Application Note AN2298

PSoC™-Based USB Device Design By Example

Author: John Hyde
Associated Project: Yes
Associated Part Family: CY8C24794
PSoC Designer Version: 4.2
Associated Application Notes: Many Apply

Abstract
The real-world interfacing capability of the Cypress PSoC™ device family is well known and now this technology can be used to exchange data with a PC application program. This note introduces the newest member of the PSoC family, the CY8C24794, which includes a full-speed USB User Module alongside the now-familiar, programmable IO user modules to provide an instant connection of a PC-to-real-world IO. Following a short introduction to the new elements available to you, this note covers some USB essentials then works through three example projects built using the CY8C24794 evaluation board. Within a day or so you will be using this board to implement many of your latent USB product ideas.

Introduction
For the first time, a full-speed USB interface has been added to a PSoC device. The CY8C24794 is a combination of technologies that will create opportunities for a new range of low-cost, mixed-signal USB products. To help you move rapidly into this product area, Cypress Semiconductor has created an evaluation board that connects directly to the ICE-Cube in-circuit emulator.

This Application Note first describes the features of the CY8C24794 and covers some essential USB theory. It then applies this theory in three example projects. We start with a basic “Buttons and Lights” project so that you can quickly familiarize yourself with the tools. We then step through two more examples with gradually increasing complexity. By the end of this introductory note you will have the confidence to tackle your own USB design. There are several features of the evaluation board that are not used in this note (serial connection, joystick, CapSense™ buttons and slider), but these will be the subject of future Application Notes (or a design guide).

CY8C24794 Overview
The CY8C24794 ships in a 56-pin MLF package, two of which are shown in Figure 1.

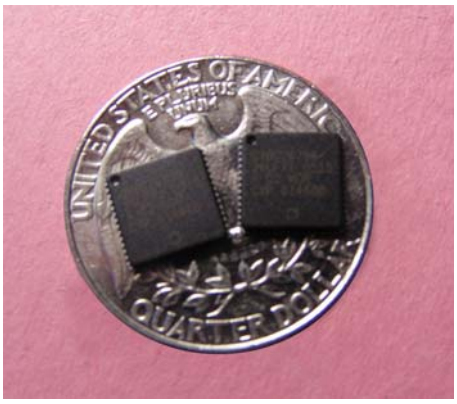

Figure 1. The MLF56 Package is Small!

This tiny 8mm x 8mm x 1mm package belies the capability of the component, which is even more evident in the block diagram shown in Figure 2.

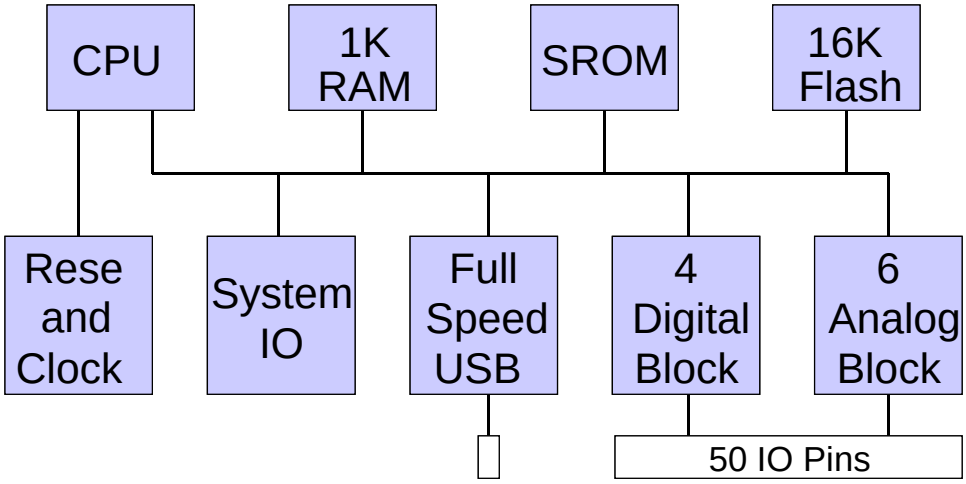

Figure 2. Block Diagram of the CY8C24794 Component

The CPU of the CY8C24794 is an M8C that can run up to 24 MHz. The M8C is an enhanced version of the M8B that is used in the Encore range of low-speed USB devices; it adds addressing modes and a TST instruction that removes the accumulator as a computational bottleneck. The RAM is extended to 1024 bytes and this is addressed as four pages of 256 bytes. An added set of auto-indexing instructions and CPU flag bits make multi-page operations efficient. The program memory includes a ROM-based supervisor program and 16 KB of Flash memory; note that the CY8C24794 can program its own memory and this will be the subject of an upcoming BootLoader Application Note. The M8C is supported by a new release of the iMAGEcraft C compiler that generates good code; all of the examples in this note are written in C.

The CY8C24794 has 4 programmable digital blocks and 6 programmable analog blocks. There are 50 IO lines and 48 of these can be used as analog inputs. There are 2 programmable analog outputs. Cypress has many Application Notes on these programmable modules so I will not “reinvent the wheel” in this note, rather, I will focus on the unique full-speed USB User Module.

The USB User Module is implemented as a separate serial interface engine with a dedicated 256-byte RAM buffer. This serial interface engine manages up to four data endpoints that can be individually programmed as IN or OUT endpoints. Endpoints are explained in the “USB Essentials” section.

Operation of the USB interface is autonomous and operates in parallel with standard CPU operations. The USB User Module includes a library of routines to manage data flow into and out of the data endpoints. In fact, the USB theory that you need to understand to use the CY8C24794 effectively is minimal (and is covered in a later section) since the library routines do almost all of the work for you. This is why I called the CY8C24794 “an instant connection” since the base software needed for USB operation is already written, debugged and included so that you can focus upon the data transfer requirements of your USB device.

To further ease the journey into USB land, the user module includes a wizard that guides you through the creation of the descriptors required by the USB standard. I will explain these in more detail in the example sections. The user module also includes an HID Wizard to simplify the construction of Human Input Device interfaces; I will also cover this in the example sections.

The CY8C24794 also includes a set of system resources such as I2C, digital clocks, decimator, two MACs, POR, and LVD reset circuitry.

Evaluation Board

An annotated photograph of the CY8C24794 evaluation board is shown in Figure 3. The sharp-eyed reader will notice that the component shown in Figure 1 is not on the board! The evaluation board uses a CY8C24794 in a 100-pin PQFP package: the extra pins of this package contain debug signals that are controlled by the ICE-Cube in-circuit emulator. The emulator adds single-step, break-pointing and trace capabilities to a CY8C24794 project. A block diagram of this "debug" component is shown in Figure 4. If you do not have an ICE-Cube you can still use the ISSP connection to download and run programs. I used the ICE-Cube connection to develop the examples that we will work through later in this Application Note.

