

# How Can I Add Universal Serial Bus To My Product?

Last Updated Wednesday, 29 April 2009

The Universal Serial Bus is now ubiquitous as the user friendly way to connect devices to your computer, but finding out how to add a USB interface to your embedded device is very confusing. This article will give you a basic understanding of what's involved and where to look for details required to implement a solution.

What USB is not! It's tempting to see the 'serial' of USB and assume that it must be like SPI or RS232 communications. They're serial, they're pretty straightforward to implement, so USB must be easy too, right? Well, while it's true that USB is a serial protocol, that's where the similarity ends. While SPI and RS232 specify the communication signals used and leave it at that, the USB standard specifies almost everything about how to use USB for communication &ndash; from connector dimensions to electrical specifications to transaction protocol. In fact, it is more instructive to think of USB communication as a layered protocol stack (a la HTTP &ndash; TCP/IP - Ethernet) rather than as a simple serial protocol. This makes implementing a USB stack much more difficult than implementing a SPI bus, for example. However, once implemented, the resultant product can be seamlessly integrated with any computer equipped with a USB master interface.

DisclaimersThe information given on this page is the result of implementing a working USB stack on the LPC2148 chip. If you spot any mistakes &ndash; by all means let us know (politely!). The lowest level of USB protocol covers the components of a single USB transaction &ndash; a token packet, optional data packet and status packet. This level is usually handled in hardware for example on the LPC2148 and so is ignored for the purposes of this discussion. Lastly, the definitive guide to the USB protocol can be found at <http://www.usb.org/developers/docs/> though USB beginners are warned that it's not the easiest document to digest... The (not so basic) basics of USBThe USB is a master &ndash; slave type of bus. Whenever you connect a USB device to your computer, the computer is the host (master) and the thing connected is the device (slave). ALL communication is host initiated. ALL directionality is host-centric. This means if the host wants to send data to a device, it just goes ahead and sends data OUT. If the device wants to send data to the host, it must wait for the host to request data before it sends data IN. Embedded peripherals are almost always devices, though as the popularity of stand alone USB connections increases (e.g. printers which can accept a camera connection), embedded host USB stacks look set to proliferate. This article only deals with device USB stacks. At the lowest level, there is the physical connection between host and device, and right above this is the transaction protocol alluded to above (commonly handled by hardware). Each device offers a function to the host, with each function consisting of one or more configurations. (Actually a device can also be a hub or a hub and several functions bundled together into a compound device, but we'll ignore those here). A device can only instantiate one configuration at a time. Each configuration provides one or more interfaces, and these interfaces act independently of one another. The interfaces are the pieces of code which actually allow data to be moved between the host and the device. Clear as mud &ndash; here's an example. The device in figure 1 has a single configuration which describes its function. The configuration consists of a human interface device (HID) interface and a mass storage device (MSD) interface. At the lowest level, the host and device exchange electrical signals over the physical connection. At the stack level, the host and device exchange USB protocol information which is sufficiently rich to allow data to be correctly routed between the host application layer and the device interfaces. At the application layer, the host has access to a HID (in this case a mouse) and to an external storage medium. Of course, it would be peculiar to place a data storage device within a mouse, but it's perfectly allowed by the protocol and works for this example. The next question to answer is how data is moved between the host and the device. Each interface moves data between the host and one or more endpoints. The logical communication between a host and an endpoint is called a pipe. A 'physical' endpoint is determined by a device's address (allocated by the host), the endpoint number and the pipe direction (either OUT from host to device or IN from device to host). There are two types of pipe, message pipes and stream pipes. Message pipes are bidirectional, linked to an endpoints IN and OUT directions and only support control transfers. Stream pipes are unidirectional, linked to either an endpoints IN or OUT direction and support all other types of transfer. Endpoint 0 is reserved for the host to control the USB device. The example above consists of: 1 configuration, 2 interfaces, 3 endpoints and 4 pipes:Endpoint 0: Control message pipe, with IN and OUT directions.Endpoint X: HID stream pipe, with IN direction (device to host).Endpoint Y: Two MSD stream pipes, one for each of IN and OUT directions. So what do I actually have to implement?The purpose of all the previous discussion is to get to the point where it's straightforward to describe what you need to implement. The first thing that is needed is to send incoming data to the correct endpoint handlers. After that, you must deal with enumeration. This is where communications over endpoint 0 allow the host to query what your configuration is and if you're behaving like a USB device ought to. If it's happy, you get allocated a bus address and the host checks for drivers to allow it to use your interfaces. The next thing to do is to implement all the interfaces your device needs. There are two choices here: you can create a new device class with its own interface, but this will also require a bespoke host-side driver; you can also use one of the pre-defined device classes (HID, MSD, etc.) to provide plug-n-play functionality. The implementation of one of these classes involves more reading, another layer of protocols, and some hardware that your device can talk to.