INDIAN INSTITUTE OF TECHNOLOGY,ROPAR

DEPARTMENT OF ELECTRICAL ENGINEERING

BTP Endsem Report

Submitted By:
Chetna Singh-2018eeb1218
Keshav Kishore - 2018eeb1158
Mahima Kumawat-2018eeb1162

Supervisor:
Dr. Neeraj Goel

Submitted in partial fulfillment of the requirements for the B.Tech degree in Electrical Engineering of Indian Institute of Technology, Ropar

18th May, 2021

Contents

0.1 Introduction

The Universal Serial Bus(USB)basically is a cable bus that supports the data exchange between host computer and a wide range of parallely accessible peripherals. The peripherals share USB bandwidth through a host scheduled, token based protocol. The bus allows the peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation(USBs are hot pluggable).

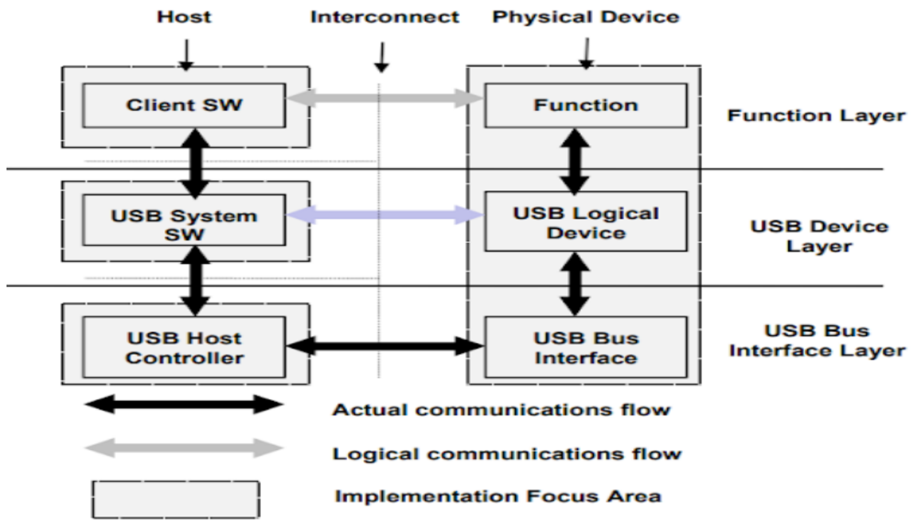
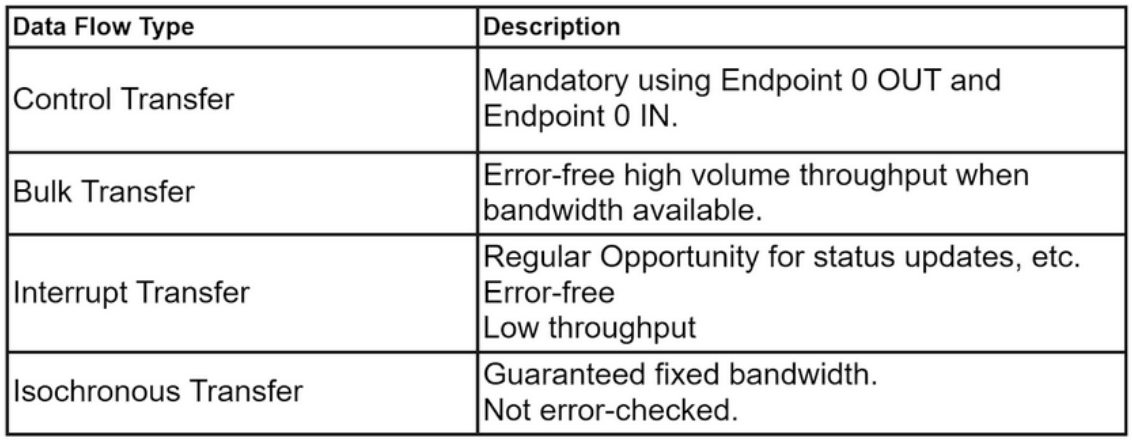


Figure shows a deeper overview of the USB, identifying the different layers of the system. In particular, our implementation area is USB Host Controller. The sorted connection of a host to a device wants an interaction between the number of layers and the entities. USB Bus Interface layer actually provides the physical/ signaling/ packet connectivity of the host to a device. The USB Device layer is the only view for the USB System Software to performing generic USB operations with a device. The Function layer basically provides additional capabilities to the host via an appropriate matched client software layer. The USB Device and Function layers each have a view of logical communication within their layer that actually uses the USB Bus Interface layer to accomplish data transfer.

0.2 Data Flow Model

The USB also supports functional data and it controls an exchange between the USB host and the device as a set of either unidirectional or bi-directional pipes. USB data transfers take place between the host software and the particular endpoint on a USB device. These associations between the host software and a USB device endpoint are called pipes. In general, data movement though one pipe is independent from the data flow in any other pipe. A given USB device may have many pipes. As an example, a given USB device could have an endpoint that supports a pipe for transaction from host to the USB device and another endpoint that supports a pipe for transaction from USB device to host.
USB architecture supports following four types of Data transfer:

1. Control Transfer: Used to configure a device at attach time and can be used for other device-specific purposes, including control of other pipes on the device.
2. Bulk data transfers: Generated or consumed in relatively large and bursty quantities and have wide dynamic latitude in Transmission constraints.
3. Interrupt data transfers: Used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics.
4. Isochronous data transfers: Occupy a pre negotiated amount of USB bandwidth with a pre negotiated delivery latency. (Also called streaming real time transfers).



Data Flow Type	Description
Control Transfer	Mandatory using Endpoint 0 OUT and Endpoint 0 IN.
Bulk Transfer	Error-free high volume throughput when bandwidth available.
Interrupt Transfer	Regular Opportunity for status updates, etc. Error-free Low throughput
Isochronous Transfer	Guaranteed fixed bandwidth. Not error-checked.