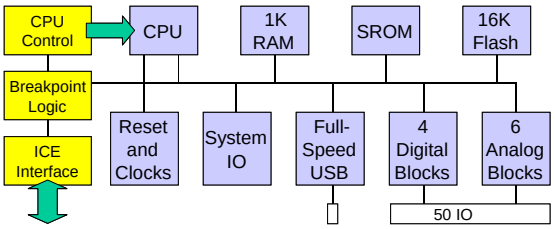

Figure 4. The "Debug" CY8C24794 is in a Bigger Package

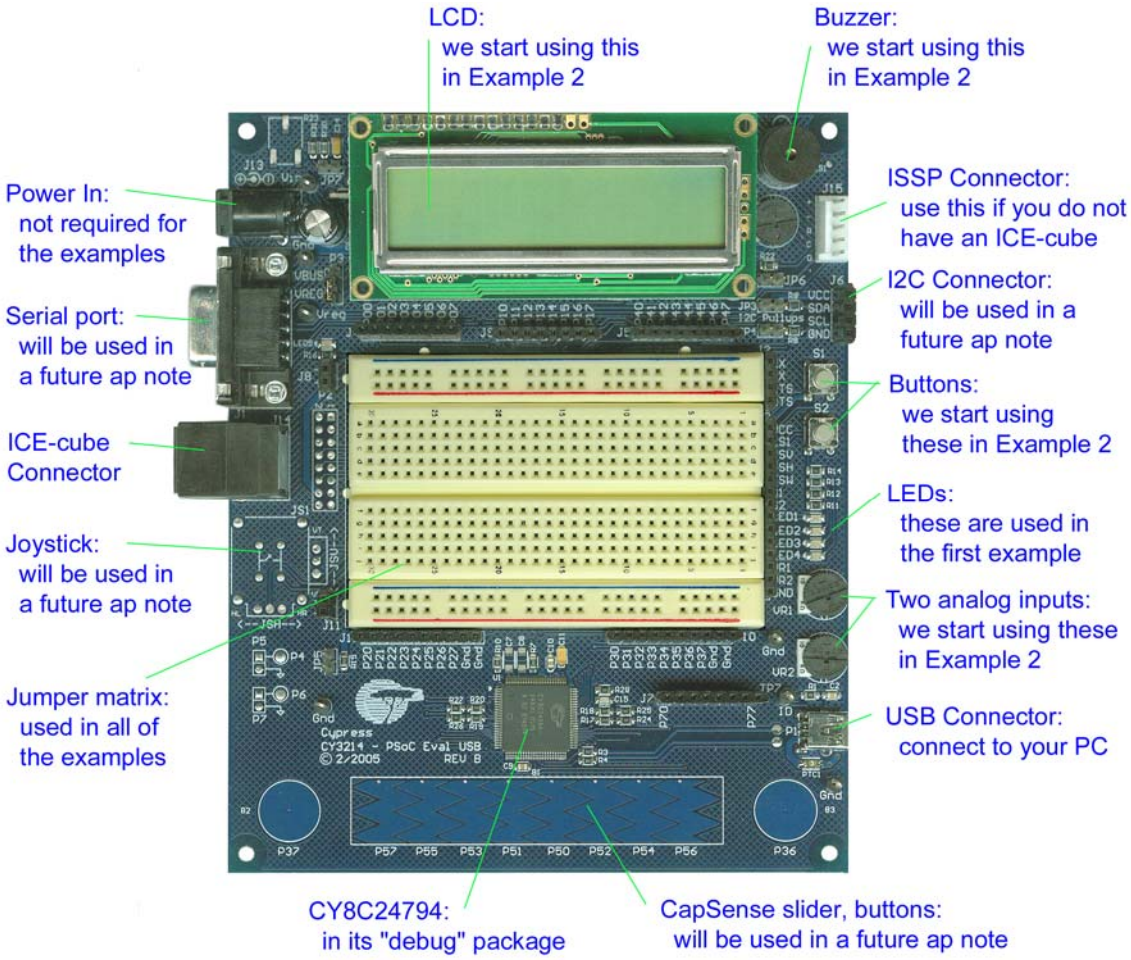

Figure 3. CY8C24794 Evaluation Board that we use for our Examples

USB Essentials

This section covers the essential USB theory that you need to know to use the CY8C24794 effectively. Note that this is not the full story since I only cover simplified, full-speed, device operation details in this note; a complete presentation takes about 100 pages and you can find these in the references. The key concept that I want you to grasp is:

$$\text{Device} = \sum (\text{Configurations} = \sum (\text{Interfaces} = \sum (\text{Endpoints})))$$

$\sum$ means "collection of." The other terms will be explained over the next two pages.

USB is a master-slave, shared, polled bus that has its data transfer protocol defined in hardware. A single master, also called a host controller, controls all bus communications and shares the available bandwidth of a USB channel between up to 126 slaves, also called devices or functions. A USB channel is just four wires: power, ground and a generally differential pair of signals (some of the protocol is implemented with individual line signaling). The signal lines are uni-directional and their direction is switched within the protocol. There is no physical clock line but a clock is embedded in the signaling scheme. USB communication uses packets that include error checking and several defined packet types are used in sequence to implement a robust data exchange. Figure 5 shows some of the more interesting packets that the CY8C24794 will handle.

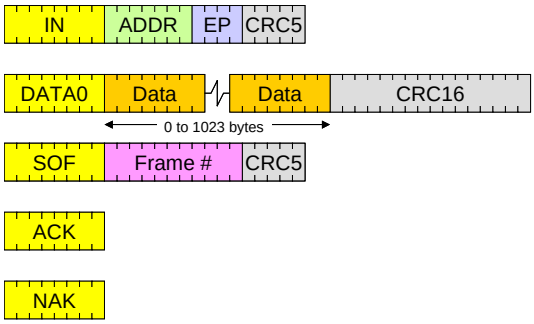

Figure 5. Representative USB Packets Handled by the CY8C24794

The master schedules packets within a 1 msec frame. It always broadcasts a Start-Of-Frame (SOF) packet every 1 msec, which can be used as a timing reference. There is a lot of software running on a host controller that decides which packets should be allocated for which devices within each frame: most packets (SOF is an exception) include the address of the target device and our device only needs to respond to those packets that are addressed to it. Our device is allocated its unique address when it initially connects to the host.

