USB in a NutShell

Making sense of the USB standard

USB Protocols

Unlike RS-232 and similar serial interfaces where the format of data being sent is not defined, USB is made up of several layers of protocols. While this sounds complicated, don't give up now. Once you understand what is going on, you really only have to worry about the higher level layers. In fact most USB controller I.C.s will take care of the lower layer, thus making it almost invisible to the end designer.

Each USB transaction consists of a

- Token Packet (Header defining what it expects to follow), an
- Optional Data Packet, (Containing the payload) and a
- Status Packet (Used to acknowledge transactions and to provide a means of error correction)

As we have already discussed, USB is a host centric bus. The host initiates all transactions. The first packet, also called a token is generated by the host to describe what is to follow and whether the data transaction will be a read or write and what the device's address and designated endpoint is. The next packet is generally a data packet carrying the payload and is followed by an handshaking packet, reporting if the data or token was received successfully, or if the endpoint is stalled or not available to accept data.

Common USB Packet Fields

Data on the USBus is transmitted LSBit first. USB packets consist of the following fields,

- **Sync**

  All packets must start with a sync field. The sync field is 8 bits long at low and full speed or 32 bits long for high speed and is used to synchronise the clock of the receiver with that of the transmitter. The last two bits indicate where the PID fields starts.

- **PID**

PID stands for Packet ID. This field is used to identify the type of packet that is being sent. The following table shows the possible values.

Group	PID Value	Packet Identifier
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

There are 4 bits to the PID, however to insure it is received correctly, the 4 bits are complemented and repeated, making an 8 bit PID in total. The resulting format is shown below.

PID_0	PID_1	PID_2	PID_3	$nPID_0$	$nPID_1$	$nPID_2$	$nPID_3$

- **ADDR**

  The address field specifies which device the packet is designated for. Being 7 bits in length allows for 127 devices to be supported. Address 0 is not valid, as any device which is not yet assigned an address must respond to packets sent to address zero.

- **ENDP**

The endpoint field is made up of 4 bits, allowing 16 possible endpoints. Low speed devices, however can only have 2 additional endpoints on top of the default pipe. (4 endpoints max)

- **CRC**

  Cyclic Redundancy Checks are performed on the data within the packet payload. All token packets have a 5 bit CRC while data packets have a 16 bit CRC.

- **EOP**

  End of packet. Signalled by a Single Ended Zero (SE0) for approximately 2 bit times followed by a J for 1 bit time.

USB Packet Types

USB has four different packet types. Token packets indicate the type of transaction to follow, data packets contain the payload, handshake packets are used for acknowledging data or reporting errors and start of frame packets indicate the start of a new frame.

- **Token Packets**

  There are three types of token packets,

  - **In** - Informs the USB device that the host wishes to read information.
  - **Out** - Informs the USB device that the host wishes to send information.
  - **Setup** - Used to begin control transfers.

  Token Packets must conform to the following format,

Sync	PID	ADDR	ENDP	CRC5	EOP

- **Data Packets**

  There are two types of data packets each capable of transmitting up to 1024 bytes of data.

  - Data0
  - Data1

  High Speed mode defines another two data PIDs, DATA2 and MDATA.

  Data packets have the following format,

Sync	PID	Data	CRC16	EOP

- Maximum data payload size for low-speed devices is 8 bytes.
- Maximum data payload size for full-speed devices is 1023 bytes.
- Maximum data payload size for high-speed devices is 1024 bytes.
- Data must be sent in multiples of bytes.

- **Handshake Packets**

  There are three type of handshake packets which consist simply of the PID

  - **ACK** - Acknowledgment that the packet has been successfully received.
  - **NAK** - Reports that the device temporary cannot send or received data. Also used during interrupt transactions to inform the host there is no data to send.
  - **STALL** - The device finds its in a state that it requires intervention from the host.

  Handshake Packets have the following format,

Sync	PID	EOP

- **Start of Frame Packets**

  The SOF packet consisting of an 11-bit frame number is sent by the host every 1ms ± 500ns on a full speed bus or every 125 µs ± 0.0625 µs on a high speed bus.

