Tech Newsletter

Golden Bits(R) Software, Inc.

Golden Bits Software, Inc.
3525 Del Mar Heights Rd, Suite 158
San Diego, CA 92130
858.259.3870 phone
858.259.7655 fax

Volume 1, Issue 2, Winter 2004

In this issue:

Technologies

- **Fibre channel**
- **Device drivers**
- **COM/DCOM**
- **Database, SQL**
- **C/C++**
- **SCSI**
- **GUIs/MFC/.NET**

USB Device Driver for Windows

The details about Windows USB device drivers

This article is featured in the April issue of Dr. Dobbs Magazine. To view online go to: www.ddj.com/articles/2004/0404/

This article discusses the details of writing an USB device driver for Windows. A working sample driver is can be downloaded from www.goldenbits.com/newsletter/issue2/usbsample.zip.

Introduction

USB devices have become very popular. Products that use USB such as MP3 players, digital cameras, and removal flash storage are now commonplace. Your next PC will probably use USB to connect your keyboard and mouse. What has made USB so popular is its low cost, flexibility, and especially its ease of use – you just plug your device into your PC.

USB Technology

Before we dive into the driver itself, it is very helpful to understand some of the basics of USB technology. USB is a serial bus (there are only 4 signals) organized in a hierarchal manner – a basic tree structure. The tree can be extended by the addition of hubs, and each hub can support additional devices. USB can support up to 127 devices, but in practice the most I've ever seen is four devices. At the top of the tree is the root hub and host controller that controls all of the device configuration and traffic on the bus. **Figure 1** shows the USB topology.

continued on page 2

.NET Top 5

Top 5 Features and Functions You'll Need

Download sample code from www.goldenbits.com/newsletter/issue2/nettop5.zip

Whenever you look at any new technology, you instinctively look for specific features and functionality that **you know** you need to develop your application, embedded system, or network utility. These are features and functions that you've relied on in the past and can't do without. You start asking yourself questions like: "I did a thread pool in C using the WIN32 API for a one of our modules. How is this done in .NET?" Or "If I have to port our monster application to .NET, how will I manage 100 socket connections?". This article discusses what I call .NET Top 5 Features and Functions.

Depending on what you're developing, there are common features that must provide certain functionality that you can rely on to enable you to develop a solid software system. These features and associated functionality may be supplied by the operating system, runtime environment, or some external module. What are the top 5 features? 1) Threading, 2) locking, 3) memory management, 4) file manipulation, and 5) network communications. What do I mean by functionality? For example: a locking primitive will in fact enforce only a single access (serialized access if you will) to a critical section of code or data structure, or that a thread can have

continued on page 8

USB (continued)

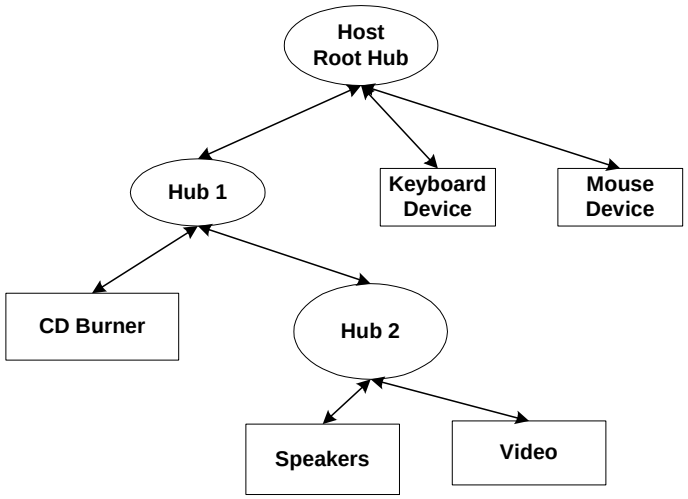


Figure 1

USB is a polled bus, which means the controller constantly polls all of the devices to see if the device has any data to transfer to the host and to also to initiate a transfer to the device. This polling enables the root hub/controller to control the bandwidth of the bus, which is important when performing isochronous transfers. During an initial setup phase, each device on the bus is assigned an address by the root hub when it connects. During this phase the device reports its configuration using a set of data structures called descriptors: the device, configuration, interface, and endpoint descriptors.

The device descriptor contains the vendor and product IDs, which is how the Windows PNP manager knows which USB driver to load. The configuration, interface, and endpoint descriptors describe how your device wishes to be connected to the bus itself. The actual connections are called endpoints and represent a unidirectional destination or source of data between the device and the host. To send and receive data, you will need two endpoints, one output to the device, and one input from the device. These descriptors are almost always predefined in the device's firmware (EEPROM or other). How your driver uses these descriptors is examined later in this article.

Data and control messages are transferred in one or more individual transactions. Transactions themselves are contained within a frame (low speed) or micro frame (high speed), a frame can contain multiple transactions. If the amount of data will not fit into one transaction, then it is broken up into multiple transactions over several frames. The frames are either 1 millisecond (low speed bus) or 125 microseconds (high speed bus) in duration. The host controller schedules all of the transactions and frames. One simple way to think about this process is each frame is a train with a fixed number of cargo cars, where each car contains a transaction; the host controller schedules these trains and fills the cars as necessary. This is a silly example, but I think it gets the point across.

continued on page 3

Bulk	Lots of data, no loss of data, error correction. Used for disks, printer, scanners.
Isochronous	Lossy, guaranteed delivery rate, no error correction, can reserve bandwidth. Used for audio, video.
Interrupt	Asynchronous data to host, guaranteed polling rate, error correction. Usually for input device.s
Control	Configuration, setup messages, relatively small data sizes, error correction. Used for setup and configuration.

Table 1

USB (continued)

USB defines four types of transfers: bulk, isochronous, interrupt, and control. These transfer types provides the designer of a USB device a nice selection of options, meaning depending on the type of device, you can select the appropriate transfer type. For example, a USB keyboard does not need the bandwidth of a bulk transfer.