There is a defined process for a new device to join the shared USB communications scheme. This process, called enumeration, requires that the device provide information in a pre-defined format, called descriptors, to the host so that it can identify the device and its characteristics. The host controller uses this information to decide if the device can be connected and, if so, it assigns a unique address and loads a device driver. Figure 6 shows the software view of a USB device connection: notice, in particular, the layered software levels in the PC and the hierarchical structure of a device.

Let us assume that the enumeration process has been successful and a device driver has been identified and loaded. A **device** can have several **configurations**: most devices only have a single configuration (all of our examples are like this) but the USB specification allows the flexibility for a single device to have multiple personalities depending upon external factors such as available power or IO capabilities. Only one configuration may be active at any time.

A configuration can have several **interfaces**: the interface defines what the device does and it is the interface that has a matching device driver on the PC. It is common for a single device to have more than one interface – the device is logically viewed as a collection of interfaces that operate independently and multiple interfaces operate concurrently. The USB specification defines classes of devices and most operating systems (Windows, OS X, Linux, etc.) provide class drivers for a wide range of devices: examples include printer class, mass storage class, Human Interface Device (HID) class, hub class and audio class. The benefit of using these existing class drivers is that we don't have to write any complex PC-based driver software when we can define the operation of our device within one or more of the supported device classes. Our first set of examples will use one or more HID class interfaces, later examples will use other class drivers. It is also possible to use a vendor-defined interface and write your own device driver but, be warned, this is a huge undertaking in development, qualification and support. We will *not* be going this route.

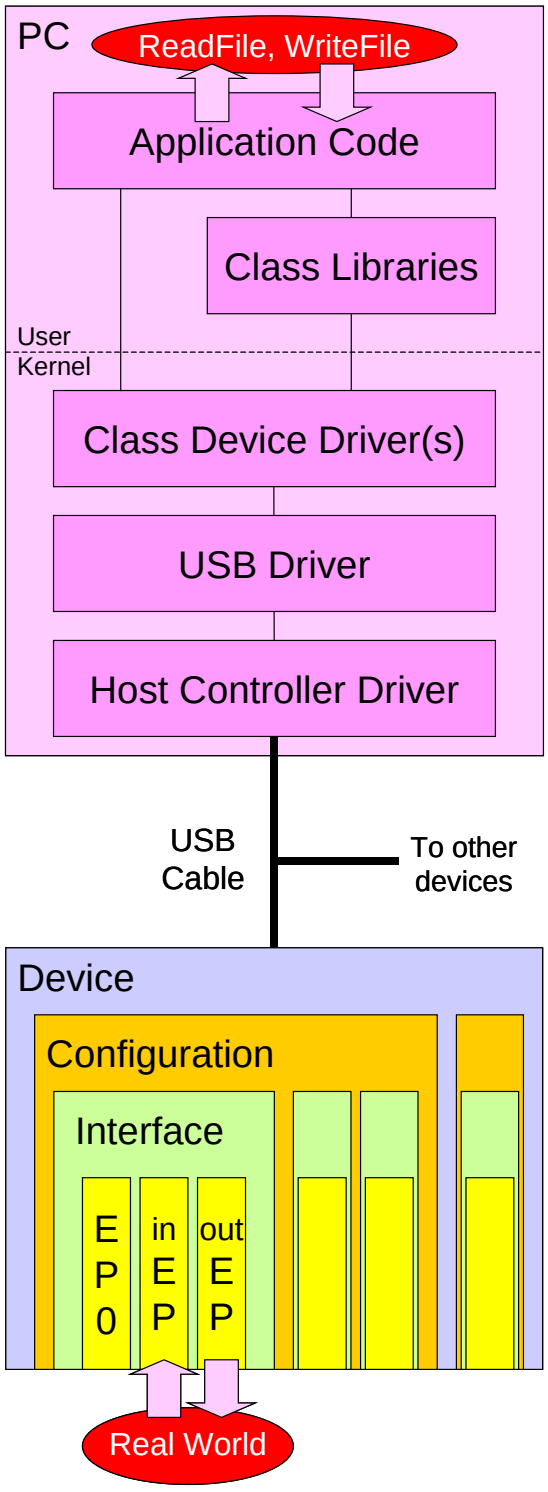

Figure 6. Software View of a USB Connection

An interface can have several **endpoints**: an endpoint is the source (IN endpoint) or sink (OUT endpoint) of data and this is where the real world attaches to USB. A device always has a control endpoint (EP0) and other data endpoints are defined as required by the data transfer needs of our application. The CY8C24794 includes four data endpoints, which enable the support of several interfaces.

Looking again at Figure 6 we see a PC application at the top of the diagram and the endpoints at the bottom of the diagram. There is a lot of software between these two and, fortunately for us, it has all been written, debugged and provided to us to use. To write data to the real world, the PC application program does a WriteFile(data) and the PC software passes this down its stack and onto the bus; here, the CY8C24794 accepts the data and passes this to the required OUT endpoint buffer. How it does this is interesting but we don't need to understand the details – we can just accept the data at the CY8C24794 endpoint and act upon it. Similarly, to provide data to the PC application, we copy real-world data into an IN endpoint and mark it valid; the PC will accept this data on its next scheduled poll of the CY8C24794 endpoint and pass it up the stack and buffer it waiting for the application program to do a ReadFile(data). Again, the CY8C24794 firmware need not be concerned on how the data gets to the application program, it just needs to know that it must load and validate endpoint data and this "appears" in the PC application program.

Are you beginning to see why I call the CY8C24794 an "instant" connection to USB?

In summary, a USB device is a collection of configurations (typically one), which is a collection of interfaces (often several), which is a collection of endpoints (EP0 always, typically one or more data endpoints). At power-on it must provide descriptors to the host and, once enabled, can accept PC data from an OUT endpoint and can provide data to a PC via an IN endpoint.

Okay, that's enough theory. Let's get on with the examples . . . . .

Example 1: Buttons and Lights

Our first example is the simplest that I could think of: the hardware equivalent of the fundamental "Hello World" program is "Buttons and Lights." This has the benefit of being useful (it moves button data from the real world and into the PC and moves lights data from the PC into the real world) but the major reason I chose it is that it is simple enough such that it does not distract us from the main goal of the first example, which is an introduction to the new aspects of the tools and development process.

Figure 7 shows the hardware setup that we will use for all of the examples in this note. I show two PCs in the figure for clarity of explanation; in reality you could use a single PC running different programs in separate windows but the text assumes two logical PCs. The preferred development method uses an ICE-Cube in-circuit emulator but, if you don't have one, the mini-programmer will suffice for these debugged examples.

