USB in a NutShell

Making sense of the USB standard

Endpoint Types

The Universal Serial Bus specification defines four transfer/endpoint types,

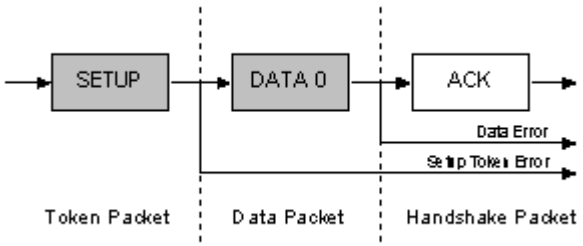
- **Control Transfers**
- **Interrupt Transfers**
- **Isochronous Transfers**
- **Bulk Transfers**

Control Transfers

Control transfers are typically used for command and status operations. They are essential to set up a USB device with all enumeration functions being performed using control transfers. They are typically bursty, random packets which are initiated by the host and use best effort delivery. The packet length of control transfers in low speed devices must be 8 bytes, high speed devices allow a packet size of 8, 16, 32 or 64 bytes and full speed devices must have a packet size of 64 bytes.

A control transfer can have up to three stages.

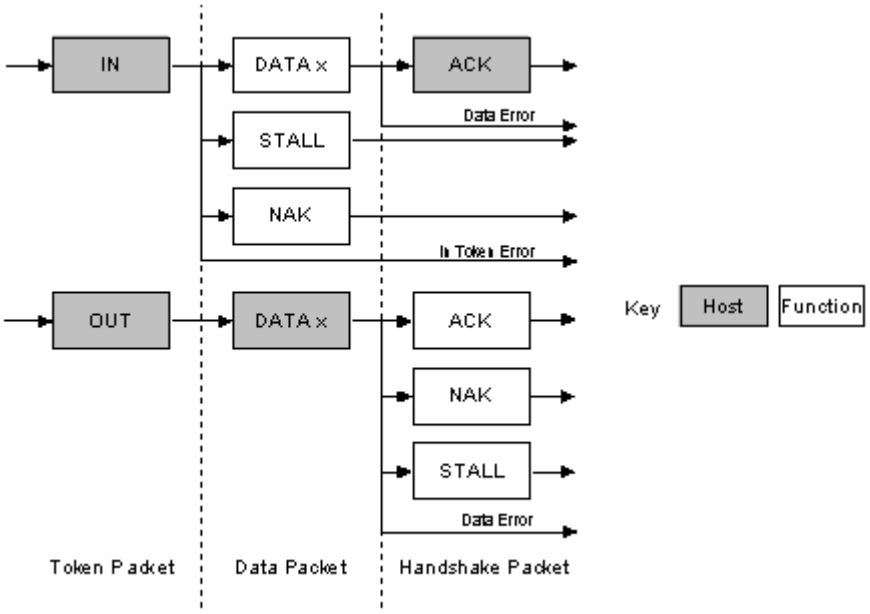
- The **Setup Stage** is where the request is sent. This consists of three packets. The setup token is sent first which contains the address and endpoint number. The data packet is sent next and always has a PID type of data0 and includes a setup packet which details the type of request. We detail the setup packet later. The last packet is a handshake used for acknowledging successful receipt or to indicate an error. If the function successfully receives the setup data (CRC and PID etc OK) it responds with ACK, otherwise it ignores the data and doesn't send a handshake packet. Functions cannot issue a STALL or NAK packet in response to a setup packet.



- The optional **Data Stage** consists of one or multiple IN or OUT transfers. The setup request indicates the amount of data to be transmitted in this stage. If it exceeds the maximum packet size, data will be sent in multiple transfers each being the maximum packet length except for the last packet.

  The data stage has two different scenarios depending upon the direction of data transfer.

