EMBEDDED USB DESIGN BY EXAMPLE

John Hyde

Part 1 & 2

USB MADE EASY

www.ftdichip.com

FTDI Chip

Embedded USB Design By Example

John Hyde

(Commissioned by FTDI Ltd)

Embedded USB Design By Example

John Hyde

Foreword by Fred Dart – Founder and CEO of FTDI.

John Hyde is an internationally recognised and renowned figure in the field of USB, having authored the seminal “USB Design By Example” series of books which have helped many engineers understand the underlying complexity of USB by leading them through a series of practical examples.

I am delighted that John has undertaken to author a new book, Embedded USB Design By Example, at our behest for those of us who would like to incorporate USB interfacing into their product designs whilst focussing on overall product development concepts rather than having to learn the intricacies of USB hardware and driver development. Written in John’s unique style, this book is intended as a supplement to the existing data sheets and application notes on our FTDI web site.

Future Technology Devices International Limited, aka FTDI, is a well known semiconductor supplier in the USB “legacy” device field. Our FT232, FT245 and Hi speed dual and quad device series of USB peripheral devices offer a seamless route for easy USB interfacing through proven, well understood serial and parallel interfaces. Coupled with a commitment to providing royalty free, multi-platform USB drivers developed in house to ensure quality and consistency, our USB interface solutions can dramatically improve time to market for USB product designs eliminating ongoing support costs in driver development.

FTDI’s Vinculum Host/Peripheral controller range offers the same approach for embedded products that require USB Host capability. They bring similar ease of design and development to your USB host designs in the same way that our current products have eased your USB device products.

For further details of FTDI’s USB solutions, please visit our website www.ftdichip.com .

Acknowledgements:

I was warmly welcomed to FTDI's head office in Glasgow, UK, where a large group of enthusiastic engineers provided me more information than I could ever have asked for. I would like to particularly thank Fred and Cathy Dart for their hospitality and Ian Dunn for organizing the presentations, training, interviews, and material reviews. While every-ones contributions are valued I alone am responsible for any errors - I welcome your feedback (to John@usb-by-example.com) so that this book may be improved in future revisions.

I must thank my family, Lorraine, BJ and CJ who all contributed to the creation of this book – I appreciate all of their hard work, support and encouragement.

I have been involved with USB since its invention and I applaud FTDI's efforts to make USB easy - I trust that with the help of this project-orientated book and FTDI's components that even more people will be able to benefit from the worlds most popular bus, USB.

Table of Contents:

Please register your book at Designbyexamplepart2@ftdichip.com so that updates may be sent when available.

List of Figures:

Chapter 1 Introduction and Essential USB theory

Our electronics industry uses the term “embedded” to describe a non-reprogrammable, or fixed function, piece of equipment or device. This book also uses this definition but with an added, more literal, meaning. Most dictionaries define embedded as “enclosed firmly in a surrounding mass” and this is the approach that I will be taking with “Embedded USB.” Yes, the designs will have USB inside but this is not their main focus. Most USB books describe USB as a technical wonder (which it is) then flood the reader with an overwhelming amount of detail. I am not going to do that, so this book will NOT make you a USB technical guru. What it will do however, is describe USB as a tool that you can use to solve a wide array of problems.

I assume that you, the reader, have a basic understanding of electronics but that this is not your primary job function. You are tasked with building an industry-specific device that is not available off-the-shelf (else you would have purchased it and carried on with your real job!). This industry-specific device must interface to a PC (defined in this book as a personal computer running a USB-aware operating system such as Windows, OS X or Linux) and must therefore use an available PC IO connection. Or you have identified a very useful and cost effective PC peripheral, such as a joystick or flash drive, which you need to connect to your equipment. In both cases the connection standard is USB. THIS is what this book is about – how can you best utilize this USB connection to solve your specific problem. This book is example based and is divided into two parts – the first includes a wide range of example solutions that connect to a PC and the second part describes a wider range of example solutions that control USB-based PC peripheral devices.

I toyed with the idea of calling this book “USB for the rest of us” in deference to Apple’s campaign around their introduction of the Macintosh computer. For those of you who don’t remember the revolution Apple caused in 1984, they positioned the existing Wintel PC as difficult to use since you needed to know how it worked to be a successful user. Apple explained how you could be immediately productive with a Macintosh since its complexity was hidden behind an easy-to-use human interface that used a mouse and graphical display. My goal is similar – I want to show you that you can **use** USB **without knowing** its intricate details.

In this introductory chapter I review the facets of USB that you need to know to be successful. There have been several books and numerous papers written that describe the intricate details of USB, but, to be frank, you don't need to know most of this information to be able to use USB successfully. In the olden days, when USB was first introduced, you had to know these details since the available silicon components that you would use to implement a USB device were quite primitive, but today almost all of the complexity of USB has been integrated into fifth generation silicon devices that are straight forward to use. In fact, we will implement all of the examples in part 1 of this book without having to refer to the USB specification or other USB-specific documentation. We will use the skills you already have, such as interfacing with simpler serial buses (RS232, I2C, SPI, etc), and with parallel buses (FIFOs, 8051 MCU etc) to create a variety of USB-based solutions.

USB History

But first, a little history. It is important to know how we got here since this will enable us to move forward with more confidence. USB was invented in the mid 1990s to solve a specific problem – desktop PC peripheral device expansion. At this time the Wintel PC industry was stalled; Intel was producing microprocessors with ever-increasing performance but this could not be delivered to the peripheral devices; everyone wanted to use the Wintel PC as the computing engine to drive their custom peripheral device since this was cost-effective, but IO expansion in those days meant unique boards or connectors and custom device drivers. It was projected that there would not be enough software engineers available on the planet to support this expanding and diverging software need. Yes, "plug-and-play" had started to take hold but the existing PC infra structure of parallel ports, serial ports, EISA and PCI buses could not support emerging telephony and video-based applications. Something fundamentally different was required.

USB Architecture

The first USB design decision was to assign another microprocessor to handle the increasing IO load – this USB host controller would manage all of the low-level interactions of the peripheral devices thus freeing up the main CPU to process user applications data. USB would be a master-slave bus with a single master, the USB host controller, and multiple slaves, the IO devices. Most of the communications complexity would be implemented in the host controller, since there was only one, and this would allow the IO devices to be simpler and therefore lower cost. It was decided that

the USB host controller would have a 1ms scheduling period and that data transfers could be synchronized to this period – this would enable time-based data (audio and video for example) to be supported. The host CPU would generate lists of data transfers for each upcoming 1 ms time interval and the USB host controller would implement the data transfers on the host CPUs behalf. Once the host controller specification was agreed it was implemented as a fixed-function ASIC. This functional partitioning and standardization of IO functions prompted a new device driver model that enabled the low level USB data transfer mechanisms to be the same across a wide variety of peripheral devices – the diverging device driver problem had been contained!

As USB evolved so did the USB Host Controller specification. There are now three specifications (UHCI, OHCI, and EHCI) and there will soon be a fourth (XHCI). All are well defined with specifications downloadable from the web and all have been implemented in silicon. Each has proven and, in the case of Wintel PCs, WHQL certified OS device drivers. But the USB development team did not stop there – to ensure that the U in USB really meant ***Universal*** they divided the diversity of known and upcoming USB devices into CLASSes and then defined a set of standardized class interfaces above the standardized host controller interface. Microsoft, Apple, the Linux community and several silicon vendors then went about implementing a wide breadth of standardized drivers. The benefit to the IO device developers is enormous – if they implemented the interfaces on their devices to match the USB class specifications then they would operate immediately with all operating systems that implemented the class driver. These standardized implementations mean that a keyboard, modem, flash drive, printer, etc can be moved around different platforms and will continue to perform as designed. Also, since all communications is protocol based it will be simple to swap out the hardware device with something faster, cheaper, or more capable. Software did not have to be redone so the large investment in applications software could be preserved.

I have taught USB to many people and a great number get hung up on a key diagram from the USB specification – Figure 5.9 reproduced (with permission) as my Figure 1.1. It is essential that you understand this figure so let's study it for a moment since it unlocks much of the insight required to conquer USB and use it as a tool.

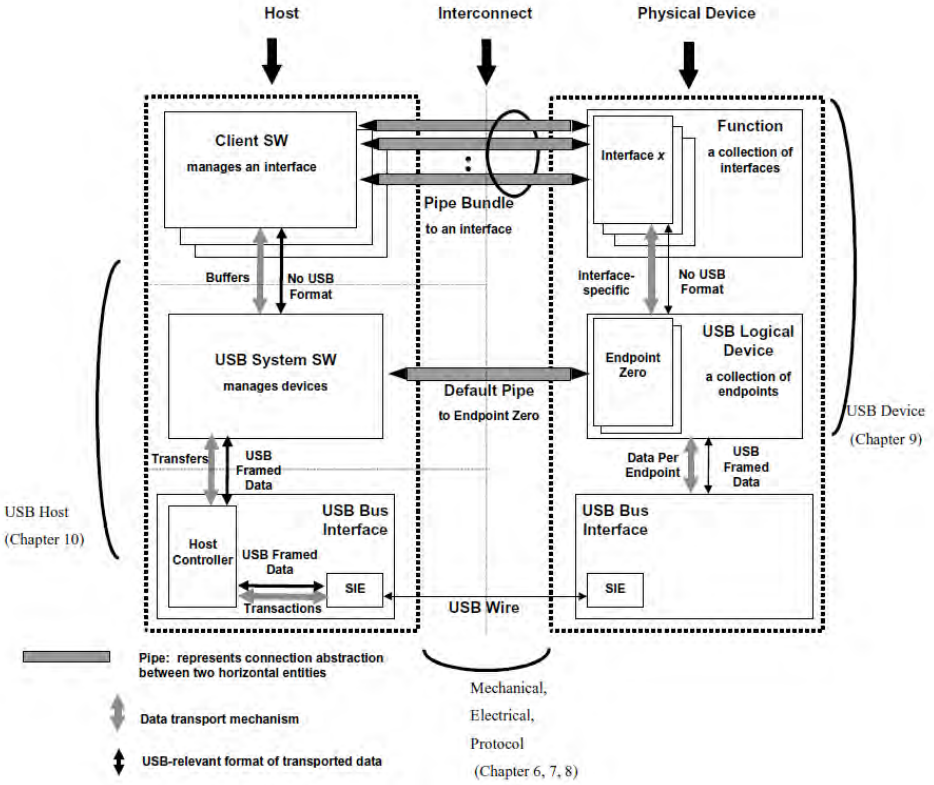


Figure 1.1: USB structure from USB specification[Ref 1]

Figure 1.1 shows two dotted boxes, ***Host*** and ***Physical Device***, interconnected with a ***USB wire***. It is important to realize that the Host (typically a PC) contains two or more CPUs and the Physical Device contains one or more CPUs – these CPUs can be reprogrammable (like the x86 Intel Processor in today's PC) or fixed function implemented as a ASIC (like the USB Host Controller) but note that they are smart. And we have a smart USB interconnecting this smart multi-CPU environment. It all looks a little daunting but, fear not, you do not need to know exactly how this all works to be successful using USB.

Figure 1.1 is a run-time diagram – it assumes that the connection between the Host and the Physical Device has already been set up (this is described later in this chapter). Data transfer starts with the Host so, as an example, let's send an ASCII string

"Hello World" from the Host to the Physical Device. We inject this 11-byte array into the ***Client SW*** block using a system call such as WriteFile( ). The Client SW may break our data into multiple buffers as it sends it to the ***USB System SW*** block. Note that this USB System SW block is receiving data transfer requests from many instantiations of Client SW blocks within different applications running on the PC. USB is a shared media but each application program treats it as a personal data connection; it is the USB System SW that manages the multiple data transfer requests from all of the Client SW blocks and it constructs a table of the ***Transfers*** necessary to service all of the requests. The USB System SW calculates the needed data transfers using a 1ms-scheduling period. The PC's x86 processor then passes this list of ***USB Framed Data*** to the USB Host Controller.

The USB Host Controller manages the low-level signaling on the ***USB wire***. It embeds our "Hello World" user data into one or more ***Transactions*** using asynchronous packets, which also includes SYNC data, device addressing data and error checking data. The ***Serial Interface Engine*** (SIE in Figure 1.1) handles automatic error retries which results in reliable data transfer between the Host and the Physical Device. The USB Interface on the Physical Device monitors all traffic on the USB wire and if it detects a packet with its assigned address then it absorbs and checks the packet and passes validated packets up to the ***USB Logical Device***. The USB Logical Device will pass user data packets up to the ***Function Block*** and our "Hello World" data will appear at the top of our Physical device.

Now focus on the horizontal bars called ***Pipe Bundle*** in Figure 1.1. The Host pushed the "Hello World" data into the top of the Host stack and it popped out of the top of the Physical Device stack. *It appeared to travel through the horizontal Pipe Bundle*. In reality it went all down the Host stack, across the USB wire and all the way up the Physical Device stack but we need not be concerned about this. The lowest level of Figure 1.1 (USB Interface, SIE and Host Controller) is fully defined by the USB specification and is implemented in fixed-function silicon. The center layer is also fully defined by the USB specification and is implemented in software, firmware or hardware (the Default Pipe is used for Link Management and is described later). The upper level is also fully defined by the USB specification and therefore, like the other layers, you have no flexibility to change it. It is interesting to know how this CPU-to-CPU communications is implemented but this knowledge is not necessary to use USB – if you accept that data effectively moves from a buffer in the Host system into a buffer in the Physical Device system (and visa

versa) then the key questions are; what is the latency, and what is the data throughput. We will address these questions in the examples chapters.

You could ask “but how do I differentiate my product within this standardized market place?” If you have the time and funds you can implement using “vendor defined” interfaces which are included within the USB specification as an option. But while you are moving along this difficult and time-consuming path don’t be surprised if your competitor introduces a similar product using a collection of standard drivers and captures most of the available customer base. I am a STRONG advocate of OS-supplied and vendor-supplied drivers and always recommend that everyone use this route. I maintain that you need an extremely compelling reason to embark on writing your own device driver; most people don’t.

The second major design decision made by the USB creators was the interconnection scheme. For ease of implementation and lowest cost, a 4-wire, serial, point-to-point connection, as shown in Figure 1.2 was chosen. A USB cable has an ‘A’ end (upstream connector, towards the host) and a ‘B’ end (downstream connector, towards the device). The ‘A’ connector included a +5V power source that could be used by a peripheral device and this could eliminate the need of many devices to include their own power (from a wall wart for example). There are rules to the amount of power that can be supplied and these are discussed later. The two signal wires are a half-duplex, differential pair that are generally driven by the host controller – the direction is switched when the host needs to read from a device.

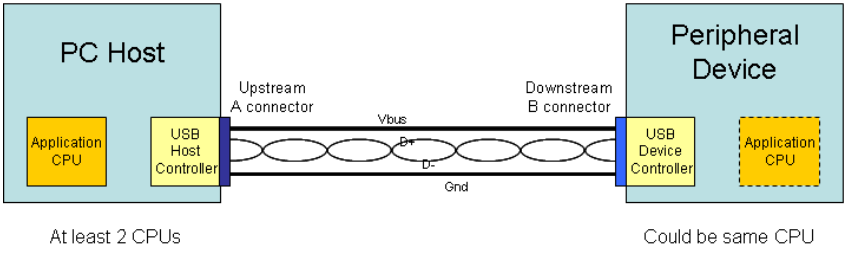


Figure 1.2: USB is a 4-wire, serial, point-to-point connection

Three standard signaling speeds are defined; low at 1.5Mb/sec, full at 12Mb/sec and high at 480Mb/s. There is a fourth option, SuperSpeed at ~5Gb/s, currently being developed by the USB Implementers Forum (USB IF). Information is transferred using asynchronous packets and these are combined with base protocols to implement four types of data transaction: control, interrupt, bulk, and isochronous. The USB specification also includes error checking and recovery mechanisms such that USB provides reliable data transfer – better still, this has been implemented in silicon by a variety of vendors so there is little reason to know every nuance. The USB IF has Compliance and Compatibility tests that silicon vendors must pass and this guarantees that the components we buy adhere to the USB specification.

Figure 1.2 shows a single USB Link connecting a USB host controller to a USB device. The host is required to support all three-link speeds (note that USB 1.1 compliant hosts will only support low and full speeds) and it is the device that selects the link speed. The USB specification includes link management commands that allow the host to interrogate the device to discover its identity and its capabilities. When the device is first connected, the host sends control transactions to the device to read pre-defined data blocks called ***Descriptors***, an example of which is shown in Figure 1.3.

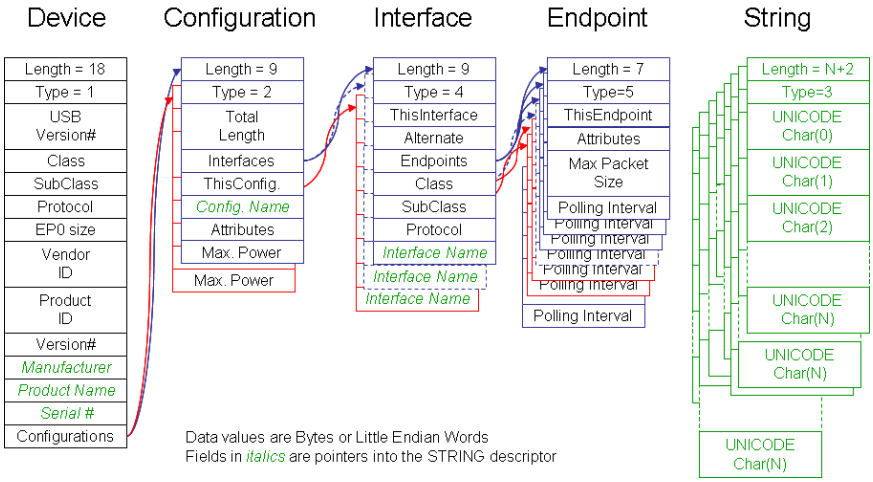


Figure 1.3: Descriptors are fixed-format blocks of data

The host operating system uses this descriptor information to determine which device driver should be used for each specific device and to assign a unique address to each attaching USB device. This process is called Enumeration and the requirement that each USB device is self-identifying is a major contributor to the plug-and-play and ease-of-use of USB. Most devices today implement the enumeration process in silicon or in canned firmware. So, once again, there is little need to understand every detail.

Importance of a USB hub

An integral part of the USB specification is a special device called a hub – this provides several bi-directional data repeaters and power injection as shown in Figure 1.4. This figure shows a USB 1.1 full/low speed hub since this is easier to explain (I cover a USB 2.0 high/full/low speed hub next). The hub contains a fixed-function USB device and the descriptors of a typical hub are shown in Figure 1.5. When this device is first attached to the host the operating system enumerates it and discovers that it is a hub – it therefore loads a hub device driver. This hub device driver manages the downstream connections. It applies power to each downstream port in turn and checks to see if a device is attached – an attached device will change the DC state of the data lines. If a device is detected then the hub device connects the downstream port to the upstream port and the host enumerates this new device – from this stage onwards the new device does not know that it is connected to the host via a hub, this is a transparent connection (yes, there is a small propagation delay through the hub but this is allowed for in the spec). The new device operates as if it were directly connected to the host. If the new device was another hub then the process would repeat – the USB specification allows for hubs up to five deep, which gives a lot of connectivity.

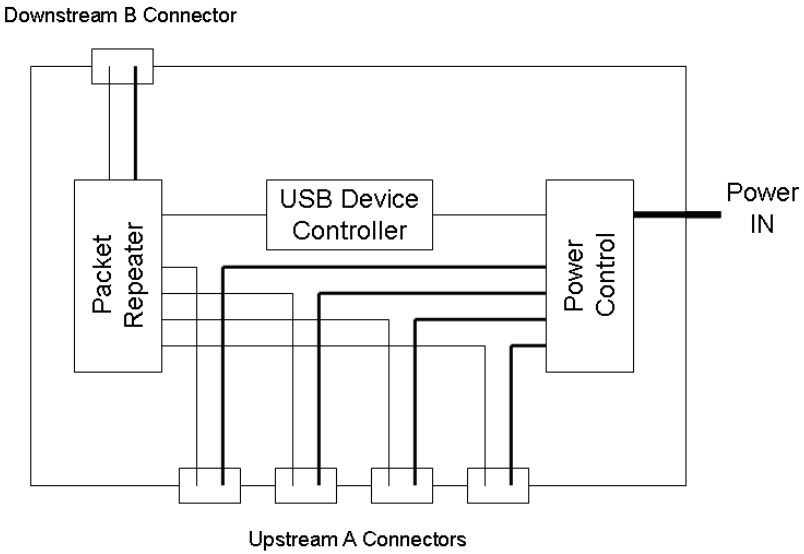


Figure 1.4: A USB hub provides connectivity

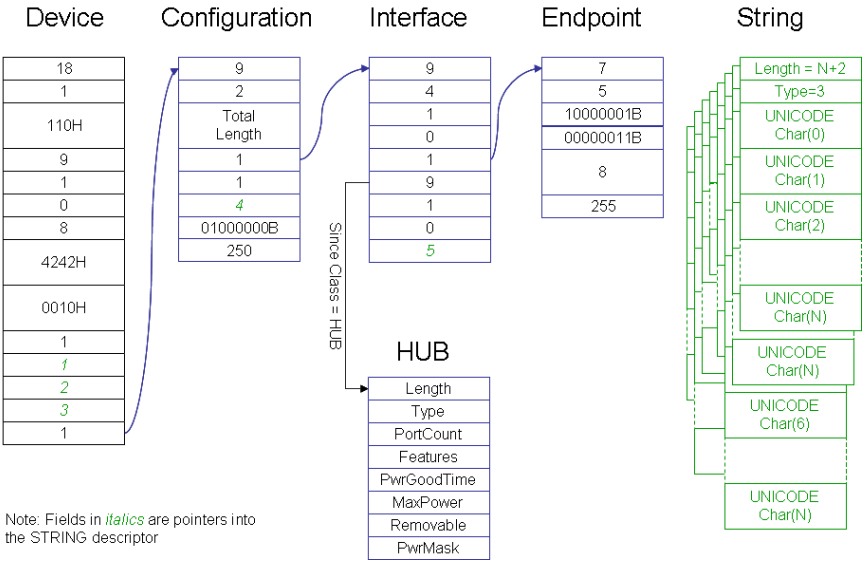


Figure 1.5: Descriptors for a basic hub

The hub also allows for the injection of power into downstream ports. The USB specification details several power levels; when first connected a device can draw up to 100mA from the upstream connection; during enumeration the device can request up to 500mA (if your device needs more then 500mA then it will need its own power source); when suspended, or not operating, a device must limit its power drain to less than 2.5mA – a PC may suspend itself and power down when not in use, and there is no point in having peripheral devices powered up when the PC is off. So the USB host controller will suspend all attached devices prior to powering down.

The basic functionality of a USB 2.0 hub, as shown in Figure 1.6, is the same as a USB 1.1 hub. Additional circuitry is included that enables more efficient use of the USB Links. A high-speed link always runs at high speed – if a low or full speed device is connected to a high-speed hubs downstream port then data transfers are "stored-and-forwarded." The data packets are sent at high speed from the host to a ***Transaction Translator*** (TT in Figure 1.6), which will then send the packet at low or full speed to the device. Similarly responses are collected at low or full speed by the TT and forwarded to the host at high speed. These operations require additional link management commands (Start Split etc.) and these are implemented by the EHCI hub driver at the host. There is no additional programming at the PC application layer nor at the device so these operations are transparent to the device and to the user.

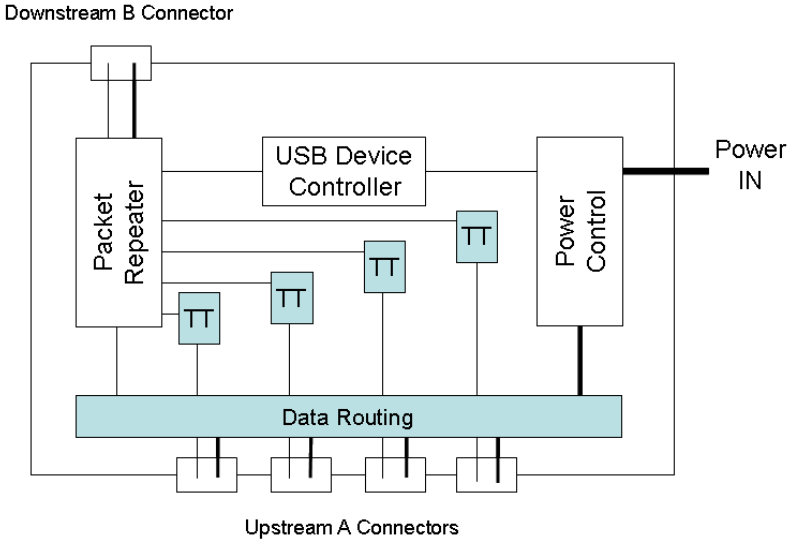


Figure 1.6: A High-Speed hub includes Transaction Translators

Figure 1.7 shows a typical PC system with a variety of devices attached via hubs. Each EHCI host controller will support a 480Mb/sec link and this bandwidth is shared by all of the devices connected via this link. Each OHCI/UHCI host controller will support a 12Mb/sec link and again this bandwidth is shared by downstream devices. A PC typically has multiple host controllers; the laptop I am using at the moment has an Intel ICH9 controller which includes 6 UHCI controllers and 2 EHCI controllers. The ICH9 also has on chip routing and the operating system will assign a UHCI controller to manage low/full speed devices and it reserves the EHCI controller connections for high speed devices. If the OS cannot route an EHCI controller to the port where a high speed device is connected it will prompt the user to move the device and plug it in elsewhere. Therefore this particular laptop can support up to 6*12+2*480=1Gb/sec of USB bandwidth.

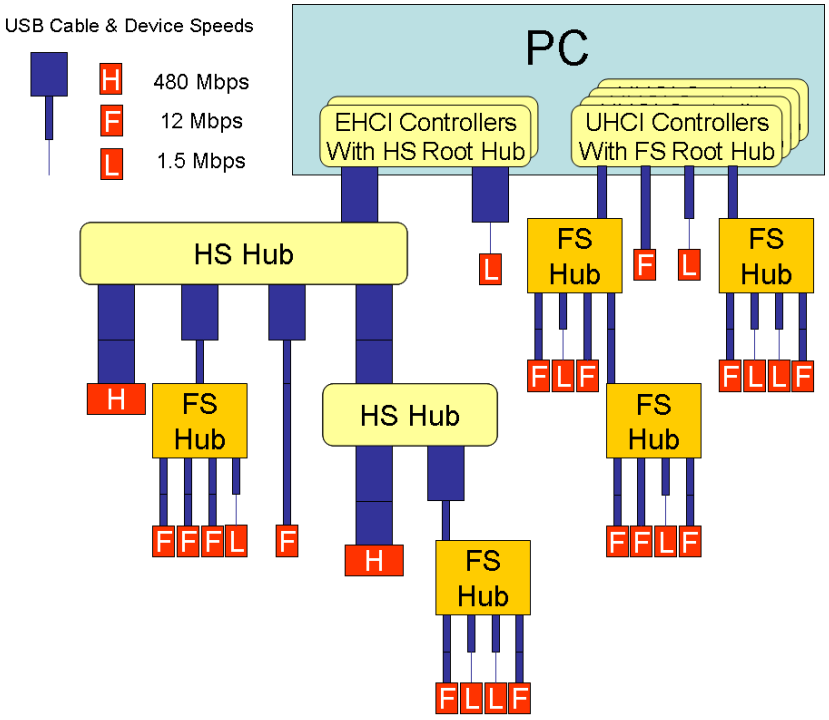


Figure 1.7: Typical PC with several hubs

Chapter Summary

In summary, USB is a shared communications media where multiple host controllers can be used to supply a desired IO bandwidth, and multiple hubs can be used to distribute this bandwidth to a diverse array of peripheral devices. All communications is standards-based and is implemented as a collection of proven class- and host controller drivers. The lower levels of this communication are implemented in fixed-function silicon. Since most of USB is standardized (and therefore cannot be changed) and most products are certified to be compliant to the USB specification then we can trust that USB works and focus our efforts on using USB to implement useful products.

So enough theory, lets implement something!!!

Part 1 focuses on designing IO devices that can be attached to a PC. I created a common source code for the Windows and Mac platforms, and I expect Linux users will be able to use the Mac OS X code with little or no modifications. I had to put an OS-specific #DEFINE to accommodate differences in library and some function names but, fundamentally, the **SAME** example code is running on all platforms. This is possible since FTDI provide their device drivers on all three platforms. I will focus on functionality and ease of understanding and not on the human interface so the code will be fundamental and written in portable C++. A set of PCBs is available (see Appendix B) to simplify working through the examples but if you don't have these then most of the examples can also be built up on a solder-less breadboard.

Ref 1: USB 2.0 Specification © 2000 Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. A free download is available from www.usb.org/developer.

Chapter 2 – A starter USB project

Let's start simple; you want to connect a single push button to a PC. On recognizing the button press, a program running on the PC initiates a series of actions one of which lights an LED adjacent to the push button (in case the PC is remote or does not have a screen, the LED provides feedback to the user that the button press has been recognized). This project could form the basis of an embedded kiosk, machine operation, sequence control, security monitoring or a range of other man-computer interactions. This project used to be easy when the PC had a parallel port but all you see now are USB ports! But you don't have time to learn USB, so you look online to buy a USB-to-ButtonAndLight adaptor that you can just use. Nobody sells one. HELP!!!

Fortunately FTDI sells and supports a USB-to-4BitIOport cable that can be used to solve this problem. FTDI don't call it that (they call it a TTL-232R) but that is how we shall use it and it is shown in Figure 2.1. It looks like a standard USB cable until you look closely at the non-USB end – there are six wires instead of the expected four. Two are power (+5V) and ground and the other four are TTL signals that can be configured as inputs or outputs. There is a fixed-function USB device molded into the plug but more of this later.

Figure 2.1: The TTL-232R is a USB-to-4BitIO port cable

I'll discuss HOW this works in a few pages time but, for now, let's WATCH it work. We learnt from Chapter 1 that all devices need a device driver – so download a driver for your operating system (OS) from www.ftdichip.com/FTDrivers.htm and load it on your PC; refer to Appendix A which includes instructions of how to do this for each supported operating system. FTDI has two sets of drivers and for the first few examples we need the D2XX driver so use that if your OS only allows a single driver for the FTDI device. Now plug the USB-end of the cable in to the PC; the OS may indicate that a new device is being added and it will match it with the device driver loaded in the previous step and this will be installed so that the OS can use it.

Figure 2.2 shows a schematic of our first example and shows this circuitry mounted on the first PCB. The button is connected on Bit3 which is pulled high by a 200K Ohm resistor inside the cable; this bit will therefore be read as a high unless the button is pressed when it will read as a low. The LED is connected on Bit2 and will be lit when this bit is driven high and will be off when this bit is driven low. Note that the resistors shown with dotted lines are included within the cable and Bit0 and Bit1 are not used in this example.

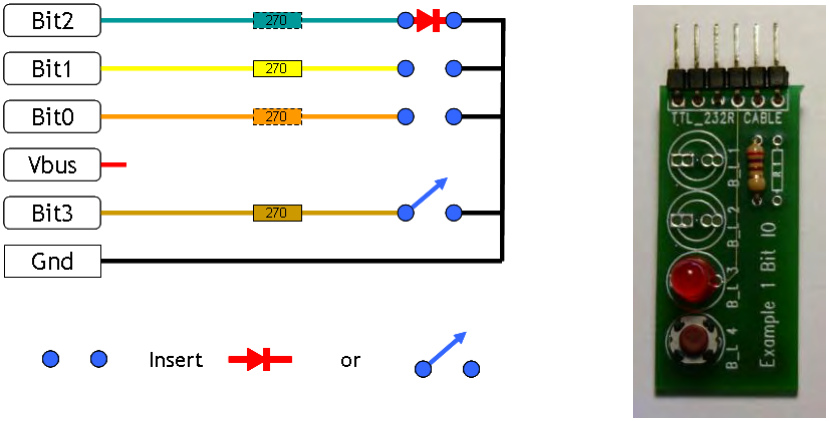


Figure 2.2: The first example, schematic and hardware

Figure 2.3 shows an edited version of the source code of our first example - I removed the error-checking for clarity but this is included in the supplied example1.cpp. Let me first explain the three helper routines, ***InitializeForBitIO***, ***WriteBits***, and ***ReadBits***, that will allow the main loop to be written more simply.