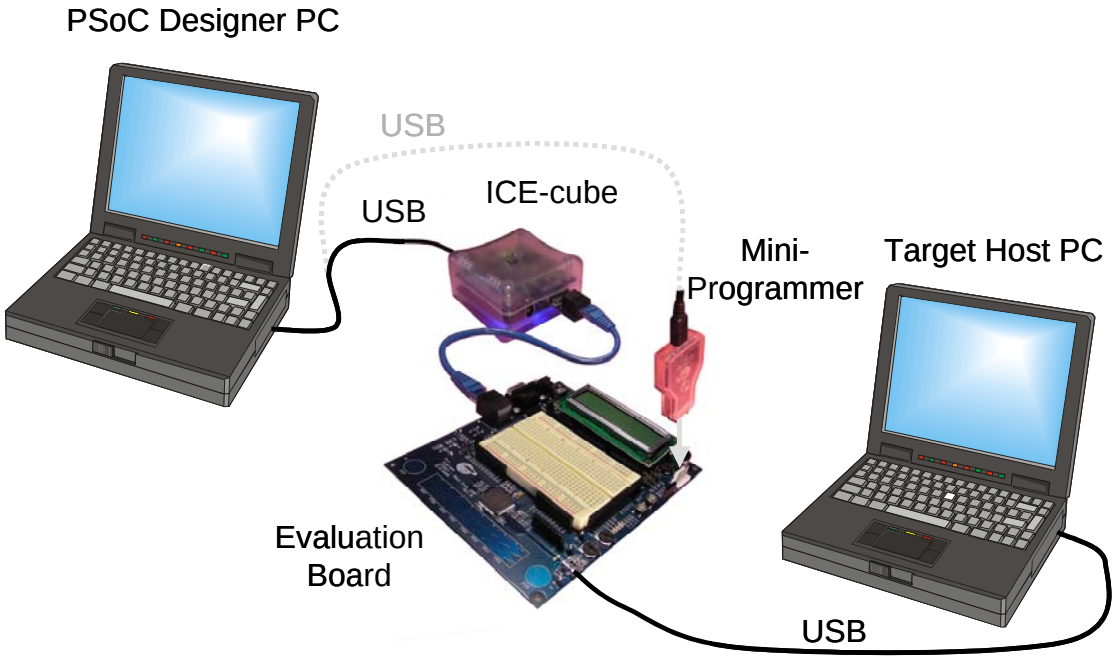

Figure 7. Hardware Setup for the Examples

We first need to set up the hardware for this example. Jumper wires should be added to connect the LEDs to port 3 and jumper wires will be used as the "buttons" on port 2. The connections are shown in Figure 8. I chose simple hardware so that we could focus on the development process.

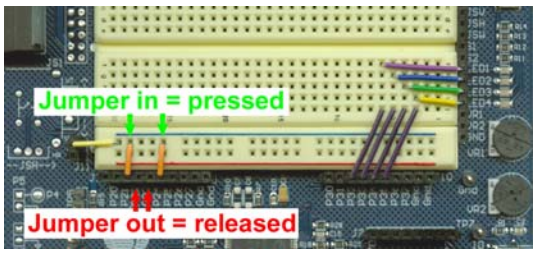

Figure 8. Jumper Wires for Example 1

I assume that you have the PSoC Designer™ software loaded on your PC and that you are familiar with it. If this is your first experience with this toolset, please read AN2010, "Getting Started with PSoC" (available on the CD-ROM included with these examples or on the Cypress web site) before proceeding further.

Start PSoC Designer and open *example1.soc* in the Debugger subsystem (we will come back later to the other views). At this time you should also start *example1.exe* on the host PC: the display will look similar to Figure 9 and the message box should tell us that it cannot find the ButtonsAndLights device. This is okay since we haven't started our device software running yet. I wrote the host program in Visual Basic and, if you have a copy, you will be able to step through the code.

I am not expecting you to have Visual Basic nor is it important for this example but the curious reader can open the source files (*.frm and *.bas using any text editor) to see exactly how I created the host program.

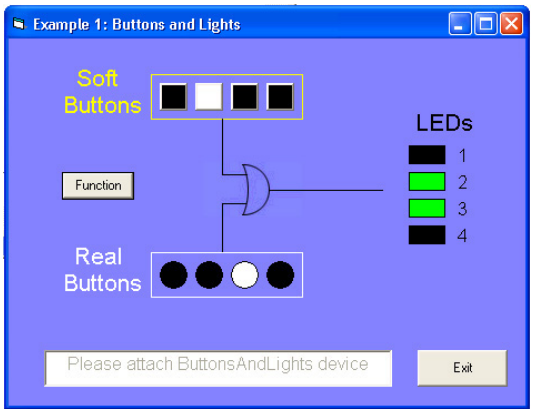

Figure 9. Companion Example 1 Host Program

The important thing to realize is that with **all** USB device designs you are **always** dealing with **two** programs: one in the device and one on the host, and these must talk nicely to each other for you to make progress! Rather than take valuable space with all the source code listings in this Application Note, I have drawn a flow chart for each program in Figure 10 (note that all of the source listings are on the CD-ROM that accompanies this Application Note).

Reviewing Figure 10 you will note that both programs are event-driven endless loops.

After some initialization, both programs wait for a timer event and the host program additionally waits for a button click (the device program handles its buttons in the timer routine).

Returning to the PSoC PC, start the ICE-Cube debugger running. If you do not have an ICE-Cube then this step should be replaced with "program the target device using the mini-programmer and ISSP cable"; once the device is programmed it will start to execute its program.

Watch the host PCs' screen and you will see the enumeration of a new device taking place. Depending upon the version of your operating system, you will see different messages appear on the screen. Once this initialization is completed, both programs will be controlled by their event loops. When the device detects a button press event it will send a "buttons" report to the host. When the host detects a button press event it updates the local LED display and sends a "lights" report to the device. When the host receives a "buttons" report it will update the local LEDs (which causes a "lights" report to be generated). When the device receives a "lights" report it will update its local LEDs.

Try this now! Click the host soft buttons or insert jumpers on the evaluation board to "press" buttons. I added a small "frill" in the host program: you can change the function that drives the LEDs but note that the host LEDs and the evaluation board LEDs are always in sync.

We have two CPUs successfully communicating using USB! The next section looks in detail how we accomplished this task.