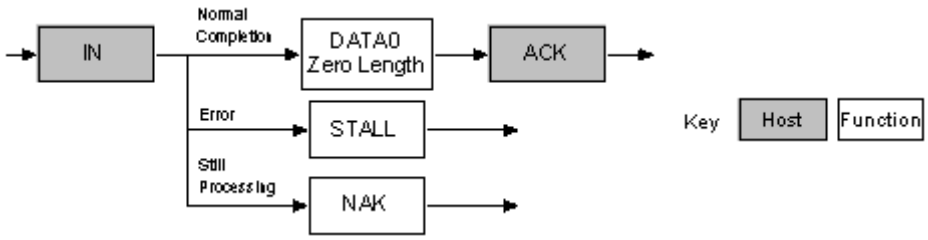
  - **IN:** When the host is ready to receive control data it issues an IN Token. If the function receives the IN token with an error e.g. the PID doesn't match the inverted PID bits, then it ignores the packet. If the token was received correctly, the device can either reply with a DATA packet containing the control data to be sent, a stall packet indicating the endpoint has had a error or a NAK packet indicating to the host that the endpoint is working, but temporary has no data to send.



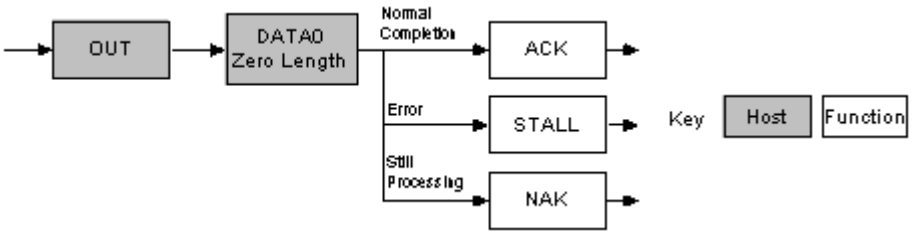
  - **OUT:** When the host needs to send the device a control data packet, it issues an OUT token followed by a data packet containing the control data as the payload. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing

the host it has successfully received the data. If the endpoint buffer is not empty due to processing of the previous packet, then the function returns a NAK. However if the endpoint has had a error and its halt bit has been set, it returns a STALL.

- **Status Stage** reports the status of the overall request and this once again varies due to direction of transfer. Status reporting is always performed by the function.
  - **IN:** If the host sent IN token(s) during the data stage to receive data, then the host must acknowledge the successful recept of this data. This is done by the host sending an OUT token followed by a zero length data packet. The function can now report its status in the handshaking stage. An ACK indicates the function has completed the command is now ready to accept another command. If an error occurred during the processing of this command, then the function will issue a STALL. However if the function is still processing, it returns a NAK indicating to the host to repeat the status stage later.



  - **OUT:** If the host sent OUT token(s) during the data stage to transmit data, the function will acknowledge the successful recept of data by sending a zero length packet in response to an IN token. However if an error occurred, it should issue a STALL or if it is still busy processing data, it should issue a NAK asking the host to retry the status phase later.



Control Transfers : The bigger picture

Now how does all this fit together? Let's say for example, the Host wants to request a device descriptor during enumeration. The packets which are sent are as follows.

The host will send the Setup token telling the function that the following packet is a Setup packet. The Address field will hold the address of the device the

host is requesting the descriptor from. The endpoint number should be zero, specifying the default pipe. The host will then send a DATA0 packet. This will have an 8 byte payload which is the Device Descriptor Request as outlined in Chapter 9 of the USB Specification. The USB function then acknowledges the setup packet has been read correctly with no errors. If the packet was received corrupt, the device just ignores this packet. The host will then resend the packe after a short delay.

1. Setup Token	**Sync**	**PID**	**ADDR**	**ENDP**	**CRC5**	**EOP**	Address & Endpoint Number
2. Data0 Packet	**Sync**	**PID**	**Data0**		**CRC16**	**EOP**	Device Descriptor Request
3. Ack Handshake	**Sync**	**PID**	**EOP**				Device Ack. Setup Packet

The above three packets represent the first USB transaction. The USB device will now decode the 8 bytes received, and determine it was a device descriptor request. The device will then attempt to send the Device Descriptor, which will be the next USB transaction.