The OS enumerated the FTDI component in the cable when it was attached and it has been added to the OS-internal plug-and-play tables. The ***FT_ListDevices*** system call queries these plug-and play tables and returns a ***DeviceCount*** of matching FTDI devices which are currently attached - my code assumes that we only have one of these cables attached. The ***FT_Open*** system call gets a handle for this device that can be used in later system calls. The ***FT_SetBitMode*** selects which pins are input, which are output and sets operation to Synchronous Bit Bang mode. The ***WriteBits*** routine is an ***FT_Write*** of one byte to our device and the ***ReadBits*** routine is a similar ***FT_Read.*** ReadBits returns the inverted value of the pins since a button press is active low.

```
BOOL InitializeForBitIO(void) {
    FT_CreateDeviceInfoList(&DeviceCount);
    if (!DeviceCount) return printf("No FTDI devices\n");
    FT_Open(0, &FT_Handle);
    FT_SetBaudRate(FT_Handle, 921600);
    FT_SetBitMode(FT_Handle, 4, SyncBitBang);
    return 0;
    }

UCHAR ReadBits(void) {
    UCHAR Value;
    DWORD BytesRead;
    FT_Status = FT_Read(FT_Handle, &Value, 1, &BytesRead);
    return ~Value;
    }

void WriteBits(UCHAR Value) {
    DWORD Written;
    FT_Status = FT_Write(FT_Handle, &Value, 1, &Written);
    }

int main(int argc, char* argv[]) {
    if (InitializeForBitIO() == 0) {
        while (1) {
            WriteBits(LED_Off);
            Idle(100);
            while (ReadBits() & Button) {
                WriteBits(LED_On);
                Idle(100);
                }
            }
        }
    FT_Close(FT_Handle);
    return 0;
    }
```

Figure 2.3: Edited source code of first example

The main loop polls the button every 100 ms and, if it is pressed, it will turn on the LED. The main loop will run until a Control+C is entered on the PC keyboard.

Let's run the program and watch it work.

Pause here while you run the program.

Now marvel at its simplicity.

So adding a push button and LED to the PC using USB wasn't difficult after all. If you don't like the gauge or the length of the cable you can just purchase the "plug + electronics" and add your own cable and case. The electronics supports 4 IO lines and these can be any combination of buttons and LEDs. Notice that you didn't see any descriptors or had to deal with any USB-ness at all.

How it works

The heart of the electronics, embedded within the plug of the FTDI cable, is an FT232R which is a self-contained, USB-ByteMover device. The block diagram, shown in Figure 2.4 shows the main elements of the FT232R.

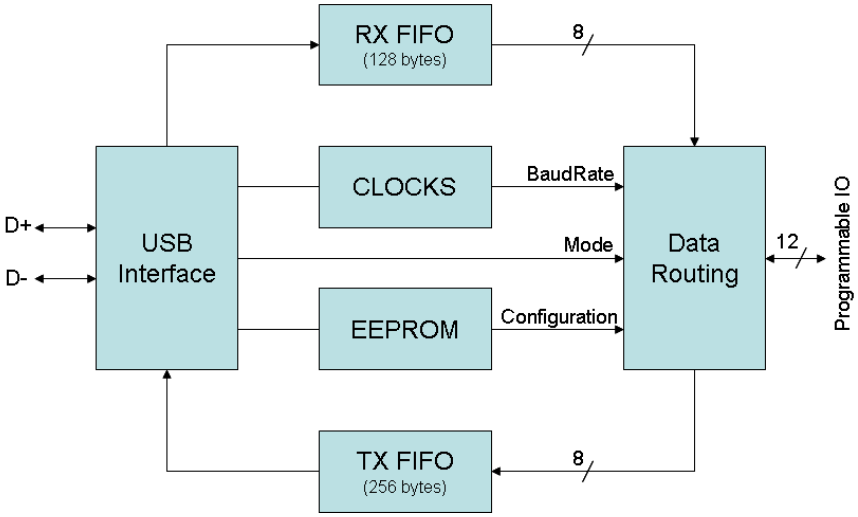


Figure 2.4: Block diagram of FT232R USB-ByteMover device

The ***USB Interface***, with the aid of data within the ***EEPROM***, enumerates with the PC host and selects FTDI's drivers. USB data packets are delivered to the ***RX FIFO*** and are then routed to the programmable IO pins using the EEPROM Configuration, Selected Mode and BaudRate generated from the ***CLOCK*** circuitry. The FT232R supports 3 data routing modes: Synchronous BitBang, Asynchronous BitBang and UART which supports RS232, RS422, and RS485 protocols with full modem control signals. The programmable IO pin block is expanded (twice!) and shown in Figure 2.5; each pin can be set as input or output, can be programmatically inverted and have higher drive current. An FTDI utility program, called FT_PROG (described in Appendix A) is used to set power-on parameters in the EEPROM. Similarly data can be routed from these programmable IO pins, including the UART protocols, to the ***TX FIFO*** where the FT232R collects this data, moves it to the PC and queues it ready for the FT_Read function. The FT232R handles all of the USB protocol on your behalf; bytes are moved between the PC application program and the programmable IO pins neatly and efficiently and the only reason to ponder about the actual data transfer is a concern about performance. In the examples in this chapter the performance bottleneck will be the IO speed at the programmable IO pins and this will be examined in later chapters. The performance limiter in this first example is the 10Hz human user – the USB operations are in the millisecond range and will be considered 'instantaneous' by the user.

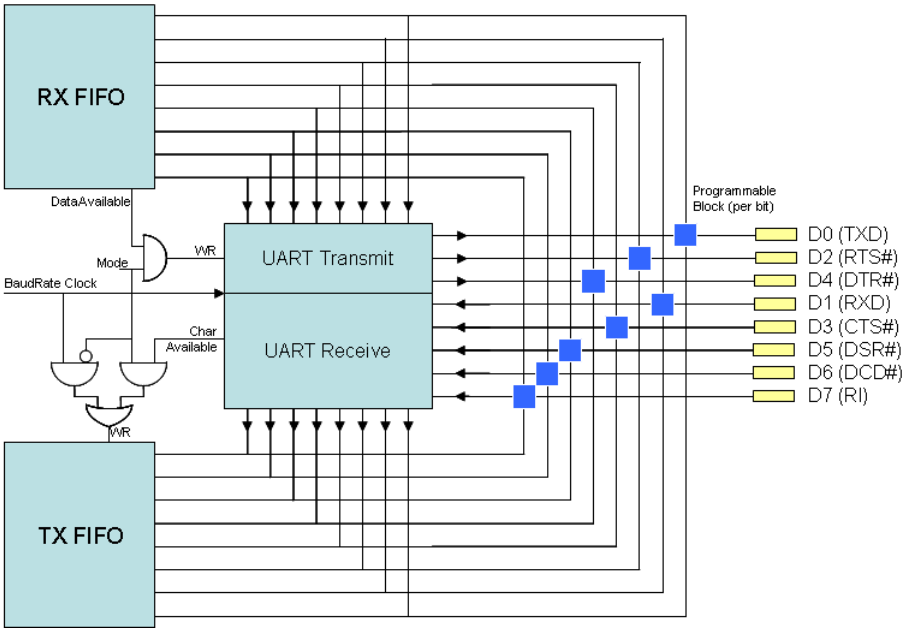


Figure 2.5A: Showing detail of ABUS data routing

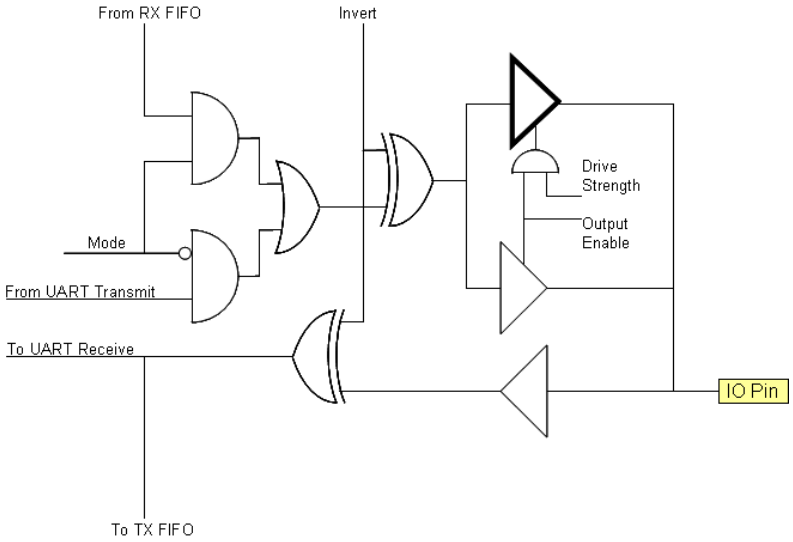


Figure 2.5B: Showing detail of programmable IO pins

Returning to the first example, we are using synchronous bit-bang mode so WriteBits copies the data byte into the RX FIFO which strobes data, at baudrate, directly to the IO pins. At the same time, the data on the IO pins is strobed into the TX FIFO and then delivered to the application program via ReadBits. Note that a WriteBits function is required before a ReadBits function can be used and note that they are paired to keep the read data in sync with the write data.

Getting more IO lines

Although the FT232R supports 12 programmable IO lines, only 4 are brought out on the TTL-232R cable. Our first example could be easily expanded to include any combination of up to 4 buttons and LEDs, and although it makes a great demo, it is a limited solution that can only solve a few problems. We need a solution that supports at least several bytes of IO. Figure 2.6 shows the schematic of a low cost component added to the non-USB end of the cable that will support up to 8 bytes of IO in any combination of inputs and outputs.

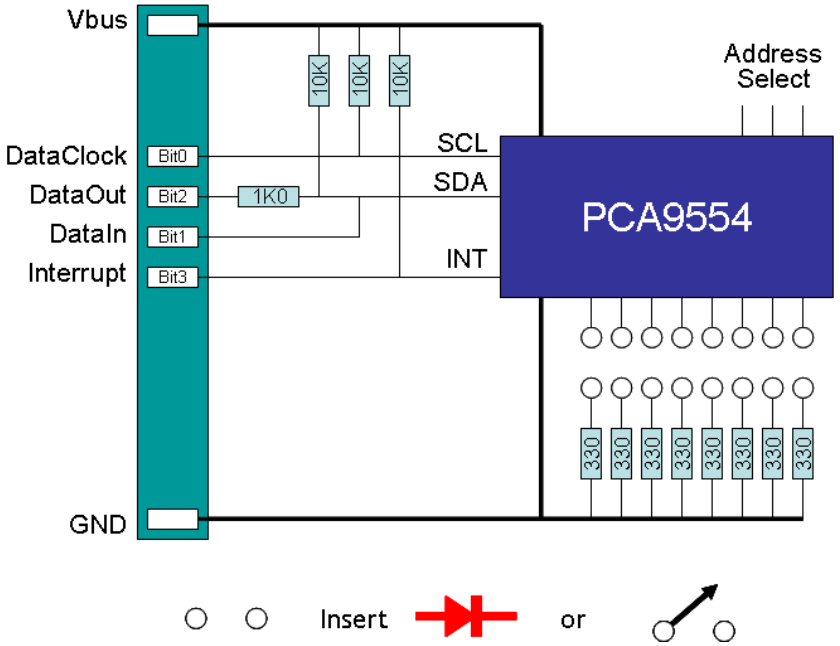
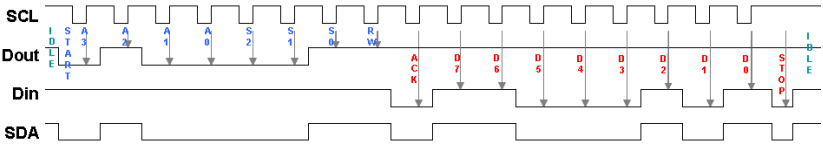


Figure 2.6: Adding an I2C IO expander to the cable

Rather than use the four programmable IO lines statically we are going to drive them with an I2C protocol and thus take advantage of the wide range of available I2C components. I chose I2C as an expansion bus since this is a multi-drop, 2-wire bus with a well-defined protocol that includes device addressing as well as data transport. This choice will allow me to easily expand the solution for later examples. I2C is a 2-wire bus but I need to use 3 connections from the cable since Output and Input functions are on two separate pins. One pin is used as ***DataClock*** or ***SCL***. I must keep my ***DataOut*** pin high when the I2C device is driving ***SDA*** low so that this can be detected by my ***DataIn*** pin.

Let's first consider the case where I have eight buttons connected to the PCA9554 component. Let's also assume that I have pre-selected register 0 using a command byte write such that an I2C read command will read the input pins. I have redrawn figure 10 from the PCA9554 data sheet as my Figure 2.7 to show the waveform that must be generated. In particular, I have separated out the wired-OR, I2C SDA line into my DataOut and DataIn lines so that it is easier to see who is driving this shared line.



Blue: FT232R driving SDA, Ax = PCA9554 Family Address, Sx = Sub Address
Red: PCA9554 driving SDA, Dx = Input data = 0xC5

Figure 2.7: Waveform needed to read an I2C byte

As you can see, each I2C bit transition needs three byte writes so, with an eight bit command, one bit ACK, eight bit data read, and a one bit STOP this will result in 54 bytes that need to be written to the FT232R. Example2 calculates this byte stream at run time and sends it to the RX FIFO where it is clocked out at the selected baudrate. The PCA9554 can operate at up to 400 KHz and the baudrate must be chosen to meet the minimum timings of their part. The limiter is a clock low time of 1.3µs which means a baud rate divisor of 4, or 5 with some margin.

The PCA9554 sub address is also calculated at run time and this three bit address field allows up to eight of these components to be used. This results in an easy expansion up to eight bytes of digital IO. Figure 2.8 shows example 1 built on a solder-less breadboard. There are many products available but I use the range from Elenco Precision since they are built from modules that may be reconfigured to give a better layout area. Figure 2.8 shows two PCA9554's, one with buttons implemented with a DIL switch and one with LEDs implemented with an LED bar graph.

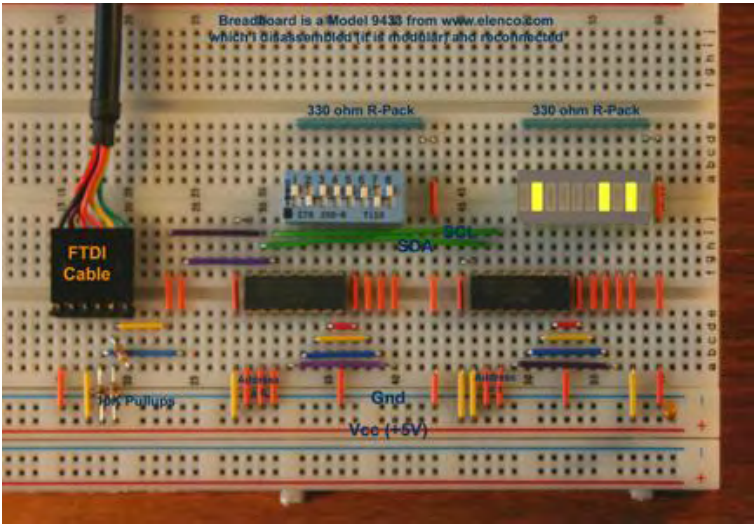


Figure 2.8: 2-way I2C bus expansion using PCA9554

Figure 2.9 shows a hardware variant of example 1. The Microchip MCP23008 has the same capability as the PCA9554 (actually, it has a lot more, but my example does not use this) and a better pin-out for this example; it allows 4 ports in about the same area and I chose 4 large seven-segment displays. You could easily add 4x8 = 32 buttons to this example, enough for most control panels.

The breadth of I2C components also allows us to have analog in and analog out modules using the Analog Devices AD799X or AD53X1 for example. These boards could be used standalone or could be used in conjunction with the buttons and LED boards. There are also sophisticated ICs such as TV tuners, sound processors and a wide range of multi-media circuits available with an I2C control interface. So, with this cable and a few standard components I can simply access up to 64 bits of digital IO and several analog channels -

enough for many control panels, system configuration, or even a distributed data gathering system. We are using PC software to bit-bang an I2C interface and this is implemented in a modular, expandable program called Example2.cpp.

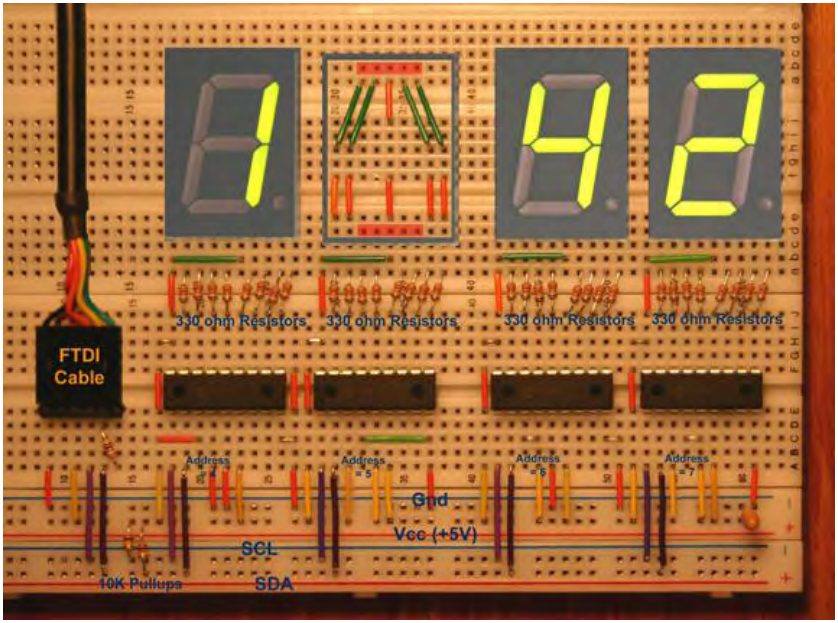


Figure 2.9: 4-way I2C bus expansion using MCP23008

Chapter Summary

This chapter has shown that is it easy to attach simple IO to a PC using USB. We used an FTDI TTL-232R cable to first drive discrete buttons and LEDs then, with the help of some low cost I2C components, we connected up both digital IO and analog IO to a PC. The USB cable also supplied the power source for our components. I have this up and running and I didn't have to even open the USB spec! All the USB-ness is handled by the FTDI device and the device driver, allowing me to concentrate on my application.

Chapter 3 - Serial and parallel device conversion

The embedded world still uses serial ports and parallel ports because they are easy, especially when compared with USB! A serial port uses just two data wires, TX and RX, for full duplex communications and a reference ground wire is also essential. Serial ports became popular with the introduction of modems in the 1977 and the RS232 standard also includes modem control signals such as DataSetReady and DataTerminalReady. The signaling levels are +/- 12 V and this tends to limit the maximum data rate to 56KBaud. Once both ends of the wire agree on the baud rate it is a simple matter to send and receive any stream of data bytes.

Unfortunately the simplicity of exchanging any stream of data bytes is also the serial ports Achilles heel. Most serial links also need to exchange some control information and this is embedded in the data stream using some kind of escape sequence. This, by itself, is not a bad technique - the issue is that there is no standard escape sequence which results in applications software being tied to a specific piece of hardware. Again, this is not a major problem except that there is no standard way for the application software to identify the attached hardware.

The real problem comes when you want to attach your serial device to a PC and you discover that there are no serial ports! PC hardware changes more rapidly than a typical embedded system, and if we are to take advantage of 'PC economics' then we need to follow their trend.

PC software has also changed dramatically. The first PCs introduced by IBM were well documented and all of their internal hardware was exposed via BIOS listings. You were actually encouraged to access the serial ports at 0x3F8 and 0x2F8. This all changed with the introduction of 'protected mode' Windows where applications software was prevented from accessing the physical hardware. The same is true today about Mac OS X and Linux. All three operating systems support multi-tasking and multi-applications so they must own the underlying PC hardware so that they can manage its use. The impact to the embedded developer is that the serial ports must be accessed via a device driver - in Windows this is COMxx, and in Mac OS X and Linux, this is /dev/tty. Once you encourage the OS to supply a *handle* to a serial port then you can read and write streams of data bytes from and to the serial port.

Representative Serial Device

So let's work through an example of converting a serial device to a USB device. My representative device is a serial display from www.robokits.co.in, as show in Figure 3.1. You can download a user manual from their web site but it is basically a 2 line by 16 character display that accepts ASCII characters through a 9600 baud serial connection. Non-displaying characters (0x00..0x1F and 0x80..0xFF) are interpreted by the on-board micro-controller to implement special functions such as the setup and display of custom characters and turning the backlight on and off. I have used this display on many embedded projects since it is low cost and only needs a single IO pin to drive the display. Adding it to an embedded PC would be much cheaper than a VGA display for those applications that only needed 2 lines by 16 characters or it could be a remote display in addition to the main display.

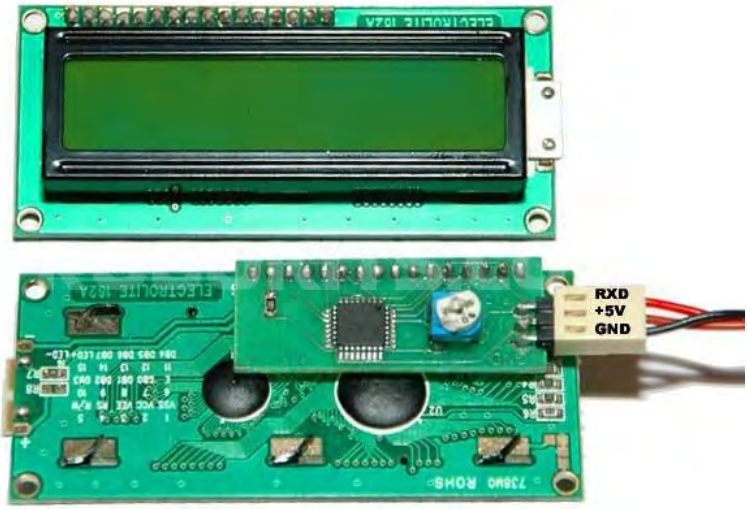


Figure 3.1: Representative Serial Device

Rather than create a custom example program for this chapter I thought it more convincing to use software for the PC that is already available and designed to support serial ports. For the Windows PC I shall use ***HyperTerminal*** and for the Mac/Linux PC I shall use ***CoolTerm*** (download from http://freeware.the-meiers.org/). Using the same TTL-232R cable introduced in chapter 2, first connect the non-USB end of the cable to the display as shown in Figure 3.2. We are using the cable to power the display and have TXD looped back to RXD.

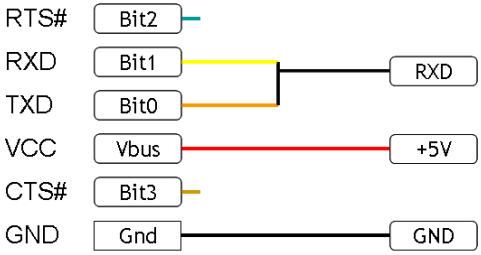


Figure 3.2: Connecting the FTDI cable to the display

Windows PC operation:

The FTDI drivers that we installed in chapter 2 includes two distinct interfaces, D2XX which we used to access low level functions and VCP, a Virtual Com Port interface. The windows driver supports both interfaces in a single installation, called CDM, but only one may be used at a time. If you have not installed this driver, do it now.

Insert the TTL-232R cable into your PC and then display the hardware configuration using the Device Manager in the system control panel. My configuration is shown in Figure 3.3. Note that the cable enumerated as a COM port - mine happened to be assigned as COM11 and yours will probably have a different number.

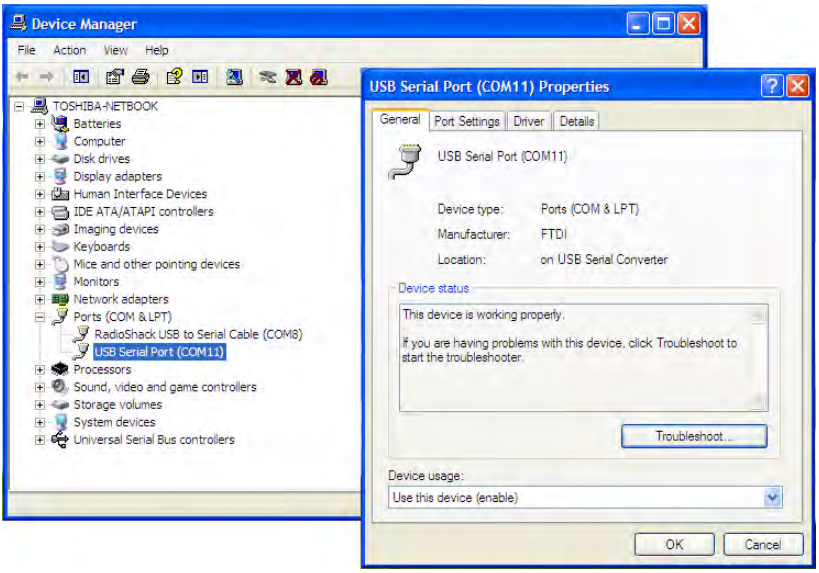


Figure 3.3: The cable is recognized by Windows as a COM port

Spin up HyperTerminal in the accessories directory and select the COM port that has been assigned to the TTL-232R cable. Now jump to the 'Configure the terminal program' section on the next page.

Mac Operation

Life is a little trickier for the Mac user since the FTDI drivers are not combined. We installed the D2XX driver in chapter 2 and it is now time to install the VCP driver. Decompress the FTDIUSBSerialDriver.dmg file that was downloaded from FTDI's web site and click on FTDIUSBSerialDriver package and follow the installation instructions.

Insert the TTL-232R cable into your PC and the OS will preferentially choose FTDI's VCP driver. Figure 3.4 shows the output of the System Profiler tool and, as seen, it lists the FTDI cable connected to USB.

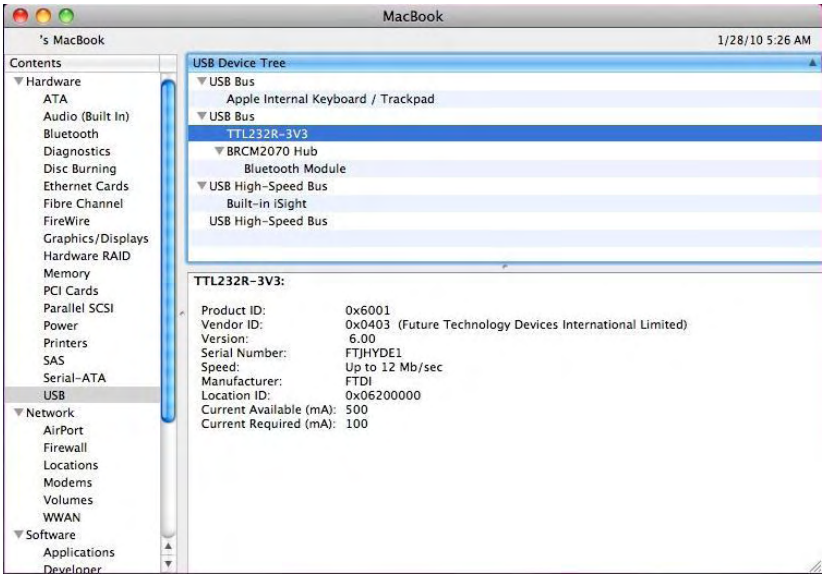


Figure 3.4: The cable is recognized by Mac OS X as a COM port

Now spin up CoolTerm and select the usbserial device.

Configure the terminal program

Choose 9600 baud and open a connection. Type "Hello World" then note that this also appears on the 2 line display. We're basically done! Yes, we are using USB but it is embedded. In fact, it is embedded so deeply that we haven't even been exposed to any USB at all. All of the USB aspects have been handled by the cable and by FTDI's VCP driver.

So conversion of a serial device to a USB device is almost trivial if we use this TTL-232R cable and driver from FTDI. All of the hard work is being done by the operating system and its drivers - they are handling the differences in hardware and we, at the application program level, need not be concerned about exactly how this is being accomplished.

Switching back to the D2XX driver.

The D2XX driver is always available to the Windows user so you may skip this section. The Mac OS X user can temporarily or permanently remove the VCP driver as follows; use the Terminal application and view the system extensions to identify the system name of the FTDI cable:

```
cd /System/Library/Extensions
ls
```

It will probably be `FTDIUSBSerialDriver.kext`. You can remove it for the current session with:

```
sudo kextunload FTDIUSBSerialDriver.kext
```

To permanently remove it (which will mean reinstalling the package if you wish to use the VCP driver again) use:

```
su
rm –R FTDIUSBSerialDriver.kext
```

Optimizing the serial connection

Now, before you rush out and make a volume purchase of this cable let us look at a few optimizing steps. My serial display is not typical in that it used TTL levels rather than RS232 voltage levels. The top of Figure 3.5 shows a more typical serial device. It has an internal microprocessor or microcontroller that drives an RS232 voltage converter for PC communications and drives custom IO specific to the embedded application.

In the center diagram of Figure 3.5 I have replaced the serial cable with the FTDI cable. This cable drives TTL levels so there is no need for the RS232 voltage converters. We have a problem with the connector however since the industry expects RS232 voltage levels on the 9 pin (or 25 pin) serial connector. I shall deal with this issue in a moment.

Now look at the third diagram in Figure 3.5 - I have moved the FT232R part from the “PC end” of the cable to the “device end” of the cable. This FT232R part replaces the RS232 voltage converter and I replace the serial port connector with a USB B connector (standard size or mini-B). This means that I use a standard USB cable to connect my new device to the PC. Another advantage of having the FT232R at the 'device-end' of the cable is that you have access to all 12 IO lines. We shall look at these in the next chapter.

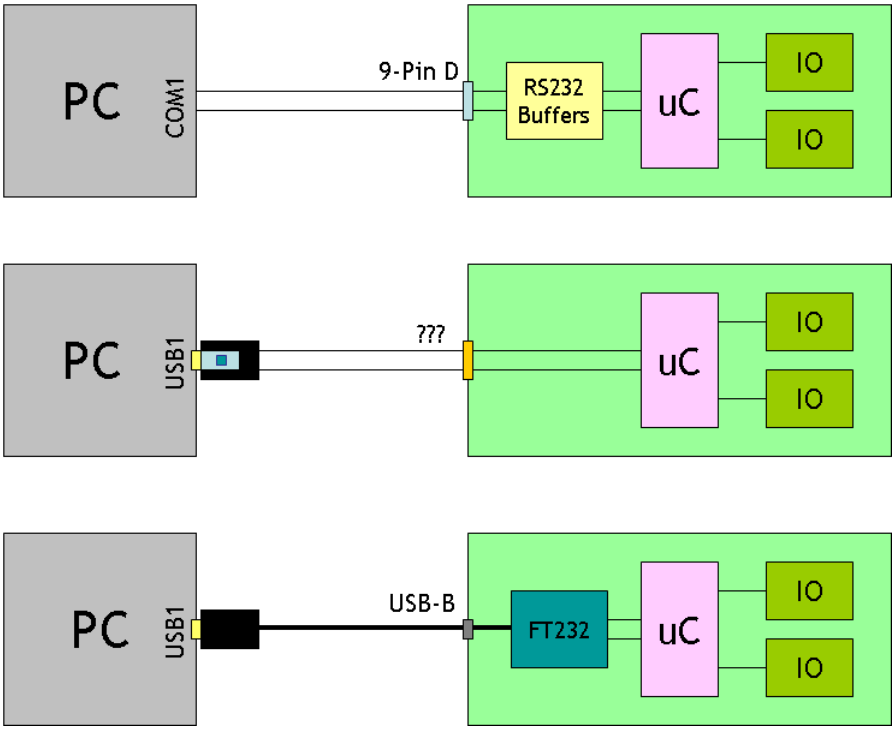


Figure 3.5: Converting a serial device

Total conversion effort is less than a day. We migrated a serial device into a USB device. But what else did we gain besides a product that is likely to sell better?