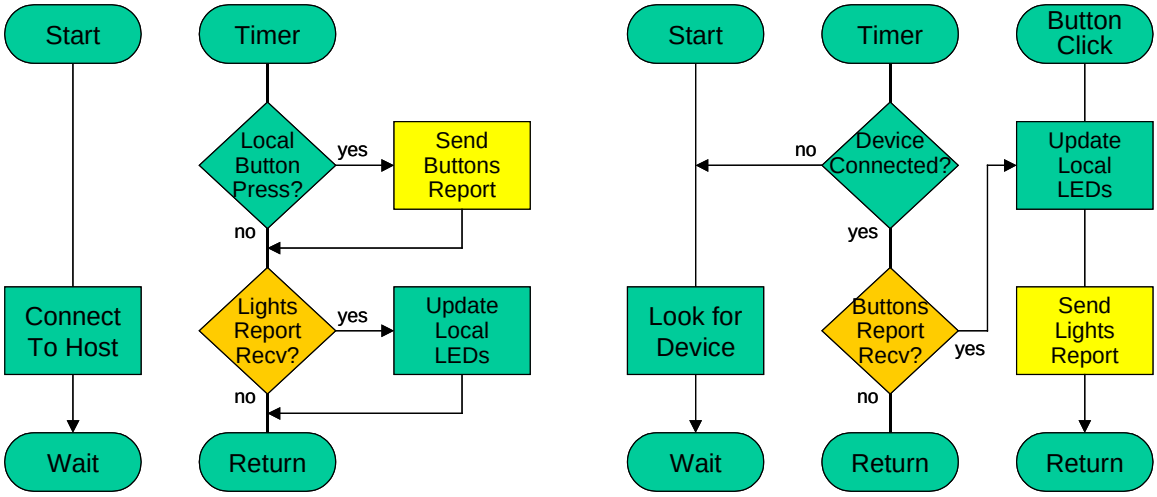

Figure 10. Flow Charts of the Device Program and the Host Program are Similar

Let us return to the PSoC Designer PC and discover how the device program is working. Stop the ICE-Cube debugger and switch to the Application Editor subsystem. Open *main.c*, it will look similar to Code 1 and, as you can see, it is straightforward.

```
void main() {
   init_hardware();
   M8C_EnableGInt;
// Start an enumeration with the host
   USB_Start(0, USB_5V_OPERATION);

// Wait until the host enables the device
   while (!USB_bGetConfiguration());
   USB_INT_REG |= USB_INT_SOF_MASK;

   while (1) {
      if (SOF_Flag) {
// Arrive here every 1 msec
        SOF_Flag = 0;
        buttons_now = scan_buttons();
        if (buttons_report != buttons_now) {
// If buttons have changed then tell the host
          buttons_report = buttons_now;
          USB_LoadInEP(1, &buttons_report, 1,
                                 USB_TOGGLE);
          }
// If a lights report has been received
// then update the local display
        lights_report =
              USB_INTERFACE_0_OUT_RPT_DATA[0];
        update_LEDs(lights_report);
        }
     }
   }
```

Code 1. main.c for Example 1

The `USB_Start()` call initiates the enumeration sequence that is handled by the USB User Module libraries; I wait for enumeration to be completed then I post a buffer to receive a lights report from the PC; I then wait for an SOF flag to be set. The SOF_ISR sets this flag every 1 msec.

Once the SOF flag is set I call `scan-buttons()`, which is a button debouncing routine. If I detect a button change then I send a buttons report to the PC *[**technical sidenote**: a USB device does not actually "send," it prepares data that the USB host will come and collect; remember that the host controls all communications and a device only "talks" when the host permits it; the result is equivalent to a "send"];* I then check to see if a lights' report was received in the previous frame and, if so, I update my local LEDs. Note that from MAIN's perspective, data is moved into and out of endpoint buffers; the USB communications is handled in the background by the Serial Interface Engine (SIE); thus, USB run-time operation is as simple as reading and writing endpoint buffers.

But it can't be this easy, can it? Well yes and no! You should be asking, "How did the host and device know that the buttons and lights reports were only one byte long? And, for that matter, how did the device know that it had to use reports at all?" To answer these questions we need to switch PSoC Designer to the Interconnect View in the Device Editor subsystem.

This example uses a single user module – the full-speed USB User Module – and this does not consume any of the programmable resources so there are still 4 digital blocks and 6 analog blocks free. At this time you should check the global resources (I did not use any system clocks in this example either) and the IO assignments (where you will see the buttons in and LED out signals).

Now right-click on the USB User Module and choose "USB Setup Wizard." Expand all of the entries so that your view looks similar to Figure 11. These are the USB descriptors that were discussed in the "USB Essentials" section – their format is defined by the USB Spec. and we enter our information to describe what our device is and does.

Figure 11 is divided into three sections: device descriptors, string descriptors and class descriptors. Strings are optional but I always include them since they make debugging simpler. I have defined three strings: my manufacturer name, the product name and a serial number. You could edit these strings, if desired.

I shall describe the device descriptors from the top down. First are device attributes.

All USB devices are required to have a Vendor ID and a Product ID. The Vendor ID is assigned by the USB Implementers Forum (at http://www.usb.org/) and I have entered my ID. You are welcome to use this for development but, should you wish to sell a product, then you **must** obtain your own Vendor ID. I assigned a Product ID to identify this example from all of my other examples. I also chose a device release of 1.00. The device class and subclass are set to 0 in the device attributes since I define these in the interface descriptor. The remaining three entries are the strings that I have already defined.

A device contains a collection of configurations and, in this example, I only have one. This is defined by a configuration descriptor. I declare a maximum power used by the ButtonsAndLights device as 100 mA and this will be provided by the USB cable. This value characterizes the device as low power and therefore can be attached to any host socket.

We can specify up to 500 mA, a high-power device, and still be bus powered but we would not be able to operate when attached to an un-powered hub.

A configuration contains a collection of interfaces and, in this example, I only have one. USB is a very flexible communications method and the underlying hardware can support a vast array of diverse devices. As mentioned in the "USB Essentials" section, the USB developers grouped similar devices into classes and the OS writers provide generic class drivers for most of the defined classes. I always encourage my clients to use standard class drivers even if it involves modifying the device slightly to fit, since the world of custom drivers is expensive and painful.

Class drivers are primarily characterized by the data transfer requirements of devices. A mass storage class device, for example, needs to move a lot of data reliably but the timing of transfers is not a concern. Contrast this with an audio class driver that needs to move time synchronized data – here late data is just as bad as no data! I chose the Human Interface Device (HID) class for this example since its data transfer characteristics (infrequent, small but reliable transfers) matched the ButtonsAndLights device. You can think of the HID driver as a "generic byte mover" since its application range extends well outside the human interface range. Many applications involving (slow) real-world activities can use the HID driver and you don't even need a human interface!