Each type of transfer and its characteristics are summarized in **Table 1**. The use of descriptors, support of different transfer types, and configurable nature make USB very flexible; you can connect almost any conceivable device. Since the USB bus interface is very flexible and can support a wide range of devices, you can think of it as the Swiss Army knife of sorts. So, there's your **extremely** quick tour of USB. The key items that relate directly to device drivers are:

- Endpoints are unidirectional. Therefore, you can't send data on an inbound pipe.
- Endpoints are mapped to pipe handles. The pipe handle is what you use to transmit and receive data.
- Transfer Types affect how much data you can send and if you can lose data.
- Descriptors, interfaces, and endpoints.

The current USB specification is version 2.0 and can be found at: www.usb.org.

USB Implementation on Windows

Microsoft provides a full stack of drivers for USB bus support; you just have to write a driver for your own USB device. Microsoft also provides a set of class drivers for audio, network, storage, printer, and HID (Human Input Device – fancy name for a mouse or keyboard). Support for these types of USB devices enables your device to easily fit into a specific device class. For example, if you have a USB disk drive, the fact the drive is located on a USB bus in transparent to the system. The MS supplied usbstor.sys handles all of the details of presenting the disk driver to the SCSI Port driver. Depending on the type of device, you may not have to even write a driver for your USB device, as is the case for audio devices (your USB device has to be compliant with USB Audio spec 1.0).

The USB driver stacks (i.e., driver layers) are different for Windows 2000 and XP/2003. The two main differences between the driver stacks are: 1) Windows XP/2003 supports high speed (USB 2.0 spec) devices and 2) includes a new generic

continued on page 4

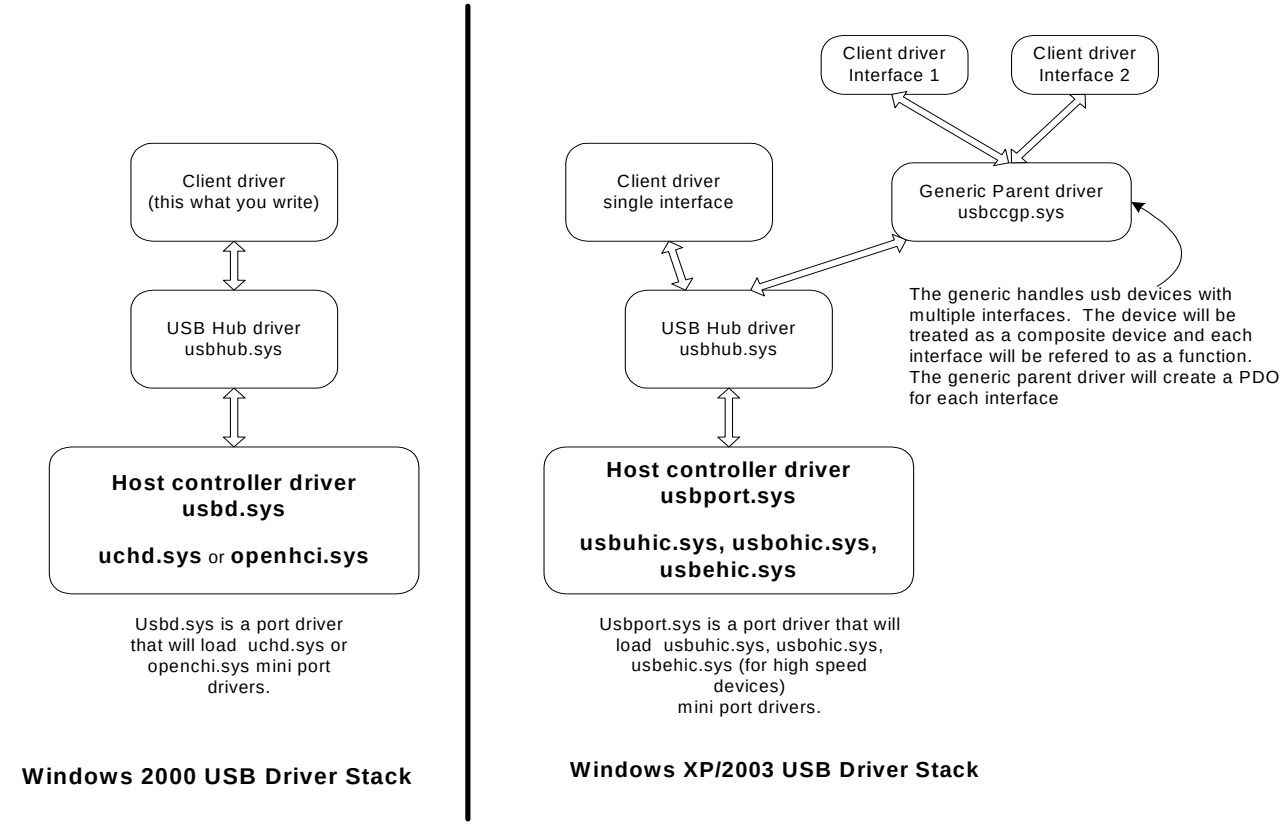


Figure 2 - USB Driver Stacks

USB (continued)

parent driver (usbccgp.sys) to support devices that expose multiple interfaces. The new generic parent driver solves the problem of managing separate function drivers for composite USB devices. For example, if you designed a new digital camera that could both take pictures and capture real time, you may want a separate driver for each function: one driver to download pictures and another driver to display real time video on the screen. **Figure 2** shows the USB driver stack for Windows 2000 and Windows XP/2003.

Your USB device is identified by a unique vendor and product ID which are managed by the USB-Implementers Forum, Inc. The cost of a vendor ID is $1,500 - $2,500 depending if you become a member of the USB-IF organization (see www.usb.org/developers/vendor). In addition to getting a vendor ID, you will probably want your device to be certified by USB-IF and pass Windows WHQL testing.

The vendor ID string is identical to how PCI devices are identified, the format of the USB vendor id string is: USB/VID_**XXXX**&PID_**ZZZZ**, where **XXXX** and **ZZZZ** are the vendor and product IDs. This information is sent to the host system, as part of the device descriptor, during the device setup phase. The Windows OS searches the list of installed drivers looking for a matching vendor and product ID - your driver's install .INF file should contain these numbers.

