Future Technology Devices International Ltd.

Application Note AN_155

Vinculum-II SD Card Example

Document Reference No.: FT_000350

Version 1.2

Issue Date: 2010-12-14

This application note provides an example of using the FTDI Vinculum-II (VNC2) to communicate with an SD Card. Drivers and source code is also provided (downloaded from FTDI website)

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 wide ranging popularity of Secure Digital (SD) Cards in consumer electronics coupled with their low cost, high capacity and small size make them an ideal option for a number of embedded applications. This application note demonstrates how to connect an SD Card to a VNC2 V2-Eval Board and then illustrates how to use FTDI's supplied drivers to communicate with the SD Card using a FAT File System interface.

The source files and libraries needed for this application can be downloaded from the FTDI website at the following location:
http://www.ftdichip.com/Support/SoftwareExamples/VinculumIIProjects/SD_Card_Beta.zip

1.1 Overview

The SD Card specification provides a subset of the SD protocol allowing communication with the devices using a Serial Peripheral Interface Bus. SPI mode is a simplistic version of the SD spec allowing for easy access to the devices via microcontrollers. The main disadvantage of SPI mode is the loss in performance when compared to a full SD Card host.

SD Cards are shipped pre-formatted with a FAT File System; this allows FAT driver access to the file structure held on the devices. SD Cards therefore provide a cost effective way of storing data, e.g. data logging, on a removable storage device when combined with VNC2's host/slave capability.

1.2 Supported SD Cards

The FAT driver and SD Card driver supplied with this application example have been tested with the following SD Cards:

SD HC

Kingston SD HC 4GB Class 4

Transcend SD HC 8GB Class 6

Patriot Memory SD HC 16GB Class 6

SD v1.0

Canon SD 32MB

2 SD Card Application Layout

Figure 2.1 below illustrates the layout of a typical SD Card application. The general flow of data from the user application travels through each of the device driver layers to the SD Card and back again. FTDI provide each of the driver layers within the system, the application layer is the only code that must be written by the user.

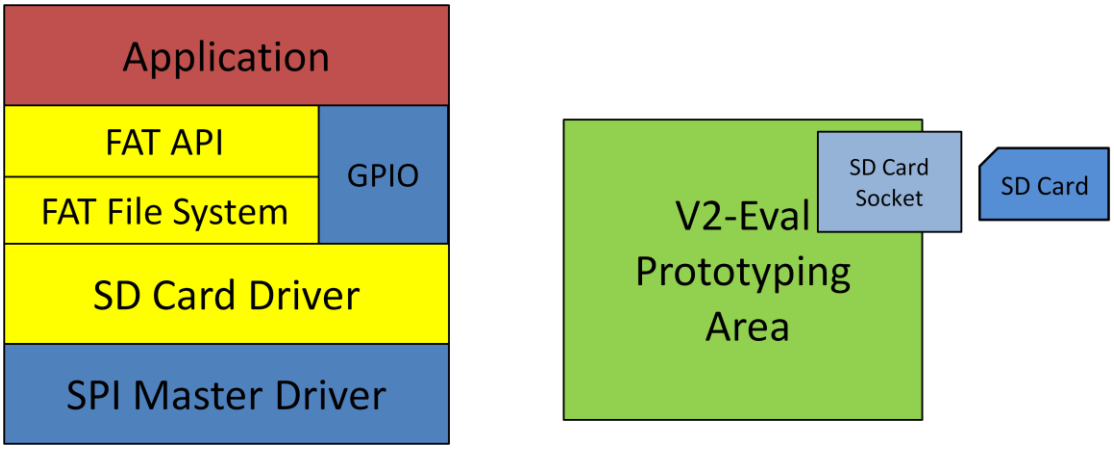


Figure 2.1 – Application Layout

The application requires the following device drivers:

The FAT API and FAT File System (FFS) driver; interfaces with the SD Card file structure allowing for open/read/write/close etc. operations on the SD Card contents.

The SD Card driver, structures data from the FAT layer into a format that conforms to the SD Card SPI specification and passes the data onto the SPI Master.

The SPI Master driver is the physical medium that transfers data from the card to the VNC2 and vice versa.

