CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

MICRO SERIES

Jim Lyle

USB Primer

Classes and Drivers

Part 2 of 4

Now that we have some of the USB basics from Part 1, we're raring to go with USB! Jim wonders if an OS can provide all the drivers for the many devices there are today. With USB classes, he explains, it's entirely possible.

One of USB's earliest and most important goals was to make it easy to use. It has to be easy because the computer marketplace is rapidly expanding to include increasingly less-technical users.

These users don't know what an interrupt or DMA channel is, let alone how to finesse them into a working configuration. Nor should they have to. Even highly technical users are tiring of the difficulties involved in configuring or upgrading their computers.

From my perspective, it's not that difficult to install an ISA or PCI card. I've been doing this for years and I know how to set the jumpers (plug-and-play usually takes care of it anyway). I rarely get the cables on backwards anymore or offset by one row of pins, either.

But, one part of the process still strikes fear into my heart. One part of the installation never goes quite the way the instructions claim (when I finally do get around to reading them). There's one element that rarely fails to "blue screen" the machine repeatedly and strangely:

THE DRIVER

I've spent days trying to install the drivers for a seemingly simple device. Sometimes, it's incompatibilities with other drivers or software. Sometimes, the driver wasn't tested well or has a bug and needs a patch or upgrade. Sometimes I never do find the problem.

Wouldn't it be nice if all the drivers you ever needed came with the OS? You'd just plug something in and it would work. No more installation headaches; no problems moving from one machine to the next or even from one type of machine to another (e.g., from PC to Mac to Linux to workstation). There would be reduced disk and memory requirements, too, and one-stop shopping for upgrades. Overall, compatibility and reliability would improve dramatically.

Developers would find tremendous advantages as well, bringing more products to more platforms in less time and with less effort. Adding USB would no longer require the expertise (and the time, often in the critical path) needed to write drivers. Testing and support requirements would be reduced, and so would the overall project risk.

There are thousands of different kinds of devices already, and more are on the way. An OS can't possibly provide all the drivers for all of these types of devices. Or can it?

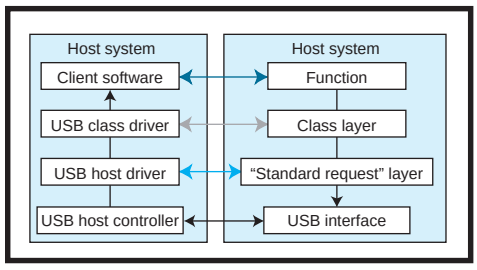

Figure 1—*USB uses well-defined protocol layers to reduce complexity and improve standardization.*

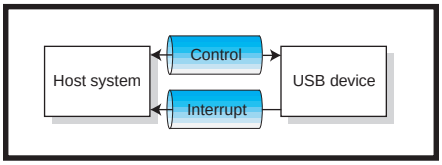

Figure 2—*USB devices use logical "pipes" to transfer information. This device uses two.*

Although lots of different products are or will be available, many of them have more similarities than differences. In some cases, identical devices are produced by different manufacturers. In other cases, the products are different but the functions are similar.

Consider mice, track balls, and touchpads. They are physically different (and there's variation even within those broad categories), but the overall function is the same—moving a cursor on the screen. They all provide an *x* and *y* displacement and two or more buttons (or the equivalent).

What about full-page scanners, hand scanners, digital still cameras, and slow-scan video cameras? All produce an image of some form. Or printers? Color or black-and-white, laser or inkjet, Postscript or not—all put an image on paper.

With a little insight and forethought, most devices can be grouped into fewer categories, each with a common purpose and set of requirements. Then, it's possible to define a common API for each category and therefore the requirements of a single, generic driver suitable for use with any of the devices in that group. USB is trying to accomplish exactly this by defining a variety of device classes.

The USB specification defines the mechanical and electrical requirements for all USB devices as well as the fundamental protocols and mechanisms used to configure the device and transport data. The class definitions are add-on documents that refine the basic mechanisms and use them to establish the class-specific blueprint for both the device and the generic driver.

There will always be unique devices as well as manufacturers that choose to differentiate their product from the competition within the driver. For these cases, vendor-specific drivers will always be necessary.

But for most products, it'll be possible to use generic drivers that are part of (or included with) the OS. That's one of the most important advantages of USB.

Comm, Printer, Image, Mass Storage, Audio, and HID (human interface device) are a few of the defined USB classes. Some devices may fit into more than one category.

For example, there are combination printers/scanners. Although physically this is one device, logically it is two. Part of the device fits into the Printer class and uses that generic driver. Part of it fits into the Image class and uses that driver. Devices in more than one class are called compound devices.

DEVICE CLASSES

Windows 98 includes many but not all of the USB class drivers. This situation is unfortunate, but it couldn't be helped because some of the class definitions weren't finished in time (some still aren't complete).