1. In Token	**Sync**	**PID**	**ADDR**	**ENDP**	**CRC5**	**EOP**	Address & Endpoint Number
2. Data1 Packet	**Sync**	**PID**	**Data1**		**CRC16**	**EOP**	First 8 Bytes of Device Descriptor
3. Ack Handshake	**Sync**	**PID**	**EOP**				Host Acknowledges Packet
1. In Token	**Sync**	**PID**	**ADDR**	**ENDP**	**CRC5**	**EOP**	Address & Endpoint Number
2. Data0 Packet	**Sync**	**PID**	**Data0**		**CRC16**	**EOP**	Last 4 bytes + Padding
3. Ack Handshake	**Sync**	**PID**	**EOP**				Host Acknowledges Packet

In this case, we assume that the maximum payload size is 8 bytes. The host sends the IN token, telling the device it can now send data for this endpoint. As the maximum packet size is 8 bytes, we must split up the 12 byte device descriptor into chunks to send. Each chunk must be 8 bytes except for the last transaction. The host acknowledges every data packet we send it.

Once the device descriptor is sent, a status transaction follows. If the transactions were successful, the host will send a zero length packet indicating the overall transaction was successful. The function then replies to this zero length packet indicating its status.

1. Out Token	**Sync**	**PID**	**ADDR**	**ENDP**	**CRC5**	**EOP**	Address & Endpoint Number
2. Data1 Packet	**Sync**	**PID**	**Data1**		**CRC16**	**EOP**	Zero Length Packet
3. Ack Handshake	**Sync**	**PID**	**EOP**				Device Ack. Entire Transaction

Interrupt Transfers

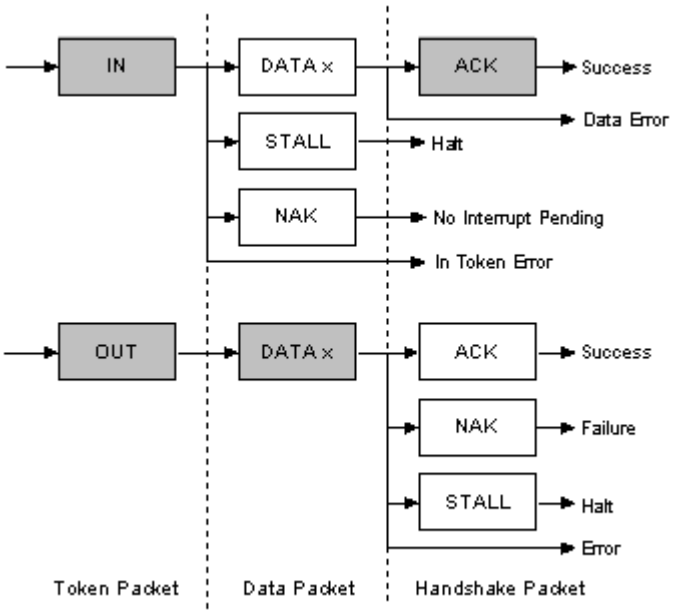
Any one who has had experience of interrupt requests on microcontrollers will know that interrupts are device generated. However under USB if a device requires the attention of the host, it must wait until the host polls it before it can report that it needs urgent attention!

Interrupt Transfers

- Guaranteed Latency
- Stream Pipe - Unidirectional
- Error detection and next period retry.

Interrupt transfers are typically non-periodic, small device "initiated" communication requiring bounded latency. An Interrupt request is queued by th device until the host polls the USB device asking for data.

- The maximum data payload size for low-speed devices is 8 bytes.
- Maximum data payload size for full-speed devices is 64 bytes.
- Maximum data payload size for high-speed devices is 1024 bytes.



The above diagram shows the format of an Interrupt IN and Interrupt OUT transaction.

- **IN:** The host will periodically poll the interrupt endpoint. This rate of polling is specified in the endpoint descriptor which is covered later. Each poll will involve the host sending an IN Token. If the IN token is corrupt, the function ignores the packet and continues monitoring the bus for new tokens.