As an added feature the GPIO driver can be used to check the state of the Card Detect and Write Protect lines from the SD Card; this will be discussed in section 4.

3 Hardware Setup

This SD Card example requires the following pieces of hardware: a host PC to connect to the VNC2; VNC2 customer evaluation board V2-Eval; a VNC2 64-pin QFN daughtercard; an SD Card socket; a FAT formatted SD Card (Transcend SD HC 8GB used in this example); a USB A-B cable to connect the V2-Eval Board to the host PC; and a soldering iron with appropriate wire to solder the SD socket to the prototype area. The USB cable is only used in this sample to program the firmware into the VNC2; after this point the VNC2 runs as a standalone system without the need for the host PC.

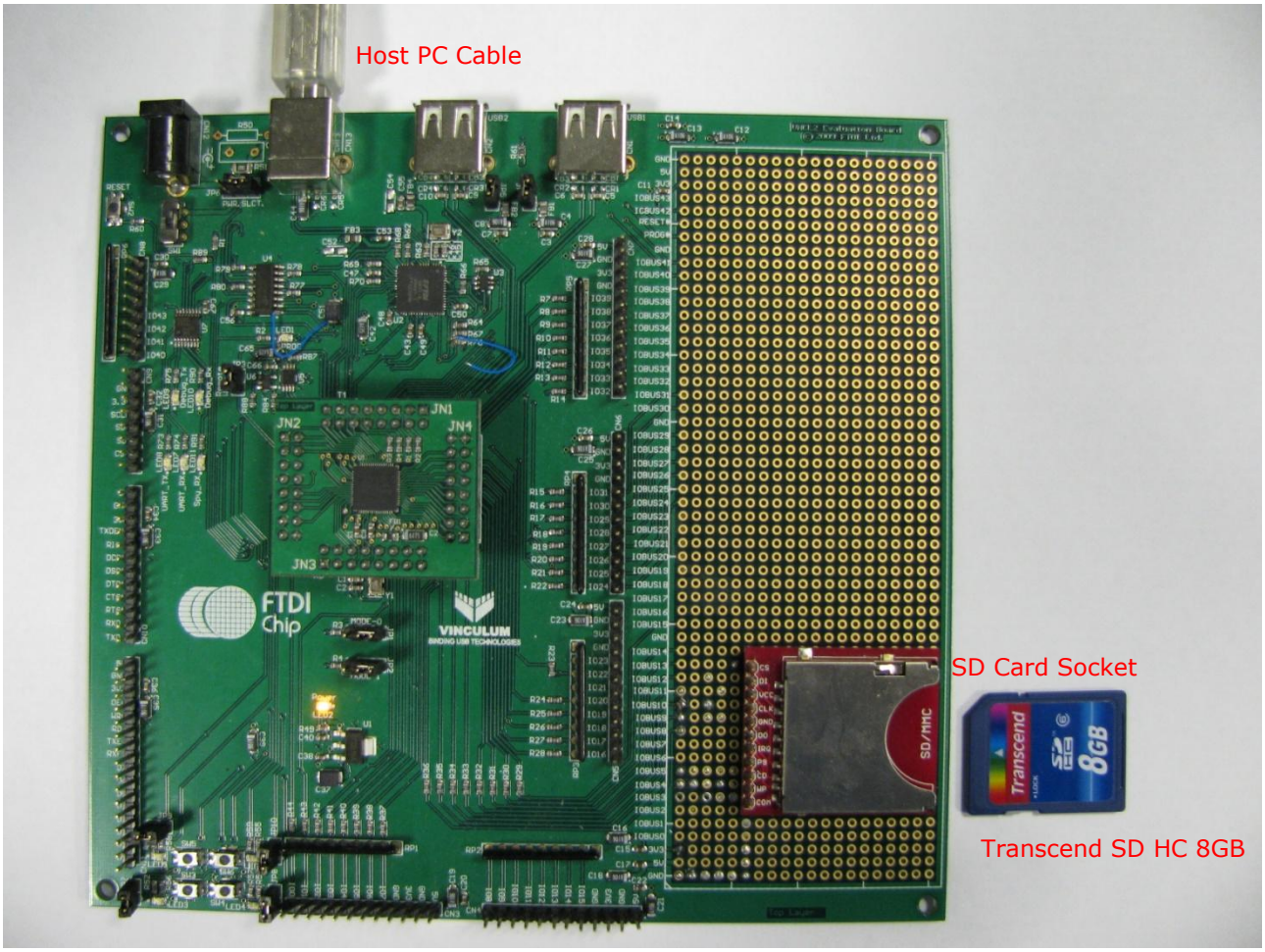
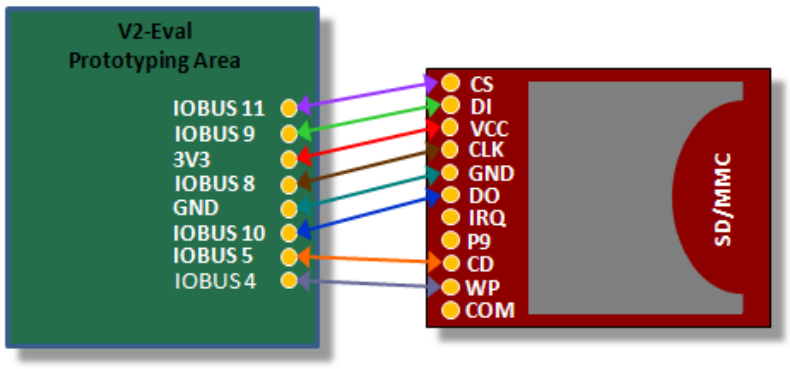


