ENHANCEMENT OF USB2PHY TEST SUITE

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF ENGINEERING

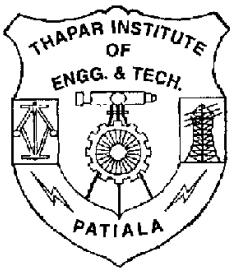
IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted By:
Shilpa Sharma
Roll No-8044124

Under the guidance of:

Mrs. Manu Bansal (TIET Patiala)
Mr. Alok Kaushik (STMicroelectronics, Greater Noida)



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(DEEMED UNIVERSITY),
PATIALA – 147004.

Year – 2006

DECLARATION

I, Shilpa Sharma hereby declare that the work which is being presented in this thesis entitled **"*Enhancement of US2PHY Test Suite*"** (Work Carried out at **LIPP**,**STMicroelectronics, Greater Noida**) by me in partial fulfillment of requirements for the award of degree of Master of Engineering in Electronics and Communication from Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried under the supervision of Mr. Alok Kaushik (Project Manager, STMicroelectronics) and Mrs. Manu Bansal, TIET Patiala.

The matter presented in this thesis has not been submitted in any other University or Institute for the award of Master of Engineering.

SHILPA SHARMA

This is certified that the above statement made by the candidate is correct to the best of my knowledge

(Mr. Alok Kaushik)
STMicroelectronics, Greater Noida

(Mrs. Manu Bansal)
Senior Lecturer, TIET Patiala

Counter Signed:

Head of Department, ECED
T.I.E.T, Patiala.

(Mr. T.P.Singh)
Dean of Academic Affairs,
T.I.E.T, Patiala

ABSTRACT

USB2PHY TEST SUITE aims to verify USB IP RTL/Netlist considering all test conditions including corner cases too.

The advantage of this suite is that the USB IP RTL/Netlist behavior can be verified for small changes made in the design. The environment is so flexible that any particular test condition can be realized and IP can be verified against that test condition. The purpose of host controller and device controller is served by test-benches.

It consists of 108(original) test-cases which are to be verified. More test conditions are to be enlisted from the available documents for USB_2.0 specifications and UTMI_1.05 specifications.

The purpose of the thesis is to enlist more test conditions and write VHDL code for those test conditions along-with the verification and debugging of existing test environment. By the time of completion of this thesis 60 more test-cases have been written. For these 60 new test conditions and also for existing 108 some new procedures have been written and some existing procedures have been modified.

Chapter 1: INTRODUCTION

1.1 Introduction to USB (Universal Serial Bus)

The original motivation for USB came from three interrelated considerations:

- With the merge of computing and communication, the movement of machine-oriented and human oriented data types from one location or environment to another depends on ubiquitous and cheap connectivity [1]. USB provides a ubiquitous link that can be used across a wide range of PC related interconnects.
- From end user's point of view, the PC's I/O interfaces such as serial/parallel ports, keyboard/mouse/joystick interfaces, etc., do not have the attributes of plug-and- play.
- The addition of external peripherals continues to be constrained by port availability. The lack of a bidirectional, low cost, low to mid speed peripheral bus has held back the creative prolification of peripherals such as telephone/fax/modem adapters, answering machines, scanners, PDA's, keyboards, mice, etc. Existing interconnects are optimized for one or two point products. As each new function or capability is added to PC, a new interface has been defined to address this need.
- The more recent motivation for USB2.0 stems from the fact that PCs have increasingly higher performance and are capable of processing vast amounts of data. At the same time, PC peripherals have added more performance and functionality. User applications such as digital imaging demand a high performance connection between the PC and these increasingly sophisticated peripherals.USB2.0 address this need by adding a third transfer rate of 480 Mb/s to 12Mb/s and 1.5Mb/s originally defined for USB.USB2.0 is a natural evolution of USB, delivering the desired bandwidth increase while preserving the original motivations for USB and maintaining full compatibility with existing peripherals.

Thus, USB continues to be the answer to connectivity for the PC architecture. It is fast, bi-directional, isochronous, low-cost, dynamically attachable serial interface that is consistent with the requirements of the PC platform of today and tomorrow.

1.2 Goals for the Universal Serial Bus

The USB is specified to be an industry-standard extension to the PC architecture with a focus on PC peripherals that enable consumer and business applications. The following criteria were applied in defining the architecture for the USB:

• Ease-of-use for PC peripheral expansion

• Low-cost solution that supports transfer rates up to 480 Mb/s

• Full support for real-time data for voice, audio, and video

• Protocol flexibility for mixed-mode isochronous data transfers and asynchronous messaging

• Integration in commodity device technology

• Comprehension of various PC configurations and form factors

• Provision of a standard interface capable of quick diffusion into product

• Enabling new classes of devices that augment the PC's capability

• Full backward compatibility of USB 2.0 for devices built to previous versions of the specification

1.3 Purpose of Industry Standard USB (USB2.0)

An industry-standard USB defines the bus attributes, the protocol definition, types of transactions, bus management, and the programming interface required to design and build systems and peripherals [2]. The goal is to enable such devices from different vendors to interoperate in an open architecture. The standard is intended as an enhancement to the PC architecture, spanning portable, business desktop and home environments. It is intended that the specification allow system OEMs and peripheral developer's adequate room for product versatility and market differentiation without the burden of carrying obsolete interfaces or losing compatibility.

1.4 Taxonomy of Application Space

Figure 1.1 describes taxonomy for the range of data traffic workloads that can be serviced over a USB. As can be seen, a 480 Mb/s bus comprehends the high-speed, full-speed, and low-speed data ranges. Typically, high-speed and full-speed data types may be isochronous, while low-speed data comes from interactive devices. The USB is primarily a PC bus but can be readily applied to other host-centric computing devices. The software

architecture allows for future extension of the USB by providing support for multiple USB Host Controllers.

PERFORMANCE	APPLICATIONS	ATTRIBUTES
LOW SPEED • Interactive Devices • 10 – 100 kb/s	**Keyboard, Mouse** **Stylus** **Game Peripherals** **Virtual Reality Peripherals**	**Lowest Cost** **Ease-of-Use** **Dynamic Attach-Detach** **Multiple Peripherals**
FULL-SPEED • Phone, Audio, Compressed Video 500 kb/s – 10 Mb/s	**POTS** **Audio** **Microphone**	**Lower Cost** **Ease-of-Use** **Broadband** **Dynamic Attach-Detach** **Guaranteed Latency** **Guaranteed Bandwidth** **Multiple Peripherals**
HIGH-SPEED • Video, Storage • 25 – 400 Mb/s	**Video** **Storage** **Imaging** **Broadband**	**Low Cost** **Ease-of-Use** **High Bandwidth** **Guaranteed Latency** **Dynamic Attach-Detach** **Guaranteed Bandwidth** **Multiple Peripherals**

Fig. 1.1 Application Space Taxonomy

Chapter 2: ARCHITECTURAL OVERVIEW OF USB

The USB is a cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripherals. The attached peripherals share USB bandwidth through a host scheduled, token-based protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation [3].

2.1 USB System Description

A USB system is described by three definitional areas:

- **USB interconnect**
- **USB devices**
- **USB host**

2.2 USB interconnect

USB interconnect is the manner in which USB devices are connected to and communicate with the host. This includes the following:

- **Bus Topology**: Connection model between USB devices and the host.
- **Inter-layer Relationships**: In terms of a capability stack, the USB tasks that are performed at each layer in the system.
- **Data Flow Models**: The manner in which data moves in the system over the USB between producers and consumers.
- **USB Schedule**: The USB provides a shared interconnect. Access to the interconnect is scheduled in order to support isochronous data transfers and to eliminate arbitration overhead.

2.2.1 Bus Topology

The USB connects USB devices with the USB host. The USB physical interconnect is a tiered star topology [4]. A hub is at the center of each star. Each wire segment is a point-to-point connection between the host and a hub or function, or a hub connected to another hub or function. Figure 2.1 illustrates the topology of the USB. Due to timing constraints allowed for hub and cable propagation times, the maximum number of tiers allowed is seven (including the root tier). In seven tiers, five non-root hubs maximum can be supported in a

communication path between the host and any device. A compound device occupies two tiers; therefore, it cannot be enabled if attached at tier level seven. Only functions can be enabled in tier seven.

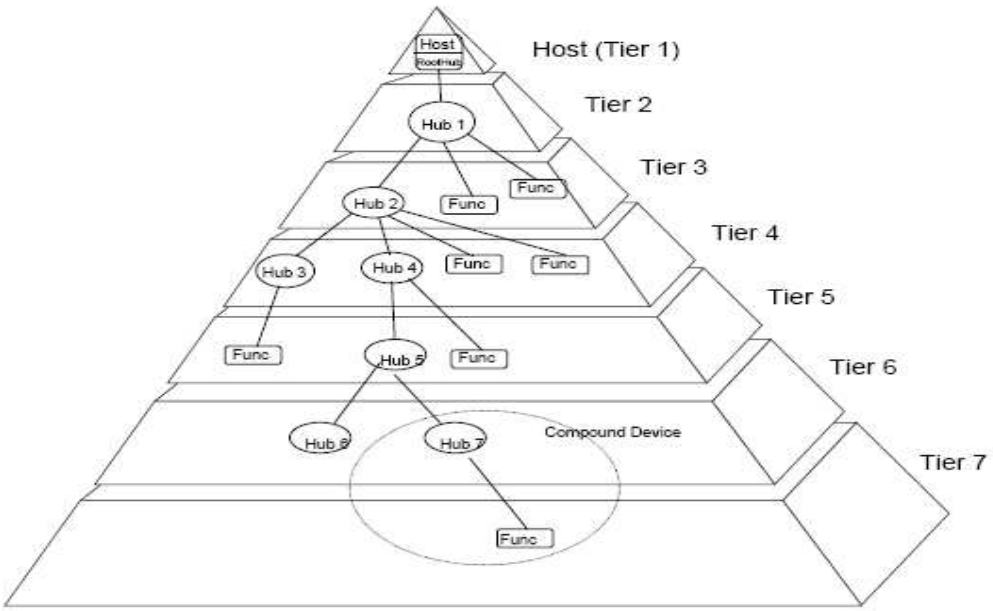


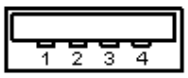
Figure 2.1 Bus Topology

2.3 Physical Interface

The physical interface of the USB is described as electrical and mechanical specifications for the bus.

2.3.1 Connectors

All devices have an upstream connection to the host and all hosts have a downstream connection to the device [5]. Upstream and downstream connectors are not mechanically interchangeable, thus eliminating illegal loop-back connections at hubs such as a downstream port connected to a downstream port. There are commonly two types of connectors, called type A and type B which are shown below.



Type A USB Connector

Type B USB Connector

Fig. 2.2 USB Connectors

USB 2.0 included errata which introduces mini-USB B connectors. The reasoning behind the mini connectors came from the range of miniature electronic devices such as mobile phones and organizers. The current type B connector is too large to be easily integrated into these devices.

Just recently released has been the On-The-Go specification which adds peer-to-peer functionality to USB. This introduces USB hosts into mobile phone and electronic organizers, and thus has included a specification for mini-A plugs, mini-A receptacles, and mini-AB receptacles.

2.3.2 Electrical

The USB transfers signal and power over a four-wire cable, shown in Figure 2.3. The signaling occurs over two wires on each point-to-point segment.

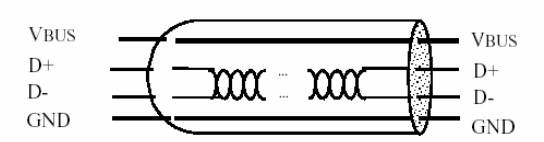


Fig .2.3 USB Cable

The cable also carries VBUS and GND wires on each segment to deliver power to devices. VBUS is nominally +5 V at the source. The USB allows cable segments of variable lengths, up to several meters, by choosing the appropriate conductor gauge to match the specified IR drop and other attributes such as device power budget and cable flexibility [6]. In order to provide guaranteed input voltage levels and proper termination impedance, biased terminations are used at each end of the cable. The terminations also permit the detection of attach and detach at each port and differentiate between high/full-speed and low-speed devices.

2.4 Speed Identification

A USB device must indicate its speed by pulling either the D+ or D- line high to 3.3 volts. A full speed device, pictured below will use a pull up resistor attached to D+ to specify itself as a full speed device [7]. These pull up resistors at the device end will also be used by the host or hub to detect the presence of a device connected to its port. Without a pull up resistor, USB assumes there is nothing connected to the bus.

Some devices have this resistor built into its silicon, which can be turned on and off under firmware control, others require an external resistor. For example Philips Semiconductor has a SoftConnectTM technology. When first connected to the bus, this allows the microcontroller to initialize the USB function device before it enables the pull up speed identification resistor, indicating a device is attached to the bus. If the pull up resistor was connected to Vbus, then this would indicate a device has been connected to the bus as soon as the plug is inserted. The host may then attempt to reset the device and ask for a descriptor when the microprocessor hasn't even started to initialize the USB function device.

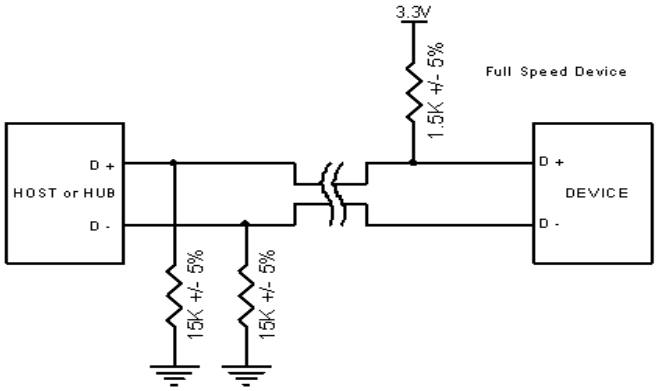


Fig. 2.4 Full Speed Device with pull up resistor connected to D+

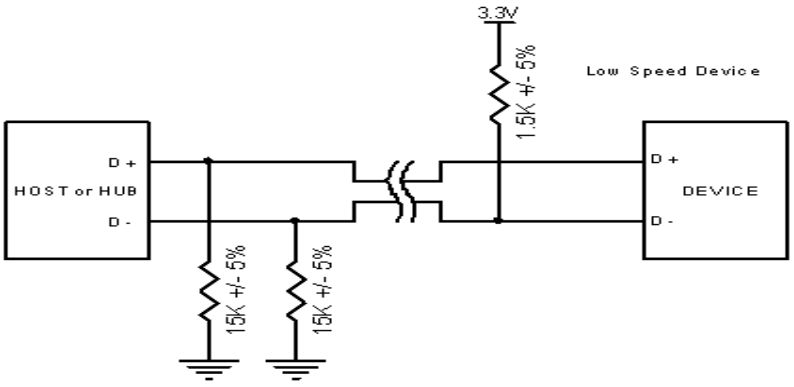


Fig.2.5 Low Speed Device with pull up resistor connected to D-

High speed devices will start by connecting as a full speed device (1.5k to 3.3V). Once it has been attached, it will do a high speed chirp during reset and establish a high speed connection if the hub supports it. If the device operates in high speed mode, then the pull up resistor is removed to balance the line.

2.5 Power (VBUS)

One of the benefits of USB is bus-powered devices - devices which obtain its power from the bus and requires no external plug packs or additional cables. However many leap at this option without first considering all the necessary criteria.

A USB device specifies its power consumption expressed in 2mA units in the configuration descriptor which we will examine in detail later. A device cannot increase its power consumption, greater than what it specifies during enumeration, even if it looses external power. There are three classes of USB functions,

- Low-power bus powered functions
- High-power bus powered functions
- Self-powered functions

Low power bus powered functions draw all its power from the VBUS and cannot draw any more than one unit load. The USB specification defines a unit load as 100mA. Low power bus powered functions must also be designed to work down to a VBUS voltage of 4.40V and up to a maximum voltage of 5.25V measured at the upstream plug of the device. For many 3.3V devices, LDO regulators are mandatory.

High power bus powered functions will draw all its power from the bus and cannot draw more than one unit load until it has been configured, after which it can then drain 5 unit loads (500mA Max) provided it asked for this in its descriptor. High power bus functions must

be able to be detected and enumerated at a minimum 4.40V. When operating at a full unit load, a minimum VBUS of 4.75 V is specified with a maximum of 5.25V. Once again, these measurements are taken at the upstream plug.

Self power functions may draw up to 1 unit load from the bus and derive the rest of its power from an external source. Should this external source fail, it must have provisions in place to draw no more than 1 unit load from the bus. Self powered functions are easier to design to specification as there is not so much of an issue with power consumption. The 1 unit bus powered load allows the detection and enumeration of devices without mains/secondary power applied.

No USB device, whether bus powered or self powered can drive the VBUS on its upstream facing port. If VBUS is lost, the device has a lengthy 10 seconds to remove power from the D+/D- pull-up resistors used for speed identification.