Descriptors Everywhere

One of the most confusing aspects of writing a USB driver is understanding all of the descriptors (device, configuration, interface, and endpoint), what they are used for, and how your driver manages them. The DDK documentation is a bit confusing. After reading the documentation four times, I still had the uneasy feeling I didn't understand exactly how these descriptors are used. Hopefully, I can do a better job than the DDK – so here it goes.

A descriptor is nothing more that a data structure. It's not a handle or some other unique object. When the designers of the USB standard tired to figure out a flexible and extensible way of configuring a USB device, they defined a set of data structures (the descriptors) that a USB device would present to describe itself and its capabilities. A USB device has one device descriptor that describes the device as a whole. This descriptor contains the vendor and product ID and the number of configurations. Each configuration is described by a configuration descriptor. The USB standard allows for multiple configurations, but in practice only one is used (the Microsoft USB class drivers only support the first configuration). Each configuration contains a set of interface descriptors where each interface defines a set of endpoints. It is these endpoints that your driver will communicate with. Your driver code will ultimately use a pipe handle to send/

continued on page 5

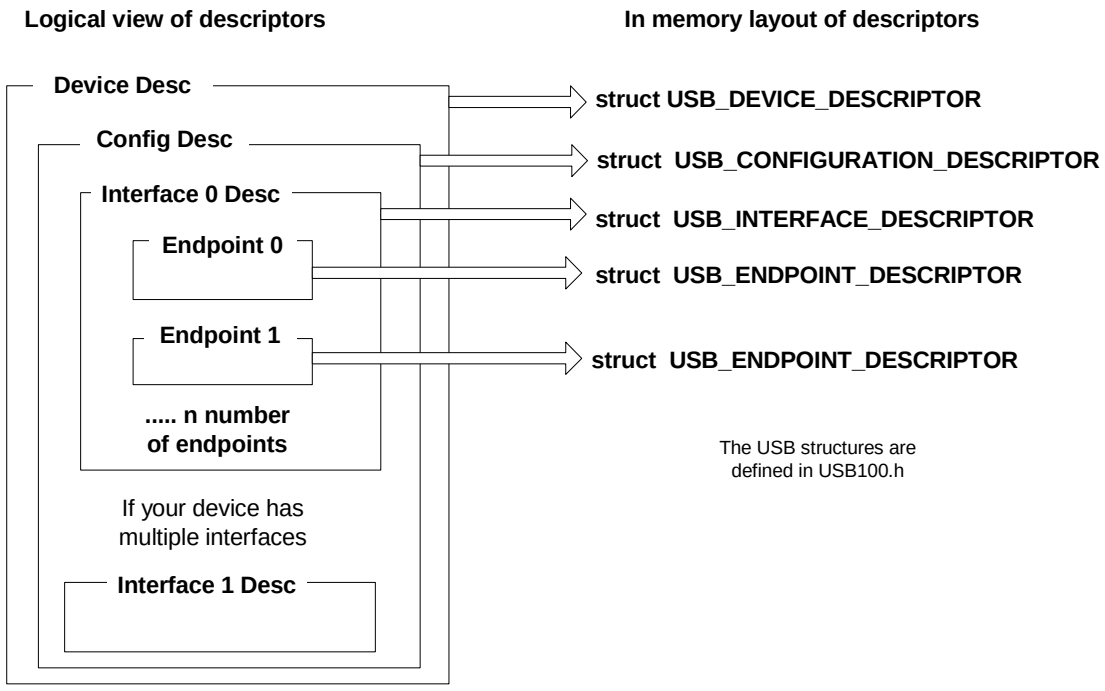


Figure 3

USB (continued)

receive data with a USB device, the pipe handle actually represents an endpoint.

When your driver gets the USB device configuration, these descriptors are returned to your driver (through a call to the Microsoft USB bus driver) in a chunk of memory containing a continuous list of descriptors. Here is one of the confusing parts of the DDK documentation: the DDK describes how to get this descriptor list but doesn't do a good job in explaining exactly what you've got. **Figure 3** shows what you have. The important piece in all of this is the interface and endpoint descriptors. These are the descriptors that define how your driver talks to the USB device.

Interfaces and Endpoints

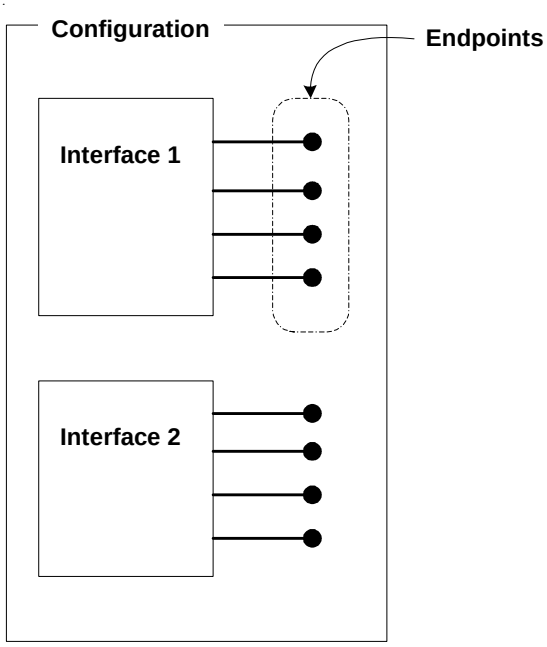


Figure 4

A USB device defines one or more interfaces where each interface describes one or more endpoints. What does this mean? Essentially, an interface is a logical grouping of endpoints. You should group similar endpoints into the same interface. In the accompanying sample driver, the encryption and decryption functions are contained in separate interfaces. As another example, you could design a USB device to perform encryption/decryption **and** compression/decompression where the en(de)cryption and (de) compression are contained in separate interfaces. **Figure 4** illustrates how interface and endpoints are logically organized.