0.3 Protocol Layer

0.3.1 Packet Identifier field:

A packet identifier (PID) is a four-bit packet type field which is followed by four bit check field. This PID indicates the type of packet and the format of the packet and the type of error detection applied to the packet. PID check field is basically generated by performing 1's complement of the packet type field. An error exists if the four PID check bits are not complements of their respective packet identifier bits.

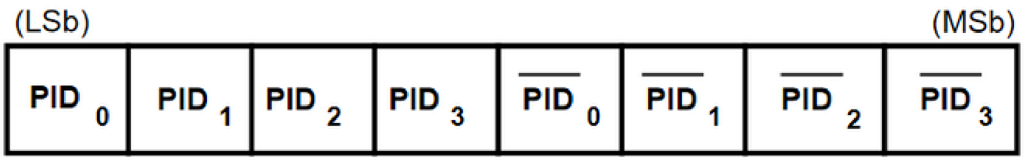


Fig: PID Format

PIDs are divided into 4 groups: token, data, handshake, and special.

PID Type	PID Name	Description
Token	IN OUT SOF SETUP	Address + endpoint number in host-to-functionTransaction Address + endpoint number in function-to-host transaction Start-of-Frame marker and frame number Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA 0 DATA1 DATA2 MDATA	Data packet PID even Data packet PID odd Data packet PID high-speed, high bandwidth isochronous transaction in a microframe Data packet PID high-speed for split and high bandwidth isochronous transactions
Handshake	ACK NAK STALL NYET	Receiver accepts error-free data packet Receiving device cannot accept data or transmitting device cannot send data Endpoint is halted or control pipe request is not supported No response yet from receiver
Special	PRE ERR SPLIT PING Reserved	(Token) Host-issued preamble. Enables downstream bustraffic to low-speed devices. (Handshake) Split Transaction Error Handshake (Token) High-speed Split Transaction Token (Token) High-speed flow control probe for a bulk/controlendpoint Reserved PID

0.3.2 Address field:

Function endpoints are addressed using two fields:

- **the function address field :** The ADDR field actually specifies the function, via address, that is either the source or the destination of a data packet, depending on the value of the token PID. Upon the reset and power up, a function's address defaults to a zero value and must be programmed by the host during the process of enumeration.

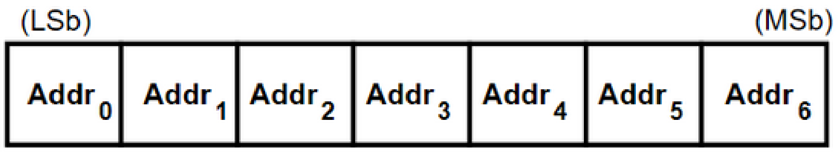


Fig: Function Address Field

- **the endpoint field :** An additional four-bit endpoint (ENDP) field, permits more flexible addressing of functions in which more than one endpoint is required. Except for endpoint address zero, endpoint numbers are function-specific. All functions should support a control pipe at endpoint number zero Low-speed devices support a maximum of three pipes per function:Full-speed and high-speed functions may support up to a maximum of 16 IN and OUT endpoints.

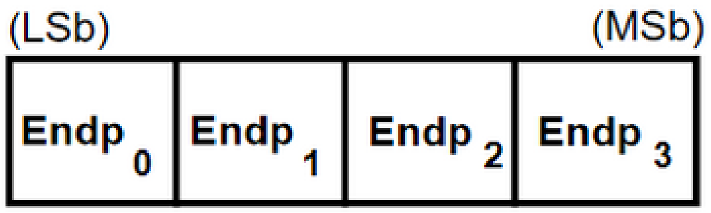


Fig: Endpoint Field

0.3.3 Frame number field:

This field is a 11-bit field that is increased by the host on a per frame basis. The field rolls over upon reaching the maximum value of 7FFH and it is send only in SOF tokens at the start of every micro-frame.

0.3.4 Data field:

Data field may range from zero to 1,024 bytes and should be the integral number of bytes. The Data bits within each byte are shifted out LSb first. This field varies with transfer type.

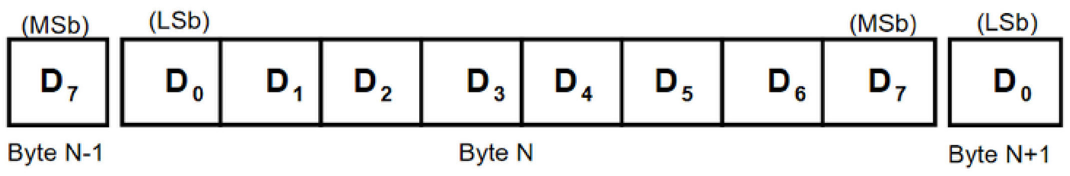


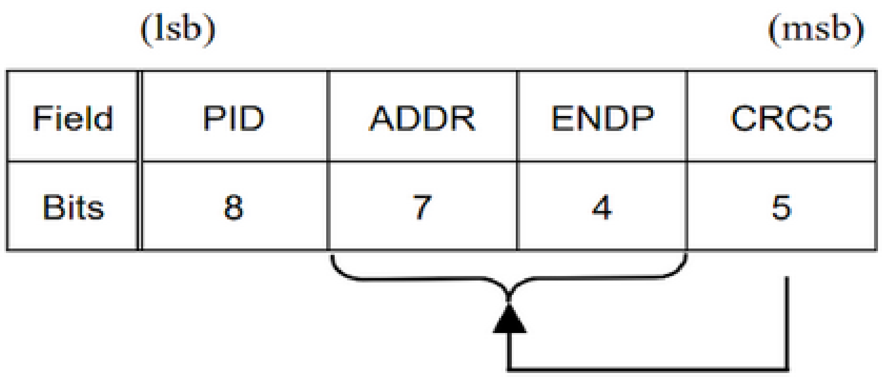
Fig: Data field format

0.3.5 Cyclic Redundancy Check field:

Cyclic redundancy checks abbreviated as CRCs are used to protect all non-PID fields in token and data packets. . For CRC generation and checking, the shift registers in the generator and checker are itroduced with an allone like pattern. For each data bit send or received, the high order bit of the current remainder is XOR with data bit and after that, the remainder is shifted by left one bit and the lower order bit is set to zero. If the result of that XOR is 1, then the remainder is XOR with the generator polynomial.

