USB: A NEAT PACKAGE WITH A *few loose ends*

RICHARD A QUINNELL,
TECHNICAL EDITOR

EDN'S Hands-on PROJECT

When *EDN*'s call for hands-on project ideas came in late 1995, I chose the Universal Serial Bus (USB). At the time, the specification was still in draft stage as Revision 0.9, and components were only a promise. Like most of you, I wanted to apply an emerging technology that looked like the next winner. By catching the bus too soon, however, I was in for a rough ride.

I chose the USB for its potential. The bus can connect a standard, closed-cover PC to industrial peripherals that now require an add-in card or embedded PC. Connecting the peripheral device automatically configures both the peripheral and the PC and prepares the system to use its newly acquired capability. Further, that connection can supply power to run the peripheral device (see **box**, "USB fundamentals").

You can use such a self-configuring, self-powered system for numerous applications. For example, you can add a diagnostic monitoring port. You can build a USB data-acquisition peripheral into a system without using system resources. The system under test need not have its own control CPU or power source. Connecting a PC to the USB device powers the device and allows the user to monitor system operation, regardless of the system's status. The ability to perform remote diagnostics over a telephone line, for example, requires only the connection of a modem-enabled PC to the diagnostic port. The need for system-hosted communications capability vanishes.

My hands-on project had two objectives. The first was to build a USB peripheral device. The second was to use the device to "kick the tires" on the bus. In particular, I wanted to evaluate the bus's timing and data-bandwidth characteristics under severe load to learn how adaptable the USB would be to embedded needs.

To meet these objectives, I planned to create a data-acquisition peripheral. Data acquisition is a typical embedded application—one that could easily adapt to the USB. In addition, such a device would give me control over the device's data rate. That control would allow me to test the bus under various loads pushing the bus to its upper limit.

Once I had my peripheral, I would be able to measure the system's attained bandwidth and compare it to the theoretical bandwidth. The USB Specifica-

The USB for PCs can automatically configure and power peripherals. But, as I learned in this hands-on project, early USB adopters may face a substantial system-level design effort. They also need to carefully examine USB data-bandwidth characteristics.

My hands-on project had three major components. The IBM PC350 (left) acted as the USB host. The other PC was the development platform for the evaluation board in the middle.

PHOTO COURTESY USB IMPLEMENTERS FORUM AND INTEL CORP

tion Revision 1.0 (**Reference 1**) calls for the host to allocate bandwidth for a bulk-transfer device based on the formula:

Bus time (nsec)=9107+83.54 (Floor(3.167+ BitStuffTime(Data_bc))) +Host_Delay,

where BitStuffTime is the increase in raw bit count due to coding, Data_bc is the number of bits in the data packet, and Host_delay is the time the host needs between successive bus transactions. For a 64-byte data packet, worst-case bit stuffing, and minimum host delay, the formula yields a bus time of 59.4 μsec. That time translates to 1.024 Mbytes/sec because the number of bus transactions each millisecond must be an integer.

This formula takes into account worst-case conditions for clock rate, signal delays, and data pattern (see **box**, "USB bandwidth analysis"). It also includes several host-system-dependent unknowns. I intended to use my peripheral to estimate these unknown