An endpoint is a source or destination of data that your driver will read from or write to. Endpoints are unidirectional, have a unique address, and have a transfer type (bulk, isochronous, control, or interrupt). From a driver's perspective, an endpoint is represented by an USB pipe. Endpoint addresses are 8 bits where the MSB (bit 7) defines if the endpoint is an input or output. If the MSB is set, then this is an input endpoint, bits 6-4 are unused and bits 3-0 are the actual endpoint number. The terms input and output are from the host system perspective – data read from a USB device is from an input (input to the host) endpoint. Data sent to a USB device is delivered to an output endpoint. For example, an endpoint address of 0x81 defines an input endpoint address of 0x01. An address for an input and output endpoint can be the same, they are distinguished by their direction.

So, how are these addresses determined? By the designer of the USB device. The OS doesn't assign an endpoint address; this address is contained in the endpoint descriptor that the USB device returns. However, the USB device address is assigned by the host controller during the initial setup phase by the USB Set Address command.

When thinking about interfaces and endpoints, it is helpful to take the perspective of someone designing a USB device. What types of interfaces and endpoints will a USB device have? It depends on how it is designed. Therefore, when writing your USB driver, you will need to collaborate with the engineer working on the USB device itself.

Sample USB Driver – USB Encrypt/Decryptor device

An sample driver is included as part of this article. The source can be downloaded from www.goldenbits.com/newsletter/issue2/usbsample.zip. This sample USB device is an encryptor/decyprtor device where the actual en(de)cryptor engine runs on the USB device itself. The TUSB3210KDBPDK development kit from Texas Instrument (see www.ti.com/usb) is used as our USB device. This kit is intended for keyboard devices, but for our sample device we'll ignore the keyboard features. The TI part, TUSB3210, is an 8052 based device intended for general purpose USB peripheral applications. TI does offer other USB peripheral devices (TUSB3410 and TUSB6250) for different peripheral applications such as streaming audio. You can purchase this development kit on-line directly from TI for $199. If you're thinking about developing a USB device, this kit along with several others, offers an inexpensive way to start to start working with USB. The only catch is you'll have to get a hold of a compiler for the USB device itself. Most vendors offer

continued on page 6

USB (continued)

demo compilers with limited capability, however, a full compiler will run you about $3,000. There are several vendors that offer very good development environment for your USB device firmware. I used the Development Kit from IAR with great success (See the IAR Compiler & Tools box below).

As I mentioned, the sample is an encryptor/decryptor USB device. The idea is to embed the actual encryption engine in the USB device itself. This enables a user to easily disconnect the device and secure it as necessary (e.g., in a vault). *Imagine yourself working for the CIA and receiving encoded messages over the internet in a cyber café in exotic Puket, Thailand. You just came back from scuba diving all day and you have a sexy agent by your side. When she disappears for a moment you quickly take your waterproof (good to 300 meters) USB en/decryptor device, plug it into the PC, and download some encrypted email, only to find out that she's really a double agent working for the Chinese!!* The CIA doesn't send encrypted messages over the internet and I'm not a CIA spy — but what the heck it sure sounded good, plus I'm sure we could figure out some way to self destruct (blowup) the thing.

Our sample de(en)cryptor device has two interfaces, the first interface is used for encrypting and the second for decrypting. The encrypting interface has two endpoints, one to send data to be encrypted and the second to read the encrypted data back. The decrypting interface uses three endpoints, the first two are used to send and received encrypted data, the third endpoint is used by the device to send the number of bytes decrypted to the host. The en(de)crypt endpoints are configured to use bulk transfer, the third endpoint is configued to use interrupt transfer.

Code Tour

The block diagram in **Figure 5** shows the main functional areas of the sample driver. The functions UsbCrypt_Write() and UsbCrypt_Read() are the main functions called when performing IO with the USB device. Both of these functions use SendBulkIntTransfer(), which builds the URB and sends it to the lower USB bus driver (provided by Microsoft). A separate thread, GetProgress() (implemented using a work item - IoQueueWorkItem()) is used to get the number of bytes decrypted. GetProgress() returns the number of decrypted bytes, but any type of device status can be returned. Also, note that a read from the USB device will block if the USB device hasn't sent anything to the host. Your driver should be able the handle this case, which means implementing a cancel routine for your pending IRPs.

The code used to get the device configuration and create the necessary USB pipes is contained in CrConfig.c. The entry point for configuring is the function ConfigureDriver(). The interfaces and pipes are created in the function SetupInterfaces() and the pipe handles for each interface are saved in the device extension in the function SaveInterfaceInfo(). The pipe handles for the decrypt or encrypt interface are also saved in the FsContext for each open file handle (see UsbCrypt_Create()), thus an open handle can only encrypt or decrypt. An application can open two file handles, one for encrypting and one for decrypting. Storing the pipe handles in

continued on page 7

IAR Compiler & Tools

I used the 8051C compiler and workbench with great success. The compiler installed easily and within a hour I had a compile working image I could download to the USB device.

IAR has been providing embedded tools for over 20 years; a lot of experience which shows in the depth of their tools. In addition to compilers, **IAR** also offers Bluetooth solutions, Visual State machine tools, and much more.

For more information about IAR you can contact their sales representative, Robert DeOlivera at (650) 287-4250

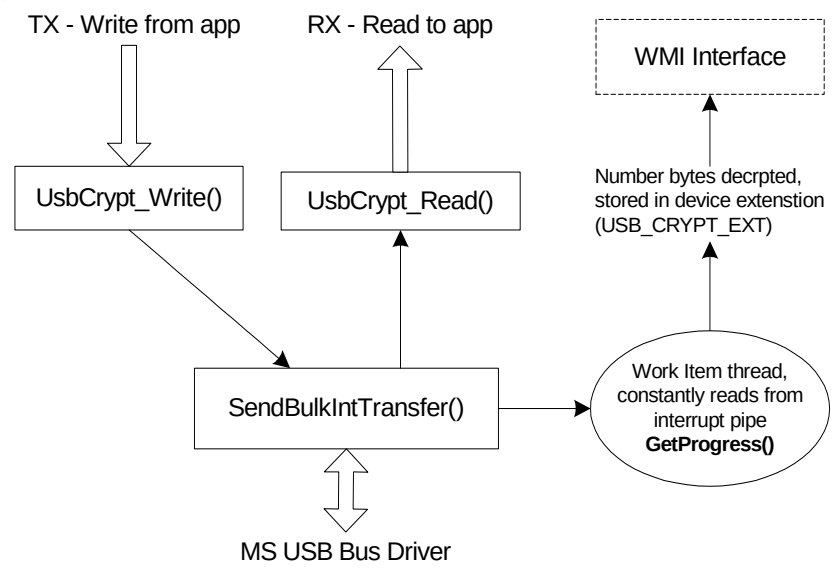


Figure 5

USB (continued)

the FsContext field is a nice technique to keep track of the individual pipes for each open handle.