If needed, we could increase the baud rate to the device. Standard serial cables can easily support 56 Kbaud and some can do 192 Kbaud. The FT232R can run at 3,000 Kbaud due to the higher data transfer rate of USB. If your device moves a lot of data then this "upgrade" would be worth implementing.

A USB cable can also supply up to 500mA at 5V. If your device can operate at or below this power level then you could eliminate the power source from your device and thus reduce its manufacturing cost. And since you will charge more for a USB version then you get a double cost benefit as well as a simpler product. This is also "low hanging fruit" and is easy to implement.

Converting a parallel device to USB

FTDI have a trio of "USB-ByteMovers" that can be used in this type of application. So far we have been using the FT232R that has a serial interface. A companion part, the FT245R replaces this serial interface with a parallel bi-directional FIFO interface for higher data throughput rates. Converting a parallel interface device to a USB device follows the same methodology as the serial device. From an applications software perspective you still treat it as a serial port but otherwise the software is un-changed. You can also reap the higher speed, and USB-provided power, benefits of the serial port conversion example. A dual-channel part, the FT2232D, is also available: the two channels can be individually programmed to operate as an FT232R or an FT245R or they can be combined to produce higher capability interface.

Chapter Summary

In this chapter we used FTDI's VCP driver that presented the USB device as a COM port. While this is convenient as a transition strategy it does not address the serial ports Achilles heel of device identification. If you have multiple devices and chose the wrong COM port number then your application software will fail, just like it did in the olden days when using real COM ports. The VCP driver does such a good job of hiding the underlying hardware details that some necessary information for a multi-device system is also hidden. I would recommend using the VCP driver as an initial step but migrate to the D2XX driver in the longer term since it has more capabilities.

Each FTDI component has an integrated, or attached, EEPROM that includes a unique ID programmed during the manufacturing process. A custom Vendor ID (VID), Product ID (PID) and friendly name can also be programmed into this EEPROM (Using FT_PROG, described in Appendix A) and any combination of these parameters can be used to specify which particular device of a Multi-USB device system should be opened by an application program.

Now that we know how easy it is to have multiple devices the next chapter will look at more capable devices.

Chapter 4 - Connecting to more capable devices

Chapter 2 showed that the bit-banged IO pins of the FT232R could be used to create a 400KHz I2C expansion bus and enabled this component to solve a wider set of digital IO and analog IO problems. FTDI has taken this fundamental IO expansion concept a major step forward and integrated two FT232Rs with more capability and bigger FIFOs into a USB high speed FT2232H product, a block diagram of which is shown in Figure 4.1. Note that the FIFOs are 32 x bigger! The I2C bus generation, including a more efficient parallel to serial conversion, has been added as a mode called MPSSE (Multi-Protocol Serial Synchronous Engine which also supports SPI, JTAG and any custom protocol). Also added are an 8051-type bus emulation and a fast, opto-isolated serial protocol. Each of these two interfaces can independently run a synchronous protocol up to 30MHz or a serial protocol up to 12Mbaud. There are also digital IOs that can be bit-banged. Let's see how this improved part can solve a wider variety of design problems.

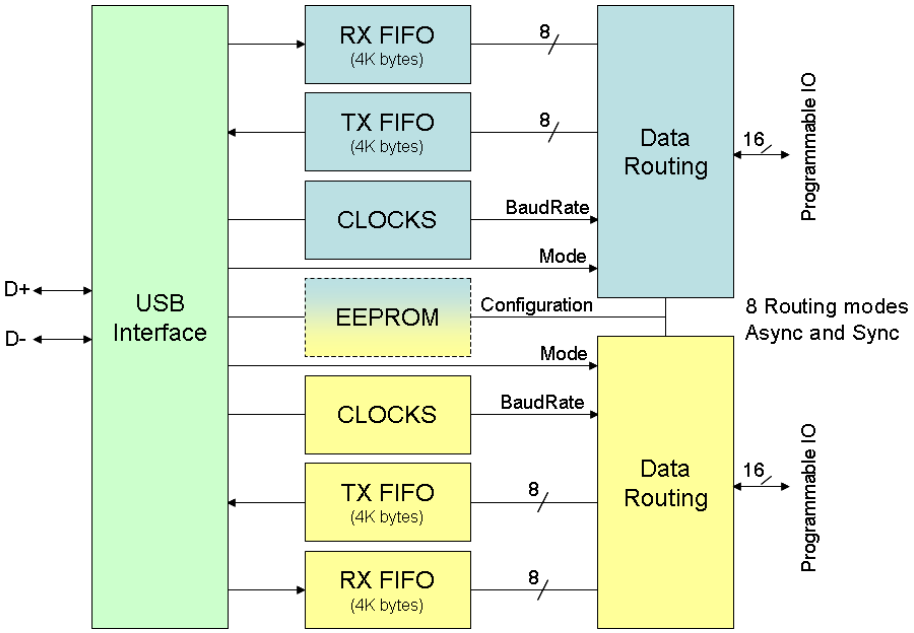


Figure 4.1: Block diagram of FT2232H

Data Collection Pod

Figure 4.2 shows a block diagram of a "Data Collection Pod." This pod is battery powered and is physically small and light to enable it to collect data from a wide range of sources. Once enabled it collects data from three analog sensors and stores these data samples in an 8MB Atmel DataFlash. At the end of the data collection period the pods are connected to a PC to extract the data and to recharge the lithium battery.

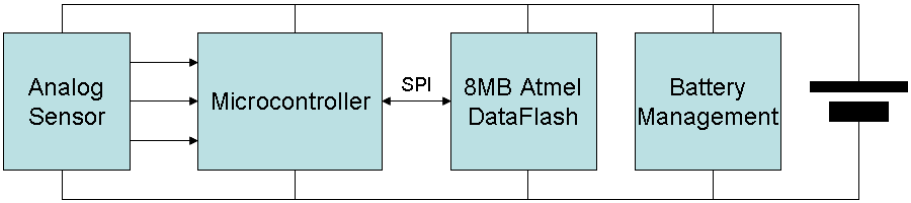


Figure 4.2: Block diagram of data collection pod

There are many applications that have a similar block diagram. In this application data is being collected but, with transducers rather than sensors, this block diagram could also be a data distribution system. My point here is that I am describing a general application using a specific example.

The obvious method of connecting the data pod to a PC is via USB. This will mean choosing a microcontroller that has a USB interface or by adding a USB component such as the FT232R as shown in Figure 4.3. Since we have a lot of data to move perhaps we should consider high speed USB. Both options increase the size, weight, and current consumption of the data pod so we look for a more creative solution.

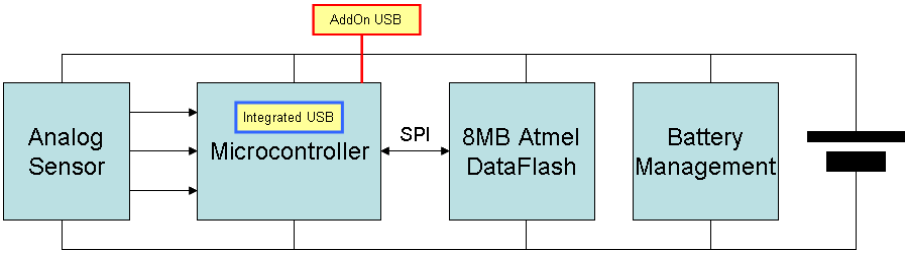


Figure 4.3: Options for adding USB

Figure 4.4 shows an optimized solution that partitions the design into a "Reader" and a lower-cost "Data Pod". The data pod connects to the reader using the SPI connection on a set of PCB gold fingers - this saved the size, weight and cost of a connector and did not involve additional circuitry in the data pod. Additionally the battery management IC was moved out of the data pod and into the reader since it is only needed during charging.

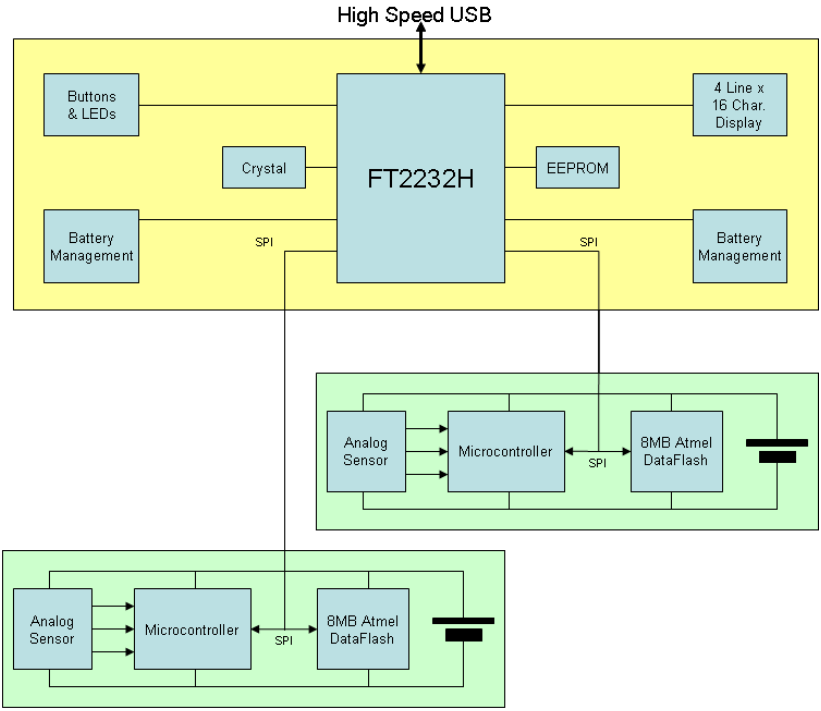


Figure 4.4: Block diagram of 'Reader' plus 'Pods'

The FT2232H can support two USB-to-SPI channels and run them both at 30MHz. There are also enough additional IO lines to manage two battery management ICs, support a series of buttons and LEDs and a 4 line by 20 character display that can give the user instructions or sales messages. This means that the "Reader" is a standalone peripheral that does not need the PC screen or keyboard to implement a human interface. So, if needed, a single PC could support several of these readers. Lets step through this example which I have modularized to create a set of easy-to-adapt building blocks for your use.

Figure 4.5 shows a more detailed block diagram of the IO connections to the FTDI FT2232H channel A which will be set up in MPSSE mode. Channel B is similar but has buttons and LEDs in place of the LCD display. I will be going into detail on the SPI interface to the Atmel AT45DB642D 8MB DataFlash and on the custom parallel interface to the LCD display. For prototyping I used the FT2232H mini module, shown in Figure 4.6 which I wired to the DataFlash and to the display. All components are powered from USB.

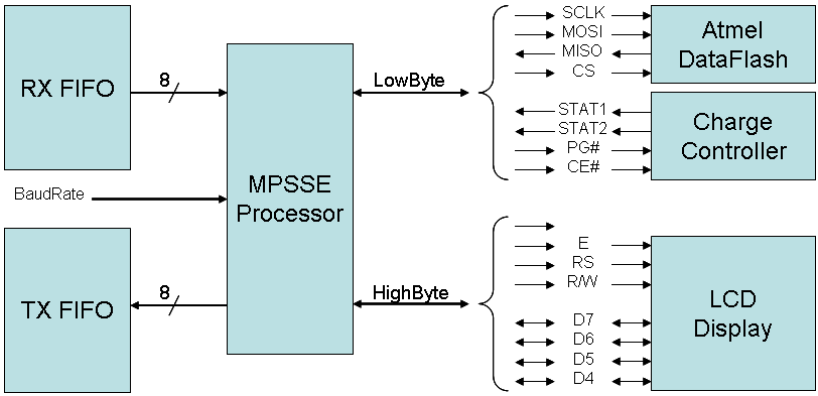


Figure 4.5: Detail of FT2232H Channel A IO connections

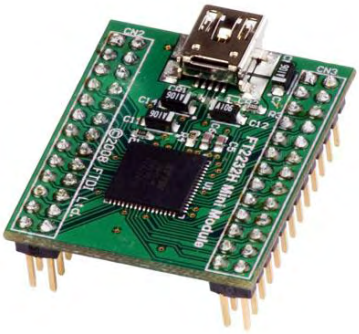


Figure 4.6: FT2232H mini module used for prototyping

SPI interface

In MPSSE mode, command bytes are intermingled with data bytes within the RX FIFO and the MPSSE processor decodes these command bytes and operates on any data bytes - this can be a little confusing at first so I shall step through this SPI example in detail. Rather than repeat a lot of information in FTDI's applications note AN_108. I recommend that you have a copy of this note to refer to as I work through this example.

The MPSSE command structure enables data to be strobed out of the RX FIFO at a bit or byte level on the rising or falling edge of the clock. Data can also be strobed into the TX FIFO with similar control. Our first task then is to choose a signaling method that is compatible with the Atmel DataFlash. Referring to figure 21.1 of the AT45DB642D data sheet we note that in SPI mode 0 SI data is latched on the rising edge of SCK and S0 data is driven on the falling edge of SCK. We therefore set up MPSSE to drive byte data out on the falling edge of SCK (i.e command 0x11 from AN_108, Table 3.3) and to read data on the rising edge of SCK (i.e command 0x20 from Table 3.3). The Atmel DataFlash requires CS to toggle to initiate commands and I will use SetDataLow commands (command 0x80 described in section 3.6 of AN_108) to drive CS low and high.

Let's first read the Device ID from the DataFlash. After driving CS low we need to send a command byte, 0x9F (See table 15 of Atmel data sheet), we then read in 2 bytes and finally drive CS high. This sequence is shown at the left hand side of Figure 4.7.

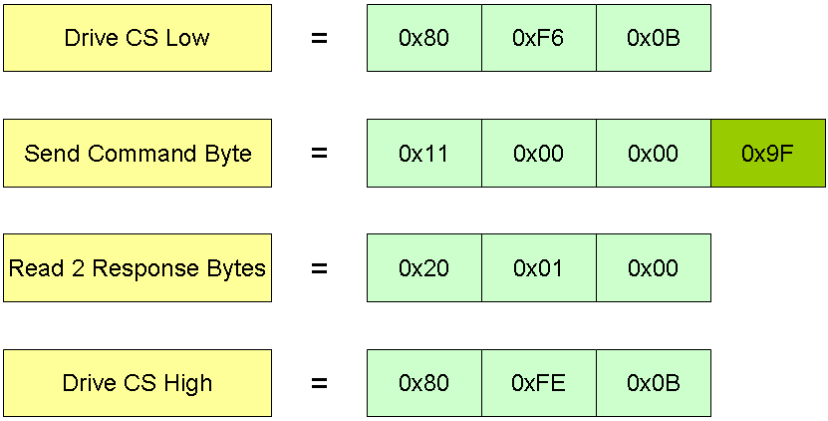


Figure 4.7: MPSSE commands used to drive SPI

A 3 byte sequence is needed to drive CS low and this is shown on the right hand side of Figure 4.7 - this SetLowByte sequence can set up to 8 bits. 3 bytes are needed to send the SPI command byte - bytes 2 and 3 are a count of the following data bytes, and, in this example, there is only one byte (0x0000 = 1 byte). This may appear to be a large overhead but the PC is running very fast and the FIFOs are large so you should not be overly concerned about this. Since count can be up to 64,536 the overhead is less for larger data transfers. 3 bytes are needed to set up the read of the response from the DataFlash. Finally 3 bytes are needed to drive CS high which will return the SPI bus to its idle state.

So, we load the RX FIFO with 13 bytes and execution of these commands by the MPSSE engine will result in 2 bytes will be written into the TX FIFO. Figure 4.8 shows the resulting SPI traffic captured with a USB DX logic analyzer (see www.usbee.com). I added an extra chip select and deselect, so that we could get a little more insight into the sequence timing. I have the baud rate set to 1 MHz during debug and I will run at 30 MHz later.

Refer now the Example3.cpp program listing - I have several helper routines that allow you to focus on the SPI operation and not on the details of the MPSSE implementation. Most DataFlash commands are 4 bytes long so I declare them as 4 byte dwords and manage the individual bytes inside the helper routines. I have implemented GetDeviceID, ReadData, and WriteData to get you started. Note that the bit-bang instructions (SetLowByte used to toggle CS) cycle the IO pins at the selected baudrate.

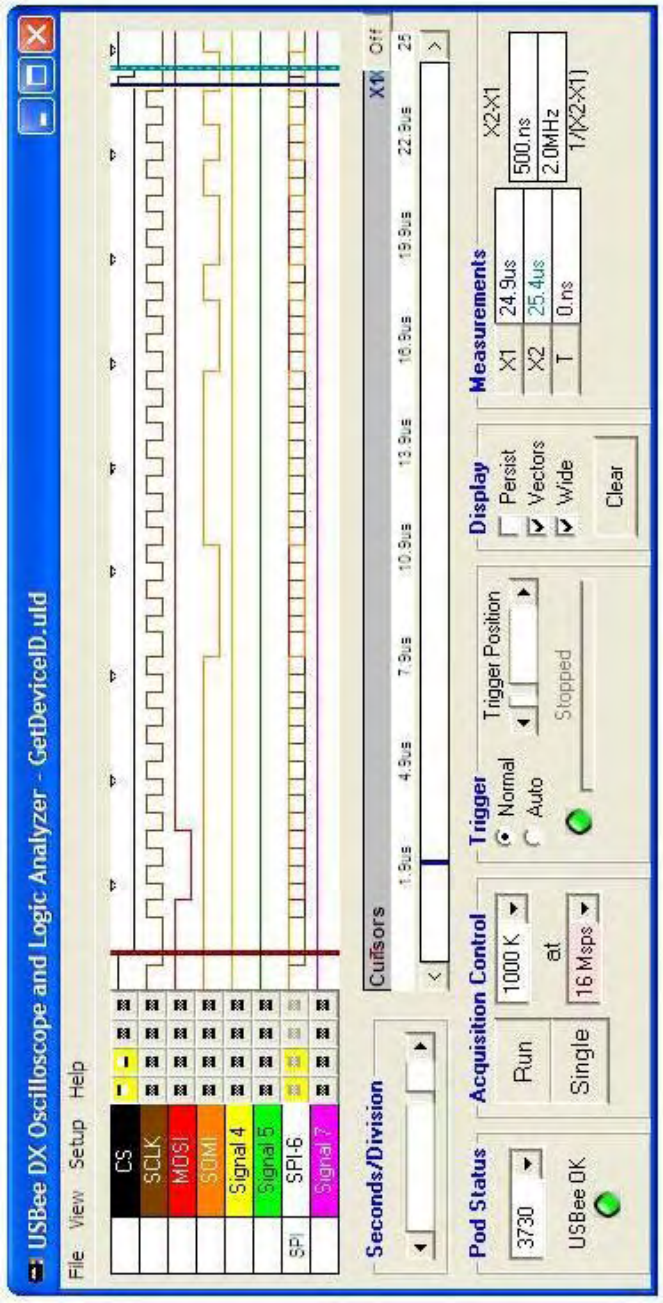


Figure 4.8: USBee trace of GetDeviceID SPI command

LCD Interface

Most 2 and 4 line character displays use the same parallel interface consisting of 3 control lines (E, R/W, and RS) and an 8 bit data bus that can be operated in 4 bit mode. In this application I have only 8 data lines available (called GPIOH0..7 when in MPSSE mode) so I implemented the data transfer in 4 bit mode. The waveforms needed to write and read the display are shown in Figure 4.9. I extracted these from the LCD display datasheet which is included in the Example 4 directory for your convenience. I use a series of SetByteHigh commands to create this custom waveform. To meet the LCD displays timing I set the baud rate to 1 MHz when writing to this high data byte.

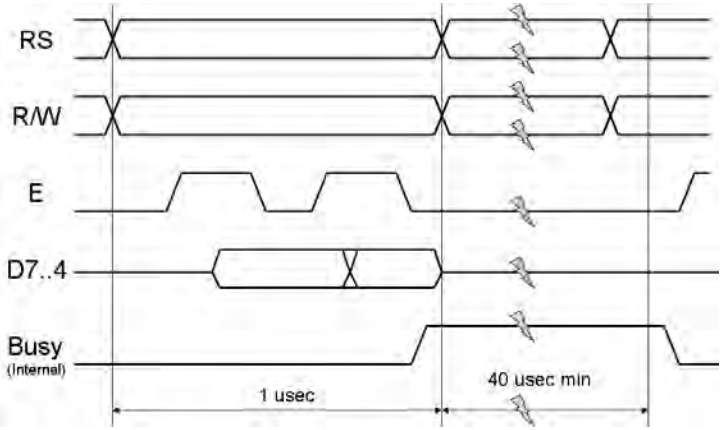


Figure 4.9: Control signals used by most LCD character displays

Sending a command to the LCD display takes about 1µs and then the display needs at least 40µs to implement the command. Some commands take much longer. Please refer to the LCD display datasheet. The datasheet also recommends polling the status register looking for a busy bit but we will NOT do this due to the fundamental operation of USB. It is time to understand the latencies involved with USB transfers.

Sending data to the RX FIFO involves an FT_Write command and reading from the TX FIFO involves an FT_Read command. If two commands are initiated (two FT_Writes or FT_Write + FT_read) then the OS will schedule these in separate USB frames. In other words, they will be at least 1ms apart. So it is not sensible to poll for a 40µs signal since it will take 1ms to do the poll! It is also a good idea to send as many bytes as possible in a single buffer; otherwise the

separate FT_WRITEs will be 1ms apart. The FT2232H has a 4KB RX FIFO and can therefore queue up a large number of commands + data for the MPSSE engine. In this example I idle for 40µs between most LCD commands using the MPSSE command 0x8F, 0x38, 0x00.

My simple example, within the Example4 directory, allows you to send any text to any line. The frame work is complete and other functions, as described in the LCD display datasheet, may be easily added. I tested the code on several displays of different physical sizes, some 2 line some 4 line, and they all operated the same way.

For the Data Collection Pod example I have SPI on both channels and I bit-bang the battery management ICs, LCD display, buttons and LEDs. The FT2232H makes an excellent dual USB-to-SPI adaptor with additional capability to read and write 24 additional IO lines.

Other examples

The MPSSE mode of the FT2232H supports SPI, I2C, JTAG and custom parallel protocols on both channels. So, as an exercise, we could restructure the FT2232H's channel B to drive the I2C protocol and run the examples from chapter 2. Or we could reprogram the FT2232H's channel A to be a serial interface and run the examples from chapter 3. The FT2232H also supports several other modes and protocols that I have not presented here (look over the datasheet!) making it an extremely versatile component suitable for many interfacing projects.

Figure 4.9 shows an alternate hardware implementation for a single channel DataPod using a solder-less breadboard. I used a DLP-1232H module since this DIL module plus straight in! A downside is that only part of a Channel A is brought out to the module pins; high byte is not brought out so I implemented the 7-pin LCD interface on an MSP23S08 expansion device (this is an SPI version of the MSP23008).

I hope that I have given you a flavour of the capability of the FT2232H - it can be any two of:

USB-to-SPI adaptor USB-to-I2C adaptor
USB-to-JTAG adaptor USB-to-custom protocol adaptor
USB-to-serial adaptor (RS232, RS422 or RS485)

And if you need four channels then look at the FTDI FT4232H.

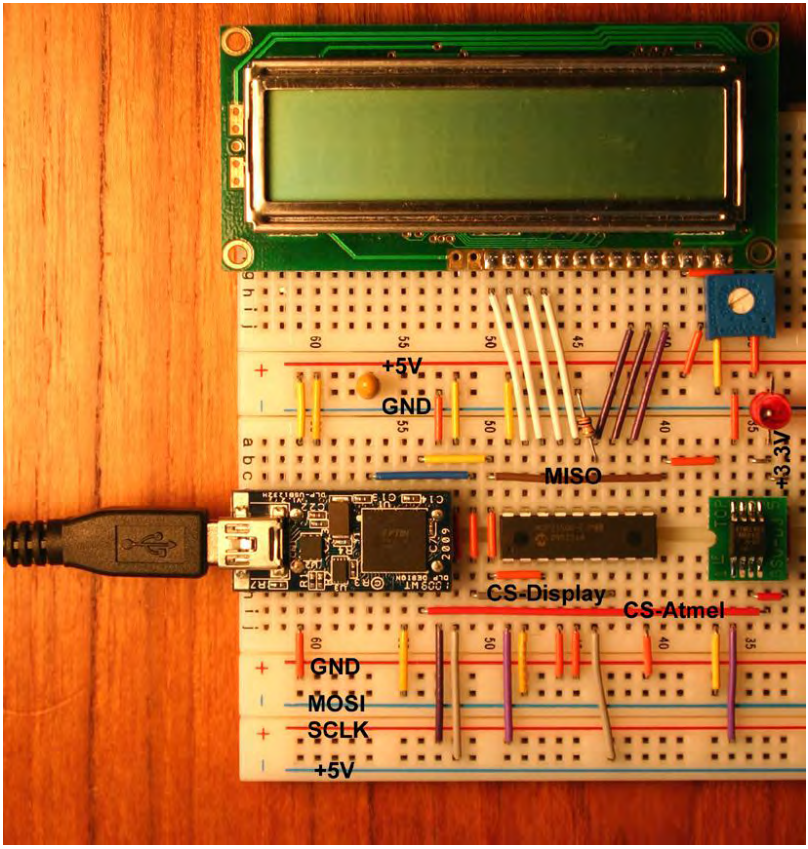


Figure 4.10: Single channel DataPod using a DLP-1232H module

Chapter summary

Notice that we write software on the PC to create our industry-specific device. The modes of the FT2232H allowed us to create a variety of solutions with a single component which is configured using software. There is no firmware to write or maintain at the USB device since the FT2232H is implemented as a fixed-function, high-speed device. The drivers hide all of the details of USB and we can focus on filling the RX FIFO and reading the TX FIFO – this enables us to be closer to our application.

Introduction to Part 2

We learnt in part 1 of this project book that most of the complexity of a USB system is within the host controller. The host controller is responsible for managing the communications to a diverse range of USB devices and this must be scheduled using predefined rules and many timing constraints. Early USB host controller implementations used a fast RISC CPU that was tuned to process the various transaction lists and to handle the required error checking and retries. FTDI took a different approach with their Vinculum-II host controller - they implemented most of the host controller functions in dedicated, special-purpose hardware such that the host controller function could be managed by a simpler 16-bit CPU. There are still timing constraints but these are handled by a real-time micro-kernel (which is described in later chapters). This 'hardware-heavy' implementation results in a USB host controller that is easy to use and is the main subject of part 2 of this project book.

FTDI provide more than just the silicon components; their full solution includes a complete C-based tool chain with a GUI-based Integrated Development Environment (IDE), a Real Time Operating System (RTOS), an on-chip debugger and evaluation modules. There is a lot of material to cover! Chapter 5 looks at the original Vinculum host controller and its applications range –Vinculum-II can do everything that the original Vinculum can do plus a lot more! Chapter 6 looks at the hardware capabilities of Vinculum-II and chapter 7 looks at the micro-kernel and device drivers wrapped around this hardware. Chapter 8 works through a design example using the Vinculum IDE and I round off part 2 with two chapters of worked design examples. You will discover that developing a product that requires USB host capability is a straight-forward, well-defined task – you will call it “easy” after the second project!

My brief from FTDI in writing Part 2 was to “keep it short and precise” so, to save space in this book and to avoid a lot of repetition; I only include snippets of the example code in this book. I expect you to be reading this book alongside a PC that has the examples on it. I will refer to the example code by filename and you will view the source code on your PC screen rather than in this book. Having a PC with the Vinculum-II toolset installed and a V2EvalBoard attached will also give you the opportunity to run the examples as I progress through each stage. This will provide the best learning experience.

Chapter 5: Vinculum-I Design Examples

FTDI introduced their first generation dual USB host controller, now called Vinculum-I, in 2006. It is a fixed-function device that supports the attached mode of operation as shown in Figure 5.1.

Figure 5.1: Vinculum-I operates as an attached device

Vinculum-I runs a firmware monitor that is controlled by an external application CPU using an SPI, FIFO or UART link. Several firmware versions are available that implement a variety of specific functions but all include the ability to read and write to a USB flash drive. This chapter looks at several examples of attaching a flash drive to an existing product using Vinculum-I.

Adding a Flash Drive to a product

The flash drive is arguably the most successful USB product. Its density has increased almost logarithmically over the past decade while its price has fallen at a similar rate. You can now buy 1GB drives for less than $10. But, up until now, they have been excluded from embedded projects due to the complexity of interfacing but I am about to change all that!

The major issue is, of course, that a flash drive is a USB device and therefore, to control it, you need a USB host controller. The USB specification deliberately put most of the communications complexity within the host controller, since there is only ever one in a system, and this enables USB devices to be simpler and therefore lower cost. A flash drive is a Mass Storage Class device and, although these specifications are a free download from www.usb.org, they are not easy to read. This is not surprising because they are specifications and not implementation guides. Additionally, these Mass Storage Class specifications only define basic track/sector, read/write operations so we also need to understand specifications of the FAT file system, as used on all commercial flash drives, to be able

to read and write user data. The amount of information that we need to understand how to "just connect a flash drive" is becoming over whelming. What we need is a component that implements all of these specifications for us; after all, they are industry standard specifications that we have almost no freedom to change anyway, we just want to use them!

Vinculum-I provides a DOS-like, command line interpreter, front-end to a flash drive. A Vinculum block diagram is shown in Figure 5.2 – internally it is implemented as a microcontroller, with specialized IO devices, running embedded firmware but we do not need to know this. Vinculum-I's command line interface is accessed via a UART, SPI, or a FIFO. Vinculum-I actually supports two USB ports and each can be programmed to be a host or a device but my series of examples will assume a single host port with a connected flash drive.

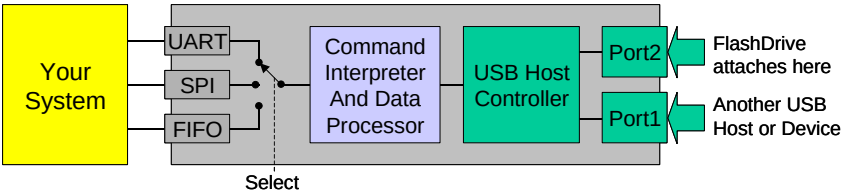


Figure 5.2: Vinculum-I uses a DOS-like command interface

To demonstrate its ease of use I am going to connect the FT232R USB-to-serial cable introduced in Chapter 2 to a PC that is running HyperTerminal at 9600 baud. I will connect this cable to an FTDI evaluation module called VMusic as shown in Figure 5.3. Ignore the name for now, we won't use the "music" part until the second example; for now, this is a Vinculum-I mounted on a board with a convenient serial connector. Choose any flash drive that you may have and plug this into the USB A socket of the VMusic board, also shown in Figure 5.3.

Figure 5.3: USB-to-Serial cable connected to VMusic board

The board will sign on and offer a D:> prompt. Now, in HyperTerminal, enter "DIR" and, Hey Presto, the contents of the drive are displayed. Now enter the following commands:

```
OPW test1
IPA
WRF 12
Hello World!
CLF test1
```

These commands first open a file called "test1" for write, then tell Vinculum that 12 bytes of data are coming. "Hello World!" is the data that is written, and CLF closes the data file.