Figure 3.1 – Hardware Setup

3.1 Connecting SD Card Socket

Figure 3.1 illustrates the SD Card socket soldered to the prototyping area of the V2-Eval Board. The following figure, 3.2, represents the solder connections that have been made from the prototyping area to the SD Card socket. The IOBUS pins used when connecting an SD socket are open to the designer, the pins used in this example simply represent a working setup. Please remember that the IOMux on the VNC2 must be configured to match the IOBUS pins selected, this will be discussed further in section 4.1.



Connection	Signal
IOBUS11 <-> CS	Chip Select
IOBUS9 <-> DI	MOSI
3V3 <-> VCC	VCC
IOBUS8 <-> CLK	Clock
GND <-> GND	Ground
IOBUS10 <-> DO	MISO
IOBUS5 <-> CD	Card Detect
IOBUS4 <-> WP	Write Protect

Figure 3.2 - SD Socket Connections

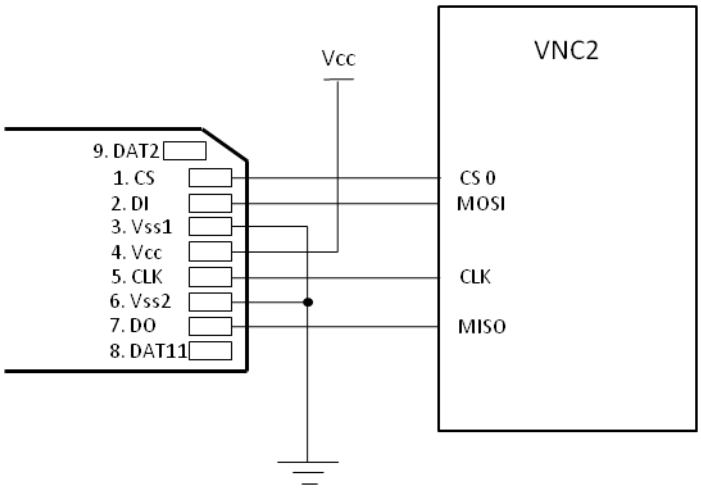


Figure 3.3 - SD Socket Schematic

4 Application Source Code

The following section demonstrates how to write an application to write a file to an SD Card using the card's FAT File System. This section anticipates that the user has a sound understanding of application structure and the fundamental concepts of how the Vinculum Operating System (VOS) works. For an introduction to VinIDE, VOS and an overview of the general application structure please refer to the Vinculum-II tool-chain Getting Started Guide. The Vinculum-II tool-chain consists of a compiler, assembler, linker and debugger encapsulated in a graphical interface (the IDE) and is used to develop customised firmware for VNC2.