You should use WMI to expose any type of device statistics, status, and events to management applications. WMI can be a pain to setup, but it is one of the necessary things to do (along with WHQL) to provide your customers a top notch, quality driver. In the sample driver, WMI is used to provide the number of bytes decrypted. The sample application reads this information using the WMI COM interfaces.

Pipe Handles

Most of the documentation regarding USB programming, usually shows a nice layered approach; you client driver sitting on top of the MS provided bus driver with pipe handles providing the logical connection to your endpoint. So, while things are nicely layered in the Windows OS, on the device side it's different. Usually USB devices are embedded systems, without layers of drivers - there's no concept of a pipe handle. For example, the TI USB device simply generates an interrupt whenever a packet is received on a specific endpoint – that's it. A pipe handle is really a Windows abstraction of a USB device endpoint.

User Application

The accompanying sample application sends data to the USB device to be encrypted or decrypted. In this sample, only 20 bytes are sent a one time. A real product should be able to handle an arbitrary large number of bytes. The radio buttons select the en(de)crypt function, if decrypting the number of bytes decrypted by the USB device is also retrieved using the WMI COM interface. The function ConnectToWMI() and GetDecryptBytes() handle the WMI COM details. On start-up, the application opens a handle to the USB device. Note that reading and writing data is done using the standard Win32 API calls Read File() and WriteFile() calls (which used by the MFC CFile class).

Screen shot of sample application

Summary

This article describes USB and how to implement a Windows USB device driver. Now you have enough information to start writing a USB driver, or, at a minimum, be able to explain to your manager what it will take. Just remember: 1) descriptors are just C language structures used by the USB device to describe itself and its capabilities, 2) the organization of interfaces and endpoints is up to you, 3) pipe handles represent endpoints, and 4) WMI is a pain, but necessary to provide a professional polished product.

References

USB Complete, Second Edition by Jan Axelson. Lakeview Research.

Programming the MS Windows Driver Model, Second Edition by Walter Oney. Microsoft Press.

Windows DDK documentation

www.usb.org

www.diving.phuket.com - Diving in Thailand

www.cia.gov/employment/clandestine.html. For information on how to apply for a spy position. Interestingly, I don't recall James Bond - Agent 007 filling out an application online.

.NET Top 5 (continued)

private variables. You need both; the desired feature with the necessary functionality.

While this is just a beginning, and some people will debate the merits of the individual items, it does represent what I refer to as the top 5 features and functions.

1. Threading

The .NET framework provides us with a full set of threading options including an asynchronous call mechanism which isn't truly a thread, but does provide thread like functionality. The two main framework classes that provide thread support are **System.Threading.Thread** and **System.Threading.ThreadPool**. The difference between the two is the amount of thread management your code will have to perform. Using the **Thread** class your code is responsible for managing the creation, running, and stopping of a thread; whereas, the **ThreadPool** class does all of this work for you. A thread in the .NET framework is ultimately implemented as an operating system thread within a process and can be viewed using the PVIEW utility. **Figure 1** contains C# code snippets on how a thread is created using the **Thread** and **ThreadPool** class. What's interesting when you compare the two classes is that you'll notice there is no "Start" method in the **ThreadPool** class. This makes sense because the **ThreadPool** class manages creating, starting, and stopping the threads.

The call to queue a work item to the thread pool:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadPoolFunc));

// actual code that gets run in thread pool
public static void ThreadPoolFunc(object state)
{
   // Here is the actual thread, you would perform your
     // thread work here
}
```

Code snippet to create threads directly:

```
Thread OneThread = new Thread(new ThreadStart(ThreadFunc));

// threads are initially started paused
OneThread.Start();

// actual thread code
public static void ThreadFunc()
{
}
```

Figure 1

Both classes use delegates for the actual implementation of the thread itself. The code you wish to execute in the thread is contained in the delegate. A delegate is a C# object that you can think of as a type safe function pointer. In addition to methods used to manage the thread, the **Thread** class contains properties such as apartment type, culture (my personal favorite), and state (running, suspended, etc). One subtle thread property is **Thread.CurrentPrincipal** that enables you to define the security context for the thread. What happens sometimes is that your code runs great during development, but fails on a customer system because the customer is logged in with a different security context than what you assumed. Therefore, you might have to set the security context accordingly.

Which threading class should you use? It depends on the task. A perfect place to use a thread pool is in server code when you are servicing client requests. As each client request is received, it is queued to the thread pool as a work item. Alternately, if you need to control your threads, then you will want to use the **Thread** class. For example, if you need to coordinate the flow of processing between each thread as in a producer and consumer scenario. You might have to pause the producer thread, enabling the consumer thread catch up.

Threads need data to work on. For both **Thread** and **ThreadPool** classes per thread data is encapsulated in a C# object that is then passed as argument to the thread. The thread itself can allocate memory as needed, and as expected, local variables are allocated on the stack and are thread safe. **Figure 2** shows how thread data is passed to a **Thread** and **ThreadPool** class. A word of caution when using threads: when a thread is used in a GUI to perform some background task, you will have to marshall your information back to the GUI. You can not make calls directly into the GUI class (Windows form class) because the Windows forms is by default single threaded. Only the thread that created the window can operate on it directly.

The .NET framework provides support for asynchronous calls. You can think of asynchronous calls as one shot threading,

continued on page 9