Now remove the flash drive and connect it to your PC, Mac or Linux system and open test1. Notice that the data written by the Vinculum is present. Now edit test1 to add a message "Hello from my PC, Mac or Linux"

Now reattach the flash drive to the VMusic board and enter "RD test1". Voila, the text is displayed!

Now stop and think what we have accomplished.

We have written, read and exchanged data files between a PC, Mac or Linux system and an embedded system using a flash drive. We did not have to learn USB, the Mass Storage Class specification or even the FAT file system. It was as easy as entering DOS-like commands on a serial connection.

Pretty amazing!

Vinculum-I powers up in Extended Command mode where all the commands and data are ASCII; some of these commands are summarized in Figure 5.4. It can be switched into Short Command mode where binary commands and data can be exchanged. The VMusic board only provides access to the UART connection but this will be enough for my first set of examples. Vinculum-I is also available in an OEM 24 pin DIP and this additionally provides access to the SPI port, the parallel port FIFO and the other USB port.

	Directory Operations
DIR	Lists the current directory
MKD <name>	Creates a new directory <name> in the current directory
DLD <name>	Deletes the directory <name> from the current directory
CD <name>	The current directory is changed to the new directory <name>
CD ..	Move up one directory level
	File operations
RD <name>	Read file <name>. This will return the entire file
OPR <name>	Opens file <name> for reading with 'RDF'
RDF <size>	Reads <size> bytes of data from the current file
OPW <name>	Opens file <name> for writing with 'WRF'
WRF <size>	Writes <size> bytes of data to the end of the current open file
CLF <name>	Closes file <name> for writing
DLF <name>	This will delete the file from the current directory and free up disk space
VPF <name>	Play an MP3 file. Sends file to SPI interface then returns
REN <n1><n2>	Rename a file or directory
	Management Commands
SCS	Switch to the short command set
ECS	Switch to the extended command set
IPA	Input data values in ASCII
IPH	Input data values in Hex
SUD	Suspend the disk when not in use to conserve power.
WKD	Wake Disk and do not put it into suspend when not in use
SUM	Suspend Monitor and stop clocks
FWV	Get Firmware Versions
FS	Returns free space in bytes on disk

Figure 5.4: Some of the monitor's DOS-like commands

There are two types of project suitable for an attached Vinculum device – data distribution and data collection and I have examples of each category. Typically, data to be distributed is created on a PC using specialist tools and then copied onto a flash drive; an embedded system then accesses this information and presents it to a user or a machine. My example is a small JPEG viewer and MP3 player – something that we would take on a business trip and that plays back images and sounds of our family, or our favorite music. If I had used a larger display I would have called this an "active photo frame" (it is on my TODO list!). My data collection example is a portable data logger that collects field data for later analysis by a PC. In both cases an application microcontroller is used to drive the Vinculum-I (since it is a peripheral device) and other circuitry. I am confident that you can dream up many more applications for this easy-to-use part.

JPEG viewer and MPEG player

I chose a Cypress PSoC for the application microcontroller since it has firmware-configurable hardware that allows me to solve a wide range of problems with a single device. I develop and debug using a "high-end" PSoC device that has ample analog and digital resources then, near project completion, I can select a lower cost device within the same family. For the first example I shall use the Cypress PSoC Evaluation board and this is shown in Figure 5.5 with the VMusic board and a 1.5" x 1.5" Micro-LCD display already attached to the breadboard area.

Figure 5.5: This example was developed and debugged using a PSoC development system

Fundamentally I have a PSoC that reads image files off a flash drive using commands sent to the Vinculum monitor, the PSoC then sends this image to the display. If a matching MP3 file is also present on the flash drive then I command Vinculum-I to play it – this could be music or a person talking. A PC is used to create the image and MP3 files and these are copied onto the flash drive. The

PSoC/Vinculum-I based player then “runs the show.” A JPEG viewer and MPEG player is the base example but an interactive display that could be used in stores, museums, product demonstrations, art galleries, etc. is a straightforward design extension. A series of flash drives in English, Spanish, Japanese, etc. could be used to create a more universal solution.

Another beneficial aspect of a PSoC-based design is that Cypress has over a hundred applications notes that describe building blocks that can be used within your own design. The PSoC could scan buttons and the application program would use these button inputs to navigate through images / MP3 files. Or the PSoC could support a touch screen using a few of its configurable analog and digital blocks. This could be a simple resistive screen overlay or a more reliable CapSense implementation.

The size of the graphical display determines the complexity and choice of the PSoC. Since this example is about embedded flash drive applications I chose the simplest display to implement here and I will cover interfacing to a larger display in a future update. I found a serial-interface, micro-LCD and was very impressed with ease of use of these 128x128 color displays. These 1.5” x 1.5” displays are not expensive – you should get some and experiment with them. I am sure that you will soon find many uses for them, just as I did. I personally found the OLED displays much better to look at when compared with the LCD displays but the firmware to drive both displays is identical. The micro-LCD module is a very capable subsystem that supports graphics rending and several fonts. My example uses about 5% of its capability as I just download images to it. These images are 128x128 by 16-bit color and I use a PC application called Graphics_Composer that converts JPEG, BMP, and GIF images into this format (this is included in the download package). In this example these images will be copied to a flash drive and called nnn.img (nnn = 000 to 999). MP3 files are also created for each image and they will be called nnn.MP3 (these could be your favorite songs renamed).

From an application software perspective we have a PSoC interfacing two serial connections, a Vinculum-I and a micro-LCD connection. The application starts by looking for 001.img and copies it to the display. If it finds 001.MP3 then it will play it, else it will wait for 60 seconds (easy to modify) before moving onto 002.img. The application keeps incrementing through filenames until one is not found then it starts at 001.img again. To change the photos and/or

music you just swap the flash drive. The complete PSoC project is one of the Vinculum-1 examples downloadable from my website and, as you will see, it supports the basic function. It is easy to expand this design to add functions – I plan to add a feature-rich alarm clock once I get some spare time. It would be easy to make this battery powered however displays tend to consume a lot of power so I would also add a battery charger in this case. A battery charger uses a few analog and digital resources of a PSoC, a few FETs, an inductor and R's and C's. This design extension is covered in detail in Cypress's application note collection.

Portable data logger

I was "persuaded" to create an example based on the PIC microcontroller. I personally didn't like this part due to its "weird" instruction set. However, my colleague Don Powrie of DLP Design introduced me to the CCS toolset and these make PIC designs actually pleasant to do! I used to be a staunch advocate of only using assembler code for microcontrollers – I argued that a compiler would always generate larger object code than my tuned assembler code. But now these microcontrollers are available with 16KB, 32KB and beyond of flash memory! So what is the point of saving a few hundred bytes when you still have over half of the flash space as unused? C code is also much easier to write and debug when compared with assembler code.

The CCS compiler was specifically designed to create optimized code for the PIC family of microcontrollers. As well as all of the standard features that you would expect from a quality C compiler it includes built-in functions to support the on-chip features of a PIC microcontroller. A good example is the **#use RS232** directive; here you specify that you need to use a serial port and you give the compiler details such as baud rate and the IO pins that will be used for TX and RX. If the chosen PIC device has a hardware UART then the compiler will use this for printf and scanf functions, else it will include subroutines to manage the low-level bit manipulations for you. Your main program uses printf statements as before. The CCS compiler also contains built-in functions to drive the on-chip ADC and the real time clock. This way you can focus upon WHAT your program is doing and not the lower level HOW.

Don designed the battery powered data logger shown in Figure 5.6 to demonstrate the capabilities of Vinculum-I. The example program uses a serial connection to control Vinculum-I, which writes data to the flash drive. Better still, the hardware connection is a standard 4-wire serial port using TX, RX, RTS and CTS.

The PIC runs an application program that has access to a flash drive using Vinculum-I, a real time clock, a temperature and humidity sensor and two analog input channels. A connector for the ubiquitous TTL-232R cable is included, as it a connector for a PIC debugger such as CCS's ICD-U40 unit.

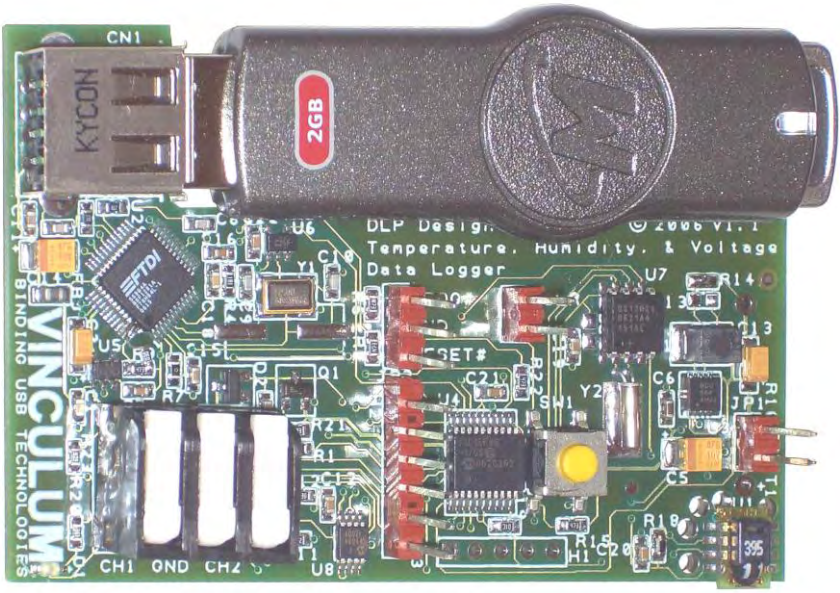


Figure 5.6: The DLP-VLOG showcases Vinculum-I's capabilities

The application program first checks to see if a flash drive is present – if one is not found then the PIC goes back to sleep since there is no point collecting data if there is no where to store it. Once a flash drive is found the PIC starts a data collection cycle: it first reads the real time from the Dallas/Maxim DS1302, then the two analog signals and the battery voltage, then the temperature and humidity. This data is then written to the flash drive and the system goes back to sleep to save battery power. This cycle repeats while a flash drive is

present and the battery is charged. The flash drive may be removed at any time and the collected data may then be analyzed using your PC.

The source code for the application is available with the Development Kit so that you can customize the data collected and the time interval between samples. Don designed the board as an evaluation tool for Vinculum-I designs but I can see many applications where this battery-operated, portable data logger would be a great fit as is.

Embedded flash drive designs now enabled

I hope that I have shown you that projects built around a flash drive are now easy. Vinculum-I encapsulates all of the required industry standard specifications and presents a simple DOS-like command line interface that is accessed via a serial port (or SPI or parallel FIFO). You add your favorite microcontroller with an application program to control the Vinculum-I peripheral. I presented a few projects to fuel your imagination. My examples used a Cypress PSoC and a MicroChip PIC but the code is readily ported to a different microcontroller architecture. Your project can collect data that is later analyzed on a desktop system or it can be used to redistribute data that was created on a desktop system via lower cost platforms. Project data may be updated by simply swapping flash drives.

If you can read and write to a serial port then, with Vinculum-I, you can read and write data files on flash drives. I would be interested to hear about projects in which you have creatively used a Vinculum and a flash drive.

Chapter 6: Getting to know Vinculum-II

Vinculum-II is FTDI's second generation dual USB host controller and it is a superset of the original Vinculum described in chapter 5 - in fact, the 48 pin LQFP version is backwards compatible (although it needs different firmware). Intense customer feedback on the Vinculum requested more package options, higher performance with lower power and more capability. Vinculum-II delivers on all of these aspects with the added ability to be user programmable - this allows Vinculum-II to support an additional standalone usage model as shown in Figure 6.1. Here the Vinculum-II CPU is also the application CPU and an adjunct microcontroller is not required which will reduce system costs.

Figure 6.1: Vinculum-II supports standalone operation

Vinculum-II was designed from the ground up to be an efficient C machine and the much larger transistor budget was spent adding hardware assist to all of the peripheral components. Figure 6.2 shows a block diagram of the Vinculum-II, all of function blocks are enhanced over the original Vinculum-I device and there are several new blocks.

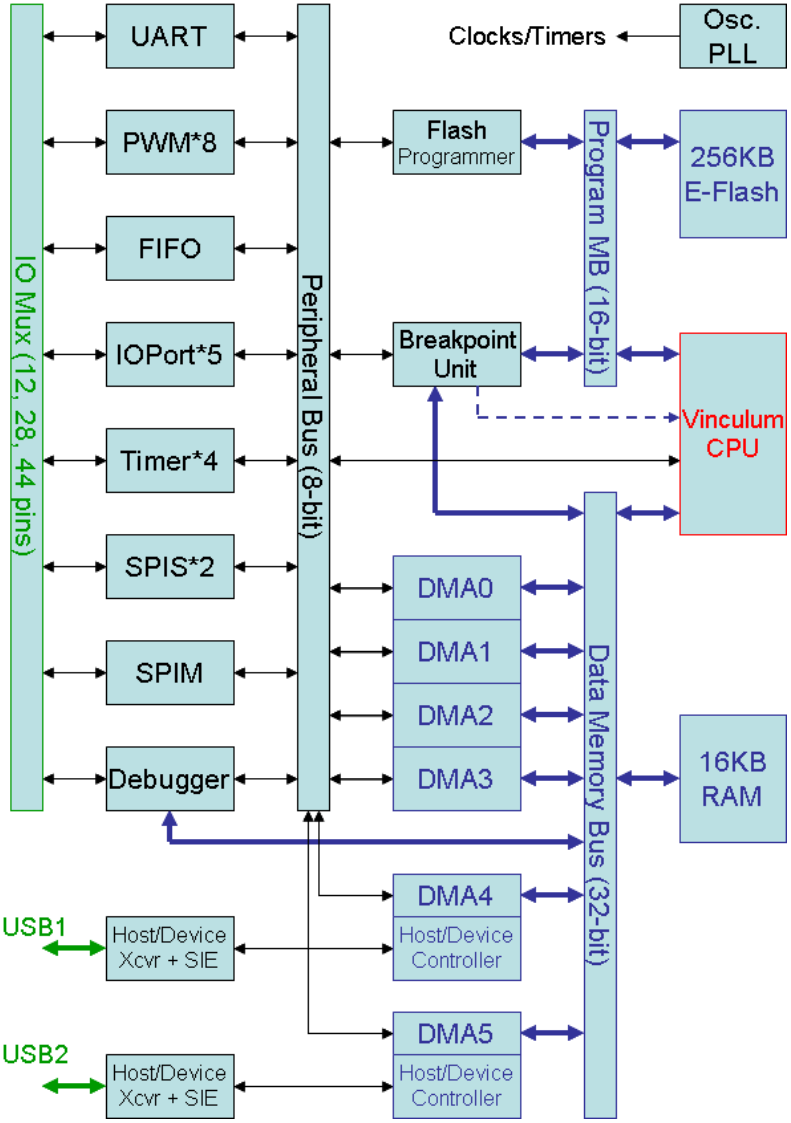


Figure 6.2: Vinculum-II hardware block diagram

The heart of Vinculum-II is a modern 16-bit Harvard Architecture CPU that controls three major buses – a 32-bit data memory accessing 16 KB of RAM, a 16-bit program memory accessing 256KB of Flash and an 8-bit peripheral bus. All buses are pipelined and

support concurrent operation. The CPU has no registers (or it has 8 K registers depending upon your point of view) and the instruction set has been designed around single clock memory accesses. The instruction set was designed to efficiently implement C code and the user is not expected to use an assembler (although one is available). In fact, all of the examples in part 2 are written in C and I haven't even opened my assembler guide!

The debugger block can take control of the CPU if necessary using the breakpoint unit. There are three hardware breakpoint registers that can trap program or data accesses in real time. The debugger port is a single pin, bi-directional 1Mbaud serial connection and it can take control of the CPU and all internal buses; it can also manage two special peripherals, the breakpoint unit and the flash programmer, enabling a blank Vinculum-II to be easily brought to life with minimal external hardware. Figure 6.3 shows the debugger module that connects to a Vinculum-II target system: the Vinculum-II DIP modules have matching pins and an FTDI Applications Note describes how to integrate this capability into your custom design. This module connects to the development PC using a standard USB cable and this will be demonstrated in Chapters 9 and 10. Note that the Vinculum-II Evaluation Board has the debugger module circuitry built in.

Figure 6.3: A debug module connects to your target system

A major Vinculum-II design goal was efficient power management and, following a reset, only the CPU, clocks, debug port and flash memory are powered. The CPU starts at 48 MHz and can be switched down to 12 MHz or it can move into standby mode where it only consumes about 150 uA. The 12MHz active current is 8 mA increasing to 25 mA at 48 MHz.

All the peripherals shown in Figure 6.2 have individual power connections that are not activated unless required by your application program. Each peripheral has one or two control/status registers and these are considered as part of the CPU core - this enables the CPU

to power down a peripheral that is infrequently used while maintaining its state for quick re-activation. Data buffering and movement is handled by a sophisticated DMA controller.

The six - channel DMA controller can move data between memory and peripheral devices or it can implement queues, FIFO or circular, in memory. The CPU is rarely involved in data movement - it sets up DMA channels and responds when data transfers have completed. Four channels are available for application use and two are dedicated to each USB host/slave controller.

The USB host controller is modeled upon the OHCI (Open Host Controller Interface) Specification (a free download from *http://www.compaq.com/productinfo/development/openhci.html*) with most of the queue handling, error checking and retries implemented in specialized hardware. This results in minimal interaction required by the CPU to support full and low speed data transfers on both host channels simultaneously. Each host controller can also run in slave mode to present a USB device interface to an external host (we shall see examples of both in later chapters).

The Vinculum-II can be used in attached mode (see Figure 5.1) where an external CPU uses the UART, SPI or FIFO peripherals to communicate with a firmware monitor program running on Vinculum-II. FTDI plan to port all of the original Vinculum-I firmware packages (VDAP, VDIF, VCDC, VMSC, and VDPS) into Vinculum-II versions and the Vinculum-II monitor will be the subject of a future FTDI Applications Note.

Additional peripherals and modes have been added to Vinculum-II to support operation in standalone mode (see Figure 6.1). A high-speed synchronous mode has been added to the FIFO function; the original SPI slave modes are supported and a standard, 'unmanaged' slave mode has been added; two slave SPI channels are now available and a master SPI channel has been added; the UART is unchanged.

Vinculum-II also includes: five general purpose 8-bit IO ports where a transition on some bits can generate an interrupt; 4 general purpose 16-bit timers, each with an optional 16-bit prescalar, that operate in a variety of modes including one shot, continuous and interrupt generation; a 32-bit WatchDog timer that will reset the CPU if not periodically cleared and 8 PWM channels that supports a variety of modes.

Vinculum-II is available in 6 packages types (32/48/64 pin, LQFP/QFN) so, after selecting which peripherals you need for your application, you would choose the smallest (and therefore cheapest) package option for your design. To allow this package flexibility Vinculum-II includes an IO Mux that is used to map peripheral resources onto physical device pins. Full cross-bar switching (i.e. any peripheral pin connectable to any physical pin) consumes a great deal of die area so, in order to keep costs down, the peripheral pins are grouped into sets of four related functions and these functions are connectable to the physical pins in groups of four. Figure 6.4 shows two examples of connecting peripherals to physical pins. The Vinculum-II toolchain includes an IO_Mux utility program that simplifies I/O pin assignment.

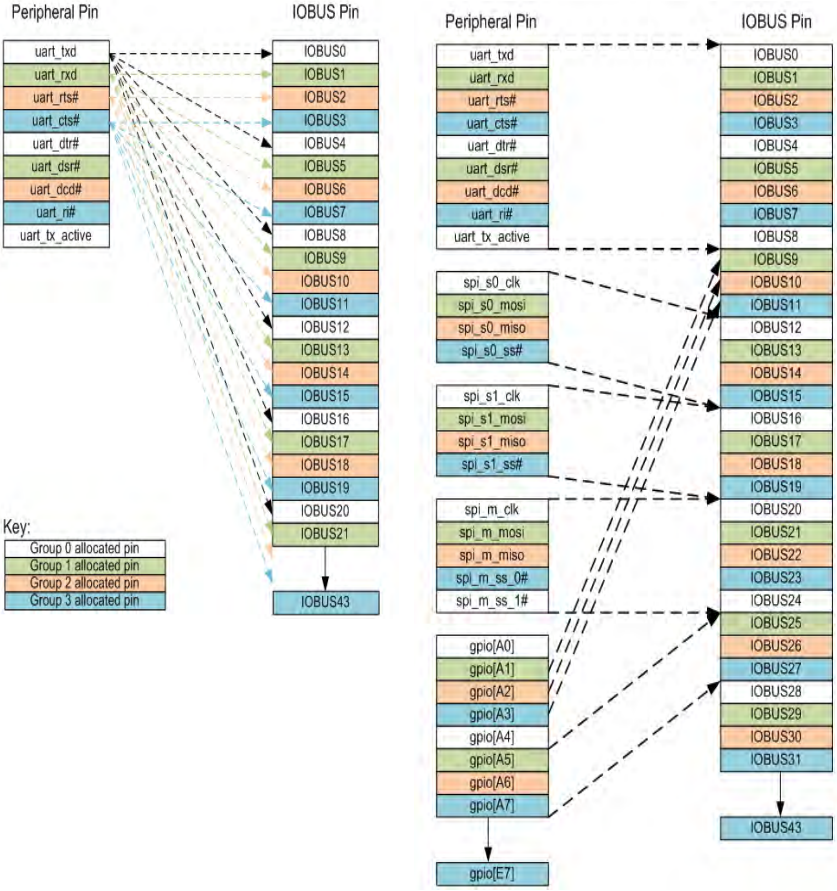


Figure 6.4: The IO Mux connects peripherals to physical pins

A schematic of each physical pin connection is shown in Figure 6.5 and is designed to accommodate a variety of load situations. The output stage operates at 3.3V levels (and is 5.0V tolerant) and may be configured to have a slow or fast (default) slew rate and a drive strength of 4mA (default), 8mA, 12mA or 16mA. An input may be configured as a normal input (default) or a Schmitt trigger input and have no termination (default) or to use a pull up or pull down 75 KΩ resistor.

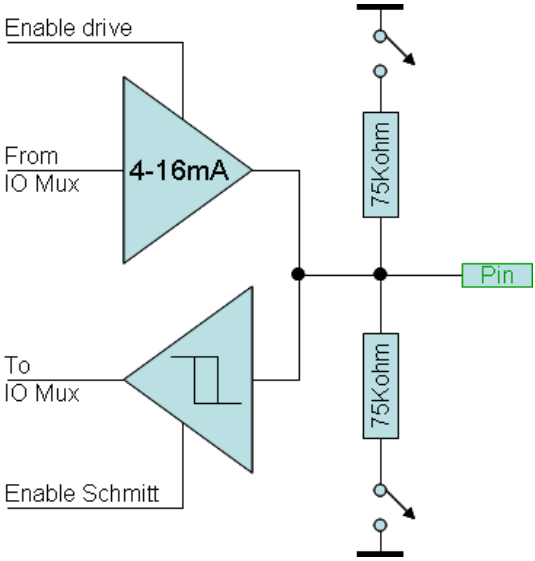


Figure 6.5: Each IO pin has a configurable driver/receiver

Vinculum-II has a fast CPU, ample on-chip program and data memory and a diverse collection of peripherals. For many applications these ample on-chip resources will enable a single-chip Vinculum-II solution. In all cases the peripherals are implemented with a lot of specialized hardware which allows the software to easily control these devices. FTDI also provide device drivers for each of the peripherals and this is the subject of the next chapter; your applications code will not access peripheral registers etc. but will use a common API on top of a micro-kernel. The micro-kernel manages all of the peripherals with tasks that are higher priority than user tasks. In this way, you need not be concerned that your selection of peripherals could change the system timing of, say, the host controller – you can focus upon your applications code.

Chapter 7: Writing software for the Vinculum-II

Writing software for a USB host controller may seem like a daunting task but I should point out that we have already done this! With example 1, in chapter 2, we wrote a program that interacted with the USB host controller on a PC (Windows, OS X or Linux). The physical hardware was masked by the operating system which presented us an API (Application Program Interface). We will follow the same methodology when writing software for our embedded host - FTDI provide an API for Vinculum-II which masks the intricate details of the embedded host controllers, and other hardware resources, so that we can focus on our application program. Figure 7.1 shows the basic structure of the different host environments - note the similarities.

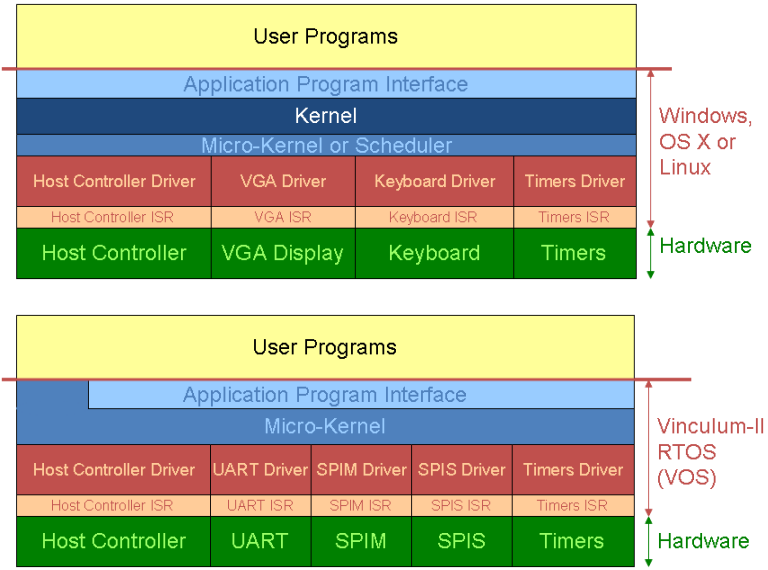


Figure 7.1: Applications programming environments

The lowest levels of drivers within the Windows, OS X, and Linux environments have a micro-kernel or scheduler that must deal with the real time nature of USB host controller communications but this level of detail is not exposed at the API level. Since FTDI expects many Vinculum-II applications to operate with real-time constraints they decided to expose a micro-kernel API to the user. They also provide device drivers for all of the on-chip peripherals. The

combination of micro-kernel plus device drivers is called the Vinculum-II Real Time Operating System, VII-RTOS, or just VOS.

If you haven't written programs using a multitasking RTOS framework before you will discover that it is a good methodology to build applications that do more than one thing, and I can't think of a previous embedded project that I have done that would not have benefited from this approach. If you are familiar with terms such as Task, Thread, and Semaphore you can skip the next section.

Multitasking RTOS 101

You have to learn some new words and concepts to be successful with a multitasking RTOS. This will take some effort so let me first explain the benefit of becoming familiar with these new terms.

You probably write your code using flow charts or state machines. Flow charts are good for describing sequential processes while state machines are good if there are small numbers of possible states with well-defined transition rules. However, both are poor at describing more complex systems with several interdependent parts. Multitasking, on the other hand, is a good fit for such systems - you define a task to handle each part then define how the parts interact.

A significant weakness of the sequential and state machine approaches is that they are inflexible. A good programmer can initially create a workable solution using these approaches but as requirements change and enhancements are demanded the workable design invariably turns into spaghetti code that is difficult to debug and even worse to maintain. The multitasking RTOS approach forces code that is structured so that it can grow and change easily. Changes are implemented by adding, deleting or changing some tasks while leaving other tasks unchanged. Since your code is compartmentalized into tasks, propagation of changes through the code is minimized. This will also reduce testing efforts. So, you have some hard work now to save time and effort later - this is a good deal.

The first paradigm shift you need to make is to partition your program into a set of smaller tasks - each will do one job and it will do it very well. You must also be comfortable with data structures since an RTOS will use a lot of them. Note too that ***task*** now has a specific meaning, it consists of a collection of code bytes that is the program, a collection of variables that are data bytes on the stack and a data

structure, also on the stack, called the task **context**. A ***thread*** is a data structure used to describe a task and its operational status.

If you have two, or more, identical peripherals (Vinculum-II has several duplicate units) you can define two threads each with the same code object but with different stack and context objects.

Once your application is divided into several tasks you will define how these tasks interact. The primary inter-task communications mechanism is a ***semaphore***, which is another system-defined data structure. Several operations are defined for a semaphore (object) such as ***Initialize***, ***Signal***, and ***Wait***. A task that creates data will signal when it has data while a task that consumes data will wait until a semaphore is set. Figure 7.2 shows a simple embedded program split into multiple tasks, three in this case.

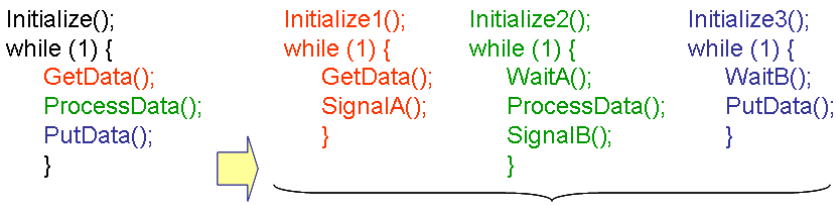


Figure 7.2: A program has several tasks that interact

We will work through an example in the next chapter using real Vinculum-II code rather than the theoretical pseudo-code shown in Figure 7.2 so don't focus upon the details yet. All will become clear with some examples.

Each task is written as if it has sole ownership of the CPU and you must now consider that GetData() runs continuously – mmm, what *did* happen to input data while you were processing and outputting data before? You could now allocate the coding of each task to different programmers with different areas of expertise. Also if a better data processing algorithm is discovered then only one task has to be changed; you need not be concerned about the impacts to the input and output processes since they now operate independent of the processing task. Are you beginning to see some of the benefits of this "divide-and-conquer" approach?

When you divide your program into multiple tasks you will decide that some are more important than others and you can assign these a higher priority. Figure 7.3 shows the classical multitasking RTOS task state diagram – specific details on the Vinculum-II implementation are covered later. As tasks are ***Create***d they are placed on the ***ReadyToRun*** list where the RTOS determines the highest priority task and makes this the ***Running*** task; execution of this task continues until it is blocked for some reason (waiting for a resource, such as a semaphore or a timer) when it is placed on the ***Waiting*** list; the RTOS then places the highest priority task on the *ReadyToRun* list as the *Running* Task; and so the process continues. There is a system-defined task, the ***IdleTask***, which has the lowest priority and is always ready to run.

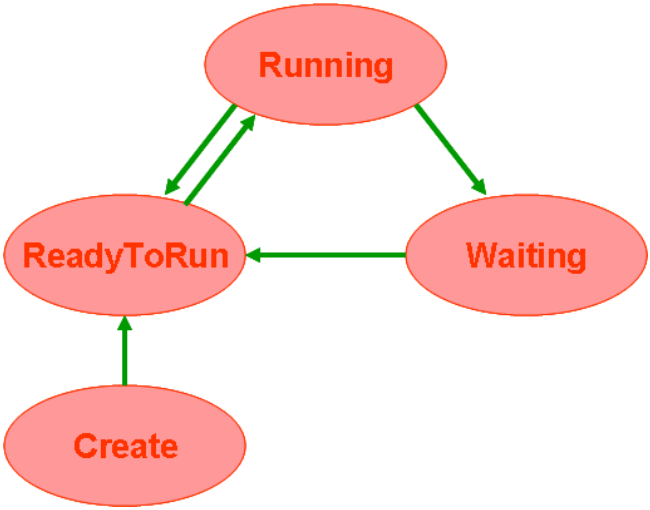


Figure 7.3: Tasks continuously move though this state diagram

Vinculum-II Software Architecture

Figure 7.4 shows a block diagram of the layered Vinculum-II software architecture. This is such an important diagram that I decided to give it a whole page (and I apologize that it is sideways!).