0.4 Packet Formats

0.4.1 Token packet:



Field	PID	ADDR	ENDP	CRC5
Bits	8	7	4	5

A token packet consists of a PID, specifying either IN, OUT, or SETUP packet type and ADDR and ENDP fields. The PING special token(only for High speed devices) packet also has the same fields as a token packet.

- For OUT and SETUP transactions(data transactions are from host to device), the address and the endpoint fields can uniquely identify the endpoint that will receive the subsequent Data packet.

- For IN transactions(data transaction from device to host), these fields uniquely identify which endpoint should transmit a Data packet.

- For PING transactions(handshake transaction from device to host), these fields uniquely identify which endpoint will respond with a handshake packet.

Only the host can issue the token packets. These Token packets have a five-bit CRC that covers the address and endpoint fields.

0.4.2 Split transaction packet:

In USB, a special token for split transactions is defined: SPLIT. This is a 4 byte token packet as compared to other normal 3 byte token packets. This split transaction

token is basically used to support split transactions between the host controller while communicating with the hub operating at high speed with full-/low-speed devices to some of its downstream facing ports. A high-speed split transaction has two parts:
1. start-split
2. complete-split.

0.4.3 Start of Frame packet:

The Start of Frame (SOF) packets are always issued by the host at a nominal rate of once every 1 ms for a full-speed bus and 125 μs for a high-speed bus.

0.4.4 The data packets:

The data packet consists of a PID, a data field containing zero or more bytes of data, and a CRC . There are total four types of data packets, identified by differing PIDs: DATA0, DATA1, DATA2 and MDATA. The DATA0 and DATA1 are defined to support data toggle synchronization. All four data PIDs, used in data PID sequencing for high bandwidth high-speed isochronous endpoints. MDATA, DATA0 and DATA1 are used in split transactions.

0.4.5 Handshake Packet:

Handshake packet consists only of a PID field. Handshake packets are used to figure out status of the transacted data and returna the values which indicate successful reception of data, command acceptance or rejection, and halt conditions.
four types of handshake packets with one special handshake packet are:

- **ACK** indicates that the data packet which was received without bit stuffing or CRC error over the data field and that PID was received correctly.

- **NAK** is used for flow control purposes and to indicate that the function is temporarily can not transmit or receive the data, but will eventually be able to do so without need of host intervention.

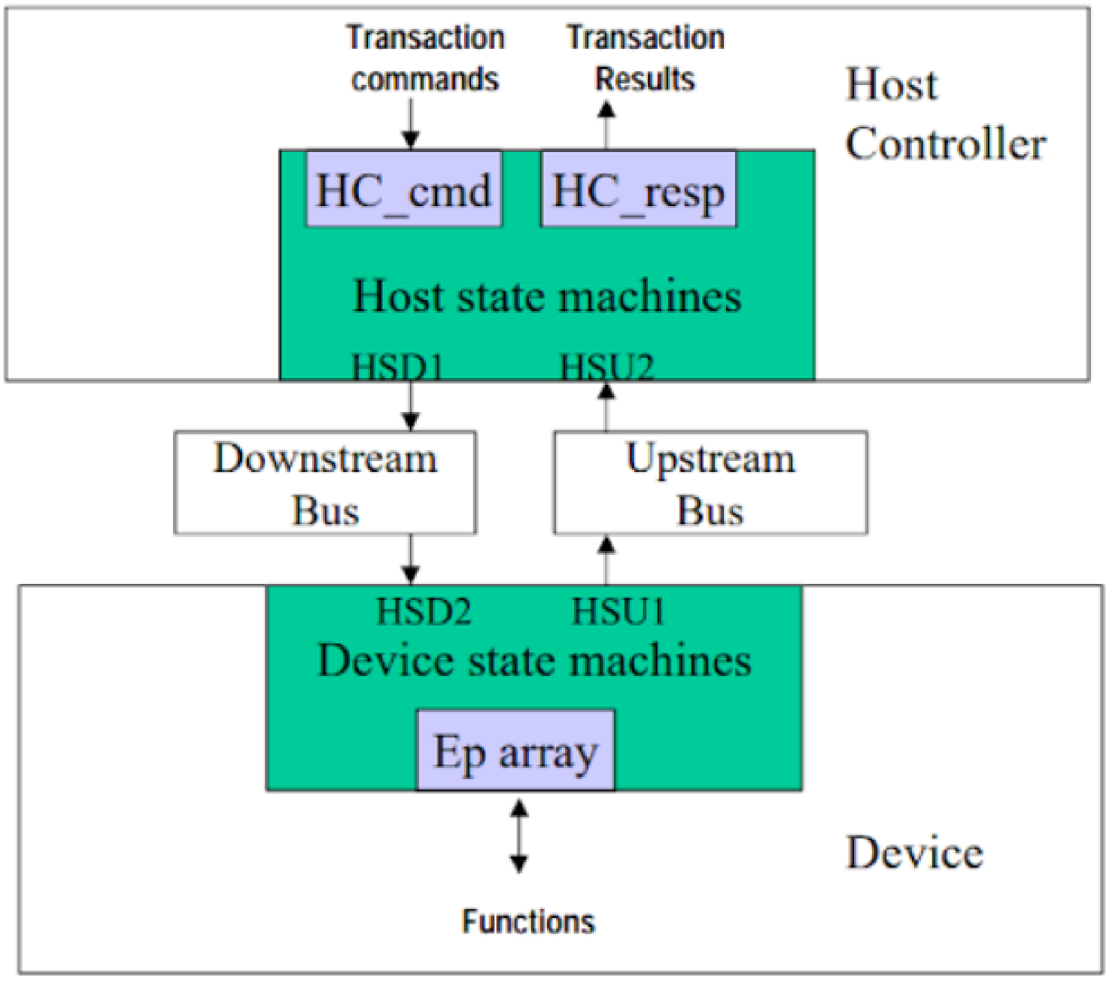
- **STALL** signifies that a function is unable to transmit or receive data, or a control pipe request is not supported. “Functional stall,” is when the Halt feature associated with the endpoint is set and the halt will be cleared only using the host intervention.