Other VBUS considerations are the Inrush current which must be limited. This is outlined in the USB specification paragraph 7.2.4.1 and is commonly overlooked. Inrush current is contributed to the amount of capacitance on your device between VBUS and ground. The spec therefore specifies that the maximum decoupling capacitance you can have on your device is 10uF. When you disconnect the device after current is flowing through the inductive USB cable, a large fly back voltage can occur on the open end of the cable. To prevent this, a 1uF minimum VBUS decoupling capacitance is specified.

2.6 System Configuration

The USB supports USB devices attaching to and detaching from the USB at any time. Consequently, system software must accommodate dynamic changes in the physical bus topology.

2.6.1 Attachment of USB Devices

All USB devices attach to the USB through ports on specialized USB devices known as hubs. Hubs have status bits that are used to report the attachment or removal of a USB device on one of its ports. The host queries the hub to retrieve these bits. In the case of an attachment, the host enables the port and addresses the USB device through the device's control pipe at the default address. The host assigns a unique USB address to the device and then determines if the newly attached USB device is a hub or a function. The host establishes its end of the control pipe for the USB device using the assigned

USB address and endpoint number zero. If the attached USB device is a hub and USB devices are attached to its ports, then the above procedure is followed for each of the attached USB devices.

If the attached USB device is a function, then attachment notifications will be handled by host software that is appropriate for the function.

2.6.2 Removal of USB Devices

When a USB device has been removed from one of a hub's ports, the hub disables the port and provides an indication of device removal to the host [8]. The removal indication is then handled by appropriate USB System Software. If the removed USB device is a hub, the USB System Software must handle the removal of both the hub and of all of the USB devices that were previously attached to the system through the hub.

2.6.3 Bus Enumeration

Bus enumeration is the activity that identifies and assigns unique addresses to devices attached to a bus. Because the USB allows USB devices to attach to or detach from the USB at any time, bus numeration is an on-going activity for the USB System Software. Additionally, bus enumeration for the USB also includes the detection and processing of removals.

2.7 Data Flow Types

The USB supports functional data and control exchange between the USB host and a USB device as a set of either uni-directional or bi-directional pipes. USB data transfers take place between host software and a particular endpoint on a USB device. Such 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 [9]. As an example, a given USB device could have an endpoint that supports a pipe for transporting data to the USB device and another endpoint that supports a pipe for transporting data from the USB device. The USB architecture comprehends four basic types of data transfers.

- **Control Transfers**: 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.
- **Bulk Data Transfers**: Generated or consumed in relatively large and bursty quantities and have wide dynamic latitude in transmission constraints.

- **Interrupt Data Transfers**: Used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics.
- **Isochronous Data Transfers**: Occupy a prenegotiated amount of USB bandwidth with prenegotiated delivery latency. (Also called streaming real time transfers).

2.8 USB Data Flow Model

The USB provides communication services between a host and attached USB devices. However, the simple view an end user sees of attaching one or more USB devices to a host is in fact a little more complicated to implement than is indicated by the figure 2.6. Different views of the system are required to explain specific USB requirements from the perspective of different implementers. Several important concepts and features must be supported to provide the end user with the reliable operation demanded from today's personal computers [10]. The USB is presented in a layered fashion to ease explanation and allow implementers of particular USB products to focus on the details related to their product.

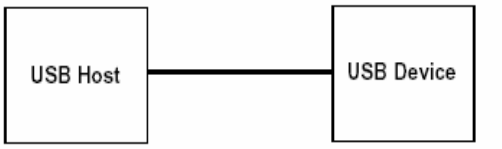


Fig 2.6 Simple USB Host/Device View

Figure 2.7 shows a deeper overview of the USB, identifying the different layers of the system .In particular, there are four focus implementation areas:

- **USB Physical Device**: A piece of hardware on the end of a USB cable that performs some useful end user function.
- **Client Software**: Software that executes on the host, corresponding to a USB device. This client software is typically supplied with the operating system or provided along with the USB device.
- **USB System Software**: Software that supports the USB in a particular operating system. The USB System Software is typically supplied with the operating system, independently of particular USB devices or client software.
- **USB Host Controller (Host Side Bus Interface)** : The hardware and software that allows USB devices to be attached to a host.

There are shared rights and responsibilities between the four USB system components. The remainder of this specification describes the details required to support robust, reliable communication flows between a function and its client.

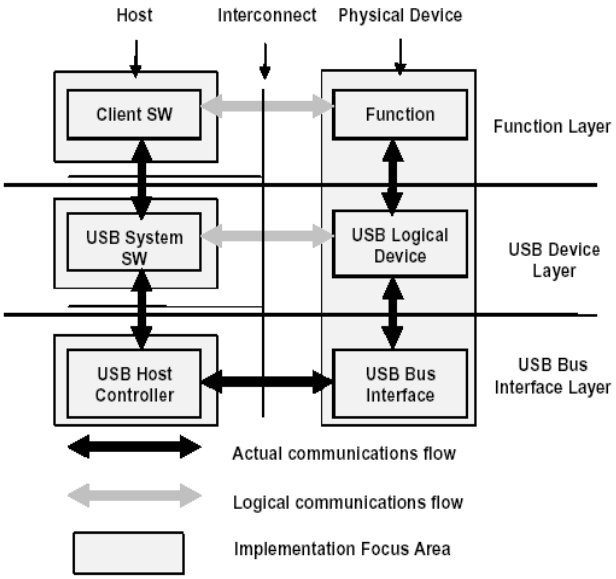


Fig2.7 USB Implementation Areas

As shown in Figure 2.7, the simple connection of a host to a device requires interaction between a number of layers and entities. The USB Bus Interface layer provides physical/signaling/packet connectivity between the host and a device. The USB Device layer is the view the USB System Software has for performing generic USB operations with a device. The Function layer 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.

2.9 USB Topology

There are four main parts to USB topology:

- **Host and Devices**: The primary components of a USB system
- **Physical Topology**: How USB elements are connected

- **Logical Topology**: The roles and responsibilities of the various USB elements and how the USB appears from the perspective of the host and a device
- **Client Software-to-function Relationships**: How client software and its related function interfaces on a USB device view each other

2.10 USB Host

The host's logical composition is shown in Figure 2.8 and includes the following:

- USB Host Controller
- Aggregate USB System Software (USB Driver, Host Controller Driver, and host software)
- Client

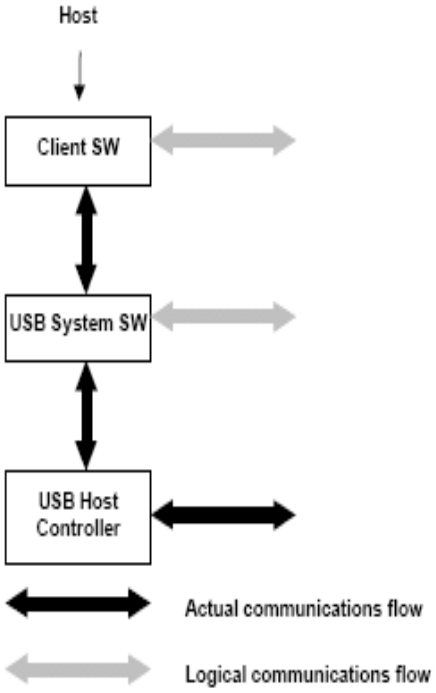


Fig. 2.8 USB host logical composition

The USB host occupies a unique position as the coordinating entity for the USB. In addition to its special physical position, the host has specific responsibilities with regard to the USB and its attached devices. The host controls all access to the USB. A USB device gains access to the bus only by being granted access by the host. The host is also responsible for monitoring the topology of the USB.

2.11 USB Devices

A USB physical device's logical composition and includes the following:

- USB bus interface
- USB logical device
- Function

USB physical devices provide additional functionality to the host. The types of functionality provided by USB devices vary widely. However, all USB logical devices present the same basic interface to the host. This allows the host to manage the USB-relevant aspects of different USB devices in the same manner. To assist the host in identifying and configuring USB devices, each device carries and reports configuration related information. Some of the information reported is common among all logical devices. Other information is specific to the functionality provided by the device. The detailed format of this information varies, depending on the device class of the device.

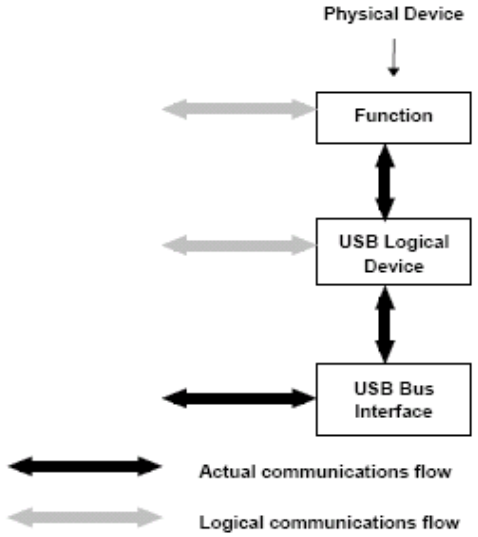


Fig.2.9 USB Physical device's logical composition

2.12 Physical Bus Topology

Devices on the USB are physically connected to the host via a tiered star topology USB attachment points are provided by a special class of USB device known as a hub. The additional attachment points provided by a hub are called ports. A host includes an embedded hub called the root hub. The host provides one or more attachment points via the root hub. USB devices that provide additional functionality to the host are known as functions. To

prevent circular attachments, a tiered ordering is imposed on the star topology of the USB. This results in the tree-like configuration as shown in figure 2.10.

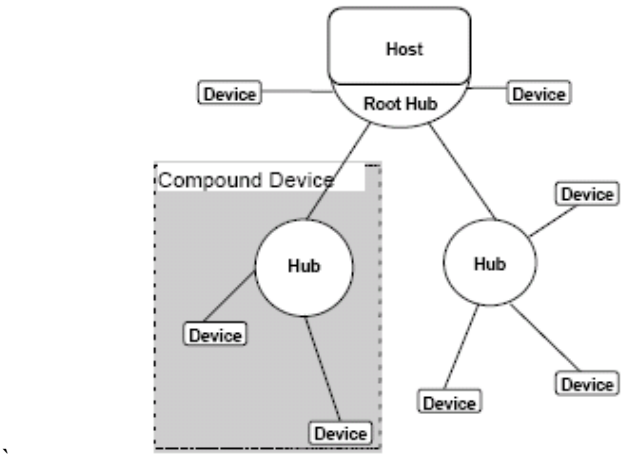


Fig. 2.10 star topology for USB Devices

Multiple functions may be packaged together in what appears to be a single physical device. For example, a keyboard and a trackball might be combined in a single package. Inside the package, the individual functions are permanently attached to a hub and it is the internal hub that is connected to the USB. When multiple functions are combined with a hub in a single package, they are referred to as a compound device. The hub and each function attached to the hub within the compound device is assigned its own device address. A device that has multiple interfaces controlled independently of each other is referred to as a composite device. A composite device has only a single device address. From the host's perspective, a compound device is the same as a separate hub with multiple functions attached.

2.13 Logical Bus Topology

While devices physically attach to the USB in a tiered, star topology, the host communicates with each logical device as if it were directly connected to the root port. This creates the logical view that corresponds to the physical topology. Hubs are logical devices also but are not shown to simplify the picture. Even though most host/logical device activities use this logical perspective, the host maintains an awareness of the physical topology to support processing the removal of hubs. When a hub is removed, all of the devices attached to the hub must be removed from the host's view of the logical topology.

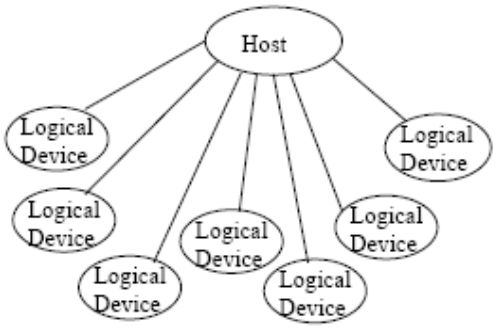


Fig. 2.11 logical bus topology

2.14 Client Software-to-function Relationship

Even though the physical and logical topology of the USB reflects the shared nature of the bus, client software (CSw) manipulating a USB function interface is presented with the view that it deals only with its interface(s) of interest. Client software for USB functions must use USB software programming interfaces to manipulate their functions as opposed to directly manipulating their functions via memory or I/O accesses as with other buses. During operation, client software should be independent of other devices that may be connected to the USB. This allows the designer of the device and client software to focus on the hardware/software interaction design details. Figure 2.12 illustrates a device designer's perspective of the relationships of client software and USB functions with respect to the USB logical topology.

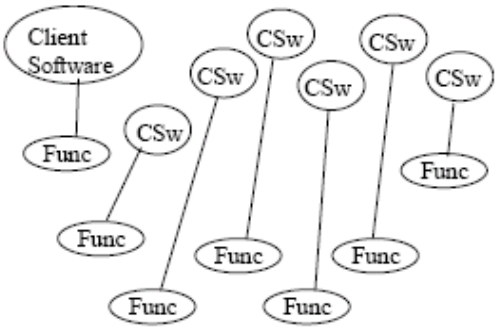


Fig 2.12 relationships of client software and USB functions

2.15 USB Communication Flow

The USB provides a communication service between software on the host and its USB function. Functions can have different communication flow requirements for different client-to-function interactions. The USB provides better overall bus utilization by allowing the separation of the different communication flows to a USB function. Each communication flow makes use of some bus access to accomplish communication between client and function. Each communication flow is terminated at an endpoint on a device. Device endpoints are used to identify aspects of each communication flow.

The complete definition of the actual communication flows supports the logical device and function layer communication flows. This actual communication flows cross several interface boundaries.

- Host Controller Driver (HCD): The software interface between the USB Host Controller and USB System Software. This interface allows a range of Host Controller implementations without requiring all host software to be dependent on any particular implementation. One USB Driver can support different Host Controllers without requiring specific knowledge of a Host Controller implementation. A Host Controller implementer provides an HCD implementation that supports the Host Controller.

- USB Driver (USBD): The interface between the USB System Software and the client software. This interface provides clients with convenient functions for manipulating USB devices.

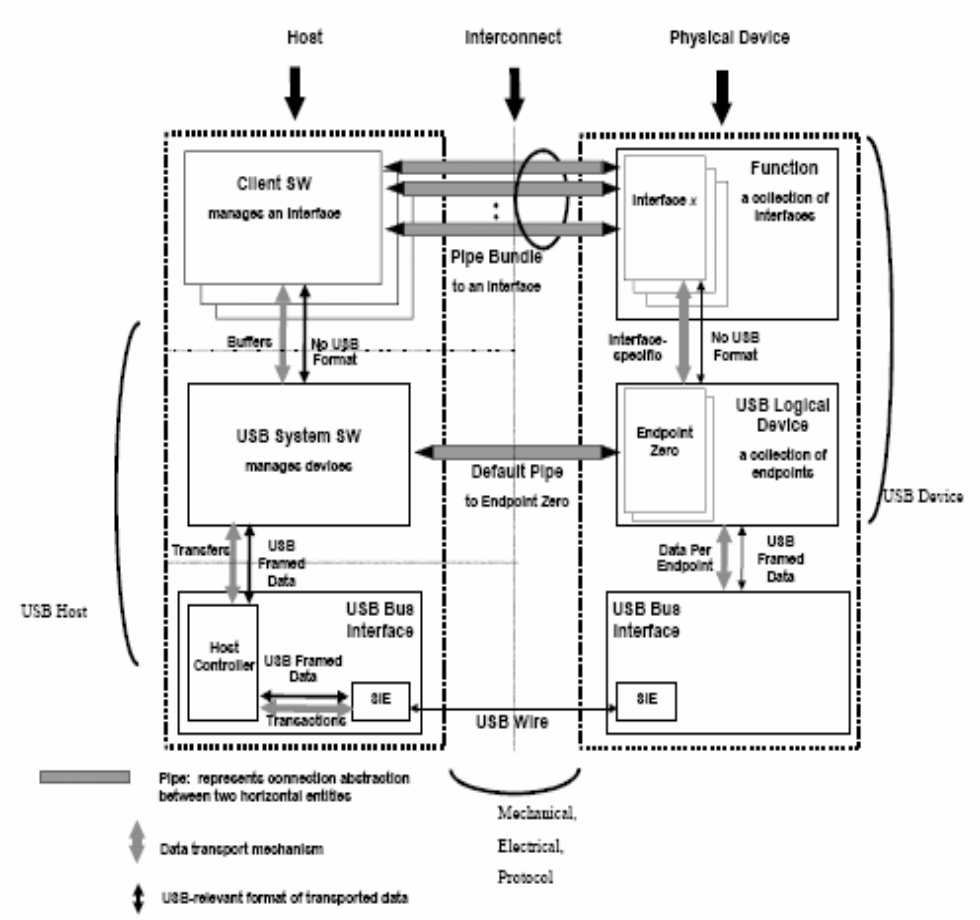


Fig.2.13 USB host/device detailed view

A USB logical device appears to the USB system as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function. The USB System Software manages the device using the Default Control Pipe. Client software manages an interface using pipe bundles (associated with an endpoint set). Client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The Host Controller (or USB device, depending on transfer direction) packetizes the data to move it over the USB. The Host Controller also coordinates when bus access is used to move the packet of data over the USB.