The following example demonstrates how to build a sample application from a blank project using the IDE. The sample demonstrates writing the string "Hello World" to a new file on an SD Card. A complete listing of the sample code is available at Appendix B at the end of this applications note. This sample source code has been tested but is provided for illustration only and is not supported by FTDI.

4.1 IOMux Setup

The VNC2 must be configured using the IOMux such that the signals coming from the device match the soldered pins on the SD socket. The following IOMux configuration source code is for the socket connection as defined within section 3.1 connected to a 64 pin VNC2 daughter card. FTDI provide an IOMux configuration utility, as part of the IDE, to aid with configuring IOMux signals; for more information search for '*IOMux Configuration Utility*' within the IDE help file.

Pins 19, 20, 22 and 23 of VNC2 are the SPI Master connections and are controlled by the SPI Master driver. Pin 16, Card Detect, will allow the user to determine if an SD Card is connected to the SD socket before attempting to communicate with a device. Pin 15, the Write Protect line, indicates whether the physical write protect lock on the SD Card is set (figure 4.1); the SD Card will not allow data to be written to the device if the write protect lock is set. Both pin 15 and 16 are controlled by a GPIO port, it is not mandatory however to have these lines connected. This will be discussed in greater detail within section 4.2.

Figure 4.1 – Write Protect

```
unsigned char IOMux_Setup()
{
    unsigned char packageType = vos_get_package_type();

    if (packageType == VINCULUM_II_64_PIN)
    {
    // SPIMaster to V2EVAL board pins
       // IOBus 8 (Pin 19) Master Clock Output
       vos_iomux_define_output(19 ,IOMUX_OUT_SPI_MASTER_CLK);
       // IOBus 9 (Pin 20) Master Out Slave In
       vos_iomux_define_output(20,IOMUX_OUT_SPI_MASTER_MOSI);
       // IOBus 10 (Pin 22) Master In Slave Out
       vos_iomux_define_input(22,IOMUX_IN_SPI_MASTER_MISO);
       // IOBus 11 (Pin 23) Chip Select Output
       vos_iomux_define_output(23,IOMUX_OUT_SPI_MASTER_CS_0);
       // IOBus 5 (Pin 16) Card Detect Input
       vos_iomux_define_input(16,IOMUX_IN_GPIO_PORT_B_1);
       // IOBus 4 (Pin 15) Write Protect Input
       vos_iomux_define_input(15,IOMUX_IN_GPIO_PORT_B_0);
    }

    return SD_SUCCESS;
}
```

4.2 SD Card Setup

The SD Card sample requires the following FTDI provided drivers: FAT; GPIO; SPI Master; SD Card; the VOS kernel and the stdio library.

4.2.1 Adding Driver Libraries

Each of the device drivers to be used in the application must be added to the current project within the IDE. Device drivers come in the form of library files that are all bundled with the IDE installer. Library files are easily added to the current project using the Project Libraries window within the IDE. For help with adding library files to VNC2 projects please refer to the Getting Started Guide.

4.2.2 Initializing Drivers

Prior to starting the VNC2 RTOS scheduler (VOS) within the main routine, all drivers to be used within the system must be initialized; that is, have the init routine called for each driver. The listing below demonstrates initializing each of the device drivers required for the sample.

Note the init routine for the SPI Master uses a context with a 512 byte buffer size, this is important as the SD Card driver will not work with a buffer less than this.