```
public class MyThreadWorkerClass {
            int Data;
            byte [] MoreData[100];
            void ThreadFunctionr() {
                // loop through data
                for(cnt = 0; cnt < 100; cnt++)
                {
                    MoreData[nCnt] = 0x88;
                }
            }
}

MyThreadWorkerClass WorkThread = new  MyThreadWorkerClass;

// fill in thread data
WorkThread.Data = 12;
Thread OneThread = new ThreadStart (
                        MyThreadWorker.ThreadFunction);

OneThread.Start();
```

Figure 2

.NET Top 5 (continued)

meaning the call itself is executed in a separate thread provide by the .NET framework. The cool thing about asynchronous calls is that it's built into the C# language itself. With asynchronous programming, you create two logical items: a worker class and callback. The worker class is the code which performs the actual asynchronous work. The callback is what the worker calls when the async is complete. The actual ascync calls are implement by the C# compiler.

2. Locking

The .NET framework contains two locking features: monitor and mutex. The **Monitor** class is used for quick lightweight locking within a process and the **Mutex** class is used for locking within or across process boundaries.

One of the new features of .NET is built in support for locking; every object created by the .NET framework has an associated locking structure! The **Monitor** class uses this structure to provide the actual lock for your critical section of code This makes implementing and using locking very simple, you don't have to worry about creating and initializing a CRITICAL_SECTION, the locking structures are automatically in place.

As I already mentioned, locking is built into the .NET framework directly, the C# **lock** keyword automatically implements a monitor lock. **Figure 3** shows how the lock keyword is used.

Creating a locking structure for every object can be a waste of resources since most objects do not require any locking. To solve this problem, the .NET runtime keeps a cache of locking structures that is uses as needed. For more information about this cache and Monitors, I recommend you read Jeffery Richter's article about the Monitor class, MSDN Magazine, January 2003, "Safe Thread Synchronization."

Mutexs in .NET work the same as Win32. Mutexs are implemented by the **Mutex** class, can be either named or unnamed, can be acquired recursively, are owned by a thread, and are signaled when this ownership is released. Mutexs can also be used across process boundaries enabling you to share resources (like shared memory) between two processes (even an unmanaged process). For example, if you have a service that is always running and you need to share access to a common data structure in shared memory, then a named mutex is the mechanism you should use.

3. Memory Management

.NET application memory is managed by the runtime engine. Your code does not have to worry about releasing memory, the runtime tracks your objects lifetime and will free up memory as necessary. When a .NET application is started, the runtime allocates a

```
 public class MyThreadWorkerClass {

// fill in thread data
 WorkThread.Data = 12;
 Thread OneThread = new ThreadStart (
                          MyThreadWorker.ThreadFunction);
 OneThread.Start();
==================================
 public class MyThreadWorkerClass {
Lock() {  code chunk, section to protect }
}
```

Very nice, since you don't have to worry about freeing the lock.

Figure 3

contiguous region of memory, called the heap. When reference types (your classes) are created, they are allocated from this heap. Allocations are done starting at the beginning of heap free space and continue in a linear fashion. On startup, a pointer is initialized to the beginning of the heap. When an allocation of N bytes is required, this pointer is simply advanced by N bytes thereby pointing to the remaining free heap memory. Allocating memory this way is very fast, but at some point the runtime needs to reclaim unused memory.

The runtime uses several techniques for releasing of unused memory. This is typically referred to as garbage collection. The runtime maintains a set of graphs representing the allocations along with generation numbers. If an object is contained in the graph, then it is referenced in some way and should not be released, otherwise the object's memory is released. Generations are interesting, they are based on the observed fact that the most recently allocated memory is usually short lived. The runtime uses 3 generations, 0, 1, 2, to track the age of each allocation. Generations are used as an optimization when the runtime performs garbage collection. When the runtime starts garbage collection, it will first examine memory allocations belonging to generation 0, since these are most likely to be short lived. If not enough memory is freed, the runtime continues checking generations 1 and 2.

After all of the memory has been released, the remaining objects are then compacted and shuffled on the heap in such a way as to create a contiguous free area of heap for new allocations. This means that some of the pointers to your objects will no longer point a valid object! The runtime does "fixup" your pointers so they point to the correct object after reshuffling the heap, but how can the runtime do this while your code is executing? The

continued on page 10

.NET Top 5 (continued)

answer, the runtime pauses all of your threads. Oh boy!! Yes, garbage collection is an expensive operation.

The runtime garbage collection is exposed to applications by the **System.GC** class. This gives your application some control over how and when garbage collection is performed. You can force garbage collection to occur using the **Collect()**. This is handy when you know that a lot of memory has been freed and is ready to be collect. For example, if your application is manipulating large bitmaps, you might want to force a collection after a bitmap is freed. The **KeepAlive()** method tells the garbage collector not to free an object.

The runtime garbage collection only knows how to release managed items. If your managed class is using an unmanaged resource, then you have to explicitly free the resource. An example is if a class opened a named pipe (an unmanaged resource), then this handle must be closed by the class. To ensure resources are freed, you should implement a **Dispose()** or add a Finalize() to your class. The **Dispose()** method is from the **IDisposable** interface and intended for consumers of your class to explicitly call. The Finalize() is called by the garbage collector before releasing the objects memory giving the object a chance to clean up. In C# a finalize method is implemented using syntax similar to a destructor in C++. The syntax for C# is:

```
public MyClass {
   public MyClass() {};
   public ~MyClass{}
   {
   …clean up code here…
   }
}
```

A couple of addition points to understand about garbage collection; garbage collection is performed at unknown time and Finalize() code is run on an separate thread within the runtime.

4. File Manipulation

Almost always you have to save your application data to some type of persistent storage, this usually means writing your data to a disk file. Well, things haven't changed with .NET (some things never do!). However some of the details are taken care of by the framework and there's a new option for storing data – Isolated Storage. One of the goals of the .NET framework is to isolate the application from any platform specifics, the file IO implementation reflects this goal. The .NET framework implements two types of storage models; streams and isolated storage. These models are implemented by the classes **System.IO.Stream** and **System.IO.IsolatedStorage.Isolated-Storage**

The **Stream** class is really an abstract class that cannot be used directly, you must use a class derived from the **Steam** class. Use of this abstract class results in a consistent interface across all of the derived classes that implement file IO. The beauty of this is your application can behave the same regardless of the type of underlying storage type, file structure, or network connection – you just don't care. For example, if you need to export data from your application into different formats, you can implement a class for each format type, such as coma separated, DB, or table. This "format" class would derive from the **Stream** class and handle all of the details of formatting your data. How is this beautiful? *Regardless of the format, the application doesn't change its behavior.*

The **FileStream** class most closely follows the traditional file IO system calls that we're accustom to using: open(), close(), read(), write(), and seek(). The methods Read(), Write(), Seek(), and Close() are the FileStream equivalents (open() is handled in the **FileStream** constructor).

An interesting IO class is **MemoryStream**. Like all IO classes, **MemoryStream** class provides the ability to read, write, and seek, the different is the underlying storage is memory. This class is perfect if you need to cache application data, you can cache your data in the same format as it persists on disk. There's no need to create a special cache structure!!

Isolated Storage

For most of us, isolated storage is new way of thinking about data storage. Traditionally, we think of storing application data in files, files that have a specific location in a file system. By convention, applications typically store data in common places such as "C:\My Documents" or "H:\My Network Location". This approach works well with a PC centric model of the world, but the Web has changed the world and this old PC model. People are now connecting to their application using vastly different methods, from web pages, wireless, cell phones, and PDAs. Stitching together the program framework for this remote and loosely connected world is one of the major goals of .NET.

Now that we have the context in which Isolated storage is used for, just exactly what is Isolated storage? Isolated storage is one of the pieces in .NET that solves a key problem with remote users. How can you isolate your application data to avoid corruption and provide security? How to uniquely identify your data, and how do you support roaming users? Isolated storage provides the mechanism to store and associate data with some property of your program such as assembly, domain, web site, or user. Isolated storage enables you to create files and directories **within the store** without concern to the underlying system. Another way to think about isolated storage is as your

continued on page 11

.NET Top 5 (continued)

own private file system, uniquely identified by assembly, user, web address, domain or other. The key point is these properties are part of your code *no matter where the code executed from*. This unique identification is one of the parts the runtime uses to actually isolate your data. A word of caution: Isolated storage **should not** be used to store confidential information.

Isolated storage is implemented by the **System.IO.IsolatedStorage.IsolatedStorage** class. Like the **Stream** class, the **IsolatedStorage** class is an abstract class that cannot be used directly. You will need to used on the derived classes, in most cases **IsolatedStorageFileStream** class is adequate for the task. **IsolatedStorageFileStream** is derived from the **Stream** class; like the **FileStream** class mentioned earlier, your code performs all of the stream operations (read, write, seek). This is the beauty of the **Stream** class, your code behaves the same.

5. Network Communication

It's a wired world and the .NET framework provides a very complete set of classes that enable an application to easily perform any network task. Networking is one of the .NET framework strengths. All of the networking classes are contained in the **System.Net** and **System.Net.Sockets** namespaces. The **System.Net** namespace contains classes for using popular network protocols such as HTTP and FTP In addition to protocol support, there are classes to handle network certificates, security, connection management, and proxies.

Protocol classes are derived from the abstract classes **WebRequest** and **WebResponse** and are referred to as "pluggable protocols". What pluggable means is, each protocol is registered with the framework and identified by the Uniform Resource Identifier (URI) used in a request. URI's essentially define a way to locate resources on the internet, for example: http://www.goldenbits.com is a URI for the Golden Bits web page. Another example of a URI is ftp://ftp.microsoft.com, the URI for the Microsoft FTP server. When a request is made using **WebRequest.Create()**, the URI used in the call determines which protocol is returned.

You can also add your own protocol to the runtime by using the **RegisterPrefix()**. The **HttpWebRequest** and **HttpWebResponse** are used to send and receive http messages to a server. All of the details of the HTTP protocol are handled by these classes, making adding web access a breeze.

The class **System.Net.Sockets** implements the socket library, Winsock, which the majority of applications use. The semantics of the **Sockets** class follows the socket model (originally BSD sockets); server listening for connections while clients connect.

The **TcpClient** and **TcpListener** classes implement this model. The **TcpListener** class will return a **TcpClient** when a connection is accepted, the **TcpClient.GetStream()** method is then used to get a **NetworkStream** class for the connection. The **NetworkStream** class is what is used to perform the actual transfer of data across the connection. What is interesting is that **NetworkStream** class is derived from the **System.IO.Stream** class, the same class used for file IO!!. Very cool. This means your application doesn't have to implement a different set of classes or techniques to handle network traffic. Also, asynchronous IO is handled the same way for files and network connections. Therefore, the .NET framework provides a uniformed model for IO, which is one of the .NET design goals. **Figure 4** shows how server code would look.

The **UdpClient** class implements UDP protocol. This class is used to send and receive UDP packets and broadcast messages. You can also manage multicast groups with this class.

Summary

So there you have it -- The .NET Top 5, David Letterman would be proud. You can download sample .NET code from www.goldenbits.com/newsletter/issue2/nettop5.zip.

■

.NET Top 5 (continued)

```
byte []ReadBuff = new byte[100];
 // listen on port 13500, noting special about
 // this port number, just random
 TcpListener Listen = new TcpListener(13500);

// start listening
Listen.Start();

// listen for connections
TcpClient ConClient = Listen.AcceptTcpClient();

// got connection at this point

// get network stream
NetworkStream ConStrm = ConClient.GetStream();

// read some bytes off stream
(ConStrm.Read(ReadBuff, 0, 100);