Figure 2.14 illustrates how communication flows are carried over pipes between endpoints and host side memory buffers. The following sections describe endpoints, pipes, and communication flows in more detail.

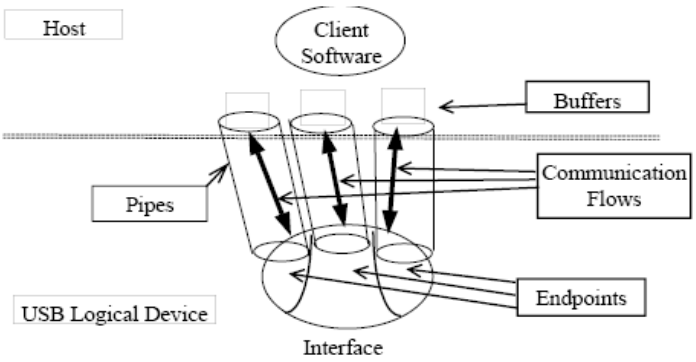


Fig. 2.14 USB communication flow

Software on the host communicates with a logical device via a set of communication flows. The set of communication flows are selected by the device software/hardware designer(s) to efficiently match the communication requirements of the device to the transfer characteristics provided by the USB.

2.16 Device Endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints. Each logical device has a unique address assigned by the system at device attachment time.

Each endpoint on a device is given at design time a unique device-determined identifier called the endpoint number. Each endpoint has a device-determined direction of data flow. The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced. Each endpoint is a simplex connection that supports data flow in one direction: either input (from device to host) or output (from host to device).

An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software. An endpoint describes itself by:

- Bus access frequency/latency requirement
- Bandwidth requirement

- Endpoint number
- Error handling behavior requirements
- Maximum packet size that the endpoint is capable of sending or receiving
- The transfer type for the endpoint
- The direction in which data is transferred between the endpoint and the host
- Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

2.16.1 Endpoint Zero Requirements

All USB devices are required to implement a default control method that uses both the input and output endpoints with endpoint number zero. The USB System Software uses this default control method to initialize and generically manipulate the logical device (e.g., to configure the logical device) as the Default Control Pipe. The Default Control Pipe provides access to the device's configuration information and allows generic USB status and control access. The Default Control Pipe supports control transfers. The endpoints with endpoint number zero are always accessible once a device is attached, powered, and has received a bus reset. A USB device that is capable of operating at high-speed must have a minimum level of support for operating at full-speed. When the device is attached to a hub operating in full-speed, the device must:

- Be able to reset successfully at full-speed
- Respond successfully to standard requests: set_address, set_configuration, get_descriptor for device and configuration descriptors, and return appropriate information The high-speed device may or may not be able to support its intended functionality when operating at full speed.

2.16.2 Non-endpoint Zero Requirements

Functions can have additional endpoints as required for their implementation. Low-speed functions are limited to two optional endpoints beyond the two required to implement the Default Control Pipe. Full speed devices can have additional endpoints only limited by the protocol definition (i.e., a maximum of 15 additional input endpoints and 15 additional output endpoints). Endpoints other than those for the Default Control Pipe cannot be used until the device is configured as a normal part of the device configuration process.

2.17 Transfer Types

The USB transports data through a pipe between a memory buffer associated with a software client on the host and an endpoint on the USB device. Data transported by message pipes is carried in a USB-defined structure, but the USB allows device-specific structured data to be transported within the USB-defined message data payload. The USB also defines that data moved over the bus is packetized for any pipe (stream or message), but ultimately the formatting and interpretation of the data transported in the data payload of a bus transaction is the responsibility of the client software and function using the pipe.

However, the USB provides different transfer types that are optimized to more closely match the service requirements of the client software and function using the pipe. An IRP uses one or more bus transactions to move information between a software client and its function. Each transfer type determines various characteristics of the communication flow including the following:

- Data format imposed by the USB
- Direction of communication flow
- Packet size constraints
- Bus access constraints
- Latency constraints
- Required data sequences
- Error handling

The designers of a USB device choose the capabilities for the device's endpoints. When a pipe is established for an endpoint, most of the pipe's transfer characteristics are determined and remain fixed for the lifetime of the pipe. Transfer characteristics that can be modified are described for each transfer type.

The USB defines four transfer types:

- **Control Transfers:** Bursty, non-periodic, host software-initiated request/response communication typically used for command/status operations.
- **Isochronous Transfers**: Periodic, continuous communication between host and device typically used for time-relevant information. This transfer type also preserves the concept of time encapsulated in the data. This does not imply, however, that the delivery needs of such data are always time-critical.

- **Interrupt Transfers**: Low-frequency, bounded-latency communication.
- **Bulk Transfers**: Non-periodic, large-packet bursty communication, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

Each transfer type is described in detail in the following four major sections. The data for any IRP is carried by the data field of the data packet.

2.17.1Control Transfers

Control transfers allow access to different parts of a device. Control transfers are intended to support configuration/command/status type communication flows between client software and its function. A control transfer is composed of a Setup bus transaction moving request information from host to function, zero or more Data transactions sending data in the direction indicated by the Setup transaction, and a Status transaction returning status information from function to host. The Status transaction returns "success" when the endpoint has successfully completed processing the requested operation.

Each USB device is required to implement the Default Control Pipe as a message pipe. This pipe is used by the USB System Software. The Default Control Pipe provides access to the USB device's configuration, status, and control information. A function can, but is not required to, provide endpoints for additional control pipes for its own implementation needs.

The USB device framework defines standard, device class, or vendor-specific requests that can be used to manipulate a device's state. Descriptors are also defined that can be used to contain different information on the device. Control transfers provide the transport mechanism to access device

descriptors and make requests of a device to manipulate its behavior.

Control transfers are carried only through message pipes. Consequently, data flows using control transfers must adhere to USB data structure definitions .The USB system will make a "best effort" to support delivery of control transfers between the host and devices. A function and its client software cannot request specific bus access frequency or bandwidth for control transfers. The USB System Software may restrict the bus access and bandwidth that a device may desire for control transfers.

2.17.1.1Control Transfer Data Format

The Setup packet has a USB-defined structure that accommodates the minimum set of commands required to enable communication between the host and a device. The structure

definition allows vendor-specific extensions for device specific commands. The Data transactions following Setup have a USB-defined structure except when carrying vendor-specific information. The Status transaction also has a USB-defined structure.

2.17.1.2 Control Transfer Direction

Control transfers are supported via bi-directional communication flow over message pipes. As a consequence, when a control pipe is configured, it uses both the input and output endpoint with the specified endpoint number.

2.17.1.3 Control Transfer Packet Size Constraints

An endpoint for control transfers specifies the maximum data payload size that the endpoint can accept from or transmit to the bus. The allowable maximum control transfer data payload sizes for full-speed devices are 8, 16, 32, or 64 bytes; for high-speed devices, it is 64 bytes and for low-speed devices, it is 8 bytes. This maximum applies to the data payloads of the Data packets following a Setup; i.e., the size specified is for the data field of the, not including other information that is required by the protocol. A Setup packet is always eight bytes. A control pipe (including the Default Control Pipe) always uses its *wMaxPacketSize* value for data payloads.

2.17.1.4Control Transfer Bus Access Constraints

Control transfers can be used by high-speed, full-speed, and low-speed USB devices. An endpoint has no way to indicate a desired bus access frequency for a control pipe. The USB balances the bus access requirements of all control pipes and the specific IRPs that are pending to provide “best effort” delivery of data between client software and functions [11]. The USB requires that part of each (micro) frame be reserved to be available for use by control transfers as follows:

- If the control transfers that are attempted (in an implementation-dependent fashion) consume less than 10% of the frame time for full-/low-speed endpoints or less than 20% of a microframe for high-speed endpoints, the remaining time can be used to support bulk transfers.

- A control transfer that has been attempted and needs to be retried can be retried in the current or a future (micro) frame; i.e., it is not required to be retried in the same (micro) frame.
- If there are more control transfers than reserved time, but there is additional (micro) frame time that is not being used for isochronous or interrupt transfers, a Host Controller may move additional control transfers as they are available.
- If there are too many pending control transfers for the available (micro) frame time, control transfers are selected to be moved over the bus as appropriate.
- If there are control transfers pending for multiple endpoints, control transfers for the different endpoints are selected according to a fair access policy that is Host Controller implementation-dependent.
- A transaction of a control transfer that is frequently being retried should not be expected to consume an unfair share of the bus time.

2.17.2 Isochronous Transfers

In non-USB environments, isochronous transfers have the general implication of constant-rate, error tolerant transfers. In the USB environment, requesting an isochronous transfer type provides the requester with the following:

- Guaranteed access to USB bandwidth with bounded latency
- Guaranteed constant data rate through the pipe as long as data is provided to the pipe
- In the case of a delivery failure due to error, no retrying of the attempt to deliver the data

While the USB isochronous transfer type is designed to support isochronous sources and destinations, it is not required that software using this transfer type actually be isochronous in order to use the transfer type.

2.17.2.1 Isochronous Transfer Data Format

The USB imposes no data content structure on communication flows for isochronous pipes.

2.17.2.2 Isochronous Transfer Direction

An isochronous pipe is a stream pipe and is, therefore, always uni-directional. An endpoint description identifies whether a given isochronous pipe's communication flow is into

or out of the host. If a device requires bi-directional isochronous communication flow, two isochronous pipes must be used, one in each direction.

2.17.2.3 Isochronous Transfer Packet Size Constraints

An endpoint in a given configuration for an isochronous pipe specifies the maximum size data payload that it can transmit or receive. The USB System Software uses this information during configuration to ensure that there is sufficient bus time to accommodate this maximum data payload in each (micro) frame. If there is sufficient bus time for the maximum data payload, the configuration is established; if not, the configuration is not established.

The USB limits the maximum data payload size to 1,023 bytes for each full-speed isochronous endpoint. High-speed endpoints are allowed up to 1024-byte data payloads. A high speed, high bandwidth endpoint specifies whether it requires two or three transactions per microframe. The table is shaded to indicate that a full-speed isochronous endpoint (with a non-zero *wMaxpacket* size) must not be part of a default interface setting.

2.17.2.4 Isochronous Transfer Bus Access Constraints

Isochronous transfers can only be used by full-speed and high-speed devices.

The USB requires that no more than 90% of any frame be allocated for periodic (isochronous and interrupt) transfers for full-speed endpoints. High-speed endpoints can allocate at most 80% of a microframe for periodic transfers.

An isochronous endpoint must specify its required bus access period. Full-/high-speed endpoints must specify a desired period as (2bInterval-1) x F, where bInterval is in the range one to (and including) 16 and F is 125 μs for high-speed and 1ms for full-speed. This allows full-/high-speed isochronous transfers to have rates slower than one transaction per (micro) frame. However, an isochronous endpoint must be prepared to handle poll rates faster than the one specified. A host must not issue more than 1 transaction in a (micro) frame for an isochronous endpoint unless the endpoint is high-speed, high-bandwidth (see below).An isochronous IN endpoint must return a zero-length packet whenever data is requested at a faster interval than the specified interval and data is not available.

2.17.3 Interrupt Transfers

The interrupt transfer type is designed to support those devices that need to send or receive data infrequently but with bounded service periods. Requesting a pipe with an interrupt transfer type provides the requester with the following:

- . Guaranteed maximum service period for the pipe
- . Retry of transfer attempts at the next period, in the case of occasional delivery failure due to error on the bus.

2.17.3.1 Interrupt Transfer Data Format

The USB imposes no data content structure on communication flows for interrupt pipes.

2.17.3.2 Interrupt Transfer Direction

An interrupt pipe is a stream pipe and is therefore always uni-directional. An endpoint description identifies whether a given interrupt pipe's communication flow is into or out of the host.

2.17.3.3 Interrupt Transfer Packet Size Constraints

An endpoint for an interrupt pipe specifies the maximum size data payload that it will transmit or receive. The maximum allowable interrupt data payload size is 64 bytes or less for full-speed. High-speed endpoints are allowed maximum data payload sizes up to 1024 bytes. A high speed, high bandwidth endpoint specifies whether it requires two or three transactions per microframe. Low-speed devices are limited to eight bytes or less maximum data payload size. This maximum applies to the data payloads of the data packets; i.e., the size specified is for the data field of the packet

The USB does not require that data packets be exactly the maximum size; i.e., if a data packet is less than the maximum, it does not need to be padded to the maximum size. All Host Controllers are required to support maximum data payload sizes from 0 to 64 bytes for full-speed interrupt endpoints, from 0 to 8 bytes for low-speed interrupt endpoints, and from 0 to 1024 bytes for high-speed interrupt endpoints.

2.17.3.4 Interrupt Transfer Data Sequences

Interrupt transactions may use either alternating data toggle bits, such that the bits are toggled only upon successful transfer completion or a continuously toggling of data toggle bits. The host in any case must assume that the device is obeying full handshake/retry rules A device may choose to always toggle DATA0/DATA1 PIDs so that it can ignore handshakes from the host.

2.17.4 Bulk Transfers

The bulk transfer type is designed to support devices that need to communicate relatively large amounts of data at highly variable times where the transfer can use any available bandwidth. Requesting a pipe with a bulk transfer type provides the requester with the following:

- Access to the USB on a bandwidth-available basis
- Retry of transfers, in the case of occasional delivery failure due to errors on the bus
- Guaranteed delivery of data but no guarantee of bandwidth or latency

Bulk transfers occur only on a bandwidth-available basis. For a USB with large amounts of free bandwidth, bulk transfers may happen relatively quickly; for a USB with little bandwidth available, bulk transfers may trickle out over a relatively long period of time.

2.17.4.1Bulk Transfer Data Format

The USB imposes no data content structure on communication flows for bulk pipes.

2.17.4.2 Bulk Transfer Direction

A bulk pipe is a stream pipe and, therefore, always has communication flowing either into or out of the host for a given pipe. If a device requires bi-directional bulk communication flows, two bulk pipes must be used, one in each direction.

2.17.4.3 Bulk Transfer Packet Size Constraints

An endpoint for bulk transfers specifies the maximum data payload size that the endpoint can accept from or transmit to the bus. The USB defines the allowable maximum bulk data payload sizes to be only 8, 16, 32, or 64 bytes for full-speed endpoints and 512 bytes for high-speed endpoints. A low-speed device must not have bulk endpoints. This maximum

applies to the data payloads of the data packets; i.e., the size specified is for the data field of the packet, not including other protocol-required information.

A bulk endpoint is designed to support a maximum data payload size. A bulk endpoint reports in its configuration information the value for its maximum data payload size. The USB does not require that data payloads transmitted be exactly the maximum size; i.e., if a data payload is less than the maximum, it does not need to be padded to the maximum size.

All Host Controllers are required to have support for 8-, 16-, 32-, and 64-byte maximum packet sizes for full-speed bulk endpoints and 512 bytes for high-speed bulk endpoints. No Host Controller is required to support larger or smaller maximum packet sizes.

During configuration, the USB System Software reads the endpoint's maximum data payload size and ensures that no data payload will be sent to the endpoint that is larger than the supported size.

An endpoint must always transmit data payloads with a data field less than or equal to the endpoint's reported *wMaxPacketSize* value. When a bulk IRP involves more data than can fit in one maximum-sized data payload, all data payloads are required to be maximum size except for the last data payload, which will contain the remaining data. A bulk transfer is complete when the endpoint does one of the following:

- Has transferred exactly the amount of data expected
- Transfers a packet with a payload size less than *wMaxPacketSize* or transfers a zero-length packet.

When a bulk transfer is complete, the Host Controller retires the current IRP and advances to the next IRP. If a data payload is received that is larger than expected, all pending bulk IRPs for that endpoint will be aborted/retired.

2.17.4.4 Bulk Transfer Bus Access Constraints

Only full-speed and high-speed devices can use bulk transfers. An endpoint has no way to indicate a desired bus access frequency for a bulk pipe. The USB balances the bus access requirements of all bulk pipes and the specific IRPs that are pending to provide "good effort" delivery of data between client software and functions. Moving control transfers over the bus has priority over moving bulk transfers.

There is no time guaranteed to be available for bulk transfers as there is for control transfers. Bulk transfers are moved over the bus only on a bandwidth-available basis. If there

is bus time that is not being used for other purposes, bulk transfers will be moved over the bus. If there are bulk transfers pending for multiple endpoints, bulk transfers for the different endpoints are selected according to a fair access policy that is Host Controller implementation-dependent.

2.17.5 Split Transactions

Host controllers and hubs support one additional transaction type called split transactions. This transaction type allows full- and low-speed devices to be attached to hubs operating at high-speed. These transactions involve only host controllers and hubs and are not visible to devices. High-speed split transactions for interrupt and isochronous transfers must be allocated by the host from the 80% periodic portion of a microframe.

2.18 USB Protocols

Unlike RS-232 and similar serial interfaces where the format of data being sent is not defined [12], USB is made up of several layers of protocols.

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)

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 a handshaking packet, reporting if the data or token was received successfully, or if the endpoint is stalled or not available to accept data.

2.19 Common USB Packet Fields

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

2.19.1 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.