After each of the drivers has been initialized the scheduler is called to start all application threads, in this case there is only one thread, tcbFirmware.

```
Void main(void)
{
    // SPI Master context
    7pymaster_context_t spimasterCtx;
    // GPIO context
    gpio_context_t gpioCtx;

    // Call VOS initialisation routines
    vos_init(50, VOS_TICK_INTERVAL, NUMBER_OF_DEVICES);
    vos_set_clock_frequency(VOS_48MHZ_CLOCK_FREQUENCY);

    //*********************************************************
    // INITIALISE DRIVERS
    //*********************************************************
    spimasterCtx.buffer_size = VOS_BUFFER_SIZE_512_BYTES;
    7pymaster_init(VOS_DEV_SPIMASTER, &spimasterCtx);
    sd_init(VOS_DEV_SDCARD);
    fatdrv_init(VOS_DEV_FTFS);
    gpioCtx.port_identifier = GPIO_PORT_B;
    gpio_init(VOS_DEV_GPIO, &gpioCtx);

    // Create threads for firmware application (no parameters)
    tcbFirmware = vos_create_thread(50, 8192, firmware, 0);

    // Device IOMux Settings
    IOMux_Setup();

    // Start VOS scheduler
    vos_start_scheduler();

main_loop:
    goto main_loop;
}
```

4.2.3 Opening Drivers

The firmware thread tcbFirmware, created at the end of the main routine, contains the code for opening each of the device handles. The VOS_HANDLE obtained from a vos_dev_open is used throughout the application to reference each respective driver layer. This is the method that allows data to pass between the device driver layers to the SD Card as discussed in section 2 and shown in figure 2.1.

At this point the GPIO driver is also configured such that our Card Detect and Write Protect pins are set as input; this will have the affect of allowing the SD Card driver to check the state of each line.

```
Void firmware(void)
{
    VOS_HANDLE hSPIMaster, hSDCard, hFat, hGPIO;
    msi_ioctl_cb_t msi_cb;
    gpio_ioctl_cb_t gpio_iocb;
    sdcard_ioctl_cb_attach_t sd_cb;
    fatdrv_ioctl_cb_attach_t fat_cb;
    fat_ioctl_cb_t fat_iocb;

    unsigned char *s = "Hello World!";
    unsigned char t[12];
    FILE *file;
    // Open a handle to each of our devices.
    hSPIMaster = vos_dev_open(VOS_DEV_SPIMASTER);
    hFat = vos_dev_open(VOS_DEV_FTFS);
    hSDCard = vos_dev_open(VOS_DEV_SDCARD);
    hGPIO = vos_dev_open(VOS_DEV_GPIO);

    // Configure the GPIO port to input.
    Gpio_iocb.ioctl_code = VOS_IOCTL_GPIO_SET_MASK;
    gpio_iocb.value = 0x00;
    vos_dev_ioctl(hGPIO, &gpio_iocb);
```

4.2.4 Attaching Drivers

The next stage is to join the device driver layers of the system allowing data to pass freely between each layer. Joining the layers is achieved through the passing of device handles to an attach ioctl for each driver. Starting at the bottom of the system device layers, the SD Card driver is connected to the underlying SPI Master, figure 4.2.

```
    // Attach SPI Master and the GPIO to the SD Card driver.
    Sd_cb.SPI_Handle = hSPIMaster;
    sd_cb.GPIO_Handle = hGPIO;
    sd_cb.WP_Bit = GPIO_PIN_0;
    sd_cb.CD_Bit = GPIO_PIN_1;
    msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_ATTACH;
    msi_cb.set = &sd_cb;
    vos_dev_ioctl(hSDCard, &msi_cb);
```

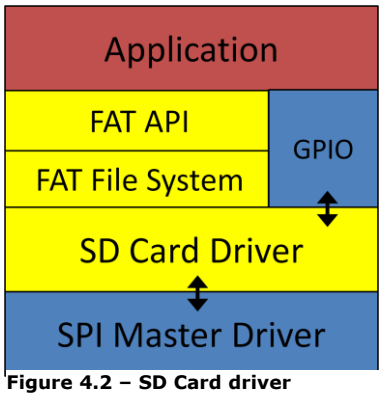


Figure 4.2 – SD Card driver attached to SPI Master and GPIO

The SD Card driver uses the `sdcard_ioctl_cb_attach_t` structure to configure the driver on an attach. The SPI_Handle member is a handle to the SPI Master driver which is a required field. The member GPIO_Handle is a handle to the GPIO port used to check the status of the Write Protect and Card Detect lines from the SD Card. As GPIO ports are 8 bits wide the WP_Bit and CD_Bit are the respective pins within the GPIO driver that the SD Card driver should monitor for changes to Write Protect and Card Detect. In relation to the above sample: if the Card Detect line (active low), GPIO pin 1, goes high the SD Card driver can report that there is no card within the socket. The SD Card driver does not require that the user provides a handle to a GPIO port, if this is the case however the handle must be set to NULL.