USB FUNDAMENTALS

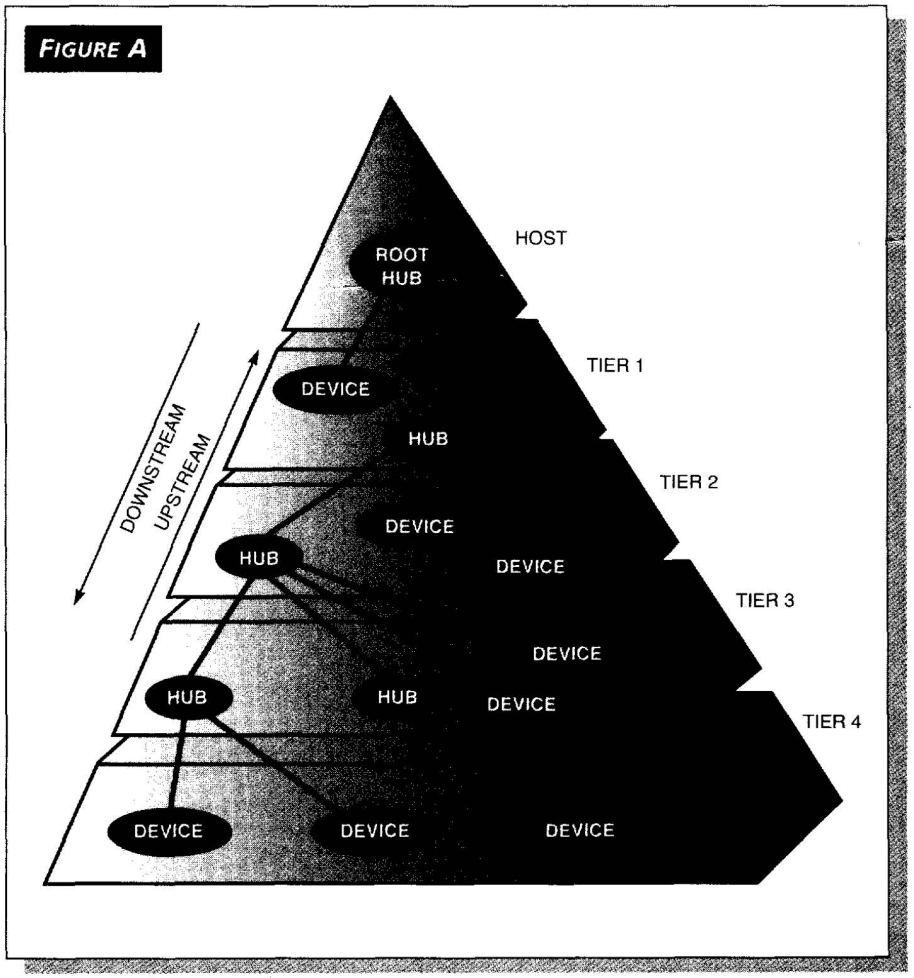
The developers of the Universal Serial Bus (USB) specification had two objectives. One was to collect all of a PC's I/O ports into one interface, providing live-insertion and automatic configuration. The other was to provide sufficient bandwidth to handle telephony applications along with typical peripherals. The design they settled on is a four-wire, half-duplex serial bus running at either 1.5 Mbps (low speed) or 12 Mbps (full speed).

From the outside, the USB appears simple. Plugging a peripheral device into the bus causes the PC to automatically respond. The operating system recognizes the device, loads the appropriate drivers, and configures the device for operation. The PC can distinguish between identical devices on the bus because the bus's physical configuration ensures a unique, identifiable connection path between device and host.

The USB's physical configuration has a tiered-star topology (**Figure A**). The PC serves as the host system and root hub. Second-tier devices, either peripherals or hubs, connect to the root hub's ports. Lower tiers connect through hubs in the tier directly above. There are no limits on the number of lower tier connection points a hub may offer. The specification allows as many as five hubs in a chain and a total of 127 devices on the bus.

A connection between a device and hub uses a four-wire, jacketed cable. Two wires form a twisted pair for data communications, the other two supply power and ground for hubs and devices without their own power source. Data flow in the cables may be downstream (hub to device) or upstream (device to hub). Power can flow only downstream and is limited to 500 mA at 5V. Cables intended for full-speed data include a shield; low-speed cables need not be shielded.

USB cable connectors come in two types (**Figure B**). The Type-A connector is a flattened rectangle that plugs into downstream-port sockets on the USB host or a hub. Cables permanently attached to a device, such as on a keyboard or mouse, use a Type-A connector. The Type-B connector, roughly square with beveled corners, plugs into upstream sockets on



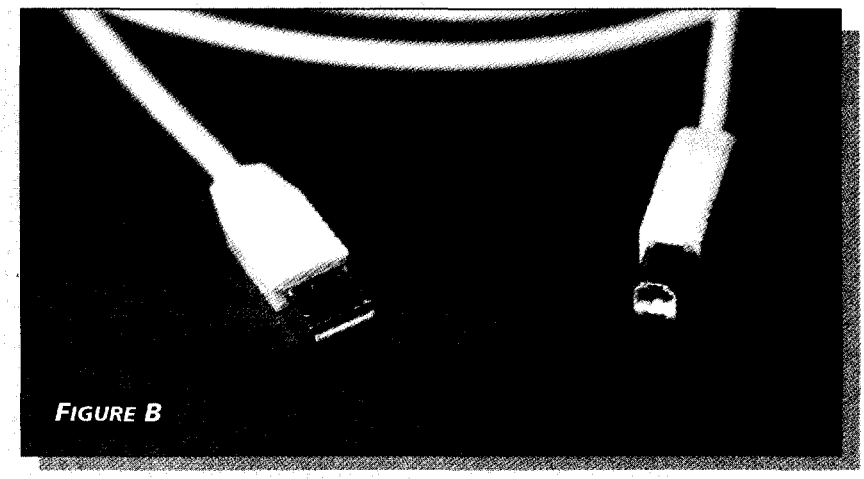
The USB's physical topology is a tiered star that can accommodate five layers and 127 devices.

values for my system and accurately estimate achievable bandwidth.