2.19.2 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.

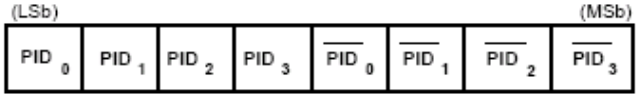


Fig. 2.15 PID Format

2.20 Address Fields

Function endpoints are addressed using two fields: the function address field and the endpoint field. A function needs to fully decode both address and endpoint fields. Address or endpoint aliasing is not permitted, and a mismatch on either field must cause the token to be ignored. Accesses to non-initialized endpoints will also cause the token to be ignored.

2.20.1 Address Field

The function address (ADDR) field specifies the function, via its address, that is either the source or destination of a data packet, depending on the value of the token PID. As shown in Figure 8-2, a total of 128 addresses are specified as ADDR<6:0>. The ADDR field is specified for IN, SETUP, and OUT tokens and the PING and SPLIT special token. By definition, each ADDR value defines a single function. Upon reset and power-up, a function's address defaults to a value of zero and must be programmed by the host during the enumeration process. Function address zero is reserved as the default address and may not be assigned to any other use.

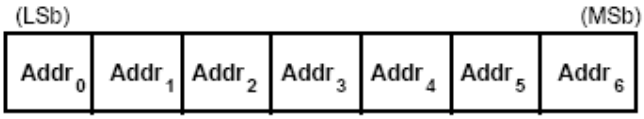


Fig 2.16 Address field

2.20.2 Endpoint Field

An additional four-bit endpoint (ENDP) field, shown in Figure 8-3, permits more flexible addressing of functions in which more than one endpoint is required. Except for

endpoint address zero, endpoint numbers are function-specific. The endpoint field is defined for IN, SETUP, and OUT tokens and the PING special token.

All functions must support a control pipe at endpoint number zero (the Default Control Pipe). Low speed devices support a maximum of three pipes per function: a control pipe at endpoint number zero plus two additional pipes (two control pipes, a control pipe and a interrupt endpoint, or two interrupt endpoints). Full-speed and high-speed functions may support up to a maximum of 16 IN and OUT endpoints.

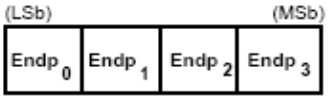


Fig. 2.17 Endpoint field

2.21 Frame Number Field

The frame number field is an 11-bit field that is incremented by the host on a per-frame basis. The frame number field rolls over upon reaching its maximum value of 7FFH and is sent only in SOF tokens at the start of each (micro) frame.

2.22 Data Field

The data field may range from zero to 1,024 bytes and must be an integral number of bytes. Figure 2.18 shows the format for multiple bytes. Data bits within each byte are shifted out LSb first.

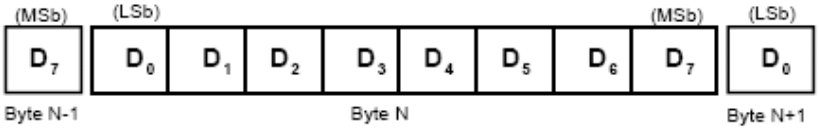


Fig. 218 data field format

2.23 Cyclic Redundancy Checks

Cyclic redundancy checks (CRCs) are used to protect all non-PID fields in token and data packets. In this context, these fields are considered to be protected fields. The PID is not included in the CRC check of a packet containing a CRC. All CRCs are generated over their respective fields in the transmitter before bit stuffing is performed. Similarly, CRCs are decoded in the receiver after stuffed bits have been removed. Token and data packet CRCs

provide 100% coverage for all single- and double-bit errors. A failed CRC is considered to indicate that one or more of the protected fields is corrupted and causes the receiver to ignore those fields and, in most cases, the entire packet.

For CRC generation and checking, the shift registers in the generator and checker are seeded with an all ones pattern [13]. For each data bit sent or received, the high order bit of the current remainder is XORed with the data bit and then the remainder is shifted left one bit and the low-order bit set to zero. If the result of that XOR is one, then the remainder is XORed with the generator polynomial. When the last bit of the checked field is sent, the CRC in the generator is inverted and sent to the checker MSb first. When the last bit of the CRC is received by the checker and no errors have occurred, the remainder will be equal to the polynomial residual. A CRC error exists if the computed checksum remainder at the end of a packet reception does not match the residual. Bit stuffing requirements must be met for the CRC, and this includes the need to insert a zero at the end of a CRC if the preceding six bits were all ones.

2.24 Signaling

The signaling specification for the USB is described in the following subsections.

2.24.1 Overview of High-speed Signaling

A high-speed USB connection is made through a shielded, twisted pair cable that conforms to all current USB cable specifications.

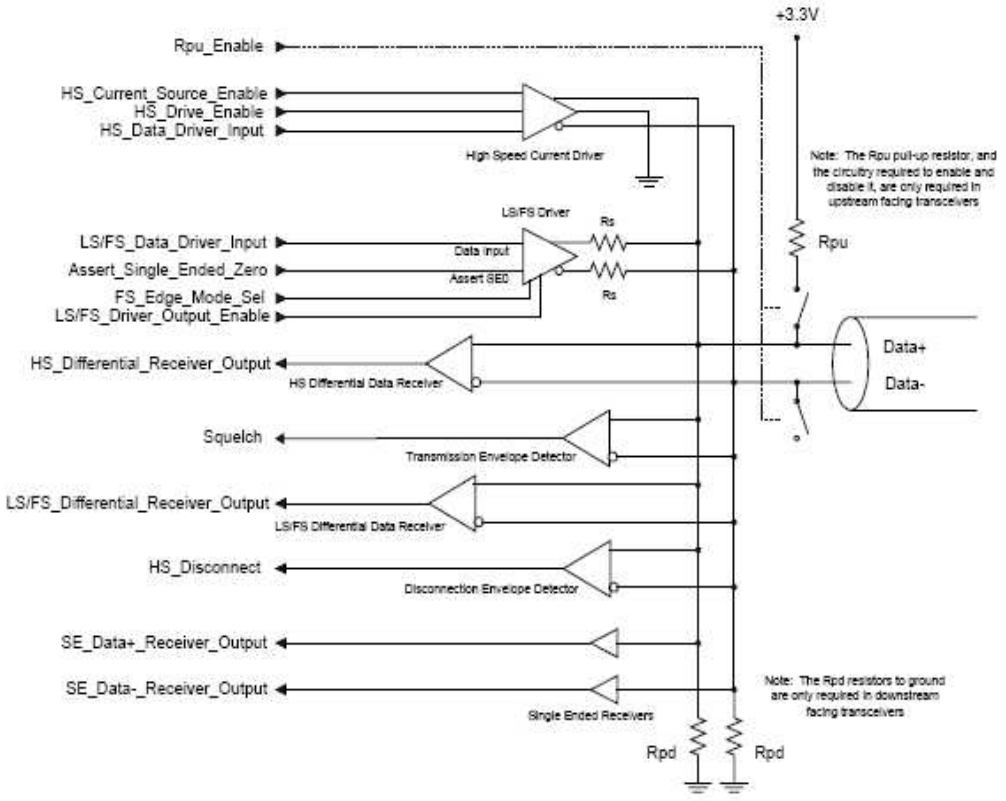


Fig. 2.19 High speed capable transceiver circuit

Figure 2.19 depicts an example implementation which largely utilizes USB 1.1 transceiver elements and adds the new elements required for high-speed operation. High-speed operation supports signaling at 480 Mb/s. To achieve reliable signaling at this rate, the cable is terminated at each end with a resistance from each wire to ground. The value of this resistance (on each wire) is nominally set to 1/2 the specified differential impedance of the cable, or 45Ω. This presents a differential termination of 90Ω.

For a link operating in high-speed mode, the high-speed idle state occurs when the transceivers at both ends of the cable present high-speed terminations to ground, and when neither transceiver drives signaling current into the D+ or D- lines. This state is achieved by using the low-/full-speed driver to assert a single ended zero, and to closely control the combined total of the intrinsic driver output impedance and the RS resistance (to 45Ω,nominal). The recommended practice is to make the intrinsic driver impedance as low as possible, and to let RS contribute as much of the 45 Ω as possible. This will generally lead to the best termination accuracy with the least parasitic loading.

In order to transmit in high-speed mode, a transceiver activates an internal current source which is derived from its positive supply voltage and directs this current into one of the two data lines via a high speed current steering switch. In this way, the transceiver generates the high-speed J or K state on the cable.

The dynamic switching of this current into the D+ or D- line follows the same NRZI data encoding scheme used in low-speed or full-speed operation and also in the bit stuffing behavior. To signal a J, the current is directed into the D+ line, and to signal a K, the current is directed into the D- line. The SYNC field and the EOP delimiters have been modified for high-speed mode.

The magnitude of the current source and the value of the termination resistors are controlled to specified tolerances, and together they determine the actual voltage drive levels. The DC resistance from D+ or D- to the device ground is required to be 45 Ω ±10% when measured without a load, and the differential output voltage measured across the lines (in either the J or K state) must be ±400 mV ± 10% when D+ and D- are terminated with precision 45 Ω resistors to ground.

The differential voltage developed across the lines is used for three purposes:

• A differential receiver at the receiving end of the cable receives the differential data signal.

• A differential envelope detector at the receiving end of the cable determines when the link is in the Squelch state. A receiver uses squelch detection as indication that the signal at its connector is not valid.

• In the case of a downstream facing hub transceiver, a differential envelope detector monitors whether the signal at its connector is in the high-speed state. A downstream facing transceiver operating in high-speed mode is required to test for this state at a particular point in time when it is transmitting a SOF packet. This is used to detect device disconnection.

In the absence of the far end terminations, the differential voltage will nominally double (as compared to when a high-speed device is present) when a high-speed J or K are continuously driven for a period exceeding the round-trip delay for the cable and board-traces between the two transceivers.

USB 2.0 requires that a downstream facing transceiver must be able to operate in low-speed, full-speed, and high-speed signaling modes. An upstream facing high-speed capable transceiver must not operate in low-speed signaling mode, but must be able to operate in full-speed signaling mode.

Therefore, a 1.5 kΩ pull-up on the D line is not allowed for a high-speed capable device, since a high-speed capable transceiver must never signal low-speed operation to the hub port to which it is attached.

Chapter 3: USB 2.0 TRANSCEIVER MACROCELL (UTM)

3.1 Introduction

High volume USB 2.0 devices will be designed using ASIC technology with embedded USB 2.0 support. For full-speed USB devices the operating frequency was low enough to allow data recovery to be handled in a vendors VHDL code, with the ASIC vendor providing only a simple level translator to meet the USB signaling requirements. Today's gate arrays operate comfortably between 30 and 60 MHz. With USB 2.0 signaling running at hundreds of MHz, the existing design methodology must change.

The intent of the UTMI is to accelerate USB 2.0 peripheral development. This chapter defines an interface to which ASIC and peripheral vendors can develop. ASIC vendors and foundries will implement the UTM and add it to their device libraries. Peripheral and IP vendors will be able to develop their designs, insulated from the high-speed and analog circuitry issues associated with the USB 2.0 interface, thus minimizing the time and risk of their development cycles.

The figure 3.1 summarizes a number of concepts expressed in UTMI specification. There are assumed to be three major functional blocks in a USB 2.0 peripheral ASIC design: the USB 2.0 Transceiver Macrocell, the Serial Interface Engine (SIE), and the device specific logic.

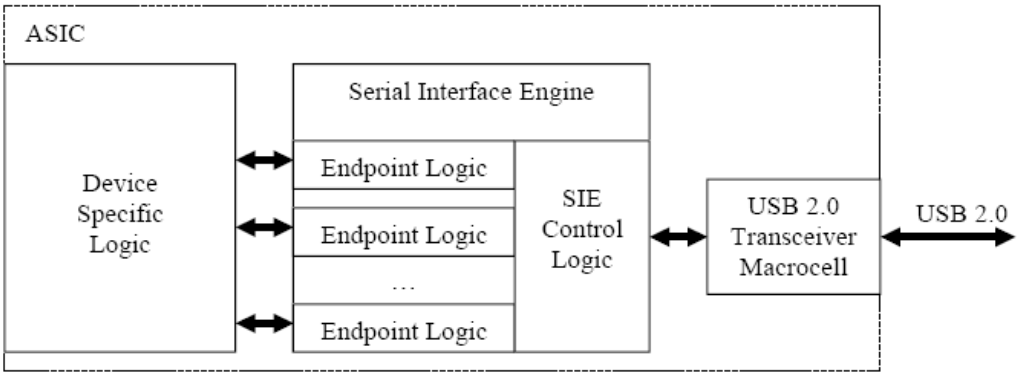


Fig. 3.1 ASIC Functional Block

3.1.1 USB 2.0 Transceiver Macrocell (UTM)

This block handles the low level USB protocol and signaling. This includes features such as; data serialization and deserialization, bit stuffing and clock recovery and synchronization. The primary focus of this block is to shift the clock domain of the data from the USB 2.0 rate to one that is compatible with the general logic in the ASIC [14].

Some key features of the USB 2.0 Transceiver are:

- Eliminates high speed USB 2.0 logic design for peripheral developers
- Standard Transceiver interface enables multiple IP sources for USB 2.0 SIE VHDL
- Supports 480 Mbit/s "High Speed" (HS)/ 12 Mbit/s “Full Speed” (FS), FS Only and "Low Speed" (LS) Only 1.5 Mbit/s serial data transmission rates.
- Utilizes 8-bit parallel interface to transmit and receive USB 2.0 cable data
- SYNC/EOP generation and checking
- Allows integration of high speed components in to a single functional block as seen by the peripheral designer
- High Speed and Full Speed operation to support the development of "Dual Mode" devices
- Data and clock recovery from serial stream on the USB
- Bit-stuffing/unstuffing; bit stuff error detection
- Holding registers to stage transmit and receive data
- Logic to facilitate Resume signaling
- Logic to facilitate Wake Up and Suspend detection
- Supports USB 2.0 Test Modes
- Ability to switch between FS and HS terminations/signaling
- Single parallel data clock output with on-chip PLL to generate higher speed serial data clocks

The UTMI is designed to support HS/FS, FS Only and LS Only UTM implementations. The three options allow a single SIE implementation to be used with any speed USB transceiver. A vendor can choose the transceiver performance that best meets their needs. A HS/FS implementation of the transceiver can operate at either a 480 Mb/s or a 12 Mb/s rate. Two modes of operation are required to properly emulate High-speed device connection and suspend/resume features of USB 2.0, as well as Full-speed connections if implementing a Dual-Mode device. FS Only and LS Only UTM implementations do not require the speed selection signals

3.1.2 Serial Interface Engine

This block can be further sub-divided into 2 types of sub-blocks; the SIE Control Logic and the Endpoint logic. The SIE Control Logic contains the USB PID and address recognition logic, and other sequencing and state machine logic to handle USB packets and transactions. The Endpoint Logic contains the endpoint specific logic: endpoint number recognition, FIFOs and FIFO control, etc.

Generally the SIE Control Logic is required for any USB implementation while the number and types of endpoints will vary as function of application and performance requirements. SIE logic module can be developed by peripheral vendors or purchased from IP vendors. The standardization of the UTMI allows compatible SIE VHDL to drop into an ASIC that provides the macrocell.

3.1.3 Device Specific Logic

This is the glue that ties the USB interface to the specific application of the device.

3.2 Functional Block Diagram

Figure 3.2 shows the functional block diagram of the USB 2.0 transceiver.