- **NYET** is a high speed handshake that is returned in two circumstances. First, a high speed endpoint as part of the PING protocol and second, a hub in response to a split transaction when the full or low speed transaction has not yet been completed .

- **ERR** is a high-speed only handshake that is returned to allow a high-speed hub to report an error on a full-/low-speed bus. It is only returned by a high-speed hub as part of the split transaction protocol.

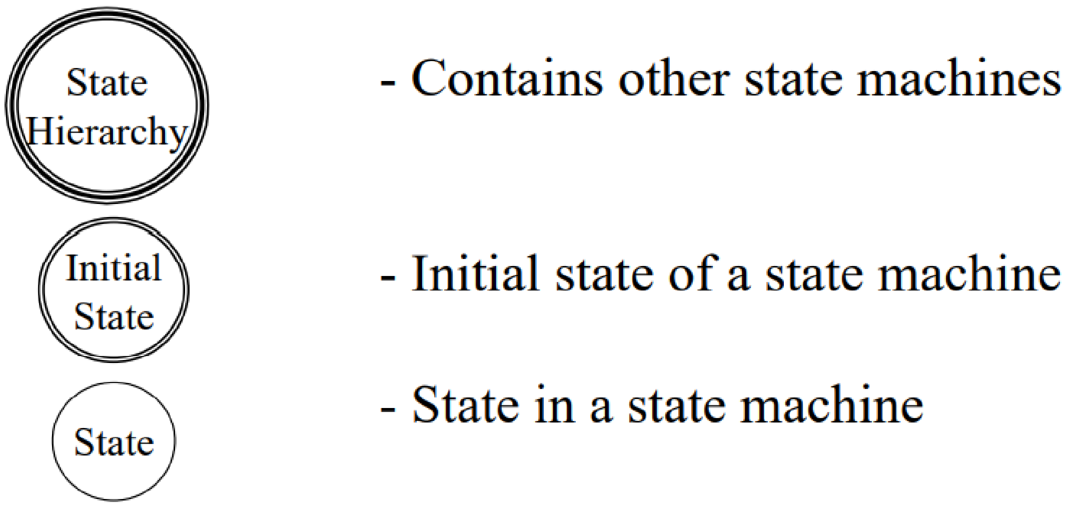
0.5 Transaction Packet Sequences

The packets that undergo a transaction, varies depending on the type of endpoint. four endpoint types are: bulk, control, interrupt and isochronous. The host controller and device each require different state machines to correctly sequence each type of transaction.



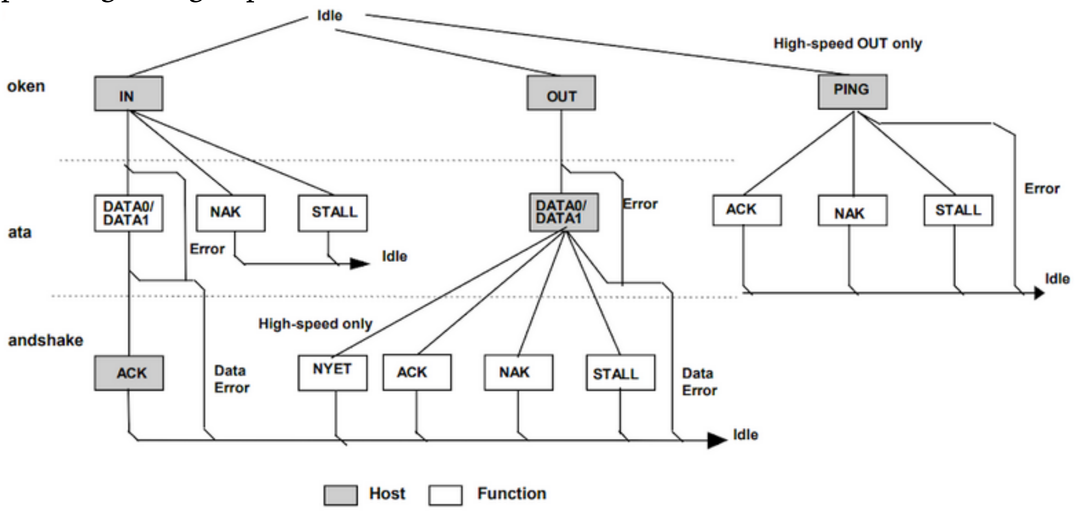
State machine context overview

figure shows the host controller and device state machines. Host controller determines every next transaction to run for the endpoint and also issues a HC cmd command to the host controller state machines. This makes the host controller state machines to produce one or more packets to move over the HSD1 downstream bus. The device receives these packets from the HSD2 bus, which reacts to the received packet, and interacts with its function through the state of the corresponding endpoint. The device responds with a packet on the HSU1 upstream bus . The state machines receives a packet from the bus HSU2 and provide a result of the transaction back to the host controller. The details of the packets are sent on the bus is guessed by the transfer type for the endpoint and bus activity the state machines observes.The diagrams should not be taken as a required implementation, but to specify the required behavior.A circle with a triple line border shows a reference to another connected state machine. The given circle with a double line border shows the initial state. The given circle with a single line border represents a simple state.