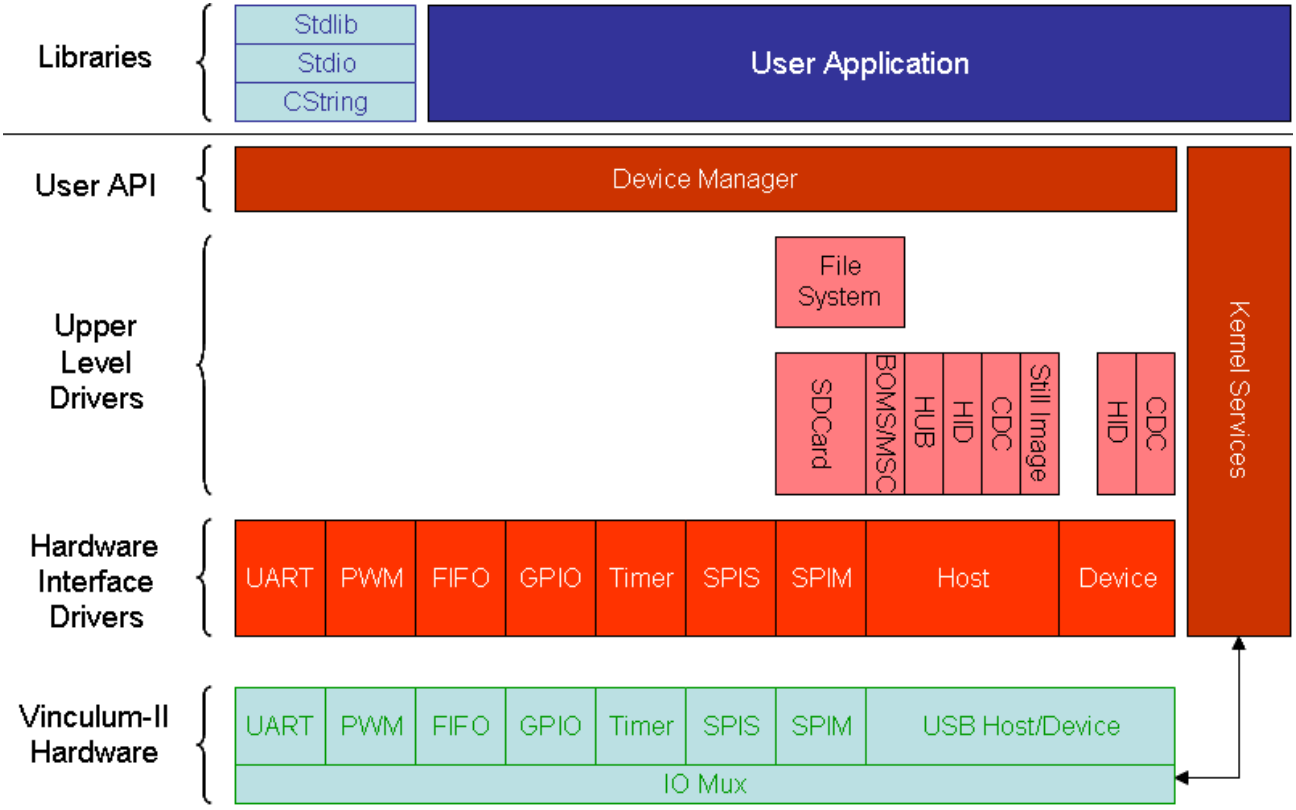


Figure 7.4: Vinculum-II Software Block Diagram

Following a RESET the software environment for Vinculum-II must be set up; the steps taken during this initialization are part of the **kernel services** module of Figure 7.4 and are shown in Figure 7.5. All Vinculum-II programs implement these steps but with different data and, once initialized, the run-time diagram shown in Figure 7.3 describes the operation of your program.

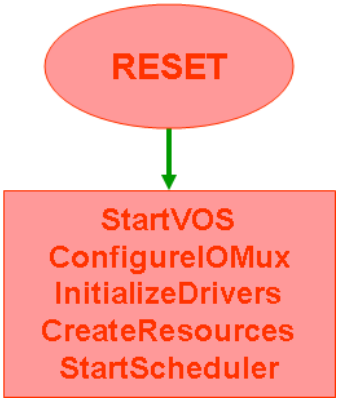


Figure 7.5: Software Initialization Steps

Kernel Services

Looking deeper into the kernel services initialization steps:

StartVOS: this function call initializes all of the internal data structures and sets up the operational parameters of the kernel. The Vinculum-II Operating System, or VOS, needs to know how many device drivers will be used so that it can organize and set aside memory for data buffers. System timing parameters are also set using StartVOS.

ConfigureIOMux: Vinculum-II is available in three package sizes (32, 48 and 64-pin) and this function call sets up the mapping of peripheral Input and Output functions with the physical pins on the package. The number of available IO pins varies with package size (12, 28 and 44) and you cannot get every peripheral signal connected to the outside world on the smallest package. Note too that you should be careful **not** to map away the debugger pin – this is pretty essential for program development and debugging!

InitializeDrivers: at the bottom of Figure 7.4 is the Vinculum-II hardware that was presented in Chapter 6. Each of the peripherals has a set of control and status registers (some more than others) but their intricate hardware details are not exposed since the micro-kernel must own the hardware. Instead, FTDI provides an optimized *Hardware Interface Driver* for each element as shown in the lower level of Figure 7.4. Each driver is tuned for the particular peripheral and handles the device interrupts. A uniform API is presented to the user (described later in this Chapter) to standardize and simplify the application program. Before a driver can be used it must be initialized. The driver for most peripherals is small but the USB Host Controller

driver, for example, will set up several threads to manage the root hub, optional downstream hubs and transaction list processing.

CreateResources: a Vinculum-II program will consist of multiple independent tasks that interact with each other. During initialization the threads for each of these tasks will be created along with the semaphores, mutexes and, perhaps, shared buffers needed for inter-task communications. Each thread has a context (object) and handles for shared objects, such as semaphores, may be provided as data within this context. Each thread has its own stack that is initialized with a known pattern so that VOS can track memory usage.

StartScheduler: once all of the program objects have been initialized we start the real time operating system which schedules tasks according to the run-time task state diagram as shown in Figure 7.3. The VOS scheduler uses a round-robin, priority-based, pre-emptive algorithm to run the highest priority task. It also tracks statistics such as thread CPU usage and this enables your application to be profiled and tuned if necessary.

Additional Device Drivers

Returning again to Figure 7.4, notice that layered above the hardware interface drivers are a set of USB Class and Other drivers. FTDI provides (at the time of writing, October 2010) Bulk Only Mass Storage Class (BOMS/MSC), HUB, HID, Communications Device Class (CDC) and Still Image drivers on top of the host controller and also HID and FT232 drivers on top of the USB device controller. This means that, out of the box, the Vinculum-II can control flash drives, cell phones, cameras, mice, keyboards, joysticks, etc etc and can also operate as a HID or as an FT232 device. More drivers will be added in future VOS releases.

For advanced users, you can write your own device driver layered on top of the hardware interface driver. FTDI provides an SD Card example layered on top of the SPI-Master driver – this enables immediate support of SD Cards or even the Atmel DataFlash component since this uses the same SPI interface (and is included in the examples in Chapters 9 and 10).

File System Driver

Layered on top of the MSC driver and the SD Card driver is a FAT file system driver. It supports devices with FAT12, FAT16 or FAT32 structures and include everything you need to simply open, read or write and then close data files. It handles all of the file

allocation tables and directory updates. It only supports 8.3 filenames but I don't see this as an issue for an embedded system. The flash drives, or SD Cards, that Vinculum-II uses are interchangeable with Windows, OS X and Linux systems, as you would expect. I did discover that writing in blocks that are a multiple of the base sector size does give you better performance. The API supports random length file reads and writes but this does cause the driver to run read-modify-write cycles on the physical device. I would recommend doing your own sector buffering as the examples in later chapters.

Device Manager

The next level of Vinculum-II software, shown in Figure 7.4, is the Device Manager that provides a consistent and standard interface to the underlying on-chip peripheral device drivers and any added device drivers. The API includes Open, Close, Read, Write and IOCtl (IO Control) functions. All devices are accessed using these standard API functions so communicating over the UART is the same as communicating over SPI, or the USB Host for that matter! This will standardize and simplify your application code and make it easier to change your hardware to match what your marketing team has (over) sold. Any differences between peripherals, such as setting the baud rate of the UART, are handled by the IOCtl API function. The read and write functions are used to stream data to and from devices and four DMA channels are available for user applications. Each host controller also has a DMA channel which the driver uses to move data into and out of any specified user data buffers. These read and write requests can be any length since the file system driver handles all USB packet size issues.

Above the kernel level is another FTDI supplied block; these are standard C run-time support such as string handling, ctype handling, stdlib support and stdio support (fopen, fclose, fread, fwrite that use the FAT file system API described above). The only non-FTDI supplied block is the user application which you write using one or more threads and how to do this is the subject of the next Chapter. FTDI supplies an application program template and several examples to get you up and productive with your Vinculum-II as quickly as possible. I have built on these FTDI examples with two chapters of examples following a Vinculum-II tool chain tutorial which is covered in the next chapter.

Chapter 8: Developing Vinculum-II Application Programs

This chapter will focus upon using the Vinculum-II development tools to create application programs. I will create a comprehensive application in stages and will explain the process as we proceed. I will admit that the example in this chapter is somewhat contrived – I did this to illustrate key aspects of writing programs based upon a real-time kernel. I would recommend that you work through each stage of this example and I have provided the source code at each stage for you to study.

FTDI offers a range of development hardware and I always find it better to start with the most powerful system such that we are not constrained. This chapter uses a Revision 1.0 Vinculum-II Evaluation Board with a 64 pin module attached; operation of the Revision 2.0 board is the same but some of the connectors have been renamed (sorry!). Rather that repeat a lot of information in this book, I assume that you have a copy of the datasheet for this board (which is 50+ pages) alongside since I will be making frequent references to figures and text from it. I will use V2Eval: as a prefix when I am referring to this datasheet. Most of this chapter's examples run on the board as is – there is a bonus section later in this chapter where we will add some hardware. The board has a lot of features and I will explain each as we use them.

FTDI provides an integrated Development Environment (IDE) for the Vinculum-II. This is a Windows-based product that should also run on a Mac which has the Parallel's Desktop installed (testing this is on my TO DO list!). The Vinculum-II IDE has a graphical interface that provides context-based menus for a variety of dockable windows. If you are familiar with the Windows Visual Studio environment or the Mac Xcode development environment then the Vinculum-II IDE will be easy to learn. New users should download and read FTDI's *AN_142 Vinculum-II Toolchain Getting Started Guide*.

All Vinculum-II application programs follow the same structure as shown in Figure 8.1. Our main program is at the end, it calls initialization routines which are at the beginning and there are run-time threads are in the center. The first stage of our first example is contained within just two modules but later stages will use many more and they will follow the same structure. This consistency will help us develop and debug larger Vinculum-II examples later on. I will build up this Chapter 8 application program in stages to demonstrate both the process and the tools.

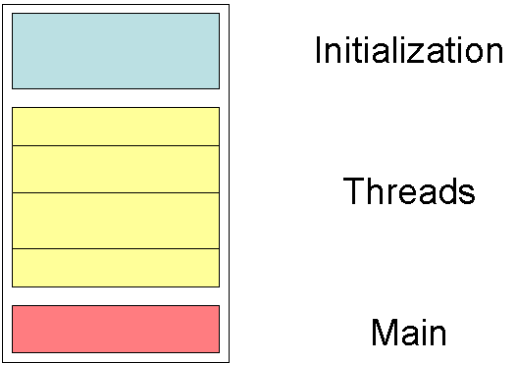


Figure 8.1: All VOS application programs have the same structure

In the first stage of this example we are going to blink an LED Sounds easy but there are several new concepts to learn when using the Vinculum-II. Before diving into the example code, I should make a general comment about Vinculum-II RTOS programming. There is a lot of initialization to do! One downside about having a fully programmable set of peripherals, and a flexible real-time operating system, is that you have to initialize it. VOS provides a consistent view of the on-chip peripherals using a driver interface and this includes IO-Control commands that are used to set the many programmable features of each peripheral. All IO-Control commands use the following format:

```
Iocb.ioctl_code = VOS_IOCTL_TYPE;
Iocb.value = ValueSpecificToType;
Status = vos_dev_ioctl (devicehandle, &Iocb);
```

A returned status value of 0 means NO_ERROR and this is what we would typically expect. It is tedious and takes a lot of code to check status after *every* system call but it is dangerous not to check status at all. So the approach I take is to start with a status value of 0 then OR in the return value then, after several commands, I check status. I am expecting 0 but if I don't get this then I have to back track and resolve the error. I use a debugger breakpoint to trap bad status.

Stage 1: Blinking an LED

Open the Chapter8/Stage1 project and note that it has two modules: `main.c` and `main.h`. First review the header file and note that it contains global declarations. We should now review `main.c` and since execution will start at `void main(void)`, then that is where I shall start my code tour.

There are no global variables to initialize in this first stage so this section is empty.

Next follow two standard functions used to initialize the Vinculum-II real time operating system (VOS). VOS will create data areas for each device driver that we use. This first stage only uses one driver, it will be a GPIO driver called `LEDs` and defined in `main.h`, and we will initialize this in `InitDevices`.

`InitDevices` first checks if I am using a 64-pin device and continues if I am. The first programmable peripheral I set up is the IOMUX. Now why did I choose:

```
vos_iomux_define_output(12, IOMUX_OUT_GPIO_E_1);
```

Remember from Chapter 6 that the Vinculum-II has five 8-bit IOPorts. We only need 1 bit and this must be connected to a physical pin to drive the LED. Refering to V2Eval:Section 10 we see that the V2EvalBoard has four user LEDs. I chose LED3 since it is on pin 12 for all package types (this will make our life easier later in the chapter!). Check the schematic and note that pin 12 is GPIO1 - Vinculum-II groups IO pins in sets of 4 so we must connect this pin to an IOPort using bit1 or bit5. I chose to use IOPortE.bit1.

I then initialize the GPIO device driver and, after checking status, I return to `main`.

The next step is to initialize the program threads – there is only one in this stage. Note that the thread does not start running yet. We are still initializing the environment.

Finally we call `vos_start_scheduler` and if all of the initialization is correct, VOS starts running, else this routine returns indicating that we have an error that we must resolve.

Our Blink thread has further initialization to do! Blink calls `StartupDevices` which opens and configures the GPIO device driver. ***NOTE that device drivers must be opened AFTER the scheduler is running.***

Our Blink program thread, repeated in Figure 8.2, is simple. It contains an initialization section then a loop that is run forever. After waiting for a VOS-controlled delay the state of the LED is toggled.

```
void Blink(void) {
    BYTE PortData = LED3;
    StartupDevices();
    while (1) {
        vos_delay_msecs(512);
        PortData ^= LED3;
// Now write pattern to the GPIO port.
        vos_dev_write(hDevice[LEDs] ,&PortData, 1, NULL);
        }
    }
```

Figure 8.2: The Blink thread is a 'do forever' loop

Using the IDE, now build and download the Chapter8/Stage1 project onto your V2EvalBoard. Refer to AN_142 if you need more information on this process. You should now see LED3 blinking at about 1 Hz.

If you click the IDE 'Pause' button you will be presented with a disassembly listing of the program – this is not a place that I want you to dwell. I have written many Vinculum-II programs, they have all been in C and I haven't even learnt all of the assembler mnemonics! So quickly close this window. If you want to stop execution, choose a line in your C program and click the left margin to set a breakpoint (a red diamond will appear). For now, choose a line within the main loop of the Blink thread and click 'start'. The debugger will stop program execution once this line is reached. You can now single step, look at program variables etc as described in AN_142.

A debug technique I prefer is 'display progress messages' and this is probably due to my early Fortran days where included printf statements were the ONLY debug method available….It is easy to include a 'debug console' into a Vinculum-II application and this is described in the stage 2 of this example.

Included within the FTDI development toolset is a terminal emulator program called V2EvalTerm. Spin this up now, choose a 3000000 baud rate, no flow control and click 'connect'.

Stage 2: Adding a debug console

The V2Eval Board contains an FT4232H and channel A of this component can be connected to the Vinculum-II as shown in V2Eval: Figure 3.2. No additional cables are required since all communications take place on the same USB connection that the IDE and debugger

are using. For our program to use this capability we must include a UART device driver and this is the main addition in stage 2.

Open the Chapter8/Stage2 project and note that there are now four modules. The initialization of the UART is extensive so I moved the `InitDevices` and `SetupDevices` procedures to their own module – you should study this now in `initialize.c`. I had to add UART IO pin routing and UART driver initialization to `InitDevices` and UART configuration to `StartupDevices`. Note that the structure of the UART code is the same as the GPIO code.

You write to the UART using the same `vos_dev_write` function that is used to write the LED. Unfortunately, I found this cumbersome for displaying text strings and variables so I wrote a dprintf function which is shown in `display.c`. I modeled this function after the C printf function but I limited my implementation to only supporting one variable per string. This solves most of my display needs.

I should mention that a real printf function is available in one of the FTDI-supplied libraries. I chose not to use this in Stage2 since it masks some of the points I need to make concerning RTOS program construction.

Use of the dprint function is shown in the Blink thread which is repeated in Figure 8.3

```
void Blink(void) {
    BYTE PortData = LED3;
    StartupDevices();
    dprint("Blink has started\n", 0);
    while (1) {
        vos_delay_msecs(512);
        PortData ^= LED3;
// Now write pattern to the GPIO port.
        vos_dev_write(hDevice[LEDs] ,&PortData, 1, NULL);
        dprint("LED %s", PortData & LED3 ? "Off":"On");
        }
    }
```

Figure 8.3: Adding progress messages to the Blink thread

Now build and download the Chapter8/Stage2 project onto the V2EvalBoard. Watch LED3 blink as before, and you will now see the new progress messages in the V2EvalTerm window.

Stage 3: Hello World

In Stage 3 we add another task, called HelloWorld, which also wants to use the dprint capability that we added in Stage 2.

When you create a thread you must assign a priority to it. A value of 1 is the lowest and the maximum value for a user task is 32; VOS uses higher priorities for its internal operation. Should VOS find more than one task on the ReadyToRun list (Figure 7.3) then it chooses the higher priority task to run. If two, or more, tasks have the same priority then it will select each in a round-robin fashion. In general you should set output threads with a higher priority than the input rates.

Open the Chapter8/Stage3 project and view `main.c`. The Blink thread calls `StartupDevices` as before to initialize the UART device driver, but what happens if the HelloWorld thread runs first, before the UART driver has been initialized? The HelloWorld thread has a dependency on the Blink thread and their operation must be synchronized. VOS has a variety of synchronization primitives and the simplest one for this situation is a semaphore.

A semaphore is a flag mechanism that two or more threads can use for signaling. A thread can wait on a semaphore and a thread can signal a semaphore. A semaphore must also be initialized before it is used; I do this in `main` where the `DevicesStarted` semaphore is cleared.

When the HelloWorld thread starts up it will discover that the `DevicesStarted` semaphore is cleared and will therefore wait until it is set. I set this semaphore at the end of `StartupDevices` in `initialize.c`. This will allow the HelloWorld thread to continue. The next thing that the HelloWorld thread does is signal the semaphore again. Although not required in this example it is important if other threads are also waiting on this semaphore. The semaphore mechanism allows threads to signal each other and can be used to control initialization sequences such as this example has shown.

Now build and download the Chapter8/Stage3 project and watch LED3 blink as before and see progress messages from the two threads in the V2EvalTerm window. If you look carefully at the progress messages you will see that occasionally the messages from the two threads are intermingled. This is typically NOT what you want and the mechanism to stop this intermingling is the subject of stage 4.

Looking at the implementation of dprint in `display.c` you will see a `Localbuffer` declared in global memory. This is a common practice in non-RTOS software but can only be used with caution in an RTOS environment where more than one thread can access this shared data. It is not obvious that we have two threads with this ability (which is why I am using this example) but note that the dprint procedure can be called from both threads. You may be thinking, but I only have one CPU, it is not possible for both threads to run at the same time. WRONG. Welcome to the world of pre-emptive, priority-based, multi-tasking operating systems; they can do you a lot of good, but, as with all power tools, if you are not careful then you can cut your hand off!

An interrupt can occur at any time and the beauty of an RTOS is that this is a managed event and is one of the key benefits of using an RTOS. An interrupt can cause the currently running thread to be suspended and another thread started. In this Stage 3 example imagine that the HelloWorld thread was running and is half-way through a dprint operation when a timer interrupt occurs causing blink to run. Blink also calls dprint so we now have two threads both executing dprint, at the same time!

Yes, this is a contrived example but I designed it to illustrate the fact that you should **treat all threads as if they are running concurrently – since they are!**

And of course, this is how an RTOS is DESIGNED to operate. VOS has mechanisms to protect shared variables and this is illustrated in Stage 4.

Stage 4: the Mutex

Stage 4 introduces the mutex which is an RTOS mechanism used to provide **MUT**ual **EX**clusion and thus stop multiple threads from accessing the same resource.

Open the Chapter8/Stage4 project and view `display.c`. Notice how the body of the dprint routine is now protected by a mutex. The function `vos_lock_mutex` checks if a mutex is locked and, if it is locked, it waits for it to unlock before locking it again. If the mutex is not locked then the function locks the mutex and returns. There is a `vos_unlock_mutux` near the end of the dprint routine. A mutex must be initialized before it is used and this is usually done in `main`.

So what happens differently now that we have a mutex? Recall that the HelloWorld thread was executing the dprint procedure when it was pre-empted by a timer interrupt; the Blink thread starts to run and it calls dprint. The Blink thread try's to set the mutex but discovers that it has already been set (by the HelloWorld thread when it started executing the dprint routine). The Blink thread is blocked by the locked mutex so VOS starts HelloWorld running again. HelloWorld finishes its dprint operation and unlocks the mutex. VOS will now restart the Blink thread which sets the mutex again and continues with its execution of the dprint routine.

Now build, download, and run the Chapter8/Stage4 project and note that the progress messages from the two threads are now not intermingled.

Stage 5: Thread Activity Monitor

We have covered some key concepts such as threads, priority, semaphores, pre-emption and mutexes. But most of the activity is happening within VOS and we don't get to see it. Yes, we have a blinking LED3 and ample progress messages being displayed but I want to **SEE** the real time operation of the VOS so I created a Thread Activity Monitor which is the subject of the Stage 5 example.

The concept of my Thread Activity Monitor is simple: I will use an 8-bit IOPort and I will assign a bit to each thread; when the thread is running it will set this bit and when it passes control to VOS it clears this bit. I then attach a logic analyzer to this IOPort and I can then capture thread activity in real time. Note that this approach is not perfect since I do not detect the pre-emption case when VOS starts up a higher priority task but I found the tool enormously educational and I am sure that you will too.

Using a Logic Analyser

I have used the USBee line of products (see www.USBee.com) since their first introduction and my USBee DX is in constant use. You saw it in Part 1 of this book and you will see it again in later chapters – I find the tool invaluable. In this chapter I will use the DX tool in its logic analyzer mode to trace 8 signals initially (more later, the DX supports 16 digital inputs and 2 analog inputs) and these are connected to the V2EvalBoard as shown in Figure 8.4. I am impressed by the DX tool's human interface – it collects data up to 24MHz and you can view this as say, six seconds across the screen or zoom in to show 6 microseconds across the screen. You choose

the zoom level to allow you to focus upon the level of detail that is important at the time. The trace can also be saved as a file and emailed to someone who then uses the USBee DX software as a viewer. I have used this ability countless times to shown both clients and vendors some timing issue with a product.

Figure 8.4: Connecting the USBee LA to the V2EvalBoard

We have driven IOPorts before and know that this requires the IOMUX and GPIO device driver. The LA pins need to be bi-directional since we do a read-modify-write operation to set or clear individual bits. So how do we handle bi-directional signals on a Vinculum-II?

We have two choices in handling bi-directional signals: we could use a single IOPort and switch its direction at run-time (set IO to input for a read, then set to output before a write) or we could use two IOPorts with one set as a input and the other set as output. I tried both methods and concluded that using two IOPorts resulted in smaller code that was easier to explain. The Vinculum-II has five 8-bit IOPorts and, so far, I have only used 1 bit on 1 of the ports so I decided that using two of the IOPorts for my LA function was OK.

Each thread is assigned an ID (how is described later) which is a single bit: 1,2,4,8,16, or 32 is enough for 6 threads. The `ThreadRunning` procedure will set the corresponding bit on the Logic Analyser port and the `CallVOS` procedure will clear it. I chose IOPorts C and D for my LA function and the pin routing is added in the `InitDevices` procedure within `initialize.c` and the attachment of two GPIO device drivers is covered in the `StartupDevices`

procedure. You should review these now and note that this is 'more-of-the-same'. Yes, the bi-directional aspect is new but it follows the same format as previous IOMUX commands.

Open the Chapter8/Stage5 project and view `TAM.c` since this is the core of the Thread Activity Monitor. I show part of this code in Figure 8.5 for convenience.

```
void ThreadRunning(BYTE ThreadID) {
// The thread now has the CPU
    BYTE PortData;
    vos_dev_read(hDevice[LA_In], &PortData, 1, NULL);
    PortData |= ThreadID;
    vos_dev_write(hDevice[LA_Out], &PortData, 1, NULL);
    }
```

Figure 8.5: The Thread Activity Monitor toggles IO pins

I felt it important that we understand the effect of the TAM routines on the overall program performance. We are adding to existing code to increase observability and, as Hiesenberg stated, this will change the system that we are trying to measure. We don't want to change it significantly so we should measure the impact of the added code on system operation. Check that the first line of main.c is:

```
#define Test_TAM 1
```

This causes only the TAM_Test thread to be created, see line 96 to 102 of `main.c` The `vos_create_thread` system call allows parameters to be passed into a thread; the 4th value is the byte count of values (we are just passing a byte) and the 5th and subsequent values are passed into the thread as parameters (we pass in a 1). The TAM_Test thread, also shown in Figure 8.6 simply calls our two TAM routines.

```
void TAM_Test(BYTE ThreadID) {
// Test VOS response times just toggling the LA signals
// Other tasks are not running for this test
    StartupDevices();
    while (1) {
        CallVOS(ThreadID);
        ThreadRunning(ThreadID);
        }
    }
```

Figure 8.6: A test to understand the impact of the TAM code

Build, download and run Stage 5 and capture the activity on the IOPort using the USBee DX (or similar Logic Analyser). Figure 8.7 shows a close up of the waveform I collected and also a view covering about 1.5 msec. As seen, the thread activity signal is toggling at about 40 usec – this is pretty impressive since the TAM routines need to make two VOS calls to toggle a bit. And 40 usec is small enough such that it can be considered an insignificant impact to our program.

Note too that the 40 usec interval lengthens to about 70 usec at regular 1 msec intervals – this is the VOS timer interrupt and we see that the service routine time is really small. FTDI have obviously tuned their kernel for best performance!

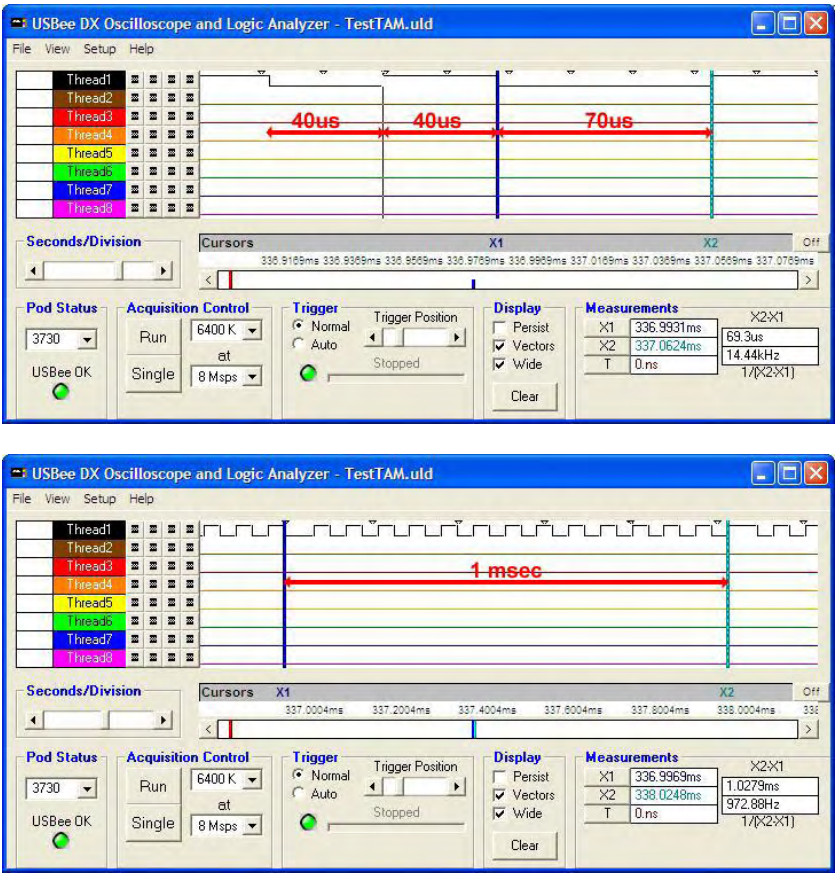


Figure 8.7: The TAM code is a minimal impact on performance

It is now time to look at the activity of our threads so stop the program and change the first line to be

```
#Define Test_TAM 0
```

This will enable a new thread that I created for illustration purposes. MyIdleTask is a thread that uses 100% of the CPU – it toggles bit 7 of the LA port to show activity. I set the priority of this thread to be the lowest value, 1, so that it can be pre-empted by any other task. The VOS Idletask has a priority of 0 and will not have the opportunity to run due to MyIdleTask. I have effectively replaced the VOS IdleTask with mine. Therefore, toggling on bit 7 of the LA trace will indicate that VOS is running MyIdleTask. Now check inside the 'do forever' loops of the Blink and HelloWorld threads. I added TAM instructions to produce activity on the LA port. I also added additional instructions within `display.c` to show when the mutex lock was ON and OFF – we will see the result in a moment.

Build, download and run Stage 5 and collect a LA trace. Figure 8.8 shows a zoom-in detail of thread 1 = Blink operation. Note the time between the X1, X2 cursors is about 1msec. I saved this USBee file in the examples directory so that you can study it.

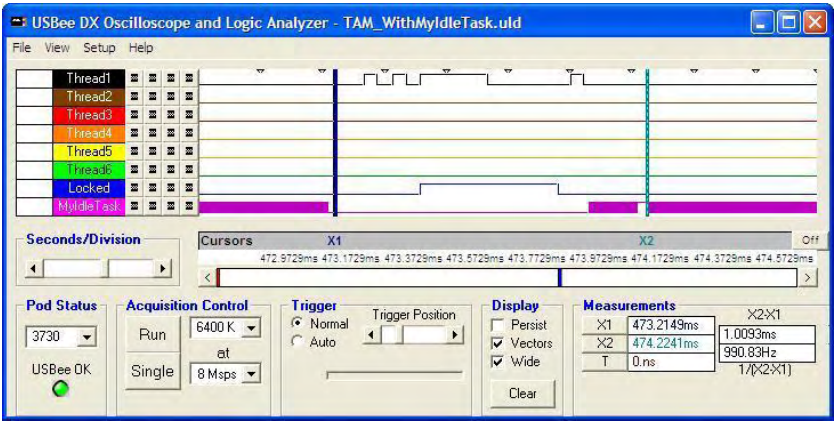


Figure 8.8: Detailed view of thread execution

An overview would show that the program is spending >99% of its time in the IdleLoop – we should expect this for such a simple example. The zoom-in detail shows that Blink started as a result of a timer interrupt and it locks the mutex while writing its data to `LocalBuffer` and then sending this on to the UART driver.