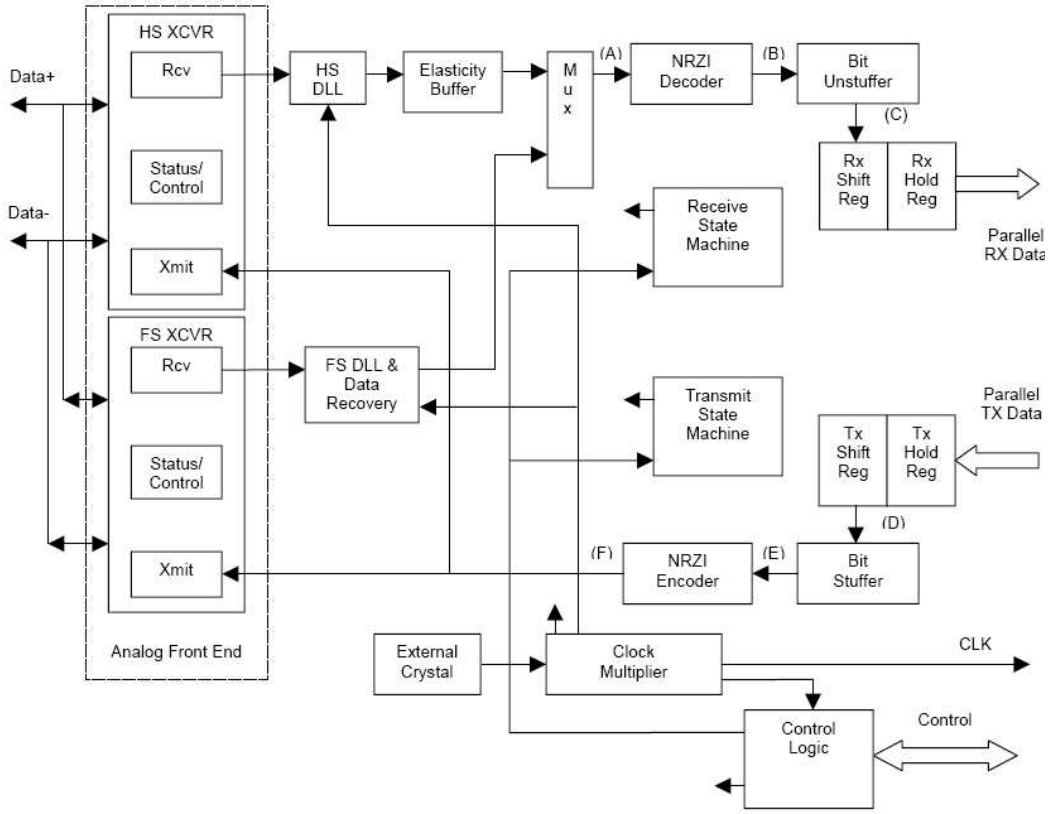


Fig.3.2 Functional Block Diagram

3.2.1 Block level description

3.2.1.1Clock Multiplier

This module generates the appropriate internal clocks for the UTM and the CLK output signal. All data transfer signals are synchronized with the CLK signal. The UTM vendor determines the frequency of the external crystal. The Clock Multiplier circuit and the External Crystal must meet the requirements defined in the USB 2.0 specification. After the release of SuspendM, the CLK signal generated by the transceiver must meet the following requirements:

1) Produce the first CLK transition no later than 5.6 ms after the negation of SuspendM.
2) The CLK signal frequency error must be less than 10% (±6.00 MHz)
3) The CLK must fully meet the required accuracy of ±500 ppm (±30.0 KHz), no later than 1.4ms after the first transition of CLK.

3.2.1.1.1Clocking HS/FS operation

In HS mode there is one CLK cycle per byte time. The frequency of CLK does not change when the UTMI is switched between HS to FS modes. In FS mode there are 5 CLK cycles per FS bit time, typically 40 CLK cycles per FS byte time. If a received byte contains a stuffed bit then the byte boundary can be stretched to 45 CLK cycles, and two stuffed bits would result in a 50 CLK delay between bytes. Figure 3.3 shows the relationship between CLK and the receive data transfer signals in FS mode.

RXActive "frames" a packet, transitioning only at the beginning and end of a packet, however transitions of RXValid may take place any time 8 bits of data are available. Figure 3.3 also shows how RXValid is only asserted for one CLK cycle per byte time even though the data may be presented for the full byte time. The Macrocell is required to present valid data for only for one clock cycle (while RXValid is asserted), although it may be presented until new data is received

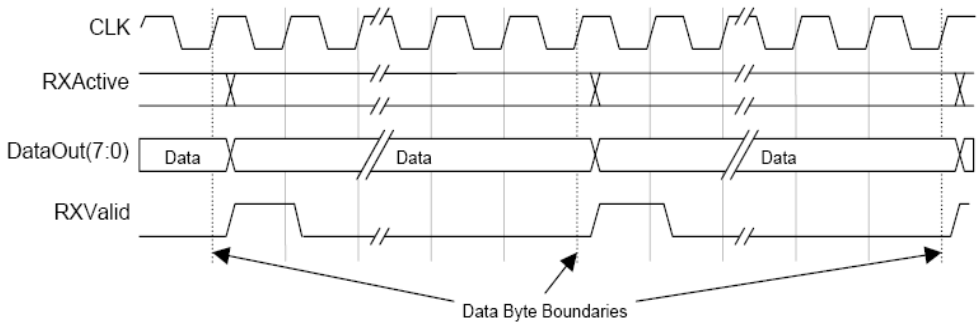


Figure 3.3 FS CLK Relationship to Receive Data and Control Signals

Figure 3.4 shows relationship between CLK and the transmit data transfer signals in FS mode. TXReady is only asserted for one CLK per byte time. This signal acknowledges to the SIE that the data on the DataIn lines has been read by the Macrocell (small arrows above DataIn signal. The SIE must present the next data byte on the DataIn bus after it detects TXReady high on a rising edge of CLK.

Transitions of TXValid must meet the defined setup and hold times relative to CLK. The delay between the assertion of TXValid and the first assertion of TXReady is Macrocell implementation dependent.

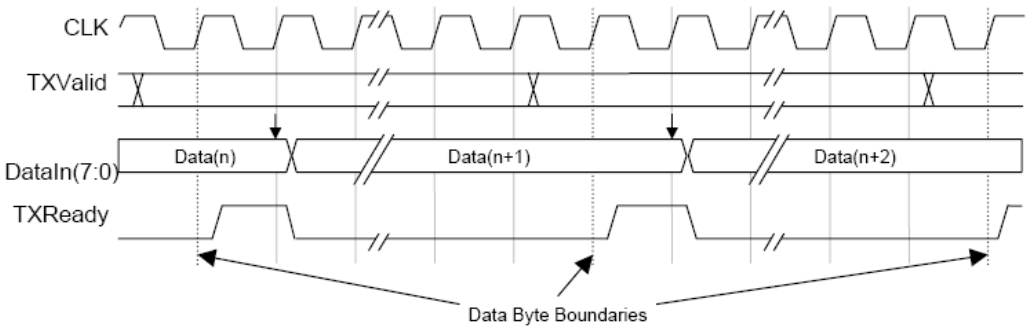


Fig 3.4 FS CLK Relationship to Transmit Data and Control Signals

The XcvrSelect signal determines whether the HS or FS timing relationship is applied to the data and control signals.

3.2.1.1.2 FS Only operation

A "FS Only" implementation of the UTM would provide 32 CLK cycles per byte time. The frequency of CLK would be 48.0 MHz. Timing is similar to a HS/FS UTM operating in FS mode.

3.2.1.1.3 LS only operation

A "LS Only" implementation of the UTM would provide 32 CLK cycles per byte time. The frequency of CLK would be 6.0 MHz. Timing is similar to a HS/FS UTM operating in FS mode.

3.2.1.2 HS DLL (High Speed Delay Line PLL)

The delay line PLL extracts clock and data from the data received over the USB 2.0 interface for reception by the Receive Deserializer. A vendor defined number of delayed clock taps are be used to sample the received data. The data output from the DLL is synchronous with the local clock.

3.2.1.3 Elasticity Buffer

This buffer is used to compensate for differences between transmitting and receiving clocks. The USB specification defines a maximum clock error of +/-500 ppm. When the error is calculated over the maximum packet size up to +/- 12 bits of drift can occur. The elasticity buffer is filled to a threshold prior to enabling the remainder of the down stream receive logic.

This block may be integrated into the DLL block. An example that will meet these requirements is a 24 bit deep, 1 bit wide FIFO with a threshold set at the midpoint. Overflow or underflow conditions detected in the elasticity buffer can be reported with the RXError signal.

3.2.1.4 Mux

The bulk of the logic in the transceiver can be used with HS or FS operations. The Mux block allows the data from the HS or FS receivers to be routed to the shared receive logic. The state of the Mux is determined by the XcvrSelect input.

3.2.1.5 NRZI Decoder

This is a standard USB 1.X compliant serial NRZI decoder module, which can operate at FS or HS USB data rates.

3.2.1.6 Bit Unstuff Logic

This is a standard USB 1.X compliant serial bit unstuff module, which can operate at FS or HS USB data rates. The bit unstuff logic is a state machine, which strips a stuffed 0 bit from the data stream and detects bit stuff errors.

In FS mode bit stuff errors assert the RXError signal. In HS mode bit stuff errors are used to generate the EOP signal so the RXError signal is not asserted. The bit rate on USB is constant; however the bit rate as presented by the UTMI to the SIE is slightly reduced due to the extraction of inserted 1 bit.

Normally a byte of data is presented on the DataOut bus for every 8 bits received, however after eight stuffed bits are eliminated from the data stream a byte time is skipped in the DataOut stream. Figure 3.5 shows how RXValid is used to skip bytes in the DataOut byte stream.

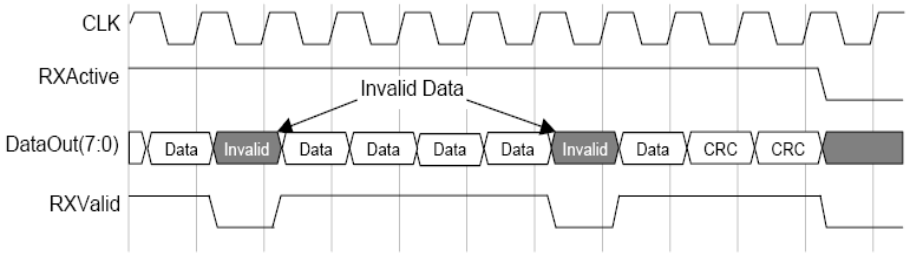


Fig 3.5 Receive Timing for Data with after Unstuffing Bits

3.2.1.7 Receive State Machine

The behavior of the Receive State Machine is described below.

The assertion of Reset will force the Receive State Machine into the *Reset* state. The *Reset* state negates RXActive and RXValid. When the Reset signal is negated the Receive State Machine enters the *RX Wait* state and starts looking for a SYNC pattern on the USB. When a SYNC pattern is detected the state machine will enter the *Strip SYNC* state and assert RXActive.

The length of the received SYNC pattern varies and can be up to 32 bits long. As a result, the state machine may remain in the *Strip SYNC* state for several byte times before capturing the first byte of data and entering the *RX Data* state. After 8 bits of valid serial data is received the state machine enters the *RX Data* state, where the data is loaded into the RX Holding Register on the rising edge of CLK and RXValid is asserted. The SIE must clock the data off the DataOut bus on the next rising edge of CLK.

Stuffed bits are stripped from the data stream. Each time 8 stuffed bits are accumulated the state machine will enter the *RX Data Wait* state, negating RXValid thus skipping a byte time.

When the EOP is detected the state machine will enter the *Strip EOP* state and negate RXActive and RXValid. After the EOP has been stripped the Receive State Machine will reenter the *RX Wait* state and begin looking for the next packet.

If a Receive Error is detected, the *Error* State is entered and RXError is asserted. Then either the *Abort 1* State is entered where RXActive, RXValid, and RXError are negated, or the *Abort 2* State is entered where only RXValid, and RXError are negated. The *Abort 1* State proceeds directly to the *RX Wait* State, while *Abort 2* State proceeds to the *Terminate* State after an Idle bus state is detected on DP and DM. The *Terminate* State proceeds directly to the *RX Wait* State. When the last data byte is clocked off the DataOut bus the SIE must also capture the state of the RXError signal.

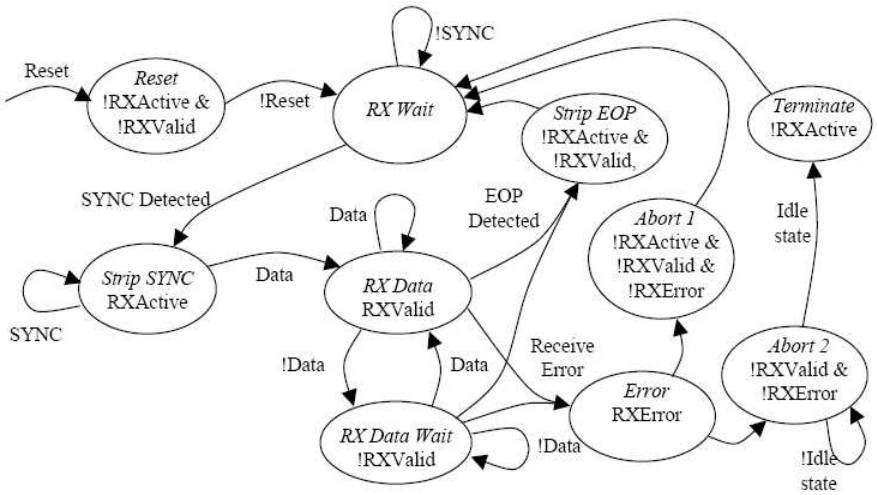


Fig. 3.6 Receive State Machine

3.1.2.8 Receive Error Reporting

If an error is detected by the receiver the receive state machine will enter the *Error* state and assert RXError. It will then transition to the *Abort1* or *Abort2/Terminate* state, terminating the receive operation. Any data received while RXError is asserted should be ignored. The Receive State Machine will then enter the *RX Wait* state and start looking for a valid SYNC pattern.

Possible sources of receive errors.

- Bit stuff error has been detected during a FS receive operation
- Elasticity Buffer over-run
- Elasticity Buffer under-run
- Loss of sync by the DLL
- Alignment error, EOP not on a byte boundary
- Vendor Specific errors

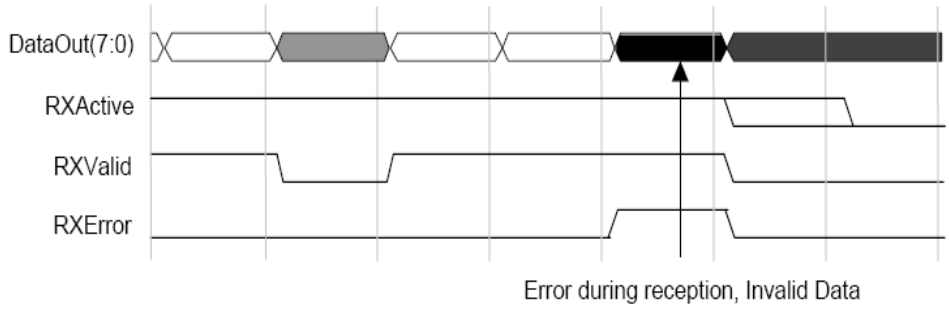


Fig.3.7 Rxerror timing diagram

3.1.2.9 Bit Suff Error Reporting

Suppose an error occurs in the middle of a packet. Depending on the UTM implementation, RXActive may be negated at the same time as RXValid and RXError, or later. For instance, if a bit stuff error occurs in the middle of a FS receive packet it will generate a receive error, however packet data is still on the bus so RXActive will not be negated until an Idle state is detected on the bus.

By definition, a "bit stuff error" during a HS packet is automatically interpreted as an EOP. In this case RXError is not asserted. However if the bit stuff error was a true bit stuff error vs. an EOP-forced bit stuff error, then there will continue to be packet data on the bus and an EOP-forced bit stuff error will occur at the end of the packet.

To prevent this collision and maintain the correct inter-packet delay, the Receive State Machine can be implemented in one of two ways:

1) If the Receive State Machine negates RXActive immediately, it must internally block TXValid to the Transmit State Machine until the USB is back to an idle state and the minimum inter-packet delay, as defined by the USB 2.0 specification, has transpired.
2) The Receive State Machine can hold RXActive asserted until an idle state is detected on the bus. Thus, holding off the SIE until the bus is Idle. In this case the SIE is responsible for timing the inter-packet delay. It is recommended that for HS packets, the internal "squelch" signal of the UTM be used to qualify the negation of RXActive.

3.1.2.10 Rx Shift/Hold Registers

This module is responsible for converting serial data received from the USB to parallel data. This module consists of an 8-bit primary RX Shift Register for serial to parallel conversion and an 8-bit RX Hold Register used to buffer received data bytes and present them to the DataOut bus.

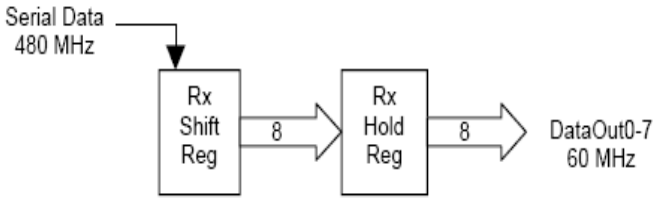


Fig 3.8 RX shift and RX hold registers

3.1.2.11 NRZI Encoder

This is a standard USB 1.X compliant serial NRZI encoder module, which can operate at full-speed or high-speed USB data rates.

3.1.2.12 Bit stuff Logic

In order to ensure adequate signal transitions, bit stuffing is employed when sending data on USB, a zero is inserted after every six consecutive ones in the data stream before the data is NRZI encoded to enforce a transition in the NRZI data stream.

Bit stuffing is enabled beginning with the SYNC Pattern and through the entire transmission. The data "one" that ends the SYNC Pattern is counted as the first one in a sequence. In FS mode bit stuffing by the transmitter is always enforced, without exception.