State Hierarchy
- Contains other state machines
Initial State
- Initial state of a state machine
State
- State in a state machine

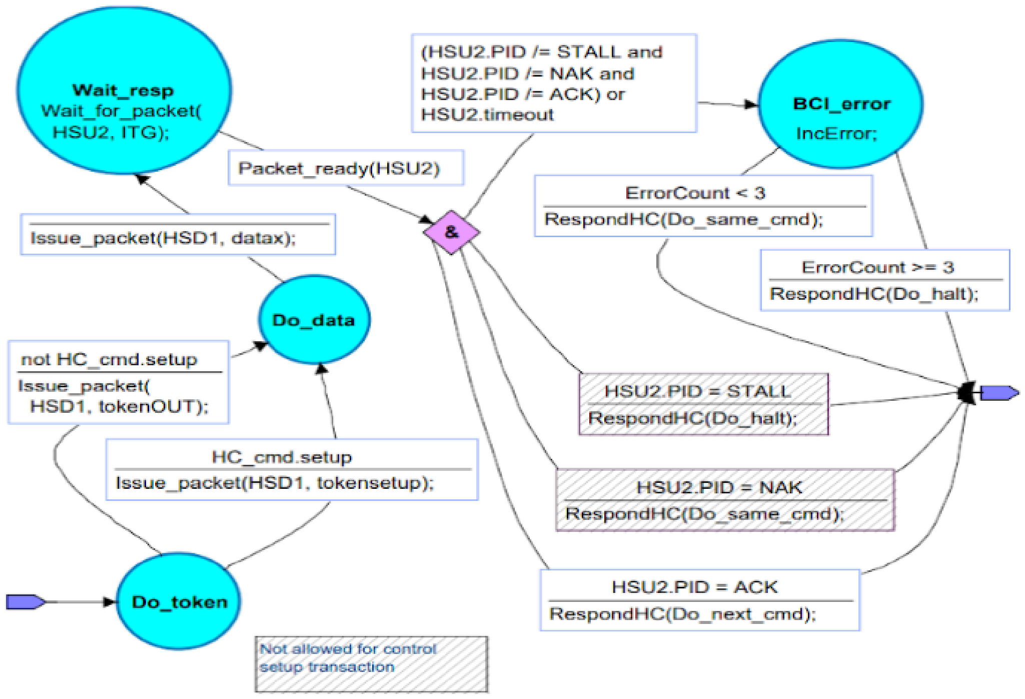
0.5.1 Bulk Transactions:

This transaction types are characterized by the ability to guarantee an error free delivery of the data between host and a function through error detection and retry.The Bulk transactions use a three-phase transaction consisting of token, data, and handshake packets. The PING and The NYET packets can only be used with devices operating at high-speed.