Stage 6: Adding Buttons

We shall now add buttons to our example so that I can illustrate the VOS-way of handling buttons. The V2EvalBoard has 4 user buttons as shown in V2Eval:Section 5.12 and I chose SW3 and SW4. SW3 is connected to pin 14 of a 64-pin device which is IOBUS3 so this needs to be attached to bit3 or bit7 of an IOPort. Similarly SW4 is on pin 32 which is IOBUS20 so this needs to be attached to bit7 or bit3 of an IOPort. The Vinculum-II has five 8-bit IO ports but note that PortB and PortA have additional capabilities with respect to interrupts. PortB includes the ability to wait for a rising, falling or changing edge on any of the 8 bits so I chose PortB to handle my two buttons.

Load the Stage6 project and view `initialize.c`. Note the added pin routing for SW3 and SW4 in `InitDevices` and the attachment of another instance of the GPIO driver to manage the buttons. We now have four instances of the GPIO driver – VOS will use the same code block for each instance but will allocate a separate data area for each instance.

Now view `main.c`, where I have added a Faster thread to manage SW3 and a Slower thread to manage SW4. I also removed the HelloWorld thread and the TAM_Test thread since they have served their illustrative purpose and are not needed any more. The Vinculum-II has five interrupts that can be waited upon, four for individual bits on PortB and one for a change on PortA. If you have up to 4 buttons then connect them to PortB. The next 8 buttons would be connected to PortA and some polling software will need to be added to discover which of these additional 8 buttons was pressed. If you have more than 12 buttons then these will be handled using an external peripheral and this is described in Chapter 11.

The Faster and Slower buttons modify a global variable, called Delay, that Blink uses as a parameter in `vos_delay_msecs`. A purist would control write access to Delay using a mutex but since this situation can only occur if both the faster and slower buttons are pressed exactly together and this results in no change in Delay then it doesn't matter if the value if increased/decreased for a brief moment.

Now build, download and run Stage6. Press SW3 and then SW4 to see the blinking period of LED3 change. You should also capture several seconds of LA trace and observe the running threads. My capture showed that the IdleTask still dominates – there is ample CPU power to do many more things!

Stage 7: Kitt scanner

Stage 7 is optional and may be skipped. It involves building a piece of hardware as shown in Figure 8.9. This is a 10 segment LED bar (I use 8 of them) mounted on a circuit board with a resistor pack and connector. It is designed to plug into the IO connectors of the V2EvalBoard. I am reluctant to solder onto the bread board area of the V2EvalBoard since I do many projects and you may be the same.

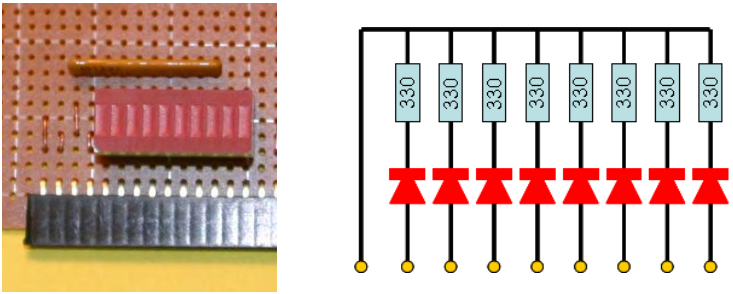


Figure 8.9: An LED bar module built for Stage 7

I liked the scanning pattern of FTDI's Kitt example and I have added a thread called Kitt, to our Stage 6 example code. It uses the same Delay variable as Blink but I would expect you to run this much faster to get the desired visual effect.

We need 8 pins to connect the LED bar module to and the example so far is using Connector CN3 (V2Eval: Section 5.2.1) for SW3, SW4 and LED3, connector CN5 for the UART (V2Eval: Section 5.2.1) and is using the connector CN7 (V2Eval: Section 5.2.5) for the Logic Analyzer connection. This leaves CN4, CN5, or CN6 to connect to our LED bar module. I decided to drive both from PortA so you can connect the LED bar module to CN4 or CN6. Note that if you build more than one LED bar module then be careful note to draw more than 500mA from the USB connection to the PC.

Open the Stage 7 project and view `main.c`. The new Kitt task is similar to the Blink task except that it does a little more processing to generate the scan pattern. `Initialize.c` is extended to include the additional pin routing and initialization of yet another GPIO driver.

Build, download, and run Stage 7 and watch the scanning kitt pattern. This can be run faster by pressing SW3. Again this simple thread adds hardly any load to the CPU which still has over 95% of its time spent in the IdleTask.

Stage 8: Using DIL modules

Stage 8 is also optional – I move our stage 6 example onto a different development platform and this stage discusses the changes made to the program. Figure 8.10 shows the new target hardware – it is a V2DIP1-32 module, a debugger module and a TTL232R cable (yes, the same cable that we used in Part 1!) used to create a serial port connection to the PC. From the PC's perspective I need 2 USB ports, one for the Debugger module and one for the TTL232R cable but the IDE and V2EvalTerm both recognize this hardware and readily connect to it.

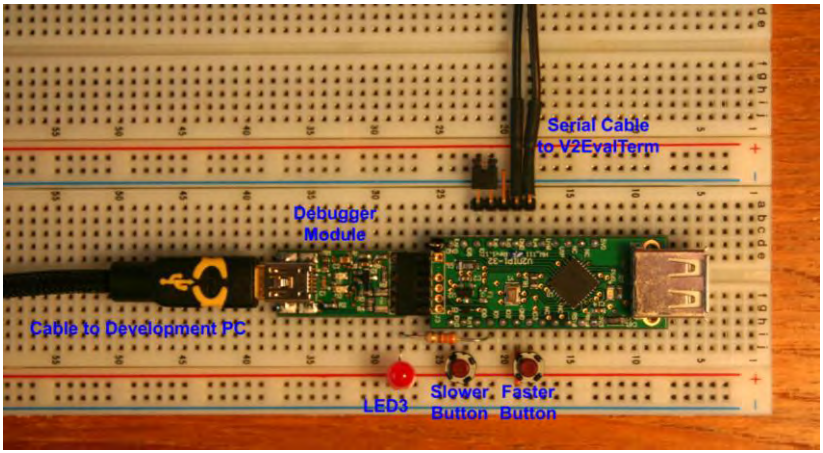


Figure 8.10: The same code runs on the DIL modules too

The solderless breadboard shown in Figure 8.10 also has the equivalent of LED3, SW3 and SW4 connected. I had to forgo the logic analyzer connection since I didn't have enough IO pins. If you have a 48- or 64- pin DIL module then you could replicate the Logic Analyzer connection.

Load the Stage 8 project and view `initialize.c` since all of the changes to run on new hardware are in this module. Note that Ports D and C are still being used for the Logic Analyzer but I have not connected these ports to physical pins (because I didn't have enough). I could remove the code from within the TAM routines but, since their impact on system operation is so small, I decided to leave the code as is, to highlight the minimal changes required to the source code when moving between Vinculum-II development modules. All that is typically required is changes to `InitDevices`.

Now build, download and run the Stage 8 example on the new hardware. LED3 blinks at a rate determined by pressing the faster and slower buttons as before. Progress messages are sent at 3 Mbaud to the V2EvalTerm that displays them as before.

Chapter Summary

I trust that this chapter has given you a good insight into the new concepts and challenges when writing applications programs for the Vinculum-II. The integrated VOS forces a structure to the application program which makes it easier to modify and expand. There is a LOT of initialization to do that sets up our program as a set of threads that use semaphores and mutexes to interact in a controlled fashion. All peripherals are accessed using a consistent device driver API. I wouldn't call it easy yet, but, by the end of Chapter 10, you'll at least consider writing Vinculum-II applications as straight-forward.

Chapter 9: Building a 'Smart Device'

This chapter explores the USB capabilities of the Vinculum-II. We will build a device, show Figure 9.1, that fits between your PC and your keyboard + mouse and can record and playback keystrokes and mouse movements. This comprehensive example will be implemented in stages and VOS programming techniques will be demonstrated on the way. You will discover that developing a USB host application or a USB device application follows exactly the same process as the GPIO and UART applications in Chapter 8; there will be more software to write since we have more data to handle but the staged implementation will make this easy to follow.

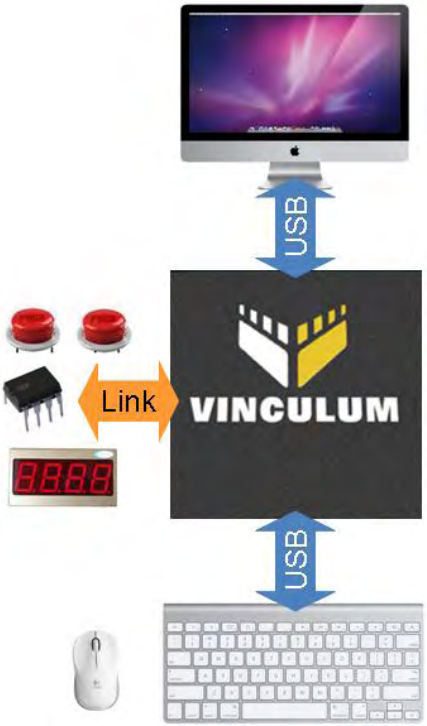


Figure 9.1: The example creates a smart device

Figure 9.1 shows a mouse connected downstream of the keyboard – my keyboard has an integrated hub, many do, but if yours does not then we will add and external hub later in the project (when we add the mouse). This will have no impact on our application

software since the Vinculum-II host driver includes a hub class driver (more details later).

I decided to start with the USB Host driver and the physical keyboard since this makes this example easier to explain. Before we jump into code however, I wanted to recap some essential USB theory.

The Vinculum-II is quite a departure from FTDI's current product line, which I covered in Part 1 of this book. You can design and deploy their current product line knowing little about USB since they are fixed-function devices that implement the details of USB within the silicon and their drivers. Vinculum-II is at the other end of the scale since it can be programmed to be **ANY** USB device and also be host to **ANY** USB device – this means that we need to be familiar with USB concepts such as descriptors and device classes.

There are two good books on USB theory – USB Design By Example by myself and USB Complete by Jan Axelson. I prefer mine but I would recommend that you get Jan's book since mine, even the Second Edition, is now quite dated. I have made several proposals to Intel for updated versions but, unfortunately, we could not make an agreement. The theory in my book covers low and full speed USB and that will be enough for our work with the Vinculum-II but the Win98-based examples do not work on the latest versions of Windows. If the next few pages are confusing then you should pause and read Jan's book, especially Chapters 11 and 12 in the fourth edition, then come back to this book.

A keyboard, and a mouse that we will add to this project later, are examples of Human Interface Devices (HID). A HID need not have a human interface and can be considered as a "generic byte mover" and, since all USB-aware operating systems include a HID class driver, it is a popular interface to implement. The HID specification includes a REPORT descriptor which describes how many bytes are moved between host and device and how these bytes should be interpreted. This report structure is extensive since there are a vast number of diverse device types that fit into this category and the interested reader is also recommended to download the HID Specification and Usage Tables from www.usb.org.

A keyboard and mouse were the first HIDs to be implemented and their definition includes a pre-defined report structure that was called the boot protocol. This is fully defined in Appendix F of the HID

Class Definition. All keyboards and mice are required to support the reports of the boot protocol, shown in Figure 9.2, since the PC's BIOS uses them before the operating system is up and running. Modern-day keyboards and mice have many more features compared with their mid 1990's counterparts so the operating system will switch from boot protocol to report protocol early in its initialization. For some reason the keyboard is required to power up using Report protocol and BIOS must issue a *SetProtocol(Boot)* – this seems backwards to me but since this is what the boot specification requires then we shall comply! My example uses boot protocol only.

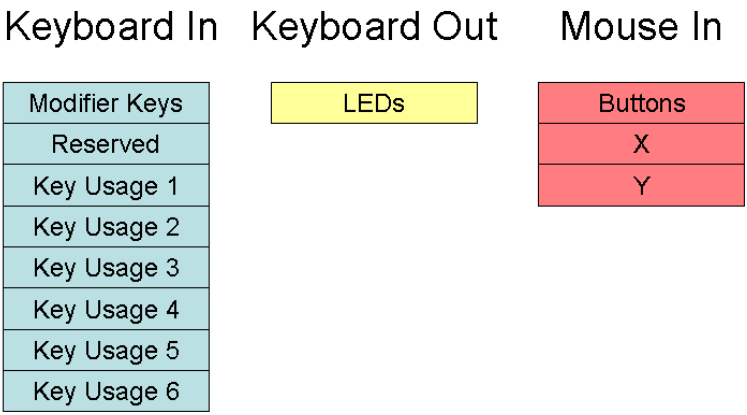


Figure 9.2: All keyboards and mice support Boot Protocol reports

Stage 1: Host Operation

Load the Chapter 9/Stage1 project and view `main.c`. Note that it is similar to Chapter8/Stage6; I started with this project, removed the Faster and Slower threads and added a FindKeyboard thread. I also removed the ThreadID and mutex display from dprint. We saw the effect of the mutex in Chapter 8 and I discovered that it was a rare occurrence for one thread to pre-empt another thread while in the dprint routine. Removing ThreadID simplified the example and that is always good (make things as simple as possible, but no simpler!). I also integrated the `CallVOS` and `ThreadRunning` procedures into a set of "instrumented" VOS calls – these can be seen in `TAM.C`. There is a small change in `Initialize.c` to initialize the USB host controller driver but most of the new code is in modules `host.c` and `DisplayDescriptors.c`.

Once started, the host driver will enumerate the device tree connected to a host port. The host driver includes a HUB class driver and it will correctly walk down all connected hubs and identify devices. It builds a table of device ***interfaces***; remember that a single USB device may contain more than one interface – this is called a composite device and is quite common. The host driver all sums all of the alternate interfaces so the resulting count can be surprisingly large. Most of the work of the new FindKeyboard thread is done in the WaitForKeyboard function in `host.c`, view this now.

After waiting for enumeration to complete we ask VOS how many device interfaces it found – we then query each interface to discover if it is a HID keyboard supporting the Boot Protocol. VOS does all the work for us and there is no real need for us to look at the descriptors. I decided, however, that it would be educational to display the descriptors of each attached device interface so I wrote a DisplayDescriptors function that you can view in `DisplayDescriptors.c`. I have found this very useful during development. I plug in random USB devices into the host2 interface (see V2Eval:Section 5.8) and its descriptors are listed in the V2EvalTerm Window.

Returning now to our FindKeyboard thread in main you note that it doesn't do anything useful after proclaiming that a keyboard was found – this will be the topic for the next stage. Now build, download and run the Chapter9/Stage1 project. LED3 should be blinking and progress messages will be displayed in the V2EvalTerm window. Now plug in a device into the host2 socket of the V2Eval Board and note that its descriptors are displayed. Now plug in a hub and a collection of devices – VOS will correctly enumerate all of them. If a keyboard is one of the devices then this will be indicated on the display. As you can see, it did not take much user software to discover devices attached to a Vinculum-II host controller. VOS is doing most of the work for us!

Stage 2: Serial-to-USB

Load the Chapter9/Stage2 project and view `main.c`. I added a new DisplayReport thread that translates the fixed format keyboard report into ASCII and displays keystrokes in the V2EvalTerm window. The main loop of this thread is shown in Figure 9.3 and it is a typical "data-consumer" thread; it waits at the DisplayReport semaphore, receives ownership of the ReportBuffer, translates and displays the information and then returns the message to the sending thread. This allows GetReports to fill in the next report.

Main loop of DisplayReports:

```
  while (1) {
// Wait for a Report to arrive
    i_vos_wait_semaphore(ThreadID, &DisplayReport);
// This thread now owns KeyboardMessage
// Display Report on V2EvalTerm
    ReportPtr = KeyboardMessage.ReportBuffer;
    Modifier = *ReportPtr;
    ReportPtr += 2;
    for (i=2; i<8; i++, ReportPtr++)
    if (*ReportPtr) HidUsageToASCII(Modifier,*ReportPtr);
// Return ownership of KeyboardMessage
    i_vos_signal_semaphore(ThreadID,
                        KeyboardMessage.ResponseSemaphore);
    }
```

Main loop of GetReports:

```
  while (1) {
// Wait for a report to arrive from keyboard
    Status = i_vos_dev_read(ThreadID,hDevice[Host],(BYTE*)&xfer,
                      sizeof(usbhost_xfer_t), NULL);
    if (!Status) return dprint("Error (%d) ", &Status);
// Send the Report off to be processed
    i_vos_signal_semaphore(ThreadID, Message>SignalSemaphore);
// Need ownership of Message to continue
    i_vos_wait_semaphore(ThreadID, &Returned);
    }
```

Figure 9.3: Threads are consumers or producers

The work to fill the report is handled in GetReports that is called from the FindKeyboard thread. I moved the code to host.c due to its size (I like to keep main.c small and we have a lot more to add incrementing our way through the stages of this example). After some initialization the GetReports main loop, also shown in Figure 9.3 is a typical "data-producer" thread. It sets up a ReportReceived semaphore which VOS will signal once a report arrives from the keyboard then signals the availability of this report using the DisplayReport semaphore.

Figure 9.4 shows a message flow diagram of Stage 2 – there are some key concepts that I must cover before we get too deep into this example. Take a quick peek at the structure of Figure 9.11 which shows the final message flow diagram – there is a lot going on!

Threads are shown in yellow, rounded rectangles and semaphores are shown as smaller, blue rectangular boxes on colored arrows. The arrows depict the flow of messages. But what is a message? And how does it flow?

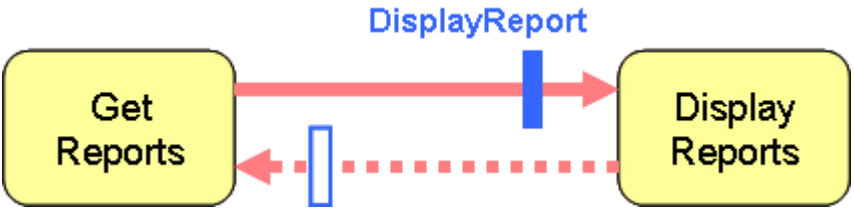


Figure 9.4: Message flow diagram of stage 2

Message Passing

I use the term ***message*** to encapsulate a user-defined block of data (see MessageType in main.h). Think of it as a piece of paper with useful data written upon it. Ownership of this message is important – if a thread owns a message then it may read and write data within it. On startup the GetReports thread owns the KeyboardMessage message and it will write useful data, such as a report from the keyboard, into this message. When the message is complete it "sends" it to the DisplayReports thread – I do this by writing to the DisplayReport semaphore and, using my definition, this means that the DisplayReports thread now owns KeyboardMessage so it can read and write within it. GetReports is not permitted to change KeyboardMessage after it has passed it to DisplayReports – this is a rule I define.

Once DisplayReports has finished with the message it sends it back to GetReports so that it can write the next report on it. Again think of a message, or piece of note paper, passing between the two threads as the mechanism to exchange data. I implement this using two semaphores and a shared MessageType data structure. No data actually moves, in fact, I keep data copying to a minimum for good system performance, but this concept of message passing makes it easier to understand the flow of data amongst the threads.

There is nothing that prevents the thread that does not own the message from reading or writing within it. This will cause data corruption and may cause our program to fail. I did consider protecting my messages using a mutex but the added lock and unlock VOS calls made the implementation cumbersome. I also realized that using a mutex does not GUARENTEE that the message data is not written by the non-owning thread. Using a mutex is a programming convention which the programmer can break by changing a

"protected" variable without first obtaining the required mutex. Using my message passing scheme requires adherence to another programming convention – **if you don't own the message then don't change it**. You own a message either by definition at startup or by receiving it from another thread. You give up ownership by sending it to another thread.

The observant reader will discover that the program names the thread FindKeyboard and this, in turn, calls a procedure called GetReports. Once a keyboard is found the FindKeyboard thread will spend all of its time in the GetReports procedure, in fact, the last few lines of the GetReport procedure. So I felt it reasonable to call this the GetReports thread in Figures 9.3 and 9.4.

Now build, download and run the Chapter9/Stage2 project. Attach a keyboard and type on it. The keystrokes that you enter will be displayed in the V2EvalTerm window. We have a serial-to-USB keyboard; interesting but probably not worth making into a product. Most people want a USB-to-Serial adaptor (ie FT232R) rather than a Serial-to-USB adaptor.

This stage2 example has shown that it is straight forward to attach a HID to the Vinculum-II – we shall see in the next chapter that connecting other device classes is also easy.

Stage 3: Device Operation

The Vinculum-II support two USB channels and either can be a host controller or a device. Let's now look at creating a HID using the other USB port. Note that you must disconnect jumper JP5 (see V2Eval:Section 5.13) and we will need to attach the USB gender changer as shown in Figure 9.6. I should mention that FTDI has a ready-built driver that makes the Vinculum-II operate like a FT232 device – this works with the same PC drivers as a real FT232R so bulk data transfers are already done for you (see FTDI's FT232R Slave example which is provided with the IDE).

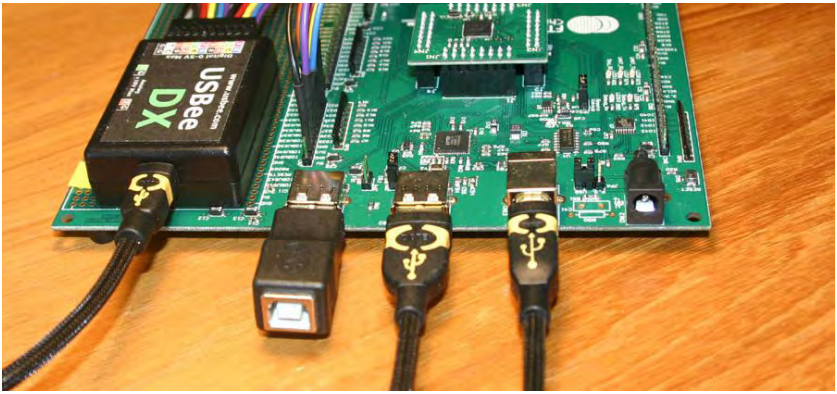


Figure 9.6: USB device development uses a USB gender changer

All of FTDI's documentation uses the term "slave" when describing operation as a USB device – I personally don't like the term but acknowledge that the term "device" is over-used and can be misunderstood so, for the remainder of this book, I shall also use the term "slave."

Load the Chapter9/Stage3 project and view main.c. The EnumerateSlave thread waits for Setup packets to be sent from the PC then handles them. Most of the slave code is in Slave.c and you should view that now. All USB devices use descriptors to define their identity and the descriptors in slave.c define a HID keyboard with boot protocol. The report descriptor describes the boot report format shown in Figure 9.2. During enumeration the slave will receive standard "Chapter 9" setup packets and also HID Class setup packets and the procedures in slave.c provide the correct responses. Initialize.c contains additional code to initialize the slave driver and to get handles for the slave's control and data endpoints.

Now build, download and run the Stage3 project. I would recommend a USB bus spy, such as the Ellisys Tracker shown in Figure 9.7 when working at this low level of USB. The bus spy is connected in series with the V2EvalBoards connection to the "target" PC and it is controlled by a different "development" PC as shown in Figure 9.7. The Stage 3 program will enumerate our device/slave and the target PC will proclaim that another keyboard has been added. Using a bus spy, and its required PC, is not essential for this example but it will save you debug time when developing your own descriptors.

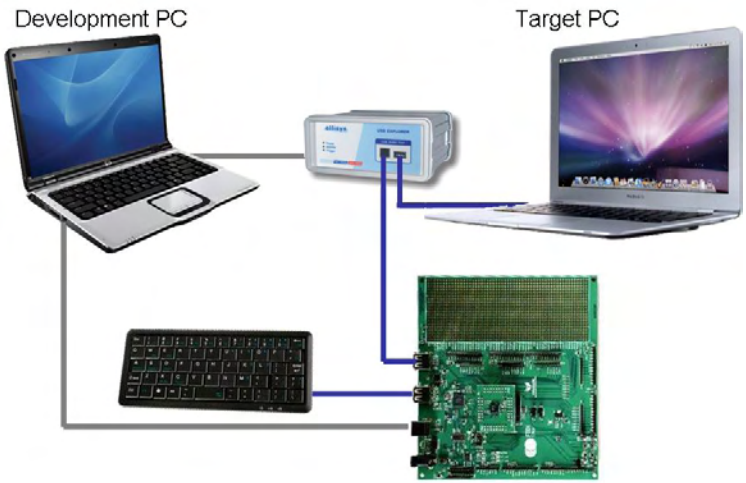


Figure 9.7: I recommend a bus spy for detailed USB work

Stage 4: Auto-typer

Now let's get our new keyboard device to do something! Load the Chapter9/Stage4 project and view main.c. I have added two threads, CreateKeystrokes and SendReports, that share a KeyboardMessage and a SendKeyboardReport semaphore. CreateKeystrokes is the producer thread that generates keydown and keyup reports exactly like a real keyboard does; it sends these to SendReports, a consumer thread, which sends them to the target PC.

Open a word processor on your target PC and then build, download and run the Stage 4 project. The V2Eval board will happily type "Hello World" into your word processor all day! We have created a USB device that operates like a keyboard.

Stage 5: Host+Slave Operation

We shall now combine Stages 2 and 4 to create stage 5. Load the Chapter9/Stage5 project and view main.c. I used the FindKeyboard thread from Stage 2 which now sends its reports using KeyboardMessage to the SendReports thread from Stage 4. The code within each of these threads is unchanged. I am combining two previously written threads to create new functionality. This is possible since each thread is self contained and exchanges data using my standardized message passing mechanism.

Build, download and run the Stage5 example code. You can type on the keyboard attached to the V2EvalBoards host2 port and the slave1 port will appear to the target PC as a real keyboard. We have the Vinculum-II operating as a short circuit!!!

I want to slow down a little here and recap what we have achieved so far. We have used Vinculum-II's USB host driver to enumerate real devices connected to USB port 2. We identified a keyboard and wrote a thread which collects keystroke reports. We created a message containing information about a keystroke report and sent this to another thread that used Vinculum-II's USB slave driver to send keystrokes to a target PC connected to USB port 1. We also have a debug console attached to Vinculum-II's UART driver which displays progress in V2EvalTerm's window. The result is keystroke reports passing into the Vinculum-II and out the other side – it's a short circuit but we have access to all of the data involved in these exchanges. Let's see what we can do with this data.

Stage 6: KeyCatcher

It is easy to add a thread between the GetReports thread and the SendReports thread, in fact, neither thread will know that we have done this and will operate unchanged. This new ForwardReports thread will forward the KeyboardMessage unchanged and SendReport will return it as before. ForwardReports will use a RecordMessage to make a copy of each keystroke report and will send this to a Record thread that will store the reports, and the time intervals between them, in a Flash memory. The data flow diagram for this keycatcher device is shown in Figure 9.8. Note that we now have two messages, shown in different colors, circulating around these threads.

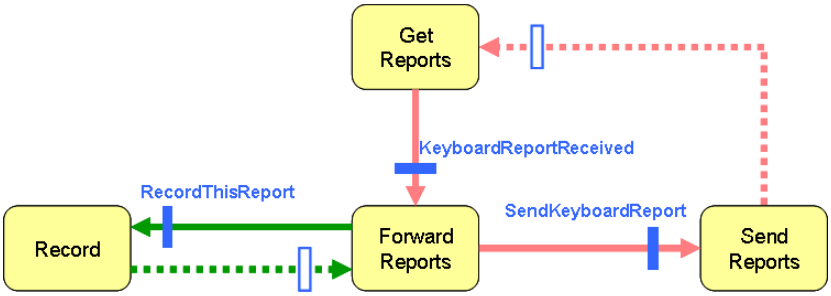


Figure 9.8: Message flow diagram of stage 6

I chose an Atmel Dataflash to store the keystroke data. I used a low-cost 2MB part but you can get these up to 32MB. The part uses an SPI interface so we need to add Vinculum-II's SPI Master driver to our project. I built a plug-on board, shown in Figure 9.9, that connects to C9 of the V2EvalBoard – see V2EvalBoard:Section 5.3. I also added a Timer driver that measures the elapsed time between keystrokes. Both drivers are started and initialized in initialize.c.

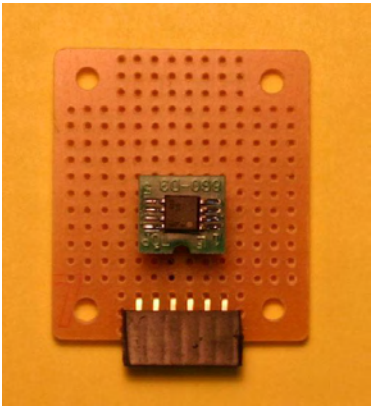


Figure 9.9: A small add-on board holds the Atmel DataFlash

Load the Chapter 9/Stage6 project and view main.c. I made a small change to the FindkeyboardThread – it initializes the KeyboardMessage.SignalSemaphore to a new semaphore; this will cause GetReports to send completed keyboard reports to a different location. A new thread, ForwardReports, makes a local copy of the keyboard report before sending it off to the SendReports thread. The ForwardReports thread owns RecordMessage at startup and uses it to send keyboard reports to another new thread, Record.

The Record thread has the most work to do and it owns writing Entries to the Atmel DataFlash. The definition of EntryType (see main.h) allows Record to save both keyboard reports and mouse reports (these will be added in Stage9). Elapsed time between keyboard/mouse reports is also recorded so that later playback will be at the same tempo as the original recorded data. The Atmel DataFlash is an impressive part – it saves data in 512 byte blocks and it has two 512 byte buffers to stage data before writing a block. The Record thread fills one buffer then, while this buffer is being copied to flash memory, it fills the other buffer. This approach effectively hides the long flash write time.

Build, download and run the Stage6 project. Externally you will not see a difference from Stage5 but the debug messages will explain what is going on. I debugged this module using three channels of the USBee DX connected to the SPI lines on V2EvalBoard:C9. I looked for the Record thread being active then turned up the resolution to see the SCLK, MISO, MOSI, and CS signals. The USBee DX includes an SPI decoder so it was easy to check that all signals were correct.

We now have the basics of a keycatcher product working. The next stage adds a Record Button so that we can control when data is recorded.

Stage 7: Add a Record Button

We already know how to manage buttons from the Chapter 8 example but this is not the complexity of this stage. The data flow diagram of Stage7, shown in Figure 9.10, shows that the Record thread must now respond to two semaphores. This is something that we have not done before.

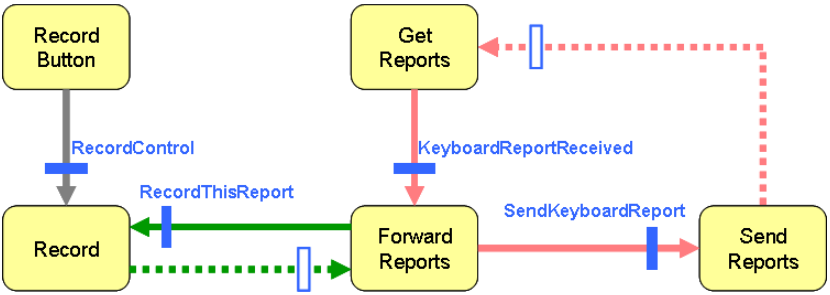


Figure 9.10: Message flow diagram of stage 7

Load the Chapter9/Stage7 project and view main.c. Note that I have added a RecordButton thread (cut, pasted and renamed from Chapter 8) that signals a RecordControl semaphore when the button is pressed – the button is implemented as a toggle function for Record. I also changed the Blink thread to keep LED3 on during record so that it is easy to know when keystrokes are being recorded. All of the new work in the Record thread so you should now focus there. Note that VOS supports waiting at a list of semaphores. A list is created with some control flags – in this example I will wait on ANY semaphore but you can also wait on ALL of the semaphores being signaled before proceeding. You use a vos_wait_semaphone_ex call

to wait on a semaphore list and, when it returns (in ANY mode) it provides an Index into the semaphore list indicating which semaphore was signaled. Since I have only two entries in my list then I will receive an Index of 0 or 1 which my code treats as a Boolean. I will wait on three semaphores in Stage 9. There I also convert this semaphore list initialization into a function that I move to Support.c.