Future releases and service packs will add additional class drivers until most or all of them are available and supported. Apple and Sun Microsystems also have class driver implementations available or underway for their respective platforms.

As the name implies, HIDs are designed for some kind of human input or output. The most common examples are keyboards, pointer devices like mice, and game controller devices such as joysticks and gamepads.

This class also includes things like front panels or keypads (e.g., on a telephone or a VCR remote control), display panels or lights, as well as tactile and audible feedback mechanisms—essentially, anything you might press, twist, step on, measure, move, read, feel, or even hear.

Seemingly, this class would include almost anything connected to a computer, but it doesn't. Its primary purpose is control. Although it's very flexible, this class definition doesn't handle large amounts of data well. It doesn't need to; other device classes can better serve that purpose.

In a USB speaker, for example, the volume, tone, and other controls fall well within the HID class. But, the sound channels are data intensive, so they are better handled by the Audio class. In fact, many products in the other classes are compound devices with HID handling the controls.

Given the diversity of USB applications in general and HID devices in particular, how can any one driver hope to do all the things required by its class? The first part of the answer comes from the physical interface. There's only one! All USB devices communicate with the host via their USB port.

This sounds self-evident, but the implications are tremendous. The USB port works according to the same basic principles for all devices, in all modes of operation. The class driver never needs to worry or know about ISA or PCI buses, SCSI, IDE, or ATAPI interfaces, serial ports, parallel ports, keyboard ports, mouse ports, game ports, or anything else for that matter.

The class driver doesn't even need to know much about USB ports. Even that physical interface is abstracted and managed by the USB Host driver. This abstraction, or layering, is another key concept that makes class drivers possible.

Each layer has its own responsibilities and it uses APIs provided by the lower levels to accomplish them. It doesn't need to know how the lower levels work or which ones are present.

Figure 1 shows a simplified view of the various protocol layers that might be present for a USB device. Note that there are connections at all levels, but most of these are logical.

The single physical connection is between the USB host controller and the device's USB interface and is at the lowest level (shown in black). This layer is the hardware—the cables, connectors, and state machines.

The first layer of software, which is required in all cases (in light blue), is

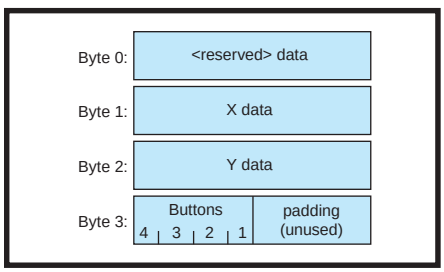

Figure 3—*This sample report for a USB joystick shows you one possible data organization.*

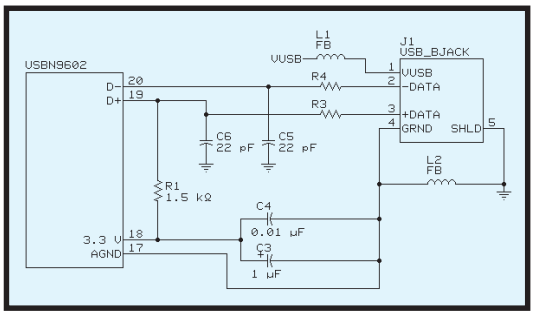

Figure 4—*This schematic shows you a typical connection between the USBN9602 and the USB connector (or cable).*

the USB host driver on the computer. On the USB device it is the essential firmware that manages the hardware and provides the standard requests (also called "chapter 9" requests because they are in that chapter of the specification). There's a logical connection between these layers for configuring and controlling the USB interface.

The device driver layer comes next (shown in grey) and is usually the class driver(s) on the host side and the corresponding firmware on the device side. The logical connection at this level carries class-specific commands and requests, although these often use protocols modeled after those in the layer below.

The top (dark blue) layer is the one the user sees and cares about. For example, the client software on the host might be a flight simulator and the associated function might be a joystick. At this layer, the only thing the client software (and user) cares about are the joystick inputs. It doesn't care (and doesn't need to know) how those inputs are read, packaged, and transported.

PROTOCOL LAYERS

To communicate with a USB device, the host software opens up a series of pipes, and uses them to transport data. The pipes correspond to hardware endpoints, which are individual channels, usually with dedicated buffers or FIFOs.

Pure HID devices use only two pipes (see Figure 2). The control (default) pipe, required by all USB devices, is used for receiving and responding to specific requests or commands. The standard requests use this pipe, and many of the class definitions (including HID) add class-specific requests.

The interrupt pipe sends asynchronous data to the host. This pipe is poorly named; USB doesn't support true interrupts but rather enables the device to predefine a maximum poll interval. This way, if a key is pressed, the mouse moved, or the joystick steered, the device can report in a timely fashion without a specific request (from the driver) to do so.