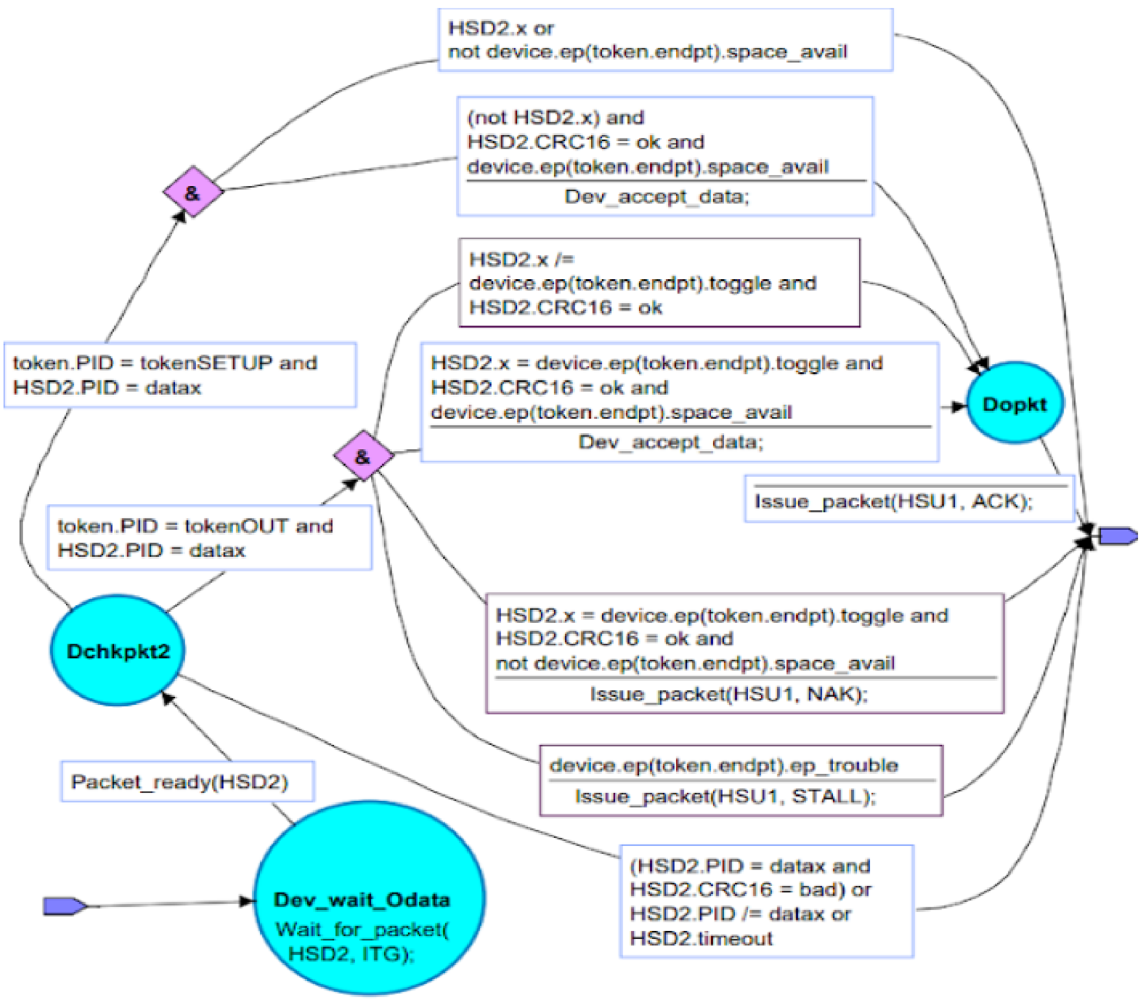


Bulk Transaction format

The following figures show the host and device state machines for bulk, control, and interrupt OUT transactions.



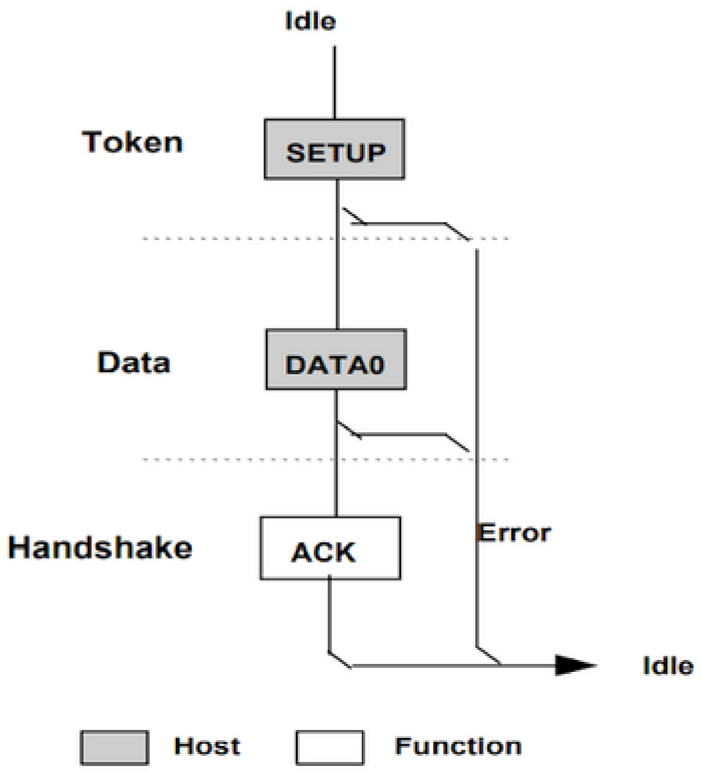
Bulk/Control/Interrupt OUT Transaction Host State Machine



Bulk/Control/Interrupt OUT Transaction Device State Machine

0.5.2 Control Transfer:

A Control transfer minimally has two transaction stages:The Setup and The Status. A control transfer may or may not contain The Data stage between the Setup and The Status stages. During the Setup stage, SETUP transaction is used to transfer information to the control endpoint of a function. The SETUP transactions are similar in format to an OUT but use a SETUP rather than an OUT PID. A SETUP always use a DATA0 PID for data field of the SETUP transaction. Function receiving a SETUP should accept the SETUP data and also respond with ACK. But if the data is corrupted, discard the data and return no handshake.



Control SETUP Transaction

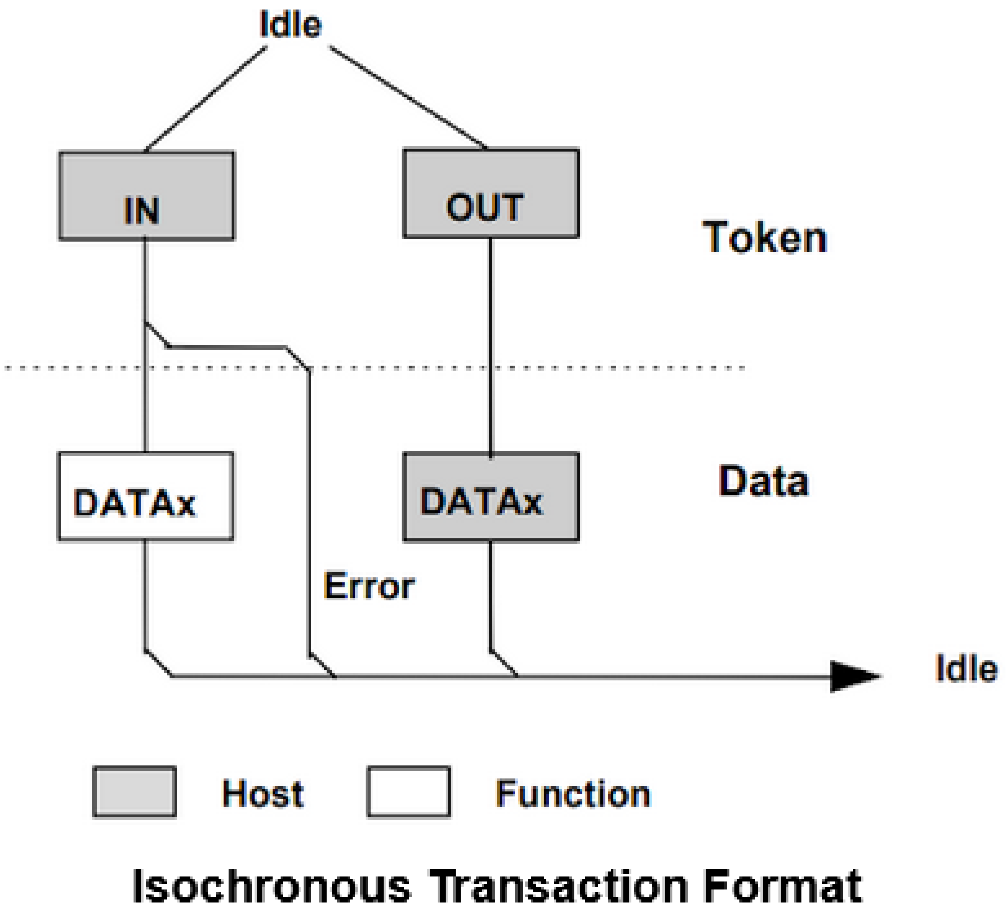
Like the case of the functional stall, protocol stall can not indicate an error with the device. This protocol STALL conditions last until the receipt of the next SETUP transaction, the function will return STALL as the response to any IN or OUT transaction on the pipe until the SETUP