Build, download and run the Stage7 project. You can now press SW3 to enable and disable recording. LED3 will stay on during recording.

Stages 7.1 & 7.2: Short diversion to reclaim memory

When I first started stage 8 I realized that I had a **BIG** problem: my stage 7 solution had used most of the available 16 KB of RAM and there was no room for more threads! I had plenty of program space available, I had used 100 K of the available 256 KB of flash but I had to do something about RAM usage. I added instrumentation code to stage 7, creating stage 7.1, which showed how the RAM was being used. The results are shown in the left side of Figure 9.11.

To my surprise the `.datarам` segment was almost 2 KB! The `.datarам` segment includes initialized data which is set up in ROM and copied to RAM at power up. Investigation showed that most of this was the text for my dprint statements; I was not expecting this since other compilers that I use put this constant data in the code segment. I tried to force the compiler to place my constant strings in the code segment by replacing the `dprint("Message", DataPtr)` with `dprint(rom "Message", DataPtr)` but the compiler complained with "unexpected ROM". To get to get my constant text in ROM I had to convert my dprint statements into two lines;

```
rom char Message1[] = "Message";
dprint(&Message1[0], DataPtr);
```

This works but it is not as elegant as `print(rom "Message", DataPtr)` so I have requested that FTDI extend the syntax of the rom keyword usage. For now, however, will have to go with the two-line declarations. I changed the dprint function to handle rom char and also all the message declarations and called this Chapter9/Stage 7.2 which you should now review.

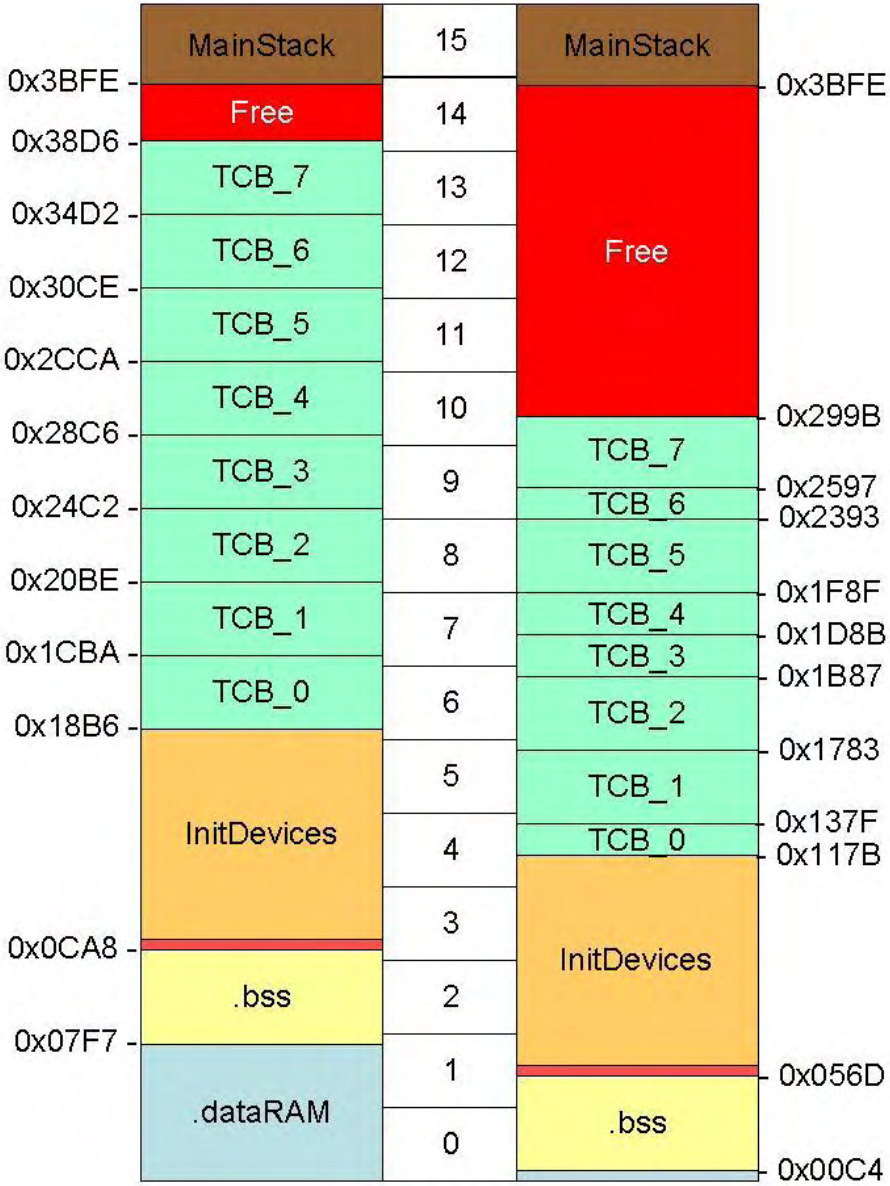


Figure 9.11: Stage 7 RAM usage, before and after

The other big RAM user is to stack space for each of the threats. VOS includes some diagnostic features (see DIAGNOSTICS section of vos.h) that can help us here. VOS initializes the stack of each thread to a known pattern and the VOS call `vos_stack_usage()` steps through the stack to determine usage; I added code in the Blink routine that displays systems statistics after the program has been running for about a minute. My test showed that some threads use little stack space so I could allocate a smaller TCB to these; this is also included in stage 7.2

FTDI have an applications note, AN_157 Vinculum-II Memory Management, which also recommended changing the compiler optimization to level to 1. This uses a different algorithm for allocating local variables and had a large impact on reducing the stack usage. So on all projects from now on, I set **Build/Options/VNC2 Compiler/ Optimization Level** to 1. The right side of Figure 9.11 shows the results of moving the dprint text to ROM and using smaller TCB's for some of the threads. I freed up almost 4 KB of RAM.

Another VOS diagnostic feature is a Thread Profiler. Once started it keeps track of which thread was running at each timer tick; it increments a thread-specific counter so I included code to extract and display the count values for each of my threads. Load and run the Chapter9/Stage 7.2 project now and record some keystrokes within the first 45 seconds of operation. After about 1 minute stats will be printed. A little math on the count values shows that the IdleTask is running 98.6% of the time; our key capture application uses only 1.4% of the CPU's time. We have ample scope to add playback and mouse tracking and this is the subject of the next two stages.

Stage 8: Add Playback

In this stage we add a Playback button which will send the keystrokes that we recorded to the target PC. Adding this playback function is about the same complexity as adding the record function and button. From a data flow perspective, we are adding a Playback thread and a PlaybackMessage that will circulate between the Playback and SendReports threads. The Playback thread initially owns PlaybackMessage and it sends keystroke reports that it reads from the Atmel DataFlash to the SendReports thread. The Playback thread is controlled by a Playback button but it does not have to wait at two semaphores simultaneously. The SendReports thread must be extended to monitor two message streams – I use a semaphore list just like the previous stage. So no new theory to learn here!

Load the Chapter9/Stage8 project and view main.c. A PlaybackButton thread is added – this is a cut, paste and renamed from the Chapter 8 example. The Playback thread is also new and you should review that now. Build, download and run this Stage 8 project and press the Playback button. Keystrokes that you recorded in Stage 6 or 7 will now be sent to the target PC as if you were typing them.

Stage 9: Add a mouse

A boot mouse is similar to a boot keyboard from a software point of view – it is a HID that generates a 3 byte report rather than an 8 byte report. This means that our example code does not have to extended very much to support the addition of a mouse. In fact, you will see that, in some cases, we will use the same code but will process different data. Figure 9.11 shows the message flow diagram of Stage 9 with the mouse functions added in a similar way to the keyboard functions. The GetReports(Mouse) thread will own a MouseMessage that it circulates around the ForwardReports and SendReports threads. All threads have to be extended to support the new mouse reports but, as you shall see, this is basically a copy, paste, and edit of existing keyboard functionality.

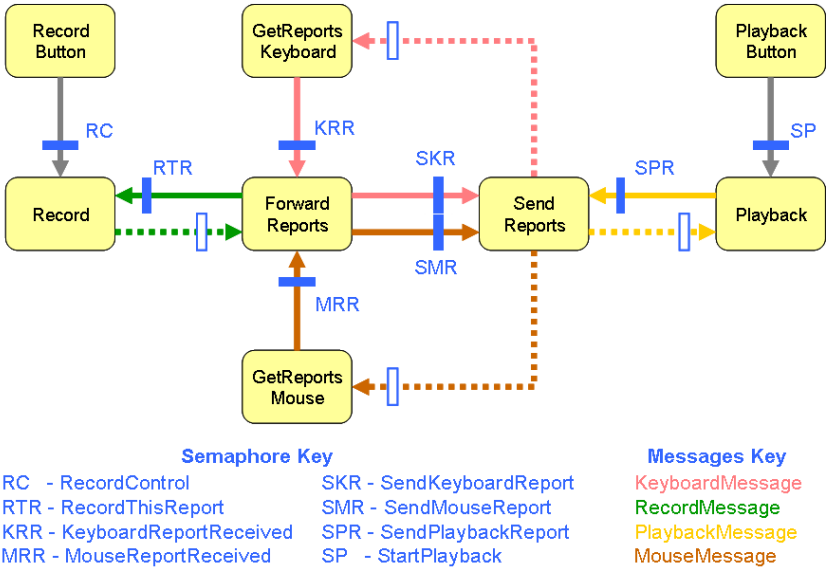


Figure 9.12: Message flow diagram of Stage 9

Open the Chapter9/Stage9 project and view slave.c. Note that I extended the descriptors of our slave device to define a composite device with two HID interfaces. Interface0 is our keyboard as before and Interface2 is a boot protocol mouse that uses EP2 for data transfers. I needed a MouseReportDescriptor to define the 3 byte report and this is shown in slave.c. Most of the Setup code for the Mouse can use the same code as the keyboard since they are similar HID devices – the only real difference is that I must supply a different report descriptor for Interface1.

Looking briefly at main.c you should note a new thread called FindMouse that is basically a copy, paste, and edit of the FindKeyboard thread. The FindMouse thread will call a WaitforMouse procedure which is in host.c – lets look at this now.

I converted the original WaitForKeyboard function into a WaitForDevice function with a few additional parameters. I then have two small front-end procedures, WaitForKeyboard, and WaitForMouse that both call WaitForDevice but with different parameters. So the additional code for a mouse is minor.

In order to additionally connect a mouse to the V2EvalBoard we will need a HUB. Some keyboards have an integrated hub and this is a convenient way to add a mouse. Another convenient way is to use FTDI's FT4232H Hub Module and this is described in Chapter 11. If your keyboard does not have an integrated hub then use any commercial hub product, attached it to the V2EvalBoard and plug a keyboard and mouse into it. The WaitForDevice function will correctly find a keyboard and mouse that are connected via hubs – there is no additional programming required for this situation.

Returning now to main.c and the FindKeyboard and FindMouse threads. Note that they both call GetReports but with different parameters. At runtime we will have two copies of GetReports running, one in the FindKeyboard context and one in the FindMouse context – only one procedure exists in ROM but we are running two copies of it, same code but different data. This is a real savings in code size. For ease of explanation I call these GetReports(keyboard) and GetReports(mouse) threads in Figure 9.11.

The ForwardReports and SendReports threads are extended to accommodate the additional circulating MouseMessage. This involves adding a new semaphore to each thread and extending the semaphore lists in each. The processing of mouse reports is the

same as processing keyboard reports. Finally the Record thread was extended to include processing of the mouse report which is handled in a similar way to the keystroke reports.

Build, download and run the Stage9 project. Press the record button, move the mouse, and type some characters. Press the playback button and watch the cursor move on the target PC and watch characters appear.

Pretty impressive.

If you haven't been checking the USBee DX Logic Analyser trace during each stage now would be a good time to turn it on. Run the program, press the Record button and type some characters while moving the mouse. I found my trace most illuminating! I discovered that the program spends most of its time in the Idle Thread waiting for something to happen. A keypress activates the GetReports(keyboard) thread, the ForwardReports thread then accepts the KeyboardMessage and in turn activates the Record and SendReports threads. I found it warming to see the threads activated in the order that my program designed them to and it was clear that the KeyboardMessage and the RecordMessage flowed as designed.

Ideas for improvement

This chapter has described a comprehensive example that has used USB host, USB slave, GPIO, UART, SPI Master, and Timer drivers. A good next step would be to add more buttons and add routines that allow for multiple individual recordings to be made and played back. This would create a smart device out of a standard keyboard and mouse where multiple repetitive sequences could be recorded and played back to simplify work flow in many applications. There is no new theory to learn to implement this smart device since Stage9 includes all of the routines that would be needed. This expansion is left as an exercise for the reader.

Stage 10: Multiple keyboards

One variant of this example that I wanted to cover before closing this chapter is the handling of multiple keyboards in a Wintel PC environment. If you connect two keyboards directly to a Wintel PC the Windows operating system does an OR on the input and combines keystrokes from the two keyboards into a single input stream. This is typically NOT why you connected two keyboards –

you wanted the separate input streams going to different places. This stage 10 creates “user keyboard” devices that the operating system will not enumerate and exclusively own.

Load the Chapter9/Stage10 project and view slave.c. I only have one keyboard report descriptor but I still have a composite device defined. I have two interfaces both of which are keyboards, one does data transfers on EP1 and the other does data transfers on EP2. The Vinculum-II supports seven data endpoints (EP1 through EP7) so you could extend this example with four more keyboards if desired.

Note that in the Interface1 descriptor I have used 0 for the Class Protocol and SubProtocol. I also changed the report descriptor to a ‘generic 8 bytes’. This will prevent Windows from recognizing it as a system keyboard and, instead, it will treat it as generic HID device. I made a change to WaitForKeyboard in host.c so that identifies two individual keyboards but the GetReports function is unchanged. Now focusing upon main.c you will see two threads FindKeyboard and FindUserKeyboard. I removed the Record and Playback functions for simplicity.

Build, download and run this Stage10 project. The target Wintel PC will correctly enumerate our composite device and typing on one of our external keyboards will result in keystrokes being recognized by the PC. The other keyboard will be ignored.

Now look for the Windows console program, TwoKeyboards, also in the Chapter9/Stage10 directory. The source code for this example is included as a Visual Studio project for your review. Run this TwoKeyboards program on the target PC; it will find our composite device and identify how many additional keyboards it found. Now type on both keyboards and note that keystrokes from one appear on one half of an output line and keystrokes from the other appear on the other half of the output line. Keystrokes from each keyboard are kept together and are not intermingled as in the default case. You can employ the techniques used inside TwoKeyboards to direct input from up to seven keyboards to separate windows in a more sophisticated application program. I have done the “hard bit” using a Vinculum-II as a smart-device and expansion of this application is left as an exercise for the reader.

Chapter Summary

This was a long chapter but we covered an enormous amount of ground. I used the HID class to demonstrate Vinculum-II's ability to talk to a USB device and to be a USB device. Both at the same time! The Vinculum-II tool suite also includes class drivers for Bulk-Only-Mass-Storage (BOMS), Communications, Still Image and Printer. We will use some of these in the next chapter. These drivers and debug tools allow us to focus upon our application program which is developed as a set of communicating threads. The threads are small, focused routines designed to do a single task well and this example has shown that they form re-usable building blocks that can be deployed in a variety of applications. As you gain experience with VOS programs you will quickly learn that using an RTOS is an efficient and productive method to write programs.

So lets move on to Chapter 10 which will connect existing USB devices together. This will definitely show that the sum is greater than the two parts.

Chapter 10: Interconnecting USB devices

This will be a fun chapter - we get to be part explorer, part detective and part engineer. We're going to look at USB devices that you currently use, or would like to use, with your PC and we will re-purpose these to operate in an embedded environment. We saw in Chapter 9 that adding a human interface device (HID) into a Vinculum-II project was easy; now let's look at other interesting USB devices that can be used in an embedded environment.

The range of available USB devices is enormous and, due to "PC economics", they are inexpensive; you can get a lot of functionality for low cost. In this chapter we will replace your PC with a sub $10 Vinculum-II and this cost reduction will open up new product opportunities! The benefit of a PC is that it is able to dynamically support **all** USB devices since it can be loaded with device drivers at runtime. Most of these devices have many configurable options which the PC must understand and select from. An embedded application is different - it has limited memory and is configured at design time. Typically it only has a few devices and the developer will pre-select the operating mode of each device. An embedded application will recognize the device by its Vendor ID (VID) and Product ID (PID) so it is not necessary to read in and parse all of the device descriptors since these are pre-known by the embedded systems designer. This will simplify our coding effort.

I assume that the USB device that we will be adding to a Vinculum-II embedded system is already running within your PC environment. An essential tool for our explorer phase is a USB bus spy. I use an Ellisys Tracker/Explorer but there are many other hardware tools available; Jan Axelson presents a variety of both hardware and software spy tools on her website at www.lvr.com. We are only interested in USB devices that operate at low or full speed since these are the speeds supported by the Vinculum-II. It is likely that your PC has high-speed USB ports so you also need a USB 1.1 hub as shown in Figure 10.1; this will ensure that a high-speed mode of the USB device under test is not activated; the USB specification requires all high-speed devices to also provide functionality at full speed.

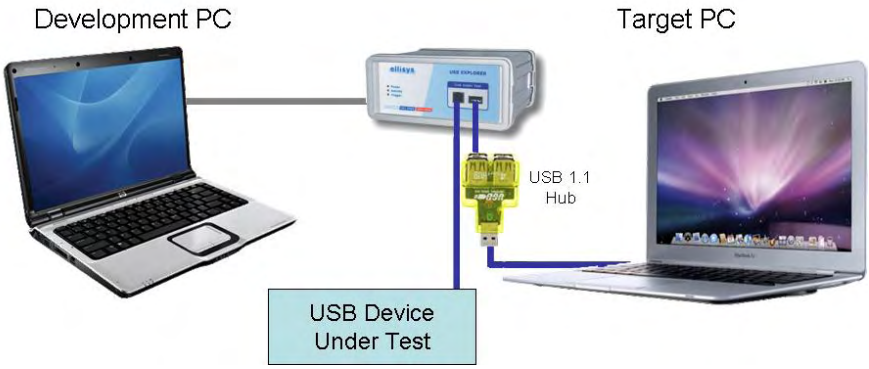


Figure 10.1: Equipment set up to explore USB device operation

Audio In/Out device.

I decided to start by exploring an audio device since this is a complicated device in the PC world but a simple device from a Vinculum-II perspective. I chose a representative USB audio adapter from cmedia as shown in Figure 10.2. This sub $10 part allows me to connect a monaural audio source, such as a microphone, and a stereo audio sink, such as speakers, to a PC. In the olden days this would have been an audio add-in card but today it is a single-chip USB component.

Figure 10.2: Representative audio device from www.cmedia.com

Enable your USB bus spy and attach an audio device to the PC. Once the device has been recognized (note that it will use standard class drivers as supplied with the operating system and you will not be required to supply a driver), run a sound recorder application and record about 10 seconds of audio. You may have to configure your PC hardware if your target PC already contains audio hardware. Now playback this audio. Finally, stop the bus spy and look at the trace of USB operation.

Figure 10.3 shows the general structure of the trace - the specific details will vary depending upon which audio device you selected and, to a certain extent, which OS you are using. The first phase of the trace shows the operating system reading the device descriptors and therefore discovering that this is an audio class device. This phase ends with the OS enabling the device via a SetConfiguration command, and selecting the zero bandwidth setting for the input and output interfaces; SetInterface(1,0), SetInterface(2,0).

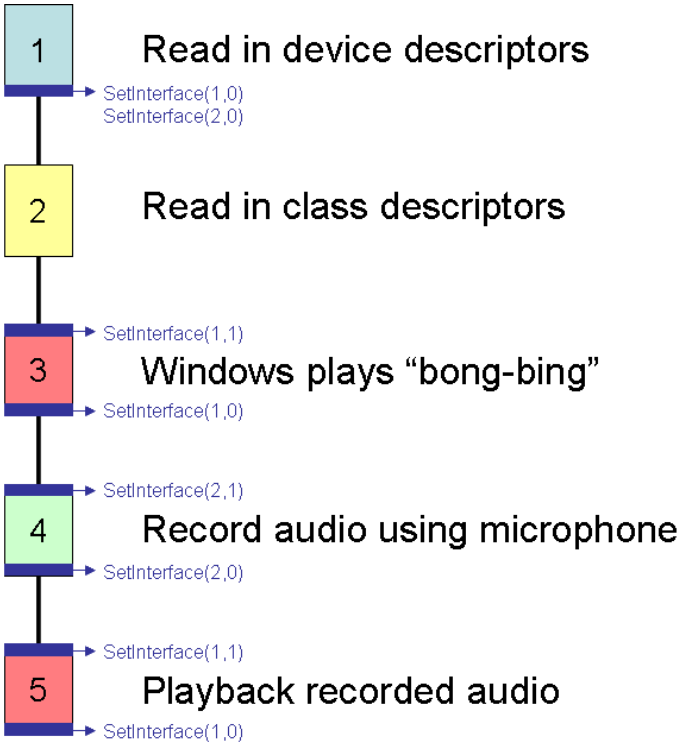


Figure 10.3: Structure of trace of an audio device operation

Note that the configuration descriptor is **much** longer than the HID configuration descriptor we saw in Chapter 9. Most of the descriptors are audio class descriptors used to define the elements and topology of the analog components within the USB interface device. We could use the audio class specification to interpret these descriptors or we could do a Google search on the VID and PID and uncover a datasheet for the product; I chose the latter.

The audio device I selected has a comprehensive datasheet at www.cmedia.com and I replicate the audio function block diagram in my Figure 10.4. The upper portion of the diagram describes a 16-bit stereo output channel that can run at 44.1 KHz or 48 KHz. The lower portion describes a 16-bit mono input channel that can run 44.1 KHz or 48 KHz. The input microphone can be directly mixed into the output stream.

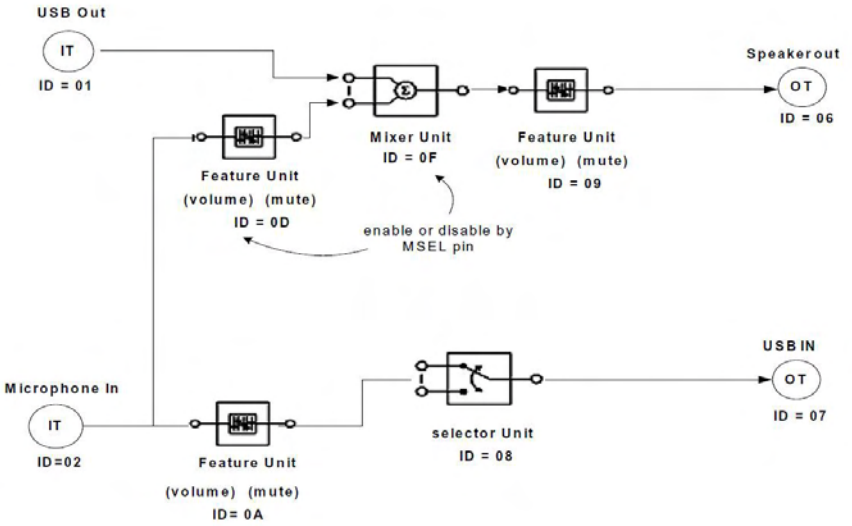


Figure 10.4: Audio topology of my representative audio device

The second phase of the USB trace shown in Figure 10.3 is the audio class driver reading in the settings of the audio hardware. The PC does this since the structure of the audio device is unknown. We will get the same information from the datasheet.

My target PC was running windows XP and phase three is the PC enabling the audio output channel with a SetInterface(1,1) and setting the sampling frequency to 48 KHz so that it can play the familiar "bing-bong" when a new device is detected. The two channel

16 bit, 48 KHz sampling will result in 196 bytes per frame of isochronous transfers. This phase lasts 0.86 seconds and ends when the PC turns off the audio channel using a SetInterface(1,0).

Phase four is the audio recording which the PC enabled via a SetInterface(2,1) and, in my case, it chose a sampling frequency of 44.1 KHz for the single channel recording. This results in mainly 88 byte isochronous transfers with about 10% 90 byte transfers to meet the 44.1 KHz sample rate. This phase ends with a SetInterface(2,0).

Phase five, the audio playback, is similar to phase three.

Wow. That took longer to explain than it took to execute! You will be pleased to hear that the embedded operation of the Vinculum-II is much simpler.

We now put our detective hat on to discover the essence of audio operation. The PC has to do a lot of work to discover the topology and details of a particular audio device. We, as embedded engineers, will choose an appropriate component that meets or exceeds our products requirements. We will identify this using its VID and PID. So there is no need to read most of the devices descriptors. To start and stop audio transfers we need to select the correct alternate interface and set the sampling rate. We will choose 48 KHz since this gives a constant byte count per frame. That's all there is to do for embedded audio input and output!

I have two audio examples that you can build upon. The first records audio onto a flash drive - it is a sound recorder. The second plays back audio from a flash drive - it is a sound player. I only implement a single file in each example but I use the FTDI BOMS (Bulk Only Mass Storage) driver so that will be easy for you to add buttons and save or playback multiple files. Once you see how easy it is to record and playback audio using the Vinculum-II you may consider adding audio cues to your application.

Sound Recorder.

The 16-bit mono, 48 KHz sampling rate produces 96 bytes every frame or 96 KB/sec. I measured the throughput of a range of flash drives and discovered that the data transfer rate was dominated by the MSC protocol; 512 byte data transfers typically completed in one or two frames but the MSC command and data phases added another three or four frames to the transfer. This overhead would be insignificant on large data transfers but the limited buffer space on the

Vinculum-II meant that I could only do single sector writes. The BOMS driver did, however, manage large transfers in a sensible way by keeping local copies of FAT tables and only updating the flash drive when absolutely necessary.

I used a 1.5 KB buffer as shown in Figure 10.5 since sixteen 96 byte transfers fit in neatly as shown. I wrote each sector buffer as it was filled and could typically keep up with the isochronous data rate. If we don't want the recording to skip then we must reduce the data rate by sub-sampling the isochronous data (or choose an audio source with a lower sampling rate!) For voice recording 8 KHz is adequate so saving every sixth sample would make the data rate very easy to manage. If you need 48 KHz sampling then review FTDI's VMusic2 example which uses MP3 compression to reduce the data rate.

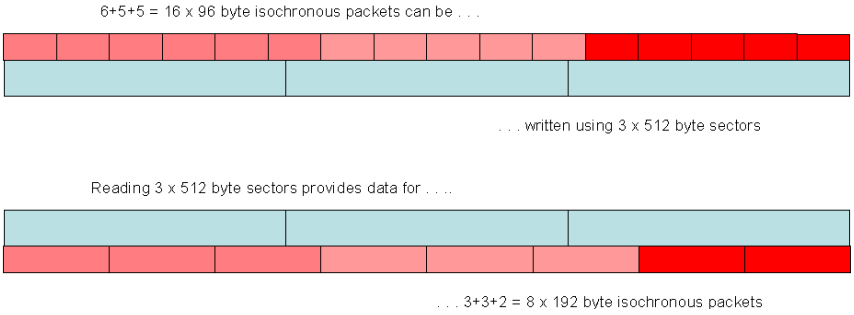


Figure 10.5: Buffering isochronous data and sector writes/reads

Open the Chapter10/SoundRecorder project and review main.c. The application program waits for a flash drive to be connected to host port 2 and waits for the Cmedia device to be connected to port 1. Once both are connected a sound.dat file is created and recording continues until either device is removed or the flash drive is full. The sound that file will not be recognized by the PC since it does not have a standard file format. I wrote a Windows console program that creates a standard Windows WAV file from sound.dat so that the data can be processed by any Windows application. If a helpful reader would like to send me an OS X version then I would be most grateful and will redistribute it to your fellow readers.

Sound Playback

The playback example uses 16-bit, stereo, 48 KHz sampling so there are 192 bytes of isochronous transfers per frame. I had similar issues with the MSC protocol as in the record example even using the 1.5 KB buffer shown in the lower half of Figure 10.5. I created a Windows console utility that removes the header from a standard WAV audio file so the audio generated on a Windows PC could be played back using the embedded Vinculum-II example. Again I would appreciate a helpful reader sending an OS X version.

Open the Chapter10/SoundPlayback project and review main.c. Again I wait for both devices to be attached and then I send sound.dat to the Cmedia audio device.

So would you agree that audio on the Vinculum-II is easy?

Position recorder.

I originally planned this third example to be a variant of the first example; I would replace the audio adapter with a USB-based GPS sensor and would record the GPS position data. But the USB GPS sensor was out of stock so I had to change the example to use a serial GPS sensor and the result is, I believe, even more useful! Most GPS modules are serial-based but my dilemma was that the Vinculum-II only has one UART and I am already using this for my debug monitor. Yes, I could use the FT232R USB-to-serial cable but I decided to resolve my need for two UARTs in a more creative, and lower cost, way since having the GPS sensor on a serial port will be better for examples later in this chapter.

My debug monitor displays progress messages on V2EvalTerm so, in fact, I only use the transmit half of the UART. This is the easy half since the Vinculum-II creates the data and does not have to synchronize with an external source. The SPI port could create the same data portion of the UART transmit signal but, of course, I also need to generate a start signal and two stop signals. Figure 10.6 shows my UART transmit signal created on the SPI port. UART data is sent lsb first so the first and every even character is a 0x7F. My UART data is sent in every odd character position and the final character is a 0xFF. The 0x7F generates my start signal and also seven stop signals. I run SPI at 6 MHz so V2EvalTerm is unaware of this hardware trick. I had to change the pin routing in initialize.c and needed to add a jumper on CN10 from pin 1 to pin 5 but otherwise this additional “UART” was a zero cost option.

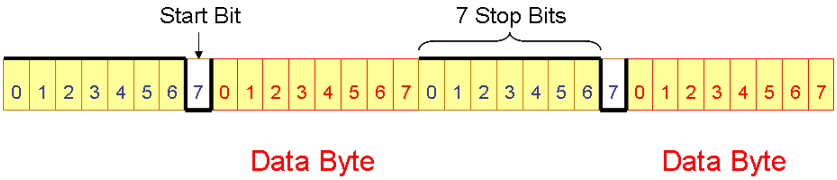


Figure 10.6: Using the SPI master to create a UART TxD signal

Having freed up the Vinculum-II UART I can now use it to receive data from the GPS module. The module runs at 4800 baud and starts transmitting standard NEMA sentences as soon that it is powered on. Figure 10.7 shows the default sentence generated by my Garmin GPS module; this NEMA data is standardized and extensively documented on the web. The interested reader should Google "NEMA sentence".

```
$GPRMC,222833,A,4554.5776,N,12357.6341,W,000.0,
       030.8,290810,017.8,E*61
```

$GPRMC = sentence type RMC is Recommended Minimum
222833 = time, 22:28:33 UTC (unified time)
A = Active, locked onto satellite signals
4554.5776, N = Latitude 45 degrees, 54.5776 mins North
12357.6341, W = Longtitude 123 degrees, 57.6341 mins West
000.0 = Speed over ground in knots (I am not moving!)
0.30.8 = Track angle in degrees (not valid since I am not moving)
290810 = 29th August 2010
0.17.8, E = Magnetic variation at this position
*61 = Checksum

Figure 10.7: Decoding the default NEMA sentence

Open the Chapter10/PositionRecorder project and review main.c. I wait for the GPS module to lock onto the satellite signals then gather data. If adjacent samples are changing then I record this data in GPS0001.txt. When five adjacent samples show no movement I close the file and wait for changing data again. I saw no point recording data if the unit is stopped. The text files are standard, comma-delimited data files that and may be opened in Microsoft Excel or similar. The data can also be pasted into Google Earth and the recorded position data can be displayed on a map.