An interface contains a collection of endpoints and an HID class device is required to have an interrupt IN endpoint. It can optionally define an interrupt OUT endpoint but, in this example, I will use EP0 for data received from the host. I do not need to declare this since EP0 is always present.

An HID class device requires a report descriptor that defines exactly the size and format of the data exchanges. Look at the report descriptor at the bottom of Figure 11. It first defines a Vendor Defined (i.e., custom) usage page, which means that the operating system will not try to own the device (Example 2 will fall into this category). I then include the minimum required report entries to define a single byte packet (report size = 8 bits, report count = 1, logical min. = -127 and logical max. = 128) input and a single byte packet output. During enumeration the host HID driver will read in the report descriptor and use it to configure its internal buffers.

In summary, the USB Setup Wizard is used to define the descriptors for our device. The data transfer characteristics of the ButtonsAndLights device indicated that we should be an HID class device so we defined a report descriptor to describe the size and format of the data we will transfer. These tables are statically defined and enable the USB run-time code to be simple.

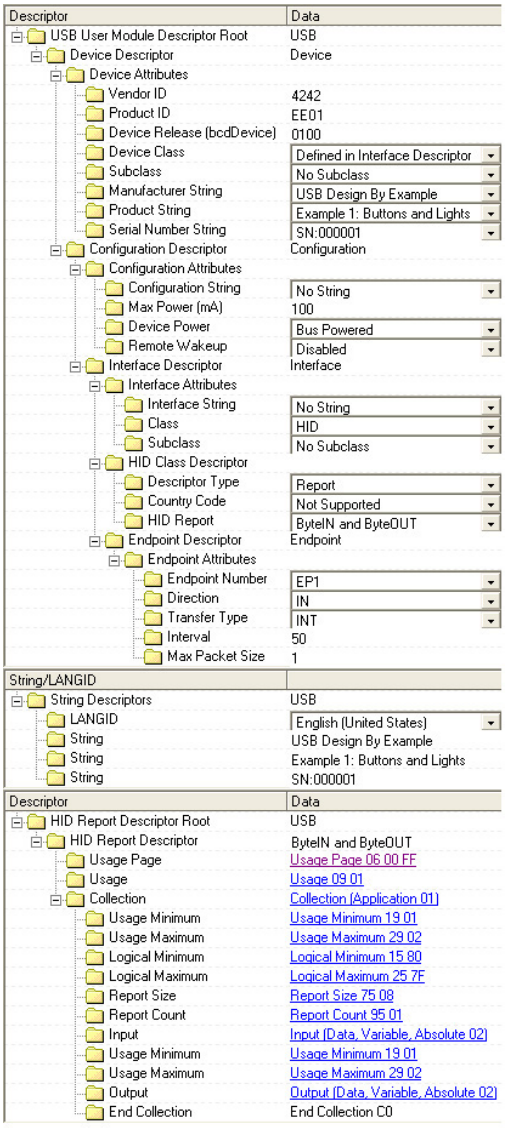

Figure 11. Descriptors for Example 1

With buttons and lights working, we can now set our sights on something more interesting.

Example 2:
Temperature Sensing Keyboard

Now that we have explored some of the USB aspects of the CY8C24794, it is time to expand our examples to use some other capability of the component. This example will operate like a keyboard although it will not physically look like a standard keyboard. This is one of the many benefits of USB: since the communications is protocol based, the host does not depend upon a particular physical implementation (above the USB Spec. -specified data transport), so you can swap out the device hardware if you discover something better or cheaper. The major investment in host software is preserved.

This example will be a temperature sensor using VR1 and VR2 as surrogates for real temperature sensors: they are easily accessible on the evaluation board and we can simply add jumper wires as shown in Figure 12.

Note that I connected VR1 and VR2 to the nearest available IO pins on port 3. This is another feature of the CY8C24794 that you will appreciate – almost all of the IO pins are identical and you can select whichever is closest, easiest to route, etc. I have completed several CY8C24794 designs and some have been on low-cost, single-sided boards thanks to this flexibility in the IO structure. Note that this routing did cost me an analog block for a PGA since the analog mux bus cannot directly drive an ADC block. Engineering is full of trade-offs!

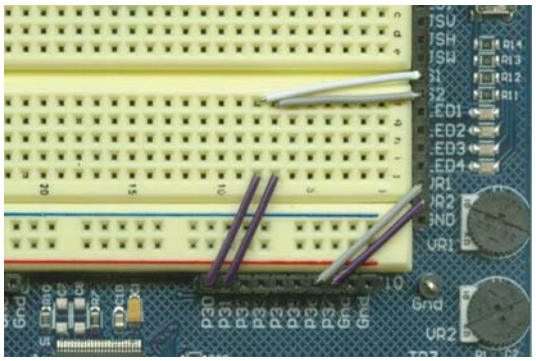

Figure 12. Jumper Wires Added for Example 2

Rather than develop a custom host application as we did in Example 1, I wanted to show the benefit of using software that is already installed on the host. I have set up descriptors so that this example looks like a standard USB keyboard. This means that this example will run with any host that supports a USB-aware operating system, such as Windows (all flavors since Win98 Gold), OS 9, OS X, Linux, VxWorks, etc.

I included the LCD User Module in this example to display the temperatures locally. This module does not use any of the programmable resources, it is just code that will reside in Flash memory. The LCD library implements all of the low-level data strobing to the LCD display mounted on the evaluation board and I will use the API to simply write to the display.

We need an ADC to read the voltages and I chose an ADCINC from the available suite of user modules. This ADC needs a counter and is placed as shown in Figure 13. Once triggered, this ADC measures the selected voltage independently so I included a task in the SOF timer loop to check for completion. The ADC subsystem produces two digital values of VR1 and VR2 every msec.