0.5.3 Interrupt Transactions :

Interrupt transactions may consist IN or OUT transfers. Upon receipt of the IN token, a function can return the data, NAK or STALL. If the endpoint has no newly interrupt information to return, the function returns the NAK during the data phase. And If the Halt feature is set for the interrupt endpoint, the function will return the STALL handshake. If an interrupt is pending thet is no new information, the function returns the interrupt information as a data packet. The host, in response to receipt of the data packet, issues either an ACK handshake if data was received error-free or returns no handshake if the data packet was received corrupted. It follows the similar format as that of Bulk transactions.

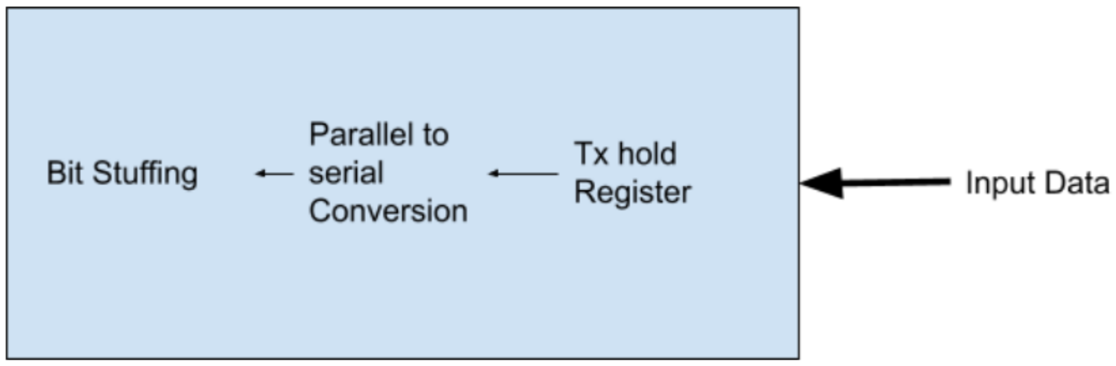
0.5.4 Isochronous Transactions:

Isochronous transactions have a token and data phase, but no handshake phase. The host issues either an IN or an OUT token followed by the data phase in which the endpoint (for INs) or the host (for OUTs) transmits data. Isochronous transactions can not support a handshake phase or retry capability. Full-speed isochronous transaction also does not support toggle sequencing. High bandwidth, high speed isochronous transactions support data PID sequencing.

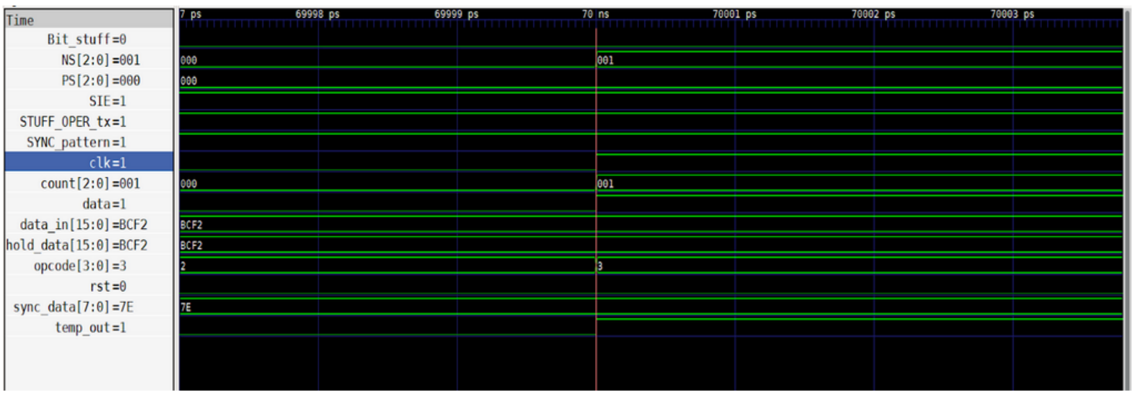


0.6 Verilog Simulation results

Using the above protocol methods, we implemented the UTMI Transmitter block. The block diagram of the same is shown below:



After Bit Stuffing, Output data will be transmitted. The Sync Pattern for this design is “01111110” which needs to be transmitted after the transmitter is started by SIE. The Transmitter module of the UTMI are designed using Verilog and they are simulated using Icarus Verilog and waveforms are analysed through GTKWave Electronic waveform viewer.



```
Time
Bit_stuff=0
NS[2:0]=001
PS[2:0]=000
SIE=1
STUFF_OPER_tx=1
SYNC_pattern=1
clk=1
count[2:0]=001
data=1
data_in[15:0]=BCF2
hold_data[15:0]=BCF2
opcode[3:0]=3
rst=0
sync_data[7:0]=7E
temp_out=1
```

From the above waveform we can see values of different variables at different times. Our clock has frequency of 20ns.

0.7 Future Work

After thorough discussion, we have decided to implement further blocks of USB like SIE , state machines, error detection, CRCs, etc. using Chisel high level synthesis or Bluespec Verilog as these methods are much more efficient and faster than traditional verilog. This semester we mainly learnt about the USB protocol layer. Design and RTL synthesis will be done next semester.