  If an interrupt has been queued by the device, the function will send a data packet containing data relevant to the interrupt when it receives the IN Token. Upon successful recept at the host, the host will return an ACK. However if the data is corrupted, the host will return no status. If on the other hand a interrupt condition was not present when the host polled the interrupt endpoint with an IN token, then the function signals this state by sending a NAK. If an error has occurred on this endpoint, a STALL is sent in reply to the IN token instead.

- **OUT:** When the host wants to send the device interrupt data, it issues an OUT token followed by a data packet containing the interrupt data. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing the host it has successfully received the data. If the endpoint buffer is not empty due to processing of a previous packet, then the function returns an NAK. However if an error occurred with the endpoint consequently and its halt bit has been set, it returns a STALL.

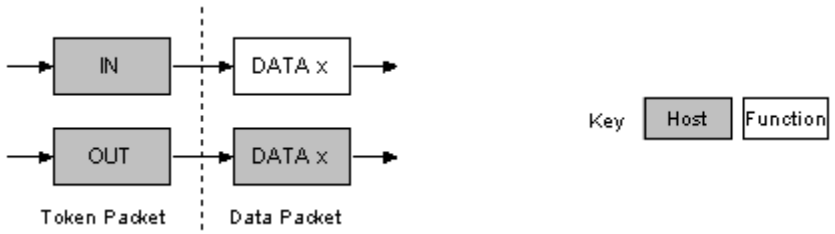
Isochronous Transfers

Isochronous transfers occur continuously and periodically. They typically contain time sensitive information, such as an audio or video stream. If there were a delay or retry of data in an audio stream, then you would expect some erratic audio containing glitches. The beat may no longer be in sync. However if a packet or frame was dropped every now and again, it is less likely to be noticed by the listener.

Isochronous Transfers provide

- Guaranteed access to USB bandwidth.
- Bounded latency.
- Stream Pipe - Unidirectional
- Error detection via CRC, but no retry or guarantee of delivery.
- Full & high speed modes only.
- No data toggling.

The maximum size data payload is specified in the endpoint descriptor of an Isochronous Endpoint. This can be up to a maximum of 1023 bytes for a full speed device and 1024 bytes for a high speed device. As the maximum data payload size is going to effect the bandwidth requirements of the bus, it is wise to specify a conservative payload size. If you are using a large payload, it may also be to your advantage to specify a series of alternative interfaces with varying isochronous payload sizes. If during enumeration, the host cannot enable your preferred isochronous endpoint due to bandwidth restrictions, it has something to fall back on rather than just failing completely. Data being sent on an isochronous endpoint can be less than the pre-negotiated size and may vary in length from transaction to transaction.



The above diagram shows the format of an Isochronous IN and OUT transaction. Isochronous transactions do not have a handshaking stage and cannot report errors or STALL/HALT conditions.

Bulk Transfers

Bulk transfers can be used for large bursty data. Such examples could include a print-job sent to a printer or an image generated from a scanner. Bulk transfers provide error correction in the form of a CRC16 field on the data payload and error detection/re-transmission mechanisms ensuring data is transmitted and received without error.