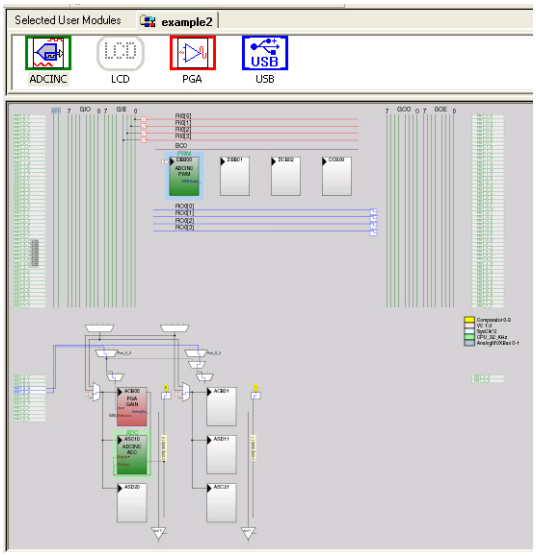

Figure 13. User Modules Configured for Example 2

Open the *example2.soc* project and select the Device Editor subsystem. Right-click on the USB User Module icon and select the USB Setup Wizard. Notice that I changed the report descriptor to define the device as a keyboard and I am also using EP2. The report descriptor describes an 8-byte IN buffer and a 1-byte OUT buffer as shown in Figure 14; the IN buffer is similar to the PS2 keyboard packet but note that HID usage codes are used in place of scan codes. The OUT buffer is used by the host to activate the LEDs on a keyboard. To learn more about HID devices in general and HID usage codes in particular, please refer to the references at the end of this Application Note.

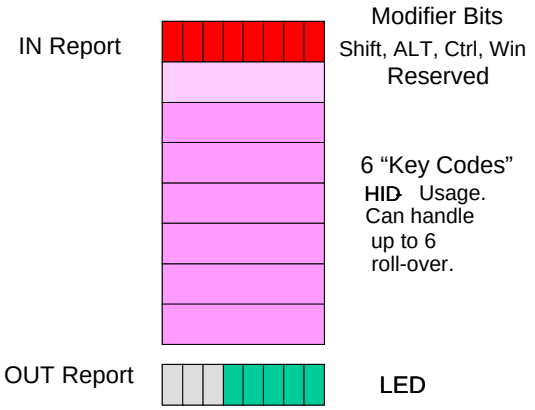

Figure 14. Data Format Defined by Report Descriptor

During enumeration, the host will recognize our device as a “standard” keyboard. If you are using Windows XP or Windows 2000 then these operating systems will also recognize the device as a “system input device” and will open it for exclusive access. This means that an application program, such as *example1.exe*, would not be allowed to open it to gain access to the reports. Microsoft tells me that allowing this would open up a security hole so it is not permitted. Please be aware of this restriction when configuring a device that the operating system will want to own!

The run-time operation of a keyboard is very easy; we just send reports using the format shown in Figure 14 to the host. One problem that I discovered during development is that the CY8C24794 is a full-speed device and can “type” much faster than a typical low-speed keyboard so therefore I had to add code to slow down the typing speed since Windows was missing characters. I decided to add two buttons, S1 and S2, also shown in Figure 12, to initiate the transfer. And we learnt in Example 1 that these buttons could be electronic buttons on the host.

When S1 is pressed, the example program reads the current temperature_1 and types the string “temp1 is xx” on the host. Similarly, S2 types the string “temp2 is xx.” Please switch PSoC Designer to the Application Editor subsystem and look at *main.c*, this is also repeated in Code 2.

This code is an extension from Example 1 and uses the same structure. Now switch to the Debugger subsystem, download the code and start it running (or program using the mini-programmer and start it running). While the enumeration messages are appearing on the host PC screen, start an editor or spreadsheet program on the host PC.

```
void main() {
   init_hardware();
   M8C_EnableGInt;

// Can now start my ADC
   PGA_Start(3);
   ADCINC_Start(3);
   ADCINC_GetSamples(1);
   LCD_Start();
   init_display();
// Connect to the host
   USB_Start(0,USB_5V_OPERATION);

// Wait to be enumerated
   while (!USB_bGetConfiguration());
   USB_INT_REG |= USB_INT_SOF_MASK;
   ADCINC_GetSamples(1);

   while(1) {
      if (SOF_Flag) {
         SOF_Flag = 0;
         Even ^= 1;
         Buttons = scan_buttons();
         if (ADCINC_fIsDataAvailable()) {
            Voltage[Even] =
                  ADCINC_bClearFlagGetData();
            DisplayTemperature(Even);
            MUX_CR3 = MUX_CR3 ^ 0xC0;
            ADCINC_GetSamples(1);
            }
         if (!Typing() && Buttons)
                  StartTyping(Buttons);
         }
      }
   }
```

Code 2. *main.c* for Example 2

Now press S1 and/or S2 and see the temperature values input directly to the editor or spreadsheet. Vary VR1 and VR2 and press S1 and S2 again.

Now connect a jumper wire between P2[7] on J12 and LS1 on J2. On the real keyboard connected to the host PC, press the CAPS_LOCK key – the buzzer on the evaluation board will turn on. Press it again to turn it off! When multiple USB keyboards are connected to a host system, it will OR the key-press inputs and will AND the LED outputs; so the OS sends an LED OUT report to both keyboards: I interpret the CAPS_LOCK LED as “turn buzzer on.” It took just 3 lines of code to add this feature: open *adcincint.asm* and search for CAPS_LOCK, it is in the ADC interrupt service routine. We were fortunate that this routine executes at about a 2 kHz rate, which is the nominal frequency of the buzzer. I check to see if the CAPS_LOCK LED is on and use this to activate the buzzer. Had this frequency not been available, we could have created it with a Timer User Module.

We have used standard building blocks to construct a simple data logger in less than an hour. Pretty amazing!

Now what other data would you like to get into your PC?

Example 3: Process Monitoring

I trust that the first two examples have shown that adding USB to a PSoC project is straightforward. In this example, we will implement a configurable process monitoring system: the device will collect analog data at a rate determined by the host, the host will then display this data in a graph, and will be able to save and reload data sets. The design will be simple and expandable and, when the example is completed, I will offer suggestions to extend it in multiple directions.

This design starts with the human interface on the host. Figure 15 shows the starting point for this example.

This example will collect 256 8-bit samples from the device at a rate determined by the rate slider on the host. The fastest rate will be 1 msec and the slowest will be 256 msec so that we can collect a wide variety of analog samples; the example will use VR1 but, once you understand how the example works, you will be able to use the output of any PSoC block.

To meet this specification we need only transfer a single byte in either direction every USB frame (similar to Example 1), so I chose an HID class interface for this example also. I did, however, specify two data endpoints to ease the later expansion of this design.

On the PSoC Designer PC, open *example3.soc* in the Device Editor subsystem, right-click on the USB User Module icon and select the USB Setup Wizard. Note that this example uses different endpoint parameters (when compared with Example 1); the data rate is much faster but still only a small fraction of what the CY8C24794 is capable of. The structure of *main.c* is also similar to Example 1 and is shown in Code 3.

Code 3 shows that *main.c* waits for a "collect samples" signal from the host; the value of the collect samples signal tells the device how many frames that it should skip when providing data. The host will poll the device endpoint at a 1-msec rate (see the value of 'interval' in the endpoint descriptor) and the device will NAK (Negative AKnowledgement) this request if data is not available. Thus, the device supplies data at the rate requested by the host.