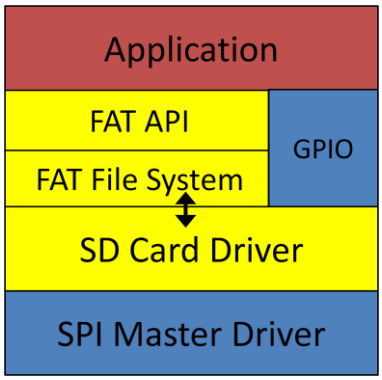


Figure 4.3 – SD Card driver attached to the FFS

Like BOMS class devices the SD Card driver interfaces to underlying hardware using the Mass Storage Interface (MSI) API. The `sdcard_ioctl_cb_attach_t` data is packaged into the `msi_ioctl_cb_t` structure and sent to the SD Card driver using the ioctl call.

Working up through the device layers to the FAT File System driver; the handle to the SD Card driver is passed to the attach ioctl to join the two layers, figure 4.3.

```
// Attach the SD Card to the Fat File System driver.
Fat_iocb.ioctl_code = FAT_IOCTL_FS_ATTACH;
fat_iocb.set = &fat_cb;
fat_cb.msi _handle = hSDCard;
fat_cb.partition = 0;
vos_dev_ioctl(hFat, &fat_iocb);
```

Finally the FaAT File System is attached to the FAT API (figure 4.4) joining each of the driver layers to the user application.

```
// Attach the Fat File System to the Fat API
fsAttach(hFat);
```

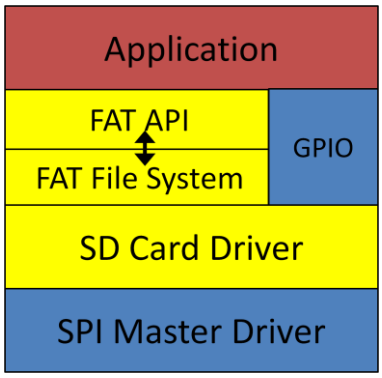


Figure 4.4 – FFS driver attached to Fat API

4.2.5 Initializing SD Card

For an SD Card to communicate via the SPI protocol it must be initialized in SPI mode at power-up. The SD Card driver provides an ioctl call that will initialize the card in SPI mode and determine the version of SD Card connected. This ioctl call must be made before attempting to read or write data to the card.

The following code initializes the card in SPI mode and uses another ioctl to return the card type. An extra check has also been added to make sure that there is an SD Card present before trying to communicate with it.

```
// Check we have an SD Card connected to the socket.
Msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_DETECT;
vos_dev_ioctl(hSDCard, &msi_cb);
if(*msi_cb.get != SD_CARD_PRESENT) {
    return;
}

// Initialize the card in SPI mode.
Msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_INIT;
vos_dev_ioctl(hSDCard, &msi_cb);

// Get the card type.
Msi_cb.ioctl_code = MSI_IOCTL_SD_GET_CARD_TYPE;
vos_dev_ioctl(hSDCard, &msi_cb);
card_type = *msi_cb.get;
```

The card_type value returned from the `MSI_IOCTL_SD_GET_CARD_TYPE` ioctl call is a char value which can be decoded using the #defines within the SD Card header file. The below table summarises the available card types.

Value	Card Type	Description
0x01	SD_V1	Ver1.x Standard Capacity SD Memory Card
0x02	SD_V2	Ver2.00 or later Standard Capacity SD Memory Card
0x03	SD_HC	Ver2.00 or later High Capacity SD Memory Card
0x04	SD_MMC	Not SD Memory Card, may be an MMC card
0xFF	SD_INV	SD Card is invalid and cannot be used

4.3 Opening File and Writing Data

At this stage it is now possible to completely control the FFS on the SD Card using the VNC2 application. This sample demonstrates how to create a new file on the device, write a string of data to that file, read that same string back and to then close the file. For a full list of all FFS operations that are available through the FAT API refer to the stdio section within the VinIDE help file.

```
// Open a new file for writing.
File = fopen("AFILE.TXT", "w+");
fwrite(s, strlen(s), sizeof(char), file);
fseek(file, 0, FAT_SEEK_SET);
fread(dir, strlen(s), sizeof(char), file);
fseek(file, strlen(s), FAT_SEEK_SET);
fwrite(dir, strlen(s), sizeof(char), file);
fclose(file);
```

5 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 8791 3570
Fax: +886 (0) 2 8791 3576

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.admin@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.