A GPS module is about $40, a Vinculum-II module is about $20 and a flash drive is about $5 so I don't understand why commercial GPS trackers retail for almost $200. And this is for the basic model! We will add more features later in this chapter using that Vinculum-II USB port which is still available.

Remote Control and Monitoring using a Cell Phone

Texting, or sending and receiving text messages using the cellular SMS messaging service, has become an integral part of our modern lives. I have provided cell phones to all members of my immediate family and this has increased my peace of mind. Cell phone operators make it inexpensive to add additional phones onto a "family plan" so I have recently provided cell phones to my extended family. My home, my vacation home and all of the family vehicles now each have their own cell phone. I will use a building block approach to the next set of examples and explain how the Vinculum-II enables a car and a house to send and receive text messages.

For texting you need a GSM/GPRS modem. This modem connects to the GSM/GPRS wireless network and needs a SIM card, or equivalent, to gain access to this network. You can purchase stand-alone modems but the simplest approach for most people is to purchase, or even receive free, a cell phone which includes this modem. You will need a cell phone with a data connector and it is encouraging to see that the phone industry moving towards a standardized micro-B, USB connector. I use Motorola and LG phones within my examples but, due to standardization within this industry, all cell phones should work.

Connecting the phone to a Vinculum-II is simple: use the phone's data cable and plug it into USB host port 1. My example looks on host port 1 and the phone will be enumerated as a COMM class device. FTDI provide a communications class driver (CDC) so all we have to do is provide the applications code. My first phone example talks to 8 buttons and 8 LEDs. We will talk to a car and a house in subsequent examples. I connect an 8 way DIL switch to PortA and an 8 segment DIL LED array to PortC (10 segment, but I only use 8); you saw the LED segment display in Figure 8.9 and the switch module uses the same design and construction.

As it turns out, sending and receiving text messages via the USB connector is very easy. The cell phone has a command monitor and it is ready to accept commands after powering up. All commands,

and responses, are in ASCII and I recommend that you explore this interface a little before proceeding with the example.

Plug your cell phone into an available USB socket on your PC. If your PC does not immediately recognize the device (a Windows PC does not have a default configuration for a CDC device - a large oversight in my opinion) then go to your cell phone vendor's website and download and install the driver. You may also want to install their messaging application too but we will not use it in my examples.

The cell phone modem will appear to the PC as a serial COM port so spin up your favorite terminal application, such as HyperTerminal or CoolTerm, and connect the phone. I list a few commands in Figure 10.8; for a full list you should Google “SMS commands”. Try AT+CGMI and AT+CNUM first to discover your phones manufacturer and your phone number.

```
AT+CGMI     Display cell phone vendor
AT+CNUM     Display cell phone number and format
AT+CMGF=1   Try to select English text mode

Syntax of following commands is different in
PDU or Text mode
AT+CMGS     Send message directly
AT+CMGW     Write text message to storage
AT+CMSS     Send a text message from storage
AT+CMGD     Delete text message
AT+CMGL     List text messages

There are many more commands!
```

Figure 10.8: SMS commands used in the texting example

An important command to try is a AT+CMGF=1. This tries to switch your SMS messaging scheme to English text mode. This is more limiting than the native PDU mode but it is easier to use and to explain. If your phone responds with ERROR then do not be too concerned since my example operates in PDU mode as well. PDU means Protocol Data Unit and all cell phones support this mode which allows more control over the message and supports non-ASCII text such as Japanese and Arabic.

Text messages can be sent directly using the AT+CMGS command or can be prepared in local storage and sent later using the AT+CMGW and AT+CMSS commands. My example sends text messages directly but you could easily modify it to create some fixed messages that can be sent later.

I check for received messages by issuing a AT+CMGL command periodically. My example reads in any new messages, interprets the message and create some action - initially this will be turning LEDs on and off. A received message also contains the phone number of the sender - I check this and reject any messages that were not sent from a predefined number (or numbers). This way I can prevent random text messages from causing the Vinculum-II to implement commands.

Open the Chapter10/Texting project and review main.c. There are two main threads: ***Buttons*** monitors the 8 switches on PortA and ***CheckMessages*** periodically checks the cell phone for received messages. If a button change is detected then a text message is sent with the current button positions to a pre-determined phone number. A duplicate system (Vinculum-II, 8 buttons, 8 LEDs and a cell phone) receives this message and sets its LEDs to the received pattern. Similarly change in the switches on this duplicate system will cause the LEDs on the first Vinculum-II to system to change. We have built a simple remote monitoring and control system using cell phones.

Now expand your mind a little. The switches could be replaced by any type of sensor and the LEDs could be replaced by any time any type of actuator; examples will follow. The text messages would be customized to match the sensors and actuators. Imagine getting a text message “The basement is flooding”. You will respond with “Turn on sump pump”. Or you receive “Movement detected in vacation home garage”; you respond with “Release the Dragon”. With very little effort you can build up a Vinculum-II plus cell phone system that gives you James Bond like features on your personal cell phone. Let's look at two examples now.

Precious item locator.

I installed the following example on my son's motorcycle but it has applications to protect any kind of mobile precious item, such as a child. I combined the cell phone example with the GPS example. I can send a text message to the motorcycles phone asking “Where are

you?”. Note that I did not send it to my son's phone since his answers are not always accurate. The Vinculum-II gets the motorcycles position from the GPS sensor and sends it back to me as a text message containing the world coordinates. I paste these into Google Earth and I now know exactly where the bike is within 3 meters. Imagine how useful this would be if the bike were stolen. Or if it was involved in some kind of accident; being able to exactly locate the bike would be invaluable.

The Vinculum-II continually monitors the GPS data stream and I get text messages if the bike speed exceeds 100 miles an hour. I also get text messages if the bike leaves a predetermined area. This would be particularly valuable if the unit were installed in a child's backpack, or miniaturized and fitted into a child’s clothing.

I've installed the same hardware in my wife's Jeep. I also purchased an ODB-II monitor from www.elmscan.com and, to my delight, it enumerates as an FT232 device. My goal was to identify car problems such as “Airbag deployed" but I have not been able to figure out the protocol to do this. I would welcome help from any car enthusiast reader who could help me with this application.

Open the Chapter10/PE_Locator project and review main.c. There is no new theory here since it is a combination of previous example building blocks. There are many ways that this design could be extended and, don't forget, that the application includes a cell phone which can always be used to make voice calls in an emergency situation.

Home/office monitoring and control.

In this example I replace the simple buttons and lights of the earlier example with sensors and actuators that monitor and control the real world. Rather than reinventing the wheel I decided to use an existing system; I chose X10 since their products are available on a worldwide basis and therefore all readers of this book can benefit. For more details visit www.x10.com and try to ignore the heavy marketing.

A typical X10 system, shown in Figure 10.9, uses two methods of signaling; power line control (PLC) and RF. Power line signaling sends patterns of bits over the home or office mains wiring while RF signaling sends patterns of bits over the air using an unlicensed band. The voltages, mains frequency and RF frequency are different in various parts of the world but we need not be

concerned about these details. An interesting reader should Google “X10 signaling” where many details and ideas can be found.

An X10 system can address up to 256 output devices which it divides into 16 house codes, A through P, each with 16 unit codes, 1 through 16. My example also uses this convention. There are also 16 commands that can be sent, such as ON, OFF, DIM, BRIGHT, etc. The devices can be lights that can be dimmed, appliances that turn on or off or devices that make a sound. More sophisticated devices, such as a room thermometer are also available. Input signals can be generated from wireless remote controls, button boxes, motion sensors, door and window sensors or even a custom signal. An X10 system is readily expandable to meet your needs.

Figure 10.9: X10 system with many real-world inputs and outputs

From this examples perspective, the most important component is the CM15A controller, called the CM15Pro in Europe, since this is the bridge between USB and the PLC and RF signaling. The CM15A is a non-standard, low-speed USB device (it does not conform to any class specification) which has an interrupt out endpoint used to send X10 commands and interrupt in endpoint that repeats the PLC and RF signals it receives. This low-level signaling scheme is not officially documented but it took me less than a day to decode the essential elements.

Open the Chapter10/X10Spy project and review main.c. After finding the CM15A controller the program waits for activity on the interrupt in endpoint then displays the raw data and an ASCII

interpretation of what I determined the bit patterns represented. To debug this example I used two CM15A's, one connected to a PC that was running the Activity Monitor of the ActiveHome application and the other connected to a Vinculum-II running the example program. I then used an X10 button box, MC10A, and an RF remote, HR12A, to sequence through the house and unit codes. The X10 product line is broad and I also tested motion sensors, keyfobs, wireless switches etc. and they all generated a signal that was recognizable by my spy program. X10 also has a range of security products which are received by the CM15A but do not generate X10 PLC signals - I'm sure there was a good marketing reason why it does this!

I recommend that you explore a new X10 input device with this spy program before integrating it into your Vinculum-II based control and monitoring system. I had no problems with X10 output modules such as light controller, appliance controller, horn, siren, etc.

From the Vinculum-II's point of view it has to create an encoded packet and send it, via the interrupt out endpoint, to the CM15A to control a real world device. It must interpret an encoded packet received from the CM17A on the interrupt in endpoint to determine which part of the real world is signaling.

Open the Chapte10/X10_Texting project and review main.c. This is a combination of the X10 spy project and the texting project. Motion sensors detect movement once armed and will send a text message if activated. I can send a text message that will turn on lights or appliances. You can extend this example to monitor and control any aspect of your home or office.

Chapter summary.

We have only started to scratch the surface of available USB devices and, unfortunately, this chapter is already over the 10 page limit! I will have to convince FTDI that another edition is needed! What would you like to see? Send feedback to John@USB-by-example.com. I would like to look at Web Cams and use the Vinculum-II to record data; the ability to playback the last 3 minutes prior to an incident would be GREAT. USB-to-network dongles is another large topic area; would you prefer Ethernet or Bluetooth connectivity?

The dual host Vinculum-II is enabling a new range of low-cost products and I hope that this chapter has fueled some of your ideas or has helped you solve some of your pending problems.

Chapter 11: Other design considerations.

This chapter contains important Vinculum-II information that I could not fit into the other chapters of Part 2! All of the Vinculum-II examples so far have been implemented with a single-chip but what happens when you need more peripheral resources; the first section of this chapter explores expansion options. I have used the V2EvalBoard for my examples since this was straight forward; I also demonstrated how the same example software would run on FTDI's range of Vinculum-II DIL modules using a debugger module. FTDI are about to announce (as of October 2010) an Arduino-inspired development platform called Vinculo; this is covered in the second section. Some of you will need features such as galvanic isolation and plug-in device functionality - I explore these component orientated issues in the third section.

Vinculum-II expansion options

The first obvious is expansion mechanism is the SPI master; we saw in Part 1 that there are many SPI devices available to expand digital I/O, analog I/O and special-purpose I/O. Vinculum-II's SPI master peripheral has 2 chip select outputs which, with an external decoder, could generate from 2 to 4 peripheral select lines. You can use other I/O lines to create more chip selects. Figure 11.1 shows some SPI expansion opportunities. Note that device 4 is another Vinculum-II! Although I don't expect you to use two Vinculum-II's in this manner, I did want to point out that the Vinculum-II does have an SPI slave peripheral (in fact it has 2) so you can use it as an attached slave device as well as a stand-alone master device. A more typical use of this attached mode is the Vinculum-II providing a self-contained USB subsystem on behalf of a host CPU.

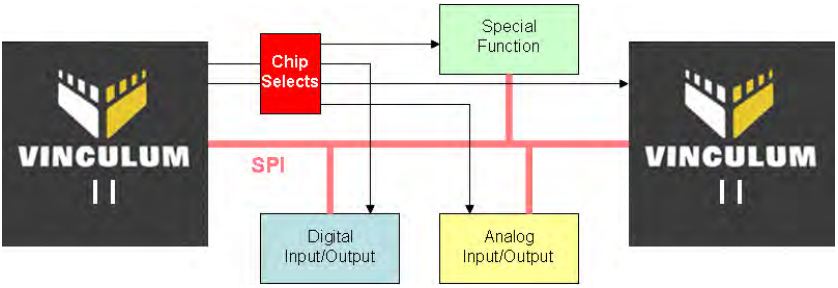


Figure 11.1: Using SPI to add Vinculum-II peripheral resources

The first resource I run out of when implementing Vinculum-II designs is the UART. Many of FTDI's customers must have provided similar feedback since FTDI have introduced an FT4232 Hub DIL module. A block diagram of this module is shown in Figure 11.2; it contains a high-speed hub and an FT4232H connected to one of the downstream ports. All four channels of the FT4232H are routed to the DIL connectors making it easy to prototype hardware. Remember from Part 1 that each channel can be a UART, SPI, I2C, FIFO, or a custom configuration.

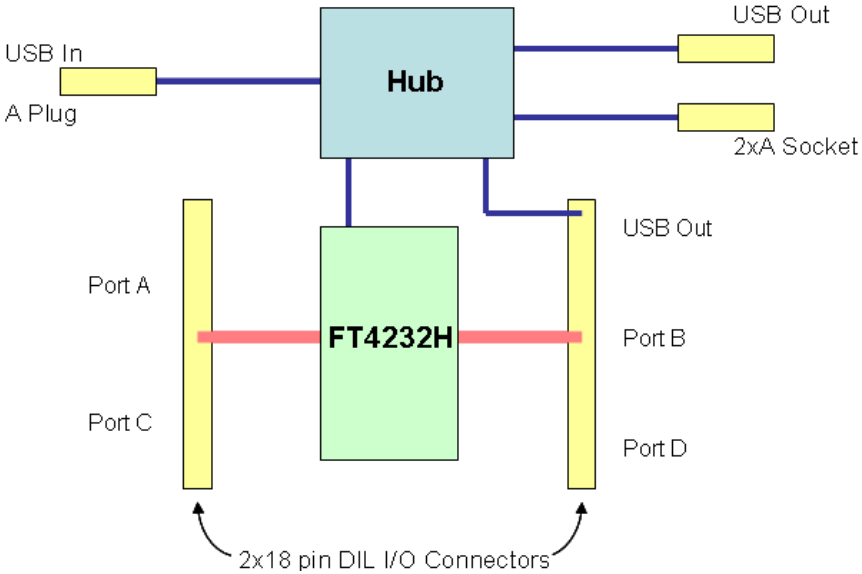


Figure 11.2: FT4232 Hub module adds USB ports and peripherals

Note that the upstream connection is a USB-A plug rather than a mini-B socket as found on the FT4232H mini-modules. This allows the FT4232 Hub to be connected directly to one of the Vinculum-II DIL modules and the combination then plugs neatly into a 0.1 inch solderless breadboard. The FT4232 Hub is shown in Figure 11.3 and this gives you ample, convenient I/O expansion for most Vinculum-II projects. And if you want even more I/O then you can daisy chain FT4232 Hub modules together! You must provide +5.0 V power from somewhere but, rest assured, you will not be limited in I/O expansion capabilities.

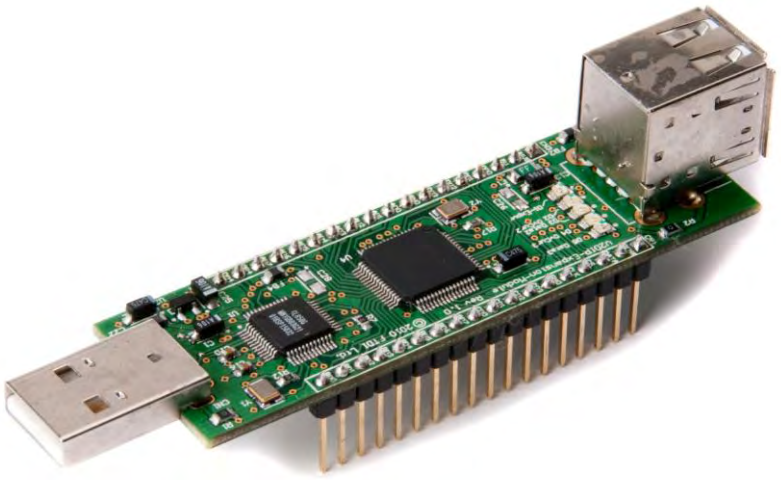


Figure 11.3: FT4232 Hub provides easy large system prototyping

When connected to a Vinculum-II the FT4232H operates at full speed so the output baud rate has a maximum value of 6Mb rather than the high-speed 30Mb. This is ample for most embedded applications. As an exercise I ported some of the Part 1 examples to the Vinculum-II with an attached FT4232 hub; you can review these in the Chapter 11 directory. FTDI includes an FT232 driver and the FT4232 hub will enumerate as four channels; you can attach to each channel and then individually set it up as a UART, SPI, MPSSE mode etc. The Vinculum-II operating system handles all of the complexity and we just use these additional peripheral resources within our application program.

Alternate development platform.

There is a lot of industry buzz surrounding the Arduino platform. It partitions the hardware a little differently than most microcontroller development systems; the CPU board has numerous I/O connections and expansion I/O is plugged on to this board. These I/O boards are called Shields and the industry has created a wide range so you should be able to purchase the expansion I/O you need and this will save you development time. You can also purchase low-cost prototyping boards for unique I/O requirements.

FTDI's Vinculo product is shown in Figure 11.4.

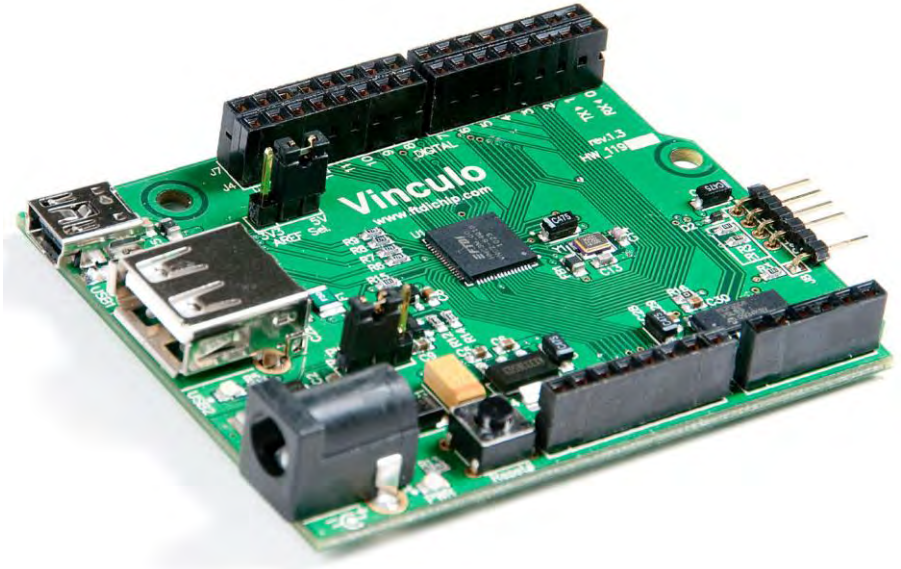


Figure 11.4: The Arduino-inspired Vinculo product

The Vinculo is mechanically compatible with the Arduino standard which means that all of the available I/O shield boards will plug directly on to the Vinculo. The Vinculo has an additional row of I/O pins giving the board greater I/O expansion capability - for example, the logic analyzer from Chapter 8 may be attached.

When compared with the V2EvalBoard, the Vinculo has host and device connectors rather than two host connectors; the SmartIO example from Chapter 9 works with the only I/O configuration changes; see the Chapter11/SmartIO project. The Vinculo does not have an integrated debugger nor an integrated FT4232 component therefore development with the Vinculo takes the same form as development with the Vinculum-II DIL modules. The Vinculo includes a connector to interface to an optional Vinculum-II debugger module.

At the time of writing (October 2010) FTDI only offer Arduino-compatible hardware. A full Arduino development system also includes Arduino open source development software. You should check FTDI's website for future announcements with respect to software. Today, software development for the Vinculo would use the IDE and tools presented in Part 2 of this book.

One the Vinculo's claims to fame is, of course, the inclusion of a USB host port. So, out-of-the-box, the Vinculo can already talk to flash drives, mice, keyboards, joysticks, cell phones, digital still cameras, X10 home automation and security, to mention a few. In fact all of the examples presented in this Part 2 can be readily ported to the Vinculo; some will need to convert the mini-B device connector to a USB A socket but that's just a cable and 100 µF capacitor!

The introduction of Vinculo now gives you three hardware choices for Vinculum-II development; V2EvalBoard, DIL modules and Vinculo. Which should you choose? Your I/O requirements should guide your choice. If your project hardware is available on an Arduino shield then the choice is easy; if it is not available then do you prefer to use solderless breadboard's or do you prefer to solder prototype hardware on an I/O shield board or a module that plugs onto the V2EvalBoard's I/O connectors? Whichever approach you take, the software development process using the Vinculum-II operating system will be the same. All three are good choices for rapid product development.

Component development issues.

The FTDI-supplied development hardware does not address issues such as galvanic isolation required for medical equipment or the construction of devices that must plug into a USB-A socket.

It used to be very difficult to isolate a USB connection since the two data lines, D+ and D-, are bidirectional and there is not a signal that indicates which direction the signals are being driven. An isolator needs to understand the USB protocol to be able to correctly drive the signals. Analog devices introduced two components, the ADuM3160 and the ADuM4160, that feature 2.5 KV and 5.0 KV rms isolation. These integrated components include all the circuitry necessary to cut the USB cable and provide an isolated connection. The datasheet also includes layout requirements for reliable operation.

If you don't want, or don't have time to prototype an isolation board then you can purchase Analog Devices' reference design implemented into a ready-to-use product from www.bb-elec.com. Their UH401 product is shown in Figure 11.5. Note that the USB connectors in Figure 11.5 are orange. This indicates a high retention USB connector which is recommended if you are designing products for the industrial or medical segments. You can get more information on these connectors from www.samtec.com.

Figure 11.5: USB isolation product from www.bb-elec.com

B & B Electronics also provide an isolated USB hub and isolated USB to serial converters that are shown in Figure 11.6. I recommend that you visit their website since they have many design recommendations and products for the industrial segment.

Figure 11.6: Other USB products from www.bb-elec.com

USB Device Design.

If you are designing a flash drive type USB device then you should consider a "Flipper" USB-A plug described at www.flipperUSB.com and shown in Figure 11.7. This is a clever idea that will increase customer satisfaction in your project. When you plug a USB-A plug into a USB A socket, you have less than a 50% chance that you will plug in it in correctly – the percentage is lower for children and older folks. If you are wrong then you flip the plug over and try again. With the "Flipper" plug you get it right 100% of the time since it is constructed to plug in both ways! The tolerances within the connector specification allow a thin circuit board with contacts on both sides to be centered in the plug. This simple idea has taken 15 years to come to fruition.

The folks at Flipper are offering samples to the readers of this book – go to www.FlipperUSB.com to get yours!

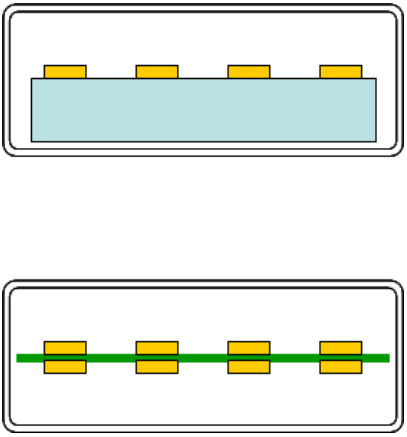


Figure 11.7: The FlipperUSB A-Plug is reversible

Chapter summary.

I hope that you will have had as much fun reading this book as I have had writing it. I am most impressed with the Vinculum-II since it's dual USB host/slave controllers enable a wider range of problems to be solved with USB. FTDI's Vinculum-II toolset also make it easy to develop and debug application programs and I trust that my examples have created solutions to some of your design challenges and have given you some good starting points.

This book is but one of the development aides available for the Vinculum-II. You should visit www.ftdichip.com often to look at their applications notes, data sheets and other examples. The IDE comes with a variety of examples covering USB devices that I have not mentioned in this book; such as a still image camera, web cam and printer – there are many examples to build upon that get you up and running quickly.

Until the next edition, happy developing! John.

Glossary:

BOMS – Bulk Only Mass Storage – FTDI use this to describe their implementation of the Mass Storage Class driver.
DIL – Dual In Line, typically on a 0.1 inch pitch such that it will plug into a solderless breadboard or mount on a prototype board.
HID – Human Interface Device. One of the standard USB classes.
I2C – Inter-Integrated Circuit. A serial expansion bus with a bi-directional Data signal and a Clock signal. Patented and licensed by Philips (now NXP).
IDE – Integrated Development Environment. A collection of tools designed to make the development process more productive.
LA – Logic Analyser.
MSC – Mass Storage Class. One of the standard USB classes.
OS – Operating System. In this book it is a USB-aware OS such as Windows (all flavors since Win98 Gold), Mac OS X and Linux.
PC – Personal Computer. In this book this runs a USB-aware OS.
SPI – Serial Peripheral Interface. A synchronous serial expansion bus with a DataIn, DataOut, Clock and Chip Select signals.
TLA – Three Letter Acronym. This book has a lot of them!
USB IF – USB Implementers Forum are the organizing body for USB matters. They also manage the USB Compliance Testing.
WHQL – Microsoft's Windows Hardware Quality Laboratory.
Wintel – An Intel x86-based PC running Windows.
VOS – Vinculum-II Operating System.

Appendix A: Examples and Schematics

The example source code can be found in the Examples/ directory.

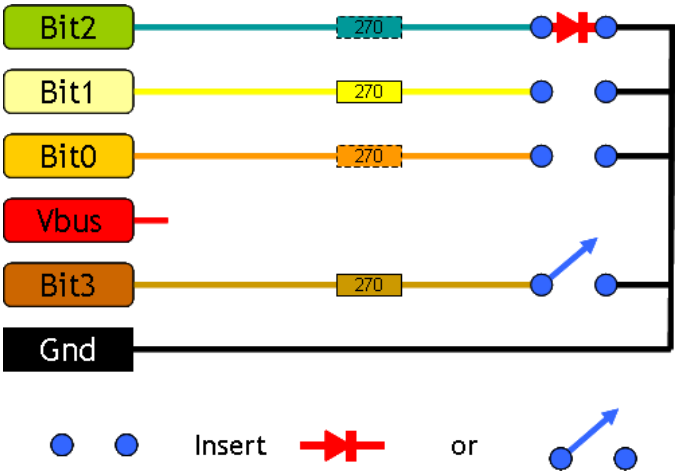
I designed the examples to give you a good start on your projects but they should NOT be considered 'production ready'. In fact, the legal folks required me to add the following disclaimer:
This software product is provided as is without any warranty of any kind, either express or implied, including, but not limited to, the implied warranty of merchantability and fitness for a particular purpose. Neither USB Design By Example nor FTDI nor their dealers or distributers assumes any liability for any alleged or actual damages arising for the use of, or the inability to use, this software.

Installing the examples

The examples are provided in multi-platform source code which may be copied and used on your PC. I have provided the Windows versions as Visual Studio project files and the OS X/Linux versions as XCODE project files. The source code is identical for all platforms and the project files will enable you to get up and running instantly.

Schematics

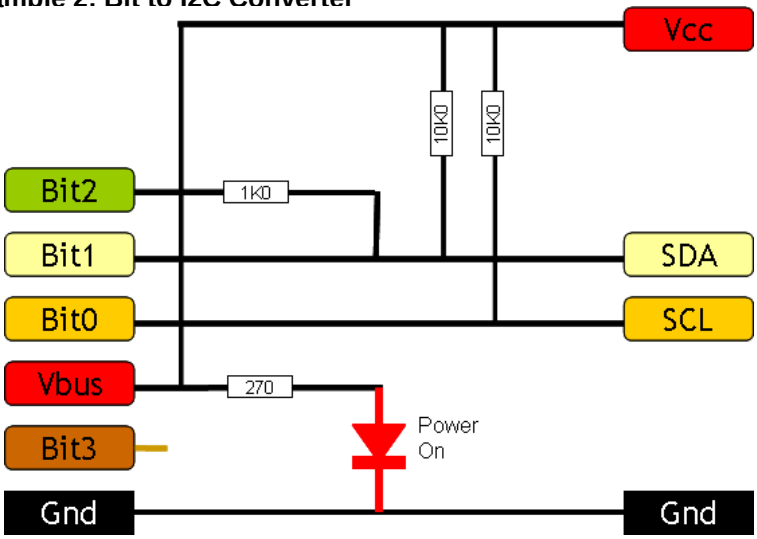
Example 1: Bit IO



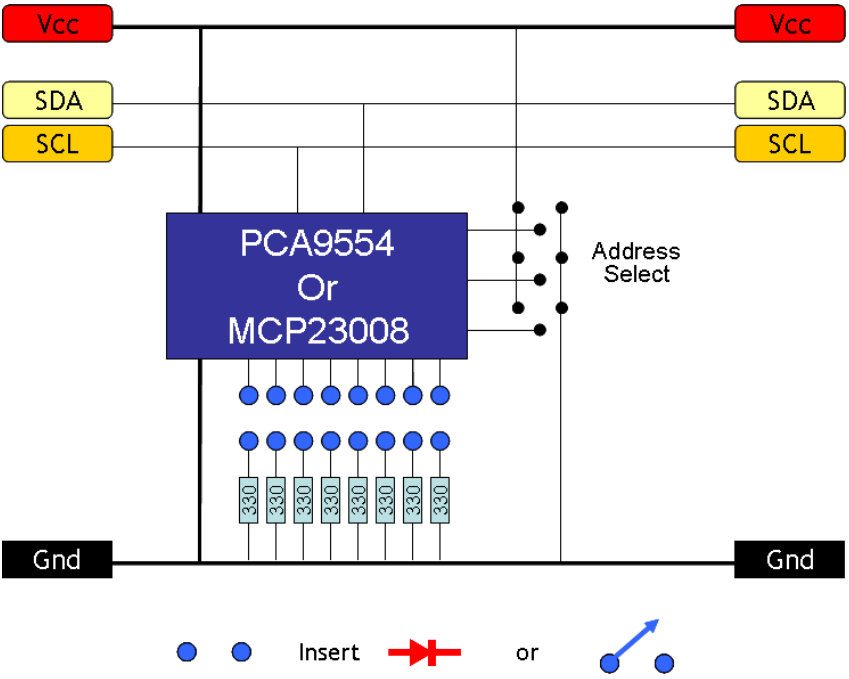
Notes:

Dotted resistors are inside the TTL-232R cable
Note colours of cable connection

Example 2: Bit to I2C Converter



Example 3: I2C to 8 bit port



Change History

Changes from Revision 1.0

- Various typographical errors fixed
- Chapter 2 examples extended
- Chapter 2 examples implemented on solder-less breadboard
- Chapter 4 example implemented on solder-less breadboard
- Chapters 5, 6 and 7 added

Changes from Revision 1.5

- Various typographical errors fixed
- Chapters 8, 9, 10 and 11 added

Changes from Revision 2.0

- Added a section in Chapter 9: Stages 7.1 & 7.2

USB is now a mature technology yet many people are not using it since they perceive it as “too complicated.” To date, all USB books and most USB technical articles have presented USB as a “technical wonder” and the reader is flooded with details such as packet types and descriptor parsing. Not surprisingly, many people who could take advantage of USB are holding back. This book treats USB as a tool that can be used to solve real world problems in embedded systems design.

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

www.ftdichip.com