The HID class driver starts with the physical/standard request API common to all USB devices and adds the HID standard pipe structure and command superset. The difference from one HID device to another is the data it returns and what the data means.

HID data is packaged into structures called reports. Figure 3 shows a sample report for a joystick. It's simple and composed of four bytes.

The first byte is unused here but is reserved for a throttle position on another model. The second and third bytes are the *x* and *y* coordinates, respectively. The fourth byte contains information about the four buttons (one button per bit, with four unused bits that are zero-filled to pad out the byte).

This is just one example for one joystick. Other HID devices have different report structures. Other joysticks may have other structures, too. Some may order the data differently or have additional functions and capabilities (e.g., force-feedback).

SAMPLE REPORT

Obviously, the HID class driver can't keep report maps for all possible implementations of all possible devices. The device has to be able to describe the report to the class driver. This too is in keeping with standard USB mechanisms.

USB devices use predefined data structures called descriptors to describe their identification, capabilities, requirements, and protocols. The USB

Listing 1—*This* `ReportDescriptor` *function corresponds to Figure 1. USB devices use descriptors to describe themselves to the host PC.*

```
unsigned char ReportDescriptor[59] = {
 0x05, 0x01,       /* USAGE_PAGE (Generic Desktop) */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x09, 0x04,       /* USAGE (Joystick)             */
 0xa1, 0x01,       /* COLLECTION (Application)     */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x26, 0xff, 0x00, /* LOGICAL_MAXIMUM (255)        */
 0x75, 0x08,       /* REPORT_SIZE (8)              */
 0x95, 0x01,       /* REPORT_COUNT (1)             */
 0x81, 0x03,       /* INPUT (Cnst,Var,Abs)         */
 0x05, 0x01,       /* USAGE_PAGE (Generic Desktop) */
 0x09, 0x01,       /* USAGE (Pointer)              */
 0xa1, 0x00,       /* COLLECTION (Physical)        */
 0x09, 0x30,       /* USAGE (X)                    */
 0x09, 0x31,       /* USAGE (Y)                    */
 0x95, 0x02,       /* REPORT_COUNT (2)             */
 0x81, 0x02,       /* INPUT (Data,Var,Abs)         */
 0xc0,             /* END_COLLECTION               */
 0x15, 0x00,       /* LOGICAL_MINIMUM (0)          */
 0x25, 0x01,       /* LOGICAL_MAXIMUM (1)          */
 0x75, 0x01,       /* REPORT_SIZE (1)              */
 0x95, 0x04,       /* REPORT_COUNT (4)             */
 0x81, 0x03,       /* INPUT (Cnst,Var,Abs)         */
 0x05, 0x09,       /* USAGE_PAGE (Button)          */
 0x19, 0x01,       /* USAGE_MINIMUM (Button 1)     */
 0x29, 0x04,       /* USAGE_MAXIMUM (Button 4)     */
 0x55, 0x00,       /* UNIT_EXPONENT (0)            */
 0x65, 0x00,       /* UNIT (None)                  */
 0x95, 0x04,       /* REPORT_COUNT (4)             */
 0x81, 0x02,       /* INPUT (Data,Var,Abs)         */
 0xc0              /* END_COLLECTION               */
};
```