If required by the bit stuffing rules, a zero bit is inserted even after the last bit before the TXValid signal is negated. After 8 bits are stuffed into the USB data stream TXReady is negated for one byte time to hold up the data stream on the DataIn bus

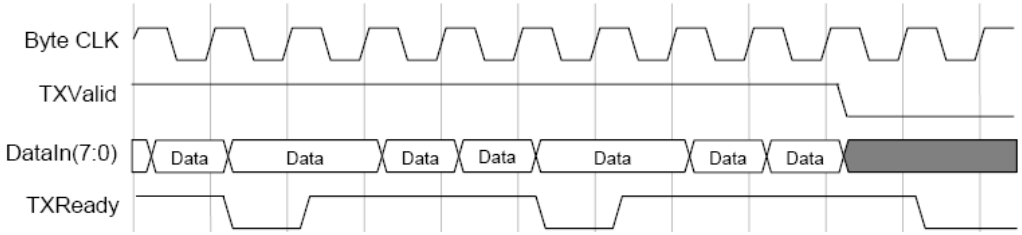


Fig .3.9 Transmit Timing delays due to Bit Stuffing

3.1.2.13 TX Shift/Hold Register

This module is responsible for reading parallel data from the parallel application bus interface upon command and serializing for transmission over USB. This module consists of

an 8-bit primary shift register for parallel/serial conversion and an 8-bit Hold register used to buffer the next data to serialize.

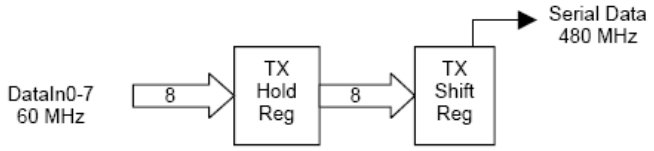


Fig 3.10 TX shift and hold register

3.1.2.14 Transmit State Machine

The behavior of the Transmit State Machine is described below and illustrated in Figure 3.11.

The Reset signal forces the state machine into the *Reset* state which negates TXReady. When Reset is negated the transmit state machine will enter the *TX Wait* state. In the *TX Wait* state, the transmit state machine looks for the assertion of TXValid. When TXValid is detected, the state machine will enter the *Send SYNC* state and begin transmission of the SYNC pattern. When the transmitter is ready for the first byte of the packet (PID), it will enter the *TX Data Load* state, assert TXReady and load the TX Holding Register.

The state machine may enter the *TX Data Wait* state while the SYNC pattern transmission is completed. TXReady is used to throttle transmit data. The state machine will remain in the *TX Data Wait* state until the TX Data Holding register is available for more data. In the *TX Data Load* state, the state machine loads the Transmit Holding register. The state machine will remain in the *TX Data Load* state as long as the transmit state machine can empty the TX Holding Register before the next rising edge of CLK.

When TXValid is negated the transmit state machine enters the *Send EOP* state where it sends the EOP. While the EOP is being transmitted TXReady is negated and the state machine will remain in the *Send EOP* state. After the EOP is transmitted the Transmit State Machine returns to the *TX Wait* state, looking for more work

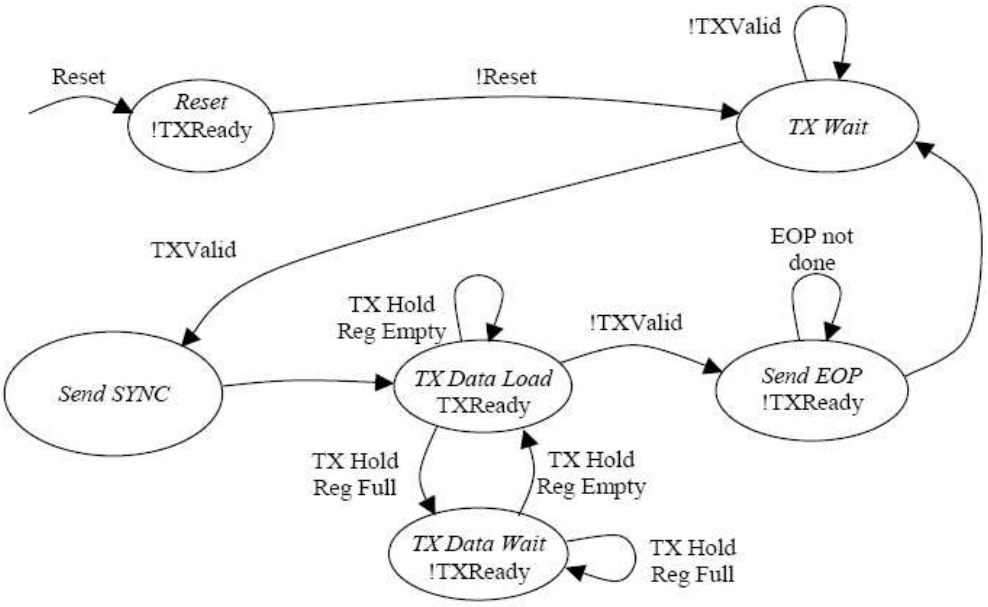


Fig 3.11 Transmit State Machine

3.1.2.15 Transmit Error Reporting

In FS mode if the data transmitter is unable, because of problems such as a buffer under run condition, to transmit the identical amount of data as was in the original data packet, it must terminate the transaction by generating a bit stuffing violation, followed by an EOP.

To accomplish this SIE must switch the OpMode to "Disable Bit Stuffing and NRZI Encoding" and load 0's into the DataIn lines for at least one byte time before negating TXValid. In HS mode, if an error condition occurs during transmission, the current transmit stream must be terminated by the transmission of a complemented version of the CRC, followed by an EOP. In this case the SIE will be responsible for presenting the complemented CRC to the DataIn lines before negating TXValid.

In either mode the negation of TXValid will cause the UTM to terminate the packet with the appropriate EOP.

3.1.2.16 USB Full Speed XCVR

The FS receiver includes the logic necessary to send and receive the FS data on USB, as well as supports the Reset, Suspend and Resume detection functions [15].

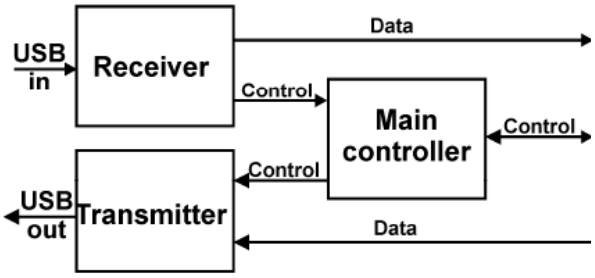


Fig. 3.12 USB core block diagram

3.1.2.17 Transmit Driver

When enabled, data from the transmit data path will be driven on to the DP/DM signal lines. The HS transmit driver is active only when Transmit is asserted, the XcvrSelect input is in HS transceiver enabled mode and the Transmit State Machine has data to send.

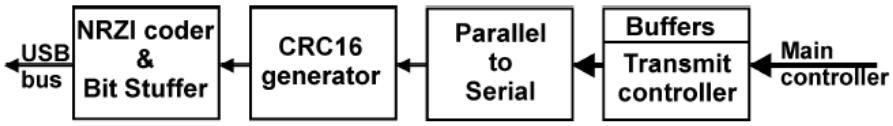


Fig. 3.13 transmitter block diagram

3.1.2.18 Receive Buffer

When enabled, received HS data will be multiplexed through the receive data path to the receive shift and hold registers. The USB 2.0 receive buffer is active only when the XcvrSelect input is in HS transceiver enabled mode.

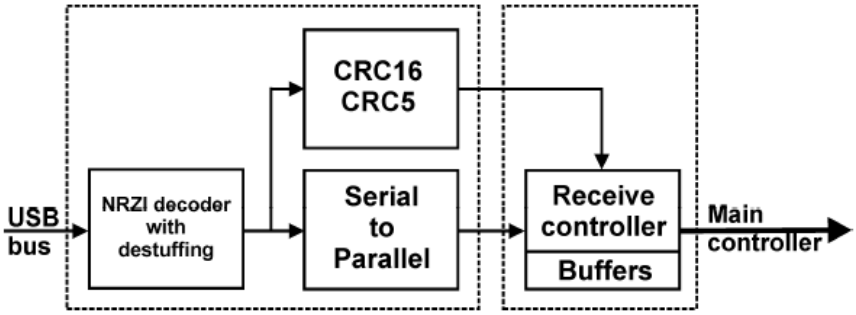


Fig. 3.14 transceiver block diagram

3.1.3 Other Components of Transceiver

3.1.3.1 Transmission Envelope Detector

The quiescent state of a HS link is for the DP and DM lines to be balanced near ground with the differential receivers listening for Start of Packet. The Transmission Envelope Detector is evoked to prevent spurious signals (e.g., noise, crosstalk, or oscillation) from triggering the Start of Packet detection process (to "squelch" the receiver). This envelope detector is used to disable or "squelch" the HS receiver when the amplitude of the differential signal falls below the minimum required level for data reception, preventing noise from propagating through the receive logic.

3.1.3.2 Full-Speed Indicator Control

In full-speed mode, a 1.5K Ohm pull-up on the DP signal line is used to indicate to an upstream port that a full-speed device is attached. In high-speed mode this resistor would introduce about a 50 mV error into the received signal so it must be removed. When it is enabled, the HS XCVR contains the circuitry to electrically detach the full-speed indicator resistor from the DP signal line.

3.3 SYSTEM INTERFACE SIGNALS

Entity diagram for 16- bit interface is as shown below:

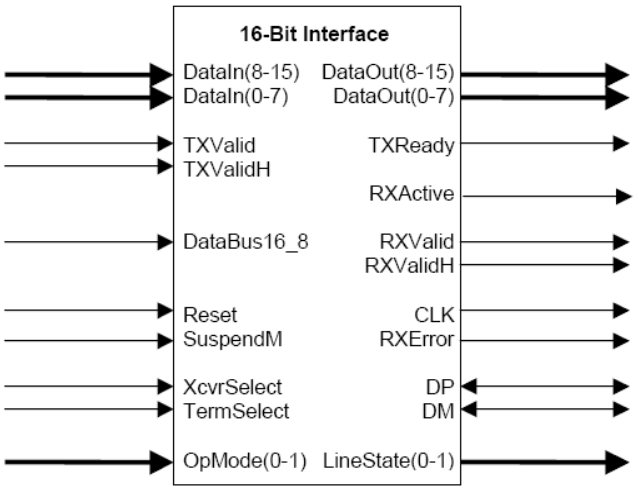


Fig.3.15 Entity diagram for 16 bit interface

System interface signals can be defined as:

3.3.1 CLK

Nominal CLK accuracy is ±500ppm for frequency, and 50±5% duty cycle. No transitions of CLK should occur until it is "usable", where usable is defined as a frequency accuracy of ±10%, and a duty cycle accuracy of 50±10%. Conceptually, there is a "CLKUsable" signal, internal to the UTM, which blocks any transitions of CLK until it is "usable". This "CLKUsable" signal is also used to switch the LineState output between CLK synchronized and combinatorial signaling. See section 5.22.2.3 for further discussion of CLK

3.3.1.1 Options

There are 3 possible implementations for a UTMI device: HS/FS, FS Only, or LS Only. The HS/FS version has 4 interface options: 16-bit unidirectional, 8-bit unidirectional, 16-bit bidirectional/8-bit unidirectional and 8-bit bi-directional. In each case, when a 16-bit option is selected CLK is at 30 MHz, and when an 8-bit option is selected **CLK** is at 60 MHz. The 16-bit bidirectional/8-bit unidirectional uses the "DataBus16_8" signal to switch between them. This signal also switches the CLK frequency.

The FS Only, or LS Only implementations only support 48 MHz and 6 MHz clocks, respectively, and always use 8 bit interfaces (either 8-bit unidirectional or 8-bit bi-directional).

3.3.2 XcvrSelect

XcvrSelect controls a number of transceiver related elements, for instance.

- Selects the receiver (source for the Mux block) in the receive data path.
- It is used as a gating term for enabling the respective HS or FS Transmit Driver.
- Switch internal UTM clocks to shared logic.

3.3.3 TermSelect

TermSelect controls a number of termination related elements, for instance.

- In HS mode the FS Driver is forced to assert an SE0 on the USB, providing the 50 Ohm termination to ground and generating the HS Idle state on the bus.
- In FS Mode TermSelect enables the 1.5K pull-up on to the DP signal to generate the FS Idle state on the bus.

3.3.4 LineState

The LineState signals are used by the SIE for detecting reset, speed signaling, packet timing, and to transition from one behavior to another. While data packets are being transmitted or received on the USB the LineState signals may toggle randomly between the 'J' and 'K' states in FS, and remain in the ‘J’ state in HS. The SIE should ignore these transitions.

3.3.4.1 Synchronization

To minimize unwanted transitions to the SIE during normal operation, the LineState is internally synchronized with CLK. When synchronized, the setup and hold timing of LineState is identical to DataOut. The exception to this is when CLK is not "usable". If CLK is not "usable" then the LineState signals are not synchronized, but driven with combinatorial logic directly from the DP and DM signal lines. The UTM must multiplex between combinatorial and synchronous LineState output depending on whether CLK is "usable". For an addition method of minimizing LineState transitions in HS mode.

3.3.4.2 Signaling Levels

The voltage thresholds that the LineState signals use for comparison on DP and DM depend on the state of XcvrSelect. LineState uses HS thresholds when the HS transceiver is enabled (XcvrSelect = 0) and FS thresholds when the FS transceiver is enabled (XcvrSelect = 1). FS Only and LS Only implementations always use FS thresholds.

There is no concept of variable, single ended thresholds in the USB 2.0 specification. The assumption was that the HS receiver would be used to detect a Chirp K or J, where the

output of the HS receiver is always qualified with the "Squelch" signal. If Squelch = 1 then the output of the HS receiver is meaningless. In the macrocell, as an alternative to using variable thresholds for the single ended receivers the following approach to encoding the LineState outputs can be used.

3.3.4.3 Minimizing Transitions

In HS mode, 3 ms of no USB activity (Idle state) signals a reset. The SIE monitors LineState for Idle state. If in HS mode, LineState is simply the output of the HS Differential receiver then LineState will toggle randomly while packets are on the USB. To minimize transitions on LineState while in HS mode the presence of Squelch can be used to force a LineState to a J State.

This scheme allows LineState to indicate a J State whenever a packet is on the USB, thus satisfying the requirement that a LineState transition occurs when there is activity on the USB, while minimizing the number of LineState transitions while there is data on the bus.

3.3.4.4 Bus Packet Timing

LineState must be used by the SIE for the precise timing of packet data on the DP/DM signal lines. The SIE uses LineState transitions to identify the beginning and end of receive or transmit packets on the bus. Due to internal UTM buffering and pipeline delays (which are implementation dependent), the receive (RXActive) and transmit (TXValid) control signals provide only a coarse indication of the actual activity on the bus. LineState represents bus activity within 2 or 3 CLK times of the actual events on the bus.

HS Mode: When XcvrSelect and TermSelect are in HS mode, the LineState transition from the Idle state (SE0) to a non-Idle state (J) marks the beginning of a packet on the bus. The LineState transition from a non-Idle state (J) to the Idle state (SE0) marks the end of a packet on the bus.

FS Mode: When XcvrSelect and TermSelect are in FS mode, the LineState transition from the J State (Idle) to a K State marks the beginning of a packet on the bus. The SIE must then wait for the end of the packet. The LineState transition from the SE0 to the J-State marks the end of a FS packet on the bus.

3.3.5 OpMode

When a device generates resume signaling to the host, it switches the OpMode to "Disable Bit Stuffing and NRZI Encoding", asserts TXValid, and presents the data on the

DataOut bus. The assertion of OpMode to “Normal” mode at the end of the 1 ms signaling period should occur until after the maximum *TX End Delay* (TXValid has been de-asserted for at least 40 bit times or in FS mode 160 CLKs).

If OpMode switched to “Normal” mode before the maximum *TX End Delay* completes, then there is the possibility that the last data still pending in the UTM will be NRZI encoded and bit stuffed (in case 6 1's occur), resulting in K and J transitions on the DP/DM signal lines at the end of resume from the device. At this time the downstream facing port will also be propagating back the K state (detected device resume) onto all enabled down stream ports. This creates bus conflict on DP/DM.

3.3.6 USB Interface Signals

DP: USB data pin data+

DN: USB data pin data-

3.4 DATA INTERFACE SIGNALS

3.4.1 Transmit Data Interface Signals

3.4.1.1 DataIn0-7

It is an input signal .It is an 8-bit parallel USB data input bus. When DataBus16_8 = 1 this bus transfers the low byte of 16-bit transmit data. When DataBus16_8 = 0 all transmit data is transferred over this bus.

3.4.1.2 DataIn8-15

It is an input signal .An 8-bit parallel USB data input bus that transfers the high byte of 16-bit transmit data. These signals are only valid when DataBus16_8 = 1.

3.4.1.3 TXValid

It is an input high signal (Transmit Valid signal). It indicates that the DataIn bus is valid. The assertion of Transmit Valid initiates SYNC on the USB. The negation of Transmit

Valid initiates EOP on the USB. In HS (XcvrSelect = 0) mode, the SYNC pattern must be asserted on the USB between 8 and 16 bit times after the assertion of TXValid is detected by the Transmit State Machine. In FS (XcvrSelect = 1), FS Only, or LS Only modes, the SYNC pattern must be asserted on the USB no less than 1 CLK and no more than 5 10 CLKs3 after the assertion of TXValid is detected by the Transmit State Machine.