Verilog Code and Testbench:

Design Code of module Transmitter:

```
module Transmitter (
clk,              //input clk signal
rst,              //reset signal for transmitter
SIE,            //SIE  = 1 , for transmitter to perform bit_stuffing
, parallel-to-serial implementation
STUFF_OPER_tx,
sync_data,            //sync_data needs to be transmitted to initiate
transmitter
data_in,              //input data to be transmitted
SYNC_pattern,
encoded_dataout,
opcode
                      );

 //port declarations

 input clk;
 input rst;
 input SIE;
 input STUFF_OPER_tx;
 input [7:0]sync_data;

 input [15:0]data_in;
 output encoded_dataout;
 output  SYNC_pattern;
 output [3:0]opcode;


 reg encoded_dataout;
 wire SYNC_pattern;


 //temporary registers

 reg[15:0]hold_data;  //hold_data is used to store the input data
 reg temp_out;
 reg [2:0]PS,NS;        //PS,NS are present states and next state
variables in bit stuff logic state machine
 reg [3:0]opcode;
 reg [2:0]count;
 wire Bit_stuff;
 wire data;


 //parameter definitions which are required in bit_stuff logic
implementation
 parameter IDLE=3'b000;
 parameter START=3'b001;
 parameter TWO=3'b010;
 parameter THIRD=3'b011;
 parameter FOUR=3'b100;
 parameter FIVE=3'b101;
 parameter BITSTUFF=3'b110;




//data_in is the input data coming in parallel format which the user
will enter in this case

always@(posedge clk)
begin
     if(rst)
     begin
     hold_data<=16'b0000_0000_0000_0000;
     opcode<=4'b0000;
     end
     else if(SIE&&~Bit_stuff&&SYNC_pattern)
     begin
     hold_data<=data_in;
     opcode<=opcode+1'b1;
     end
     else
     opcode<=opcode;
end


//Parallel to serial Conversion
always@( posedge clk)
begin
     case(opcode)
     4'b0000:temp_out=hold_data[15];
     4'b0001:temp_out=hold_data[0];
```

```
      4'b0010:temp_out=hold_data[1];
      4'b0011:temp_out=hold_data[2];
      4'b0100:temp_out=hold_data[3];
      4'b0101:temp_out=hold_data[4];
      4'b0110:temp_out=hold_data[5];
      4'b0111:temp_out=hold_data[6];
      4'b1000:temp_out=hold_data[7];
      4'b1001:temp_out=hold_data[8];
      4'b1010:temp_out=hold_data[9];
      4'b1011:temp_out=hold_data[10];
      4'b1100:temp_out=hold_data[11];
      4'b1101:temp_out=hold_data[12];
      4'b1110:temp_out=hold_data[13];
      4'b1111:temp_out=hold_data[14];
      endcase

 end

//Bit stuffing implementation
 always@(posedge clk)
 begin
      if(rst)
      begin
           PS<=IDLE;
           count=3'b000;
      end
      else
           PS<=NS;
end

always@(PS or temp_out)
begin
      case(PS)
      IDLE:if(temp_out&&STUFF_OPER_tx)
                begin
                     count=3'b001;//count+1'b1;
                     NS=START;
                end
                else
                begin
                NS=IDLE;
                count=3'b000;
           end

      START:if(temp_out)
```

```
begin
     count=3'b010;
     NS=TWO;
end
else
     begin
          NS=IDLE;
     end

TWO:if(temp_out)
begin
     count=3'b011;
     NS=THIRD;
end
else
     begin
          NS=IDLE;
     end

THIRD:if(temp_out)
begin
     count=3'b100;
     NS=FOUR;
end
else
     begin
          NS=IDLE;
     end

FOUR:if(temp_out)
begin
     count=3'b101;
     NS=FIVE;
end
else
     begin
          NS=IDLE;
     end

FIVE:if(temp_out)
begin
     count=3'b110;
     NS=BITSTUFF;
end
else
```

```
            begin
                 count=3'b000;
                 NS=IDLE;
            end
    BITSTUFF:
                 begin
                 count=3'b111;
                 NS=IDLE;
                 end


   endcase

   end


   assign Bit_stuff=(count==3'b111)?1'b1:1'b0;            //if
count==7 , It means 6 consecutive 1's have been transmitted thus bit
stuffing needs to be done.

   assign data=Bit_stuff?0:temp_out;                      //if
bit_stuff ==1 , data transmitted is 0 else data transmitted is the
hold_data

   assign SYNC_pattern=(sync_data==8'b0111_1110)?1'b1:1'b0;//SYNC
PATTERN CHECKING where sync data needs to be transmitted.



endmodule
```

Testbench:

```
`timescale 1 ns / 100 ps
module Transmitter_tb;
reg clk;
reg rst;
reg SIE;
reg STUFF_OPER_tx;
reg [7:0]sync_data;
reg [15:0]data_in;
wire encoded_dataout;
wire  SYNC_pattern;
wire [3:0]opcode;
```

```
 Transmitter dut(
                            clk,
                     rst,
                     SIE,
                     STUFF_OPER_tx,
                     sync_data,

                     data_in,
                     SYNC_pattern,
                     encoded_dataout,
                            opcode
                );

initial
begin
$dumpfile("test.vcd");
$dumpvars(0, Transmitter_tb);

#1
clk = 0; // clock in test bench
rst = 1;
SIE = 1;
data_in = 16'b1011_1100_1111_0010;
STUFF_OPER_tx = 1;

#15
rst = 0;
SIE = 1;
sync_data=8'b0111_1110;
data_in = 16'b1011_1100_1111_0010;
STUFF_OPER_tx = 1;

#1000 $stop;
end

always
#10 clk = ~clk;


endmodule
```