Start running the program on the PSoC device and then start the companion *example3.exe* on the host PC. Set the rate to slow and click "Get Samples." Now vary VR1 and watch the waveform being recorded. You can also set the rate to high and click "Continuous" but, with VR1 as the source, you will just see a slowly varying horizontal line.

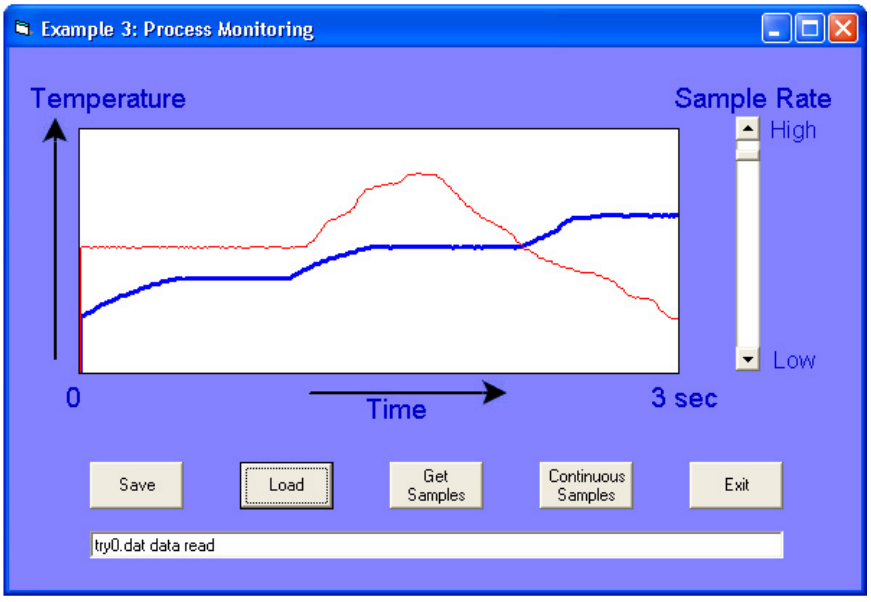

Figure 15. Human Interface for Example 3

```
void main() {
   init_hardware();
   M8C_EnableGInt;

// Can now start my ADC
   PGA_Start(3);
   ADCINC_Start(3);
   ADCINC_GetSamples(1);
   LCD_Start();
   init_display();
// Connect to the host
   USB_Start(0,USB_5V_OPERATION);

// Wait to be enumerated
   while (!USB_bGetConfiguration());
   USB_INT_REG |= USB_INT_SOF_MASK;
   ADCINC_GetSamples(1);
// Post a buffer to wait for a command
   USB_EnableOutEP(4);

   while(1) {
      if (SOF_Flag) {
         SOF_Flag = 0;
         if (ADCINC_fIsDataAvailable()) {
            Voltage =
                 ADCINC_bClearFlagGetData();
            DisplayTemperature();
            ADCINC_GetSamples(1);
            }
// Am I supplying data
         if (OutReport) {
// Have I finished supplying data
            if (--SkipCount == 0) {
               SkipCount = OutReport;
               if (SampleCount--) {
// Send another sample
                  DisplaySampleCount();
                  USB_LoadInEP(3, &Voltage, 1,
                               USB_TOGGLE);
                  }
               else {
// Finished with these samples, restart
                  OutReport = 0;
                  init_display();
                  }
               }
            }
// Did I receive a command?
         if (USB_bGetEPAckState(4)) {
            Count = USB_bReadOutEP(4,
                           &OutReport, 1);
// Setup to supply the samples
            SampleCount = 255;
            SkipCount = OutReport;
// Wait for another command
            USB_EnableOutEP(4);
            }
         }
      }
   }
```

Code 3. *main.c* of Example 3

You can also save the collected waveform using the “Save” button and reload and display a saved waveform using the “Load” button. The host application program is quite small and the source code is provided on the CD-ROM that accompanies this Application Note.

Project Expansion

This example is impressive at the high-data rate but is still only using about 2% of the capability of the CY8C24794. We can simply extend the application range while staying within the HID class or we can dramatically extend the range by using a different class driver.

The current firmware takes its 256 samples immediately after receiving a trigger command from the host. The firmware could treat this as an “arm” function and use an external signal to start data collection; this would be useful for transient analysis.

The CY8C24794 can use a 64-byte endpoint buffer to send data to the host. The device could take more samples within the 1-msec frame; it could take wider samples; or it could monitor several waveforms at the same time. The HID class can transfer one interrupt packet every frame so with a 64-byte packet on a full-speed bus, we could be transferring 64KB/sec into the PC using this example program.

If we wanted an even higher data rate then we could use isochronous (i.e., time dependent) or bulk data transfers. A full-speed isochronous data transfer would enable a rate of 1MB/sec and a bulk data transfer would enable a rate of 896KB/sec – so there is a lot of headroom for expansion. Isochronous and bulk transfers will require the use of a different class driver, which will be the subject of a future Application Note.

Summary

For those of you who thought that building a USB device was difficult, I must point out that you have built three of them while following along with this Application Note! And it was reasonably straightforward, wouldn’t you agree?

We have not, by any means, covered all aspects of USB or explored the full capabilities of the CY8C24794, but we have made a good start! Many USB devices can be built using the HID class framework and even more can be built using other class drivers: these will be the subject of future Application Notes but, for now, enjoy experimenting with these very capable tools.

I hope that my examples have started you thinking of other devices that you could create.

References

To learn more about USB I would recommend two books: *USB Design By Example*, by myself and *USB Complete* By Jan Axelson. I prefer mine but you may prefer Jan's style. If possible, you should get both of them.

You can download the USB Specification and all of the Class documents from the developers' section at http://www.usb.org/. They are a free download but, be warned, they are specifications and not easy to read.

There are also many PSoC Application Notes on the Cypress web site, several of which would benefit from a connection to a PC.

Happy developing!

About the Author

Name: John Hyde
Title: Design Consultant
Background: John has been involved with USB since its inception and has written several "USB Design By Example" books. He was also an early adopter of PSoC technology and was delighted when Cypress offered a single product with both of his favorite technologies. This introduction article will be the first of many, I am sure!
Contact: john@USB-By-Example.com

Cypress Semiconductor
2700 162nd Street SW, Building D
Lynnwood, WA 98037
Phone: 800.669.0557
Fax: 425.787.4641

http://www.cypress.com/