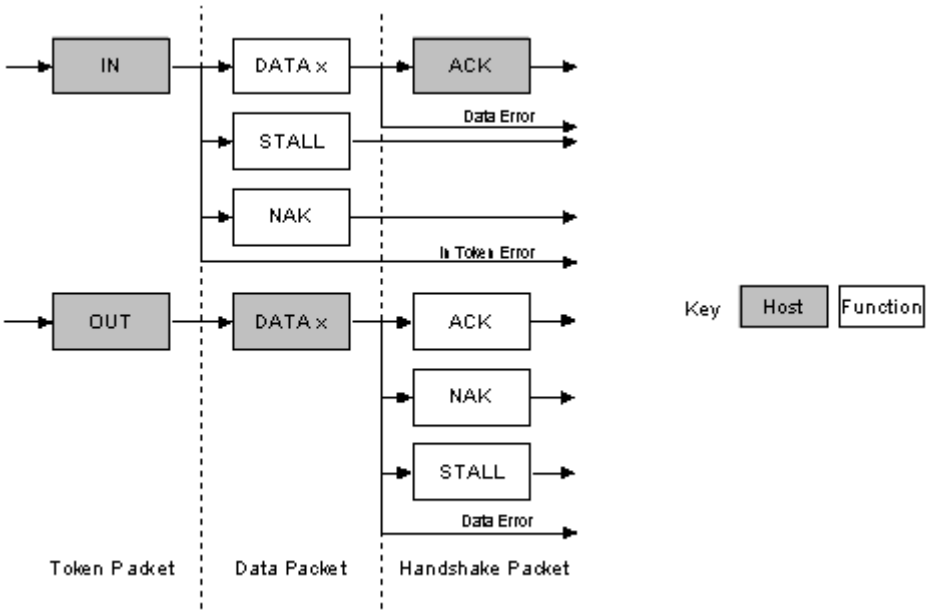
Bulk transfers will use spare un-allocated bandwidth on the bus after all other

transactions have been allocated. If the bus is busy with isochronous and/or interrupt then bulk data may slowly trickle over the bus. As a result Bulk transfers should only be used for time insensitive communication as there is no guarantee of latency.

Bulk Transfers

- Used to transfer large bursty data.
- Error detection via CRC, with guarantee of delivery.
- No guarantee of bandwidth or minimum latency.
- Stream Pipe - Unidirectional
- Full & high speed modes only.

Bulk transfers are only supported by full and high speed devices. For full speed endpoints, the maximum bulk packet size is either 8, 16, 32 or 64 bytes long. For high speed endpoints, the maximum packet size can be up to 512 bytes long. If the data payload falls short of the maximum packet size, it doesn't need to be padded with zeros. A bulk transfer is considered complete when it has transferred the exact amount of data requested, transferred a packet less than the maximum endpoint size of transferred a zero-length packet.



The above diagram shows the format of a bulk IN and OUT transaction.

- **IN:** When the host is ready to receive bulk data it issues an IN Token. If the function receives the IN token with an error, it ignores the packet. If the token was received correctly, the function can either reply with a DATA packet containing the bulk data to be sent, or a stall packet indicating the endpoint has had a error or a NAK packet indicating to the host that the endpoint is working, but temporary has no data to send.

- **OUT:** When the host wants to send the function a bulk data packet, it issues an OUT token followed by a data packet containing the bulk data. If any part of the OUT token or data packet is corrupt then the function ignores the packet. If the function's endpoint buffer was empty and it has clocked the data into the endpoint buffer it issues an ACK informing the host it has successfully received the data. If the endpoint buffer is not empty due to processing a previous packet, then the function returns an NAK. However if the endpoint has had an error and it's halt bit has been set, it returns a STALL.

Bandwidth Management

The host is responsible in managing the bandwidth of the bus. This is done at enumeration when configuring Isochronous and Interrupt Endpoints and throughout the operation of the bus. The specification places limits on the bus, allowing no more than 90% of any frame to be allocated for periodic transfers (Interrupt and Isochronous) on a full speed bus. On high speed buses this limitation gets reduced to no more than 80% of a microframe can be allocated for periodic transfers.

So you can quite quickly see that if you have a highly saturated bus with periodic transfers, the remaining 10% is left for control transfers and once those have been allocated, bulk transfers will get its slice of what is left.

Chapter 3 : Protocols

- USB Protocols
- Common USB Packet Fields
- USB Packet Types
- USB Functions
- Endpoints
- Pipes

Chapter 5 : USB Descriptors

- Device Descriptors
- Configuration Descriptors
- Interface Descriptors
- Endpoint Descriptors
- String Descriptors

Comments and Feedback?

Comments :
Email Address : (Optional)
Send