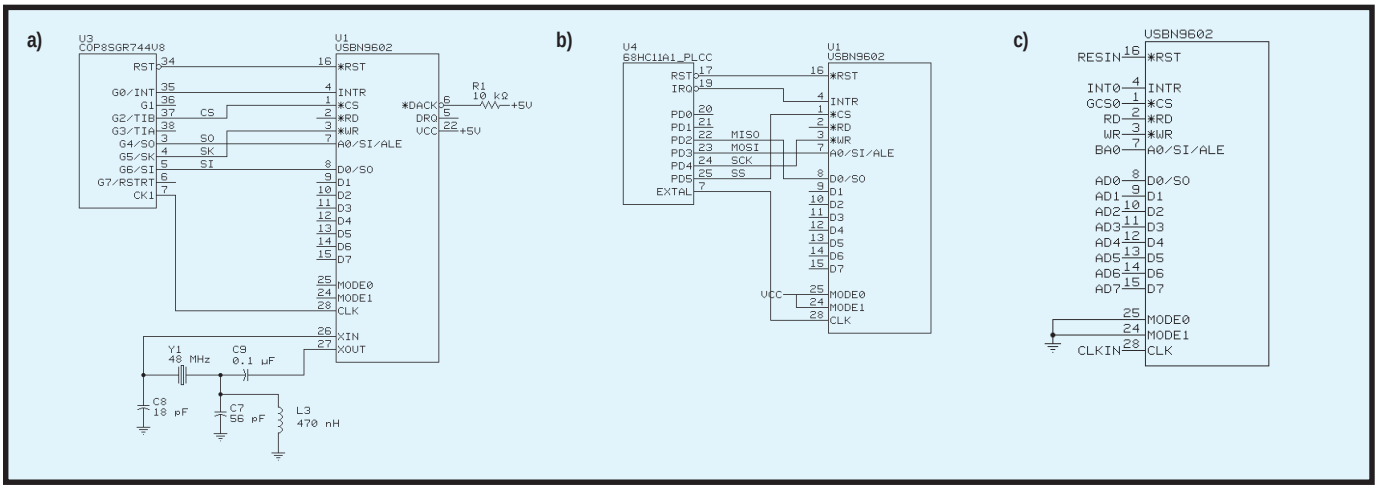

Figure 5—*This schematic shows a serial interface between the USBN9602 and a COP8 microcontroller. It also shows the oscillator circuit.* ***b***—*Here's another serial interface. In this case, the microcontroller is a 68HC11.* **c**—*In this parallel interface to the USBN9602, the microcontroller is an Intel 80C188EB. But, this example would be typical of any case where an 8-bit data bus is available.*

spec defines device and configuration descriptors that must be provided by all devices. The HID class definition adds information to these and goes on to define a report descriptor.

The report descriptor provides the map that the HID class driver needs to understand and interpret the report. The structure of the report descriptor is complex, though flexible. Fortunately, it doesn't complicate the device-side firmware because it is a data structure that can be written and compiled externally and then remain constant.

Listing 1 shows a sample report descriptor. The details are beyond the scope of this article, but note that it defines the type of application, size, maximum and minimum values, and subtypes of the various report fields.

So, a class-compliant USB product can entirely specify what it is and how it works in the onboard firmware. This makes the job of building, testing, and modifying a USB interface easier and more modular, and it brings it within the capabilities of most developers.

In the joystick, there are only three essential blocks—the ADC, USB interface, and microcontroller. The micro ties it all together, sampling the joystick at intervals and passing the data up through the USB interface (also managed by the micro).

The only new element is the USB interface. There are many varieties available: some are integrated with the microcontroller and some are separate components. These interfaces contain the state machines and buffers necessary to transmit and receive serial data on the USB. Conceptually, it's a smarter-than-average UART-like function.

National Semiconductor's USBN-9602 is one example of a USB interface. One side is attached to the USB cable or connector with a circuit like the one in Figure 4. (This figure and the ones following are not complete schematics; they merely highlight specific functions and interfaces.)

C3 and C4 bypass the USBN9602's internal voltage regulator (used by the internal USB transceiver). R1 is the required pullup that the device uses to signal its presence (and data rate) on the bus. The other components reduce EMI and transmission line effects to provide a cleaner signaling environment.

TYPICAL CONNECTIONS

The other side of the USBN9602 is the data path to the microcontroller. This data path is flexible and allows easy use with a variety of serial or parallel interfaces (there's even a DMA interface for high data rates).

Figure 5a shows a Microwire interface to a COP8 microcontroller, as well as the requisite dot clock oscillator circuit. Figure 5b shows an SPI interface to a 68HC11, and Figure 5c shows a parallel interface to an 80C188EB.

To make it even easier, several USB device manufacturers provide sample firmware source code. For the USBN-9602, National provides source code in C with compiler options for all of the microcontrollers mentioned here (and readily ported to others). This code is available on the web. Such firmware provides a ready-made solution to the some or all of the necessary device-side protocol layers.

If you want to build a mouse, keyboard, or other HID device, just modify the descriptor tables and a few top-level (function and class layer) firmware routines. Even if you're not building an HID device, the firmware layer that manages the USB interface device and responds to the standard requests provides a solid basis to start with.

PLAIN AND SIMPLE

USB simplifies the lives of developers and experimenters alike. It's possible for OSs like Windows 98 to provide most of the drivers you'll ever need for USB devices via class drivers, which make USB easier to incorporate into products and embedded systems. ◙

Jim Lyle is a staff applications engineer at National Semiconductor where he has worked with flash memory, microcontrollers, and USB products. Jim has also worked as a development engineer and technical marketing engineer for Tandem Computers, Sun Microsystems, and Troubador Technologies. You may reach him at jim.lyle@nsc.com.

RESOURCES

USB information, www.usb.org
HID device information, www.usb.org/developers/hidpage.htm and www.microsoft.com/hwdev/hid
USBN9602 firmware source code, www.national.com/sw/USB

SOURCES

USBN9602, COP8
National Semiconductor
(408) 721-5000
Fax: (408) 739-9803
www.national.com

68HC11
Motorola
(512) 895-2649
Fax: (512) 895-1902
www.mcu.motsps.com

80C188EB
Intel Corp.
(602) 554-8080
Fax: (602) 554-7436
www.intel.com

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.