// all done, close
Listen.Stop();
ConStrm.Close();
ConClient.Close();
```

Figure 4

Notes -

Project Experience - Several projects Golden Bits has successfuly completed.

SCSI Port driver for Fibre Channel. Designed the operating system layer for a SCSI storage driver (XP, Win2K, NT, Linux) for a fibre channel HBA (host bus adapter – PCI/SBUS card).

Embedded Network Appliance. Developed an embedded monitoring device for web sites and/or other data center systems. The device uses uC/OS real time kernel running on Motorola ColdFire processor (MCF5206e).

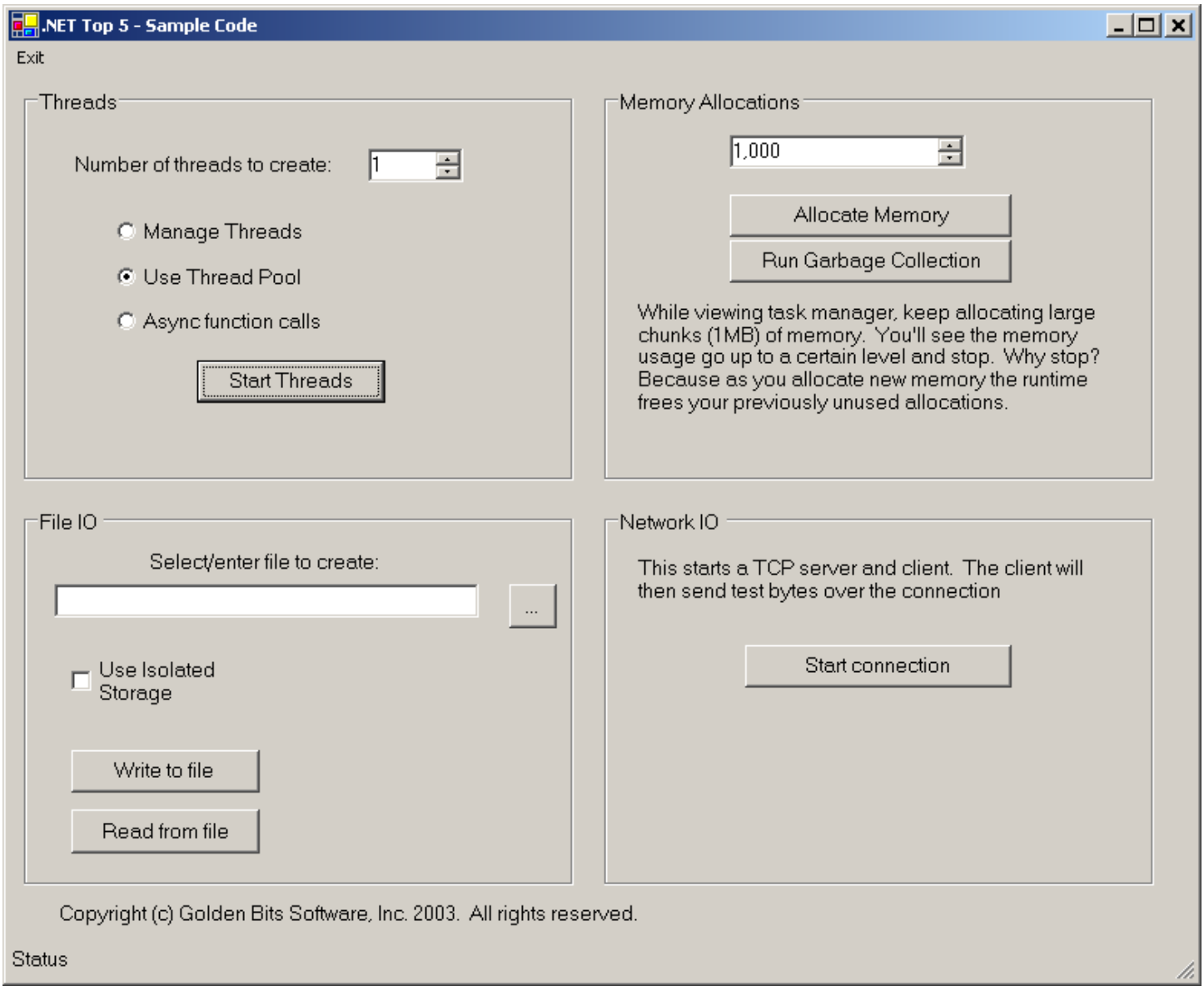
WDM, NDIS Device Drivers . Developed a WDM and NDIS device driver for a prototype wireless system.

Parallel Search Engine. Developed a search engine that distributes database query to other systems; the search runs in parallel on the supporting systems and the results are written (through bulk inserts) back into the database.

Satellite Set Top Box. Developed a script language and compiler used to code the television UI (guide menus, channel select).

Camera Control. Wrote highly customized Windows user interface with special graphics and custom controls. The application presents the user with camera images with graphic information overlaid (in near real time), camera configuration information, and product inspection information.

Embedded TCP/IP Protocol Stack. Wrote a NT packet driver using NDIS driver subsystem to simulate a mobile network for a military application. The embedded stack executed under NT, and the packet driver simulated network device IO.



Screen shot of GUI demonstrating .NET Top 5

What is 2+2?

An accountant will say “What do you want the answer to be?”A mathematician will say “I believe it is 4, but I will have to prove it.”A statistician will say “The population is too small to give an accurate answer, but on the basis of the data supplied the answer lies between 3 and 5.” An economist will say “Based on today’s thinking, the answer is 4 but the answer may be different tomorrow”. An engineer will say “The answer is 4, but adding a safety factor we will call it 5”.

The Car

There were three engineers in a car; an electrical engineer, a chemical engineer, and a Microsoft engineer.Suddenly, the car stops running and they pull off to the side of the road wondering what could be wrong.The electrical engineer suggests stripping down the electronics of the car and trying to trace where a fault may have occurred.The chemical engineer, not knowing much about cars, suggests maybe the fuel is becoming emulsified and getting blocked somewhere.The Microsoft engineer, not knowing much about anything, came up with a suggestion. “Why don’t we close all the windows, get out, get back in, and open all the windows and see if it works?”