The first task, therefore, was to build a USB peripheral. Because the project began only a few months after the specification's final approval, I had few choices for hardware- and software-development tools. Following the project's genesis, more tools and improved versions became available (**Table 1**), so the choices I made represent availability, not a comment on relative merit.

I began my project by acquiring Intel's evaluation board for its 82930 USB microcontroller. The 82930 microcontroller blends an 80251 core with USB-specific hardware, including FIFO buffers, clock generation, and bit-stuffing and protocol hardware. The evaluation board includes the 82930, clock circuits, USB line drivers, external memory, and a ROM-based reduced-instruction-set monitor (RISM). The board also has parallel and serial ports that I planned to use to connect the board to an ADC module. One serial port links the board to

devices and hubs. The Type-B connector is used only for the device end of a removable cable, such as between a hub and a printer. This two-connector scheme prevents a user from accidentally creating a loop.



USB cable connectors come in two flavors. The flat connector is for devices with built-in cables. The second connector type allows detachable cables without the risk of cabling mistakes.

Simple operation, complex behavior

Although the USB appears simple overall, its internal workings are complex. You need to carefully examine those workings if you intend to use the bus as a pathway into your PC. In particular, you need to understand how the USB operates logically and the nature of its data transfers so you can evaluate its adaptability to your intended use.

Even though the USB's physical architecture has a tiered-star topology, its logical connection is point-to-point. The host system establishes independent communications channels, or pipes, between application software and individual control or data ports on peripheral devices. Channels can carry data (stream) or control/status (message) information. Pipes have several attributes, including bandwidth allocation, packet size, information type (stream or message), and direction of stream data flow.

The host establishes these pipes by assigning a unique 7-bit address to each device on the bus, a process called "enumeration." Enumeration occurs on power-up and whenever a device attaches to the bus. During enumeration, the device reports its configuration to the host, identifying the device's accessible data and control pathways. Each pathway has its own 4-bit subaddress, called an "endpoint." The address and endpoint definitions allow the host to determine the correspondence between application-software functions and device pathways and create the necessary pipes. An application program, thus, has a direct logical connection to a device's data channels or control registers.

Having multiple pipes on a single half-duplex bus requires some form of multiplexing. The USB uses time-domain multiplexing under control of the host system, working with 1-msec frames as the basic time segment. The host initiates all bus transfers, giving it control over the allocation of time within frames to each pipe. The allocation is not static, however, but varies from frame to frame. Specific allocation depends on the type of information transfer the pipe must provide and whether the application software has requested a transfer.

The USB specification recognizes four transfer types: isochronous-data, bulk-data, control, and interrupt. Isochronous-data transfers typically carry time-critical information, such as audio, so the host must allocate time in each frame for an isochronous stream pipe. Interrupt transfers receive time slots every nth frame. All other transfers receive time on an as-available basis.

To ensure that all transfer types have access to the bus, the USB specification limits the time that the host can allocate to transfers. Isochronous-data and interrupt transfers together can occupy no more than 90% of available bus time. Control transfers have the next priority, with bulk-data transfers receiving any remaining time. The host is responsible for resolving contention among control-transfer requests and among bulk-data-transfer requests for the time available.

The host reserves bus bandwidth for isochronous and interrupt transfers during device enumeration. If a transfer needs more bandwidth than is available, however, the host does not finish that device's enumeration. This inaction effectively denies the device access to the bus.