6 Appendix A – References

Document References

Vinculum-II Embedded Dual USB Host Controller IC Data Sheet

Vinculum-II IO Mux Explained

Vinculum-II Tool Chain Getting Started Guide

Acronyms and Abbreviations

Terms	Description
IOMux	Input Output Multiplexer – Used to configure pin selection on different package types of the VNC2.
V2-Eval	Vinculum II Evaluation Board- Customer evaluation board for the VNC2 allowing prototype development.
FFS	FAT File System – A file system used to structure data within a block of memory.
VinIDE	Vinculum Integrated Development Environment – Development environment for writing and building application code for the VNC2.
VOS	Vinculum Operating System -
MSI	Mass Storage Interface – A common interface allowing file system drivers to communicate with the underlying hardware.
API	Application Programming Interface – An abstract layer between user applications and underlying system drivers.
VNC2	Vinculum II – FTDI's second generation dual Host/Slave IC.
IDE	Integrated Development Environment
SD	Secure Digital

7 Appendix B – Code Listing

```
/*
This software is provided by Future Technology Devices International Limited "as is" and
any express or implied warranties, including, but not limited to, the implied warranties
of merchantability and fitness for a particular purpose are disclaimed. In no event shall
future technology devices international limited be liable for any direct, indirect,
incidental, special, exemplary, or consequential damages (including, but not limited to,
procurement of substitute goods or services; loss of use, data, or profits; or business
interruption) however caused and on any theory of liability, whether in contract, strict
liability, or tort (including negligence or otherwise) arising in any way out of the use
of this software, even if advised of the possibility of such damage.
*/
//SD_Card_Sample.h
#define NUMBER_OF_DEVICES 4

#define VOS_DEV_SPIMASTER 0
#define VOS_DEV_SDCARD    1
#define VOS_DEV_FTFS      2
#define VOS_DEV_GPIO      3

// SD_Card_Sample.c
#include "vos.h"
#include "devman.h"
#include "IOMUX.h"
#include "msi.h"
#include "fat.h"
#include "gpio.h"
#include "sdcard.h"
#include "SD_Card_Sample.h"
#include "SPIMaster.h"
#include "string.h"
#include "stdio.h"

// Forward Declarations
void firmware(void);
unsigned char IOMux_Setup();

// Globals
vos_tcb_t *tcbFirmware;

unsigned char IOMux_Setup()
{
    unsigned char packageType = vos_get_package_type();

    if (packageType == VINCULUM_II_64_PIN)
    {
        // SPIMaster to V2EVAL board pins
        vos_iomux_define_output(19 ,IOMUX_OUT_SPI_MASTER_CLK); // Slave Clock
        vos_iomux_define_output(20,IOMUX_OUT_SPI_MASTER_MOSI); // Master Out Slave In
        vos_iomux_define_input(22,IOMUX_IN_SPI_MASTER_MISO);   // Master In Slave Out
        vos_iomux_define_output(23,IOMUX_OUT_SPI_MASTER_CS_0); // Chip Select
        vos_iomux_define_input(16,IOMUX_IN_GPIO_PORT_B_1);     // Card Detect
        vos_iomux_define_input(15,IOMUX_IN_GPIO_PORT_B_0);     // Write Protect
    }

    return SD_SUCCESS;
}

void main(void)
```



```
{
    // SPI Master context
    15pymaster_context_t spimasterCtx;
    // GPIO context