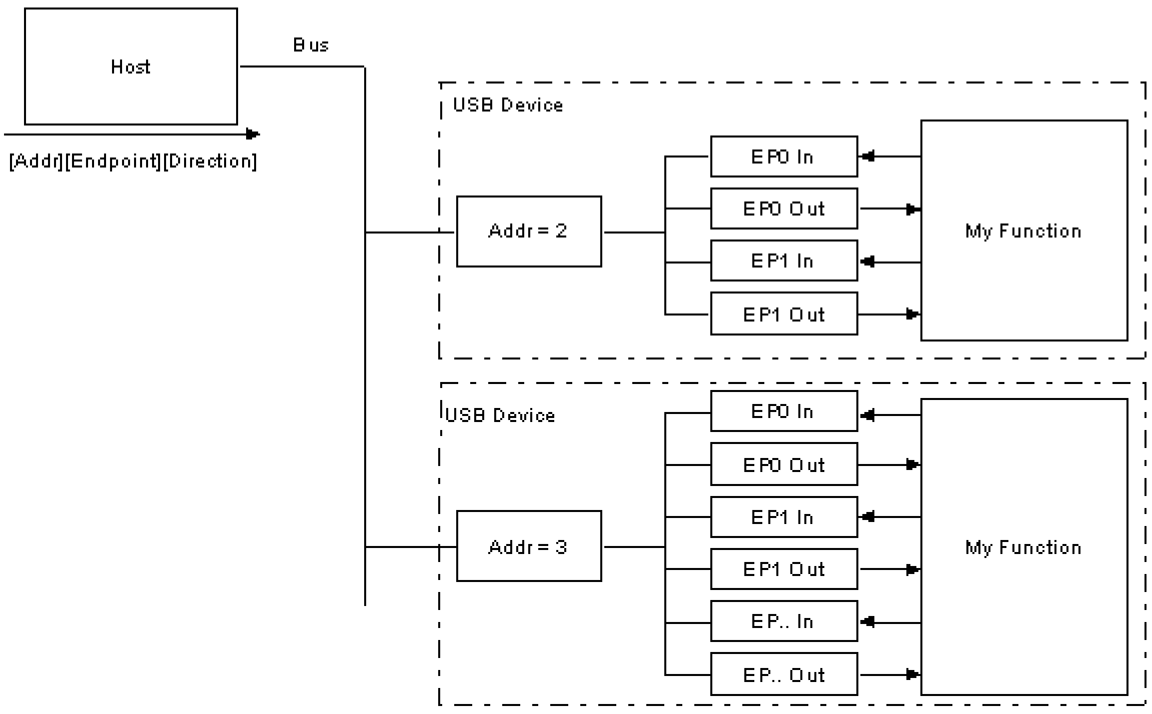
Sync	PID	Frame Number	CRC5	EOP

USB Functions

When we think of a USB device, we think of a USB peripheral, but a USB device could mean a USB transceiver device used at the host or peripheral, a USB Hub or Host Controller IC device, or a USB peripheral device. The standard therefore makes references to USB functions which can be seen as USB devices which provide a capability or function such as a Printer, Zip Drive, Scanner, Modem or other peripheral.

So by now we should know the sort of things which make up a USB packet. No? You're forgotten how many bits make up a PID field already? Well don't be too alarmed. Fortunately most USB functions handle the low level USB protocols up to the transaction layer (which we will cover next chapter) in silicon. The reason why we cover this information is most USB function controllers will report errors such as PID Encoding Error. Without briefly covering this, one could ask what is a PID

Encoding Error? If you suggested that the last four bits of the PID didn't match the inverse of the first four bits then you would be right.



Most functions will have a series of buffers, typically 8 bytes long. Each buffer will belong to an endpoint - EP0 IN, EP0 OUT etc. Say for example, the host sends a device descriptor request. The function hardware will read the setup packet and determine from the address field whether the packet is for itself, and if so will copy the payload of the following data packet to the appropriate endpoint buffer dictated by the value in the endpoint field of the setup token. It will then send a handshake packet to acknowledge the reception of the byte and generate an internal interrupt within the semiconductor/micro-controller for the appropriate endpoint signifying it has received a packet. This is typically all done in hardware.

The software now gets an interrupt, and should read the contents of the endpoint buffer and parse the device descriptor request.

Endpoints

Endpoints can be described as sources or sinks of data. As the bus is host centric, endpoints occur at the end of the communications channel at the USB function. At the software layer, your device driver may send a packet to your devices EP1 for example. As the data is flowing out from the host, it will end up in the EP1 OUT buffer. Your firmware will then at its leisure read this data. If it wants to return data, the function cannot simply write to the bus as the bus is controlled by the host. Therefore it writes data to EP1 IN which sits in the buffer until such time when the host sends a IN packet to that endpoint requesting the data. Endpoints can also be seen as the interface between the hardware of the function device and the firmware running on the function device.

All devices must support endpoint zero. This is the endpoint which receives all of the devices control and status requests during enumeration and throughout the duration while the device is operational on the bus.

Pipes

While the device sends and receives data on a series of endpoints, the client software transfers data through pipes. A pipe is a logical connection between the host and endpoint(s). Pipes will also have a set of parameters associated with them such as how much bandwidth is allocated to it, what transfer type (Control, Bulk, Iso or Interrupt) it uses, a direction of data flow and maximum packet/buffer sizes. For example the default pipe is a bi-directional pipe made up of endpoint zero in and endpoint zero out with a control transfer type.

USB defines two types of pipes

- **Stream Pipes** have no defined USB format, that is you can send any type of data down a stream pipe and can retrieve the data out the other end. Data flows sequentially and has a pre-defined direction, either in or out. Stream pipes will support bulk, isochronous and interrupt transfer types. Stream pipes can either be controlled by the host or device.

- **Message Pipes** have a defined USB format. They are host controlled, which are initiated by a request sent from the host. Data is then transferred in the desired direction, dictated by the request. Therefore message pipes allow data to flow in both directions but will only support control transfers.

Chapter 2 : Hardware

- Connectors
- Electrical
- Speed Identification
- Power (Vbus)
- Suspend Current
- Data Signalling Rate

Chapter 4 : Endpoint Types

- Control Transfers
- Interrupt Transfers
- Isochronous Transfers
- Bulk Transfers

Comments and Feedback?

Comments :
Email Address : (Optional)
Send