3.4.1.4 TXValidH

It is an Input high signal (Transmit Valid High). When DataBus16_8 = 1, this signal indicates that the DataIn (8-15) bus contains valid transmit data. This signal is ignored when DataBus16_8 = 0. This signal is not provided in 8-Bit transceiver implementations.

3.4.1.5 TXReady

It is an Output High Transmit signal .it indicates data to be transmitted is ready. If TXValid is asserted, the SIE must always have data available for clocking in to the TX Holding Register on the rising edge of CLK. If TXValid is TRUE and TXReady is asserted at the rising edge of CLK, the UTM will load the data on the DataIn bus into the TX Holding Register on the next rising edge of CLK, at that time, SIE should immediately present the data for next transfer on the DataIn bus. If TXValid is asserted and TXReady is negated, the SIE must hold the previously asserted data on the DataIn bus. From the time TXValid is negated, TXReady is a don't care for the SIE.

3.4.2 Receive Data Interface Signals

3.4.2.1 DataOut0-7

It is an 8-bit parallel USB data output bus. When DataBus16_8 = 1 this bus transfers the low byte of 16-bit receive data. When DataBus16_8 = 0 all receive data is transferred over this bus.

3.4.2.2 DataOut8-15

It is an 8-bit parallel USB data output bus that transfers the high byte of 16-bit receives data. These signals are only valid when DataBus16_8 = 1.

3.4.2.3 RXValid

It is an Output High Receive Data Valid signal. It indicates that the DataOut bus has valid data. The Receive Data Holding Register is full and ready to be unloaded. The SIE is expected to latch the DataOut bus on the clock edge.

3.4.2.4 RXValidH

It is an output High Receive Data Valid signal. When DataBus16_8 = 1 this signal indicates that the DataOut (8-15) bus is presenting valid receive data. This signal is ignored when DataBus16_8 = 0. This signal is not provided in 8-Bit transceiver implementations.

3.4.2.5 RXActive

It is an output High Receive signal. It indicates that the receive state machine has detected SYNC and is active. RXActive is negated after a Bit Stuff Error or an EOP is detected. In HS mode (XcvrSelect = 0), RXActive must be negated no less than 3 and no more that 8 CLKs after an Idle state is detected on the USB. And RXActive must be negated for at least 1 CLK between consecutive received packets.

In FS (XcvrSelect = 1), FS Only, or LS Only modes, RXActive must be negated no more than 2 CLKs after a FS Idle state4 is detected on the USB. And RXActive must be negated for at least 4 CLKs between consecutive received packets.

3.4.2.6 RXError

It is an output High Receive signal. 0 Indicates no error. 1 Indicates that a receive error has been detected. This output is clocked with the same timing as the DataOut lines and can occur at anytime during a transfer. If asserted, it will force the negation of RXValid on the next rising edge of CLK.

3.5 Other Functions

3.5.1 Linestate

The LineState signals are used for many functions. The LineState signals reflect the current state of the DP/DM signal lines. The thresholds used by the LineState to determine the state of DP/DM depend on the value of XcvrSelect. LineState uses HS thresholds when the HS transceiver is enabled (XcvrSelect = 0) and FS thresholds when the FS transceiver is enabled (XcvrSelect = 1).

FS Only and LS Only implementations always use FS thresholds. The following sections make a distinction between "soft" SE0 and "driven" SE0. Soft SE0 is the bus

signaling that results from the DP and DM signal lines being pulled down exclusively by the 15K pull down resistors (Rpd).

Driven SE0 is the result of generating a SE0 condition by enabling the FS Transmitter. In this case the DP and DM signal lines are being pulled down by the 45 Ohm serial (Rs) termination resistors.

3.5.2 SE0 handling

For low-speed and full-speed operation, Idle is a J state on the bus and SE0 is used as part of the EOP or to indicate reset. When asserted in an EOP, SE0 is never asserted on the bus for more than 2 low-speed bit times (1.3 μs). The assertion of SE0 for more than 2.5 μs is interpreted as a reset by device operating in low-speed or full-speed. For high-speed operation, Idle is an SE0 state on the bus. SE0 is also used to reset a high-speed device.

A high-speed device cannot use the 2.5 μs assertion of SE0 (as defined for FS operation) to indicate reset since the bus is often in this state between packets. If no bus activity (Idle) is detected for more than 3 ms. A high-speed device must determine whether the downstream port is signaling a suspend or a reset. If a reset is signaled the high-speed device will then initiate the HS Detection handshake protocol.

3.5.3 Suspend Detection

If a HS device detects SE0 asserted on the bus for more than 3 ms. (T1) its UTM is placed in FS mode(XcvrSelect and TermSelect = 1). This enables the FS pull-up on the DP line, asserting a continuous FS 'J' state on the bus.

The SIE must then check the LineState signals for an 'J' State condition. If 'J' State condition is asserted at time T2, then the upstream port is asserting a Soft SE0 and the USB is in a 'J' state indicating a suspend condition. By time T4 the device must be fully suspended.

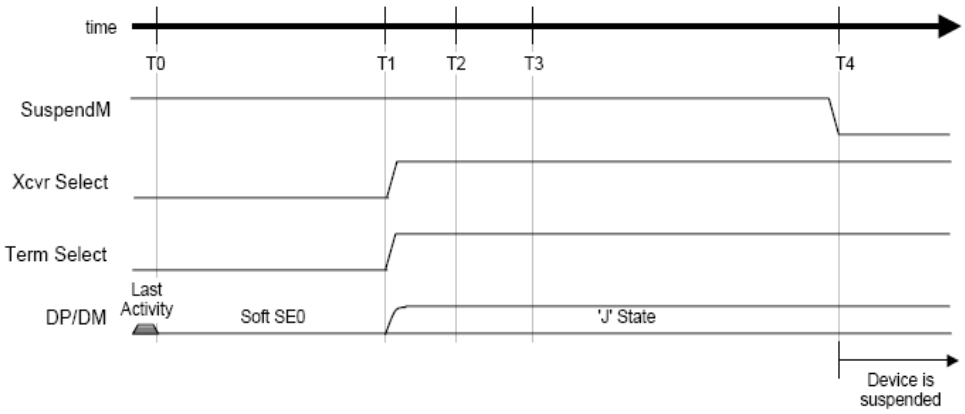


Fig. 3.16 Suspend Timing Behavior (HS Mode)

3.5.4 Reset Detection

If a device in HS mode detects bus inactivity for more than 3 ms. (T1) its UTM is placed in FS mode (XcvrSelect = 1 and TermSelect = 1). This enables the FS pull-up on the DP line to attempt to assert a continuous FS 'J' state on the bus. The SIE must then check the LineState signals for the SE0 condition. If SE0 is asserted at time T2, then the upstream port is forcing the reset state to the device (i.e. Driven SE0). The device will then initiate the HS Detection Handshake protocol.

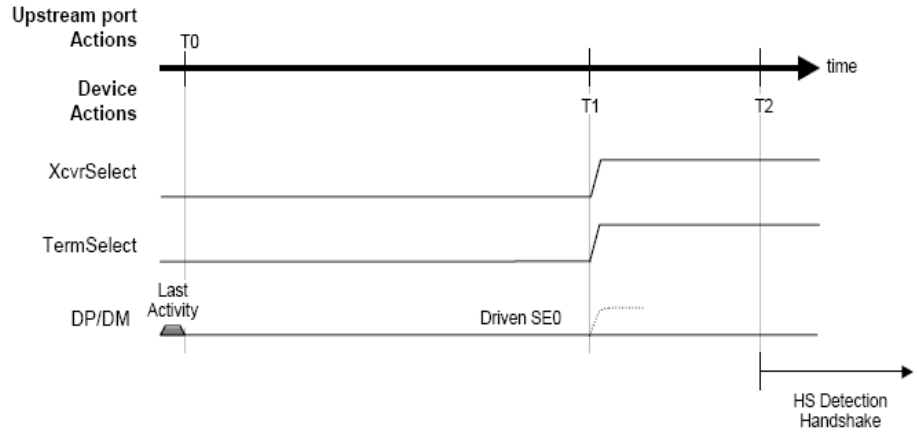


Fig 3.17 Reset Timing Behavior (HS Mode)

3.5.5 HS Detection Handshake

The High-Speed Detection Handshake process is entered from one of three states: suspend active FS or active HS. The downstream port asserting an SE0 state on the bus initiates the HS Detection Handshake. Depending on the initial state, an SE0 condition can be asserted from 0 to 4 ms before initiating the HS Detection Handshake.

There are three ways in which a device may enter the HS Handshake Detection process:

1) If the device is suspended and it detects an SE0 state on the bus it may immediately enter the HS handshake detection process.
2) If the device is in FS mode and an SE0 state is detected for more than 2.5 μs. It may enter the HS handshake detection process.
3) If the device is in HS mode and an SE0 state is detected for more than 3.0 ms. it may enter the HS handshake detection process. In HS mode, a device must first determine whether the SE0 state is signaling a suspend or a reset condition. To do this the device reverts to FS mode by placing XcvrSelect and TermSelect into FS mode. The device must not wait more than 3.125 ms before the reversion to FS mode.

After reverting to FS mode, no less than 100 μs and no more than 875 μs. Later the SIE must check the LineState signals. If a J state is detected the device will enter a suspend state. If an SE0 state is detected, then the device will enter the HS Handshake detection process.

In each case, the assertion of the SE0 state on the bus initiates the reset interval. The minimum reset interval is 10 ms. Depending on the previous mode that the bus was in, the delay between the initial assertion of the SE0 state (HS Reset T0) and entering the HS Handshake detection process and can be from 0 to 4 ms.

3.6 FS Operations

3.6.1 FS Start of Packet

The assertion of the TXValid signal initiates packet transmission. With the bus initially in the Idle state (‘J’state), a SYNC pattern ("KJKJKJKK") in its NRZI encoding is generated on the USB. To generate this pattern the SYNC data pattern (0x80) is forced into the Transmit Data Shift Register by the Transmit State Machine. TXValid will remain asserted if valid packet data bytes are available to be loaded into the Transmit Data Holding Register.

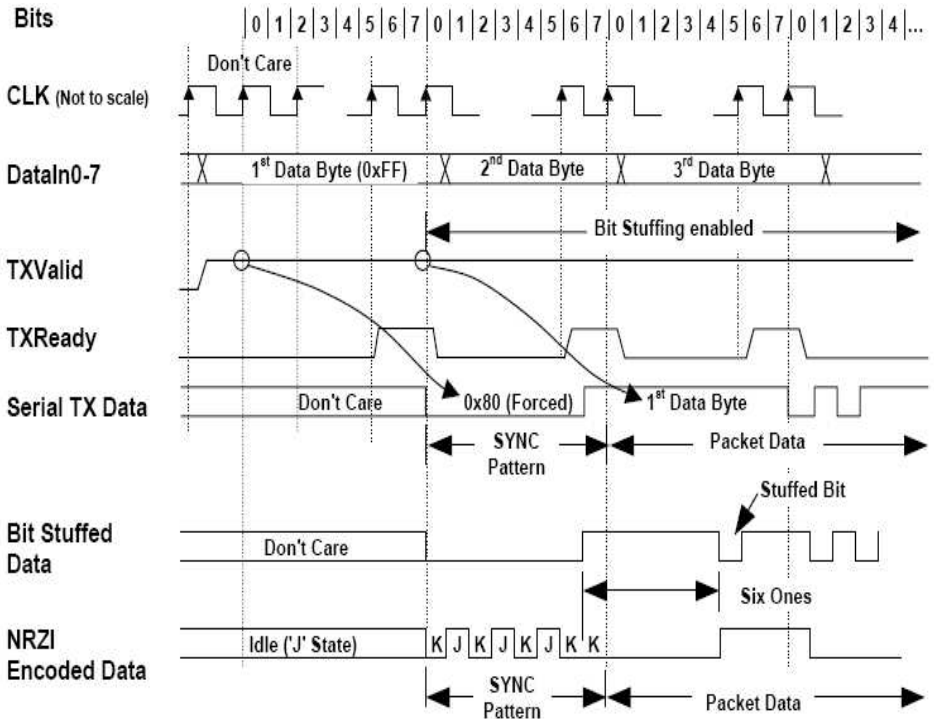


Fig.3.18 Data Encoding Sequence: FS SYNC

3.6.2 FS End of Packet

In FS mode, a single ended zero (SE0) state is used to indicate EOP and a 'J' state indicates Idle. The SE0 state is asserted for 2 bit times then a 'J' state is asserted for 1 bit time. The bus will be held in the idle state by the FS pull-up on the bus.

Negating the TXValid signal initiates the FS EOP process; bit stuffing will cease, the bit stuff state machine will be reset, and the FS EOP (two bit times of SE0 followed by a single 'J' bit) will be asserted on the bus. TXReady is negated after TXValid is detected false and cannot be reasserted (dashed line) until after the EOP pattern and 'J' state bit are transmitted. The delay between the assertion of TXValid and the first assertion of TXReady is UTM implementation dependent.

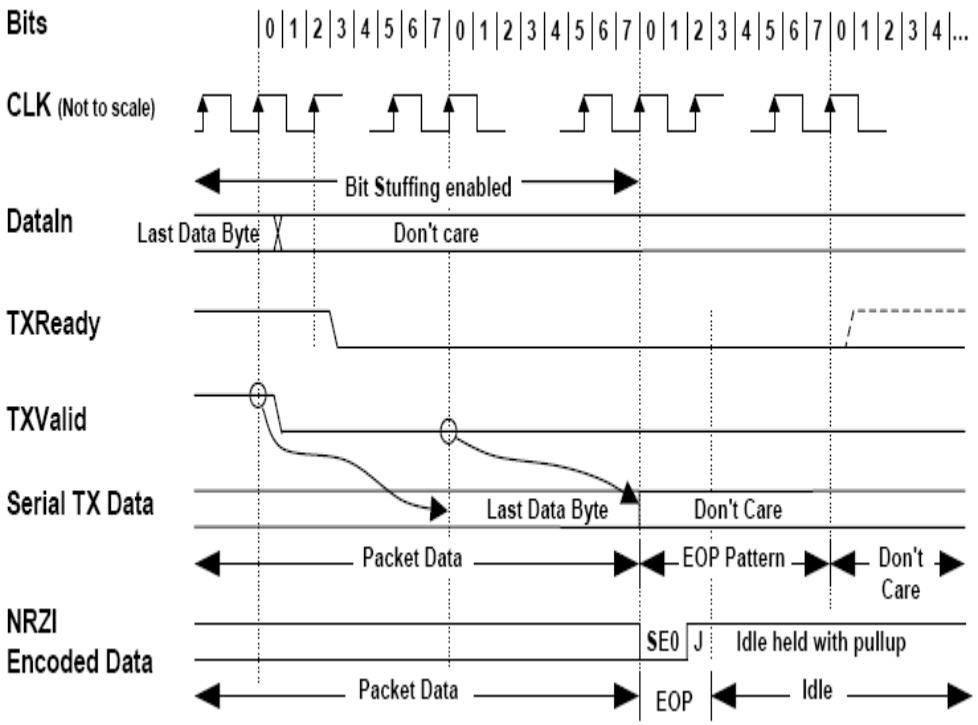


Fig 3.19 Data Encoding Sequence: FS EOP

3.7 HS Operation

3.7.1 HS Start of Packet

The assertion of TXValid will initiate the generation of the HS SYNC pattern on the USB by the Transmit State Machine. A SYNC pattern is the 32-bit binary string "KJKJK…JKJKJKK", in its NRZI encoding. To generate this pattern four bytes of SYNC data pattern (0x00, 0x00, 0x00, 0x80) are forced into the Transmit Shift Register by the Transmit State Machine. As long as TXValid is asserted the Transmit State Machine will assume that valid packet data is available on the DataIn bus to be loaded into the Transmit Holding Register.

Figure 3.16 demonstrates how the detection of a zero to one transition of TXValid forces the transmit state machine of the transceiver to send a 32-bit Sync pattern then begin transmission with bit stuffing enabled.

TXReady is negated to indicate to the SIE that the data on the DataIn bus has not been loaded into the Transmit Data Holding Register.

