos_delay_msecs(1000);
                break;
            }

            vos_dev_close(hBoms);

            leds = 0;  vos_dev_write(hGpio,&leds,1,NULL);

            vos_delay_msecs(5000);
        }

    } while (1);
}
```

7 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales)	sales1@ftdichip.com
E-mail (Support)	support1@ftdichip.com
E-mail (General Enquiries)	admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com
Web Shop URL	http://www.ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan , R.O.C.
Tel: +886 (0) 2 8797 1330
Fax: +886 (0) 2 8751 9737

E-mail (Sales)	tw.sales1@ftdichip.com
E-mail (Support)	tw.support1@ftdichip.com
E-mail (General Enquiries)	tw.admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Hillsboro, Oregon, USA

Future Technology Devices International Limited (USA)
7235 NW Evergreen Parkway, Suite 600
Hillsboro, OR 97123-5803
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales)	us.sales@ftdichip.com
E-Mail (Support)	us.support@ftdichip.com
E-Mail (General Enquiries)	us.admin@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited (China)
Room 408, 317 Xianxia Road,
Shanghai, 200051
China
Tel: +86 21 62351596
Fax: +86 21 62351595

E-Mail (Sales): cn.sales@ftdichip.com
E-Mail (Support): cn.support@ftdichip.com
E-Mail (General Enquiries): cn.admin1@ftdichip.com
Web Site URL: http://www.ftdichip.com

Distributor and Sales Representatives

Please visit the Sales Network page of the FTDI Web site for the contact details of our distributor(s) and sales representative(s) in your country.

8 Revision History

Revision	Changes	Date
V1.0	Initial Release	2010-05-11
V1.2.0	Update to use stdio library and FAT file system driver	2010-08-19