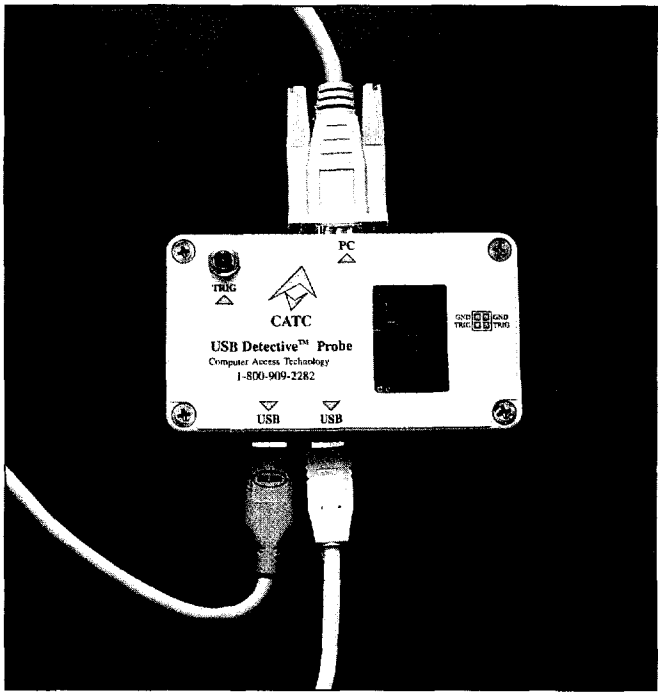
Reservation of bandwidth does not allocate bus time but

(continued on pg 42)

the development system.

The evaluation board is part of a peripheral developer's kit available from Intel. The original kit comprised the evaluation board, a host system, driver software, and sample application software. Unfortunately, no host systems were available for me to borrow, so I had to locate my own USB-ready PC. The sample software was specific to the Intel host system, so I couldn't adapt it to my project.

The first USB host systems to enter production were the IBM PC 300 series PCs. I borrowed a PC 350 from IBM. It has two USB ports on the back panel and BIOS extensions that support the USB. To be completely ready as a USB host, however, the system needed an updated version of Windows 95. The commercially available version would not recognize the USB port or run USB device drivers. I obtained from Microsoft a beta version of the latest Windows 95 OEM release for my system, along with some USB loopback test drivers.



Deciphering the USB's activity with a basic logic analyzer would be a nightmare. Fortunately, bus-analysis tools, such as the CATC USB Detective, are available.

To turn the evaluation board into a peripheral device, I would have to program the 82930. For that task, I needed software-development tools. The evaluation board came with demonstration tools from Keil Software, Production Languages Corp (PLC), and Tasking. I chose the Tasking software for its debugger and because its crippled-for-demo state was sufficient to meet my anticipated development needs.

I also needed tools for developing device drivers and application software under Windows 95. This requirement, in turn, required that I obtain a copy of the Windows NT Driver Developer's Kit from Microsoft. I was able to obtain several sample drivers but could not get what I needed to run Windows NT and develop the custom drivers that my project would require.

Too little, too late

At this point, I had to reevaluate my approach to the project. Given my limited resources and a six-week deadline, I faced an impossible development effort. The pieces of this puzzle were all too new and had raw edges. These pieces weren't going to go together very well, and I had few tools at my disposal that would let me trim the pieces to fit. And, I was missing a few pieces.

Figure 1 helps illustrate the effort I faced. Of the six major blocks in a USB

USB FUNDAMENTALS (continued)

merely tracks anticipated demand. The host system allocates time within a frame only if the application software has requested time. An audio CD device, for example, may have 50 kbytes/sec reserved on the USB, but the host doesn't allocate time if the CD is not playing. Any reserved, but unused, bus time becomes available for bulk-data transfers.

The mix of transfer types offers design trade-offs among data rate, latency, and data integrity. Isochronous data transfers have a guaranteed data rate and bounded latency. They receive time every frame, although a given transfer's position within the frame may vary. The drawback to isochronous transfer is that the data is not guaranteed. If data loss or error occurs, the USB does not resend isochronous data. Isochronous transfers can have data packets as long as 1023 bytes and must always run at full speed.

Interrupt transfers also have a guaranteed data rate and bounded latency but may run at full or low speed. Every n frames, the host queries a device through its interrupt-stream pipe. If the device has interrupt information, it returns a single data packet as long as 64 bytes. If an error occurs in transmission, the device resends the information at the next query.

Bulk data and control transfers have no guarantees on their data rate, although control transfers have the first claim to 10% of the bus bandwidth. Errors in bulk data and control transfers prompt a retry. If several transfer requests are pending for the same pipe, the retry occurs before the pending transfers. Bulk- and control-data packets can be as long as 64 bytes and may run at full or low speed.

Understanding these trade-offs and the underlying logic of the USB's operation should allow you to estimate how well the bus can meet its intended application's interface needs. For more detailed design work, you need a copy of the specification. It is available for $35 from the USB Implementer's Forum, JF2-51, 2111 NE 25th Ave, Hillsboro, OR, 97124. You can also download it free from the USB home page, http://www.teleport.com/~usb.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.