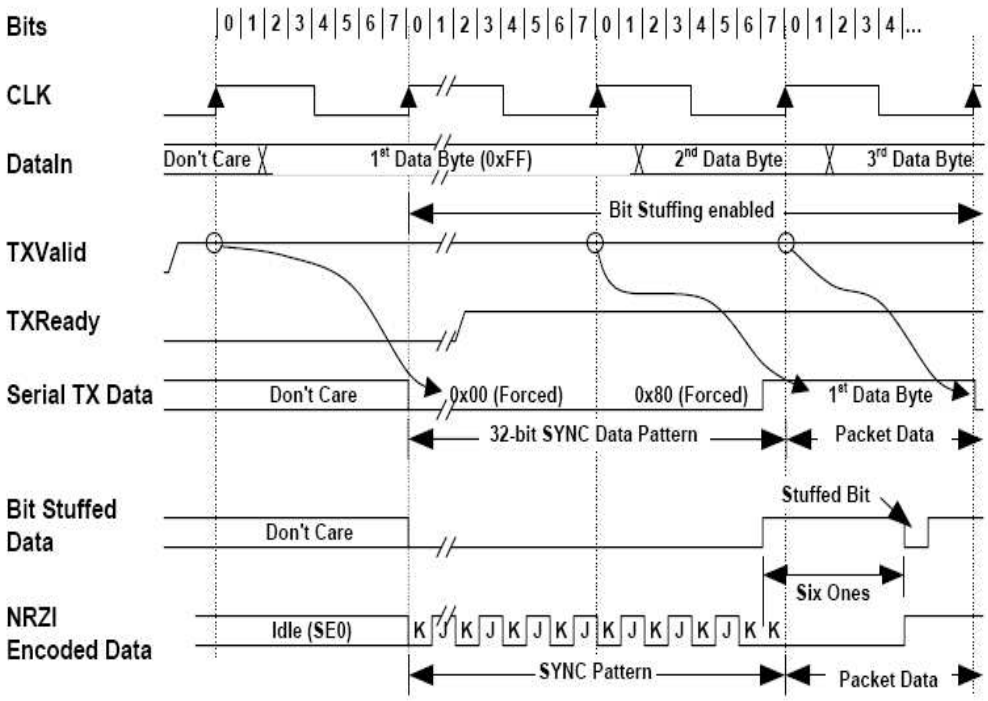


Fig. 3.20 data encoding sequence: HS sync

3.7.2 HS End of Packet

The negation of TXValid will initiate the EOP sequence in the Transmit State Machine. If required by the bit stuffing rules, a zero bit is inserted even after the last data bit.TXReady is negated for at least one byte time while the EOP pattern is being transmitted.

TXReady is negated after TXValid is detected false and cannot be reasserted (dashed line) until after the EOP pattern is transmitted. The delay between the assertion of TXValid and the first assertion of TXReady is UTM implementation dependent.

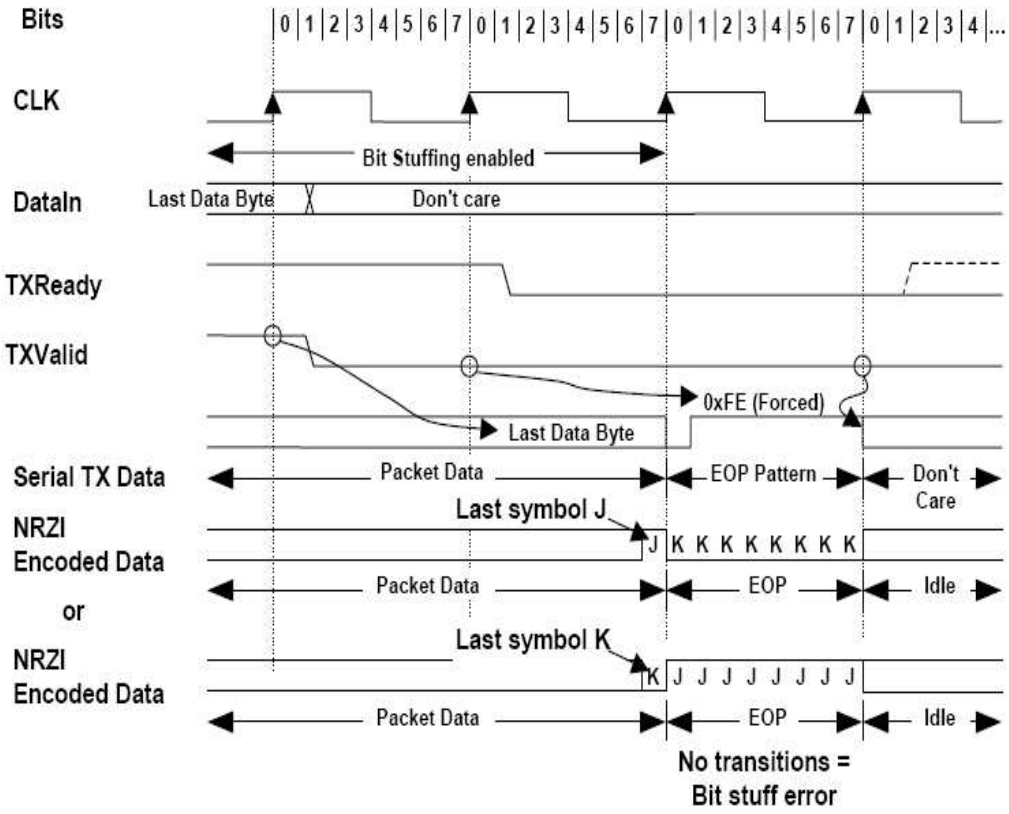


Fig. 3.21 data encoding sequence: HS EOP

3.8 Inter-Packet Delay

3.8.1 HS Inter-packet delay for a receive followed by a transmit

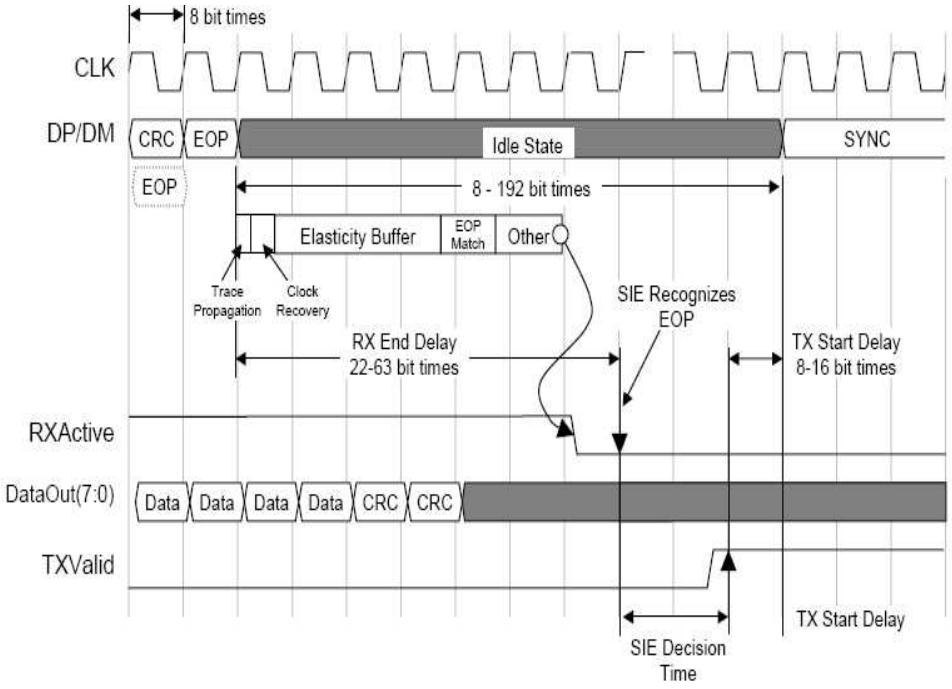


Fig 3.22 HS receive to transmit inter-packet delay

3.8.1.1 Receive End Delay Components

The *Receive End Delay* is the time between the beginning of the Idle state on the bus after a receive packet and the CLK edge that the SIE detects the negation of RXActive.

Bit Times	Reason
0-7	Data Synchronization
0-2	Trace Propagation
4	Clock Recovery
0-32	Elasticity Buffer purge
8	EOP Decode
10	Other miscellaneous
22-63	Total Receive End Delay

The Data Synchronization Offset is delay between the end of EOP and the synchronization with CLK, representing an additional 0 to 7-bit time delay.

Receive End Delay budget discussed in the table above is a representative example. The actual contribution of each component of the Receive End Delay is implementation specific. However, in all cases a device must meet the constraints of the Total Receive End

Delay (22-63 bit times). The exception is if RXError is asserted and RXActive is negated immediately (RX *Abort 1* state).

The *Receive End Delay* is the time between the beginning of the Idle state on the bus after a receive packet and the CLK edge that the SIE detects the negation of RXActive.

The *Transmit Start Delay* is the time between the CLK edge that the UTM transmit state machine detects the assertion of TXValid and the assertion of the SYNC pattern on the bus.

The *SIE Decision Time* is the delay between the SIE detecting the negation of RXActive and the UTM detecting the assertion of TXValid.

If the *Receive End Delay* is 3 to 8 CLK times and the *Transmit Start Delay* is 1 to 2 CLKs, then the *SIE Decision Time* must be between 0 and 14 CLKs to meet the requirements of the USB spec.

The worst case *SIE Decision Time* assumes that the elasticity buffer is full, the EOP just missed synchronizing with CLK, maximum Trace Propagation, and 32 bits are in the Elasticity Buffer (8 CLKs), and it takes 2 CLKs between the assertion of TXValid and the beginning of the SYNC pattern.

The *SIEDecision Time* must not take more than 14 CLKs to ensure that it does not exceed the inter-packet delay maximum of 192 bit times (24 CLKs).

In this example, the best case inter-packet delay assumes that the elasticity buffer is empty, the EOP is perfectly synchronized with CLK, 0 Trace Propagation, and 0 bits are in the Elasticity Buffer (4 CLKs), it takes 1 CLK between the assertion of TXValid and the beginning of the SYNC pattern, and the *SIE Decision Time* takes 1 CLK. In this case there will be a minimum of 38 bit times (5 CLKs) between EOP

and SYNC. This timing exceeds the 8 bit time minimum inter-packet delay required by the spec. UTM implementations that minimize the *Receive End Delay* will provide better performance and more closely meet the minimum 8 bit time delay.

3.8.2 HS Inter-packet delay for a Transmit followed by a Receive

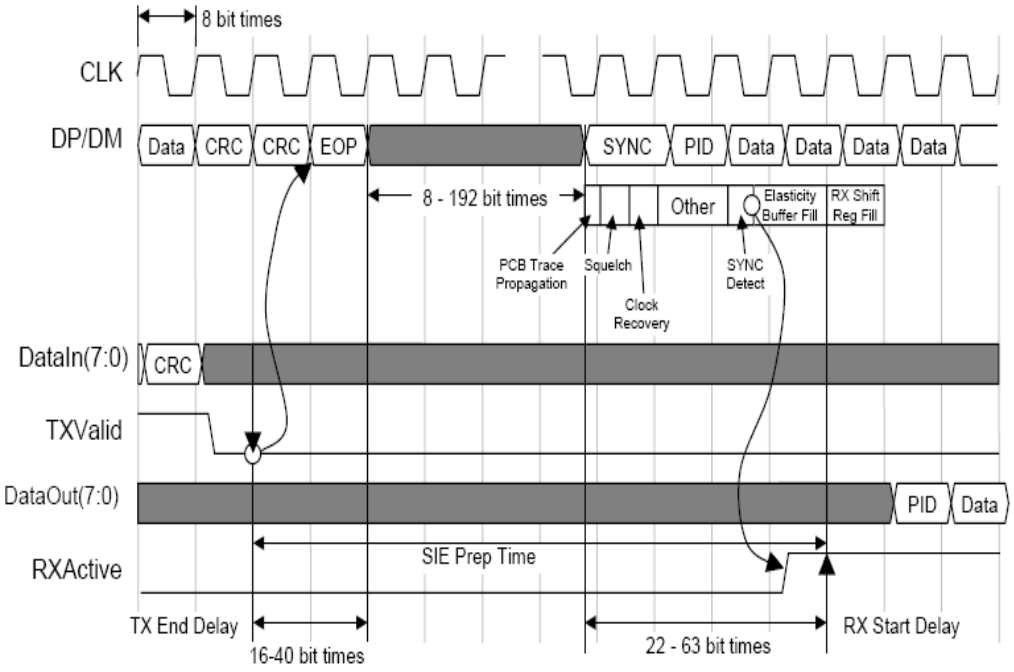


Fig .3 23 HS Transmit to Receive inter-packet delay

3.8.3 HS Inter-packet delay for a receive followed by a receive

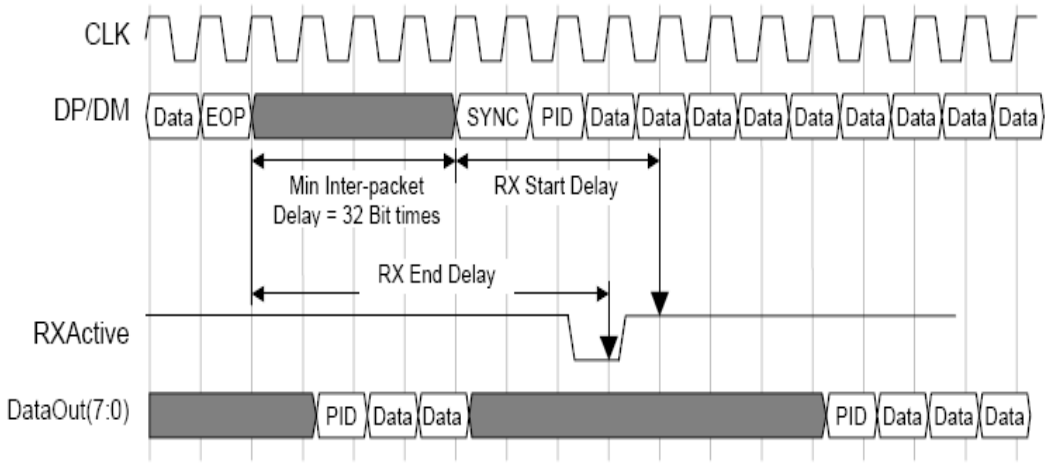


Fig 3.24 HS Inter-packet delay for a receive followed by a receive

3.8.4 FS Inter-packet delay for a Receive followed by a Transmit

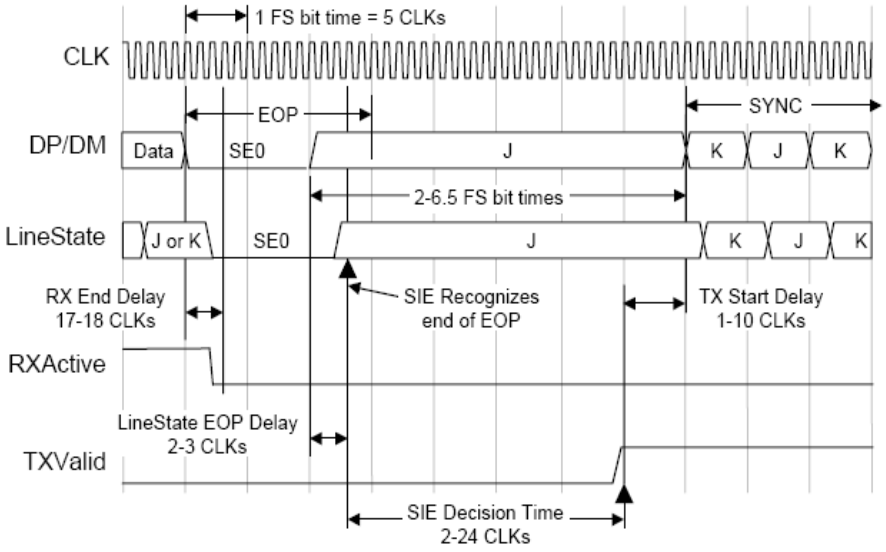


Fig 3.25 FS Inter-packet delay for a Receive followed by a Transmit

LineState delay	2-3 CLKs
TXValid to first K of SYNC	1-10 CLKs
Macrocell overhead	3-13 CLKs
Spec inter-packet Gap time	32 CLKs (6.5 bit times)

For timing the inter-packet delay the SIE must utilize LineState to determine the EOP transition from SE0 to the J-State. RXActive must also be negated before TXValid can be asserted because the SIE must parse the received data to determine what data to return with the following transmit operation. FS SIE Decision Time must be between 7-19 CLKs10 to ensure that 6.5 bit times FS inter-packet gap is met.

3.8.5 FS Inter-packet delay for a Transmit followed by a Receive

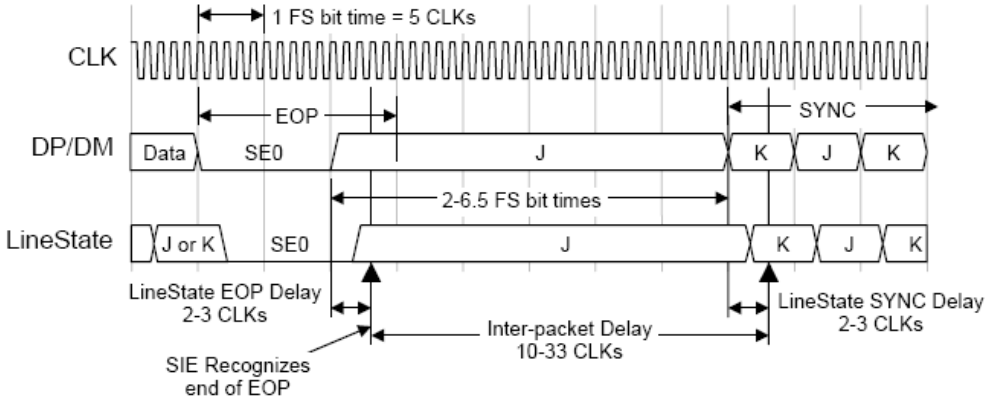


Fig. 3.26 FS Inter-packet delay for a Transmit followed by a Receive

4.1 USB Physical IP Test Environment

USB