    gpio_context_t gpioCtx;

    // Call VOS initialisation routines
    vos_init(50, VOS_TICK_INTERVAL, NUMBER_OF_DEVICES);
    vos_set_clock_frequency(VOS_48MHZ_CLOCK_FREQUENCY);

    //*********************************************************
    // INITIALISE DRIVERS
    //*********************************************************
    spimasterCtx.buffer_size = VOS_BUFFER_SIZE_512_BYTES;
    15pymaster_init(VOS_DEV_SPIMASTER, &spimasterCtx);
    sd_init(VOS_DEV_SDCARD);
    fatdrv_init(VOS_DEV_FTFS);
    gpioCtx.port_identifier = GPIO_PORT_B;
    gpio_init(VOS_DEV_GPIO, &gpioCtx);

    // Create threads for firmware application (no parameters)
    tcbFirmware = vos_create_thread(50, 8192, firmware, 0);

    // Device IOMux Settings
    IOMux_Setup();

    // Start VOS scheduler
    vos_start_scheduler();

main_loop:
    goto main_loop;
}

void firmware(void)
{
    VOS_HANDLE hSPIMaster, hSDCard, hFat, hGPIO;
    msi_ioctl_cb_t msi_cb;
    gpio_ioctl_cb_t gpio_iocb;
    sdcard_ioctl_cb_attach_t sd_cb;
    fatdrv_ioctl_cb_attach_t fat_cb;
    fat_ioctl_cb_t fat_iocb;

    unsigned char *s = "Hello World!";
    unsigned char t[12];
    unsigned char card_type;
    FILE *file;

    // Open a handle to each of our devices.
    hSPIMaster = vos_dev_open(VOS_DEV_SPIMASTER);
    hFat = vos_dev_open(VOS_DEV_FTFS);
    hSDCard = vos_dev_open(VOS_DEV_SDCARD);
    hGPIO = vos_dev_open(VOS_DEV_GPIO);

    // Configure the GPIO port to input.
    Gpio_iocb.ioctl_code = VOS_IOCTL_GPIO_SET_MASK;
    gpio_iocb.value = 0x00;
    vos_dev_ioctl(hGPIO, &gpio_iocb);

    // Attach SPI Master and the GPIO to the SD Card driver.
    Sd_cb.SPI_Handle = hSPIMaster;
    sd_cb.GPIO_Handle = hGPIO;
```

```
    sd_cb.WP_Bit = GPIO_PIN_0;
    sd_cb.CD_Bit = GPIO_PIN_1;
    msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_ATTACH;
    msi_cb.set = &sd_cb;
    vos_dev_ioctl(hSDCard, &msi_cb);

    // Attach the SD Card to the Fat File System driver.
    Fat_iocb.ioctl_code = FAT_IOCTL_FS_ATTACH;
    fat_iocb.set = &fat_cb;
    fat_cb.msi_handle = hSDCard;
    fat_cb.partition = 0;
    vos_dev_ioctl(hFat, &fat_iocb);

    // Attach the Fat File System to the Fat API
    fsAttach(hFat);

    // Check we have an SD Card connected to the socket.
    Msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_DETECT;
    vos_dev_ioctl(hSDCard, &msi_cb);
    if(*msi_cb.get != SD_CARD_PRESENT) {
        return;
    }

    // Initialize the card in SPI mode.
    Msi_cb.ioctl_code = MSI_IOCTL_SD_CARD_INIT;
    vos_dev_ioctl(hSDCard, &msi_cb);

    // Get the card type.
    Msi_cb.ioctl_code = MSI_IOCTL_SD_GET_CARD_TYPE;
    vos_dev_ioctl(hSDCard, &msi_cb);
    card_type = *msi_cb.get;

    // Open a new file for writing.
    File = fopen("AFILE.TXT", "w+");
    fwrite(s, strlen(s), sizeof(char), file);
    fseek(file, 0, FAT_SEEK_SET);
    fread(t, strlen(s), sizeof(char), file);
    fseek(file, strlen(s), FAT_SEEK_SET);
    fwrite(t, strlen(s), sizeof(char), file);
    fclose(file);

}
```

8 Appendix C – Revision History

Revision	Changes	Date
1.0	Initial Release	2010-10-14
1.1	Modified front page format	2010-10-28
1.2	Modified figure 3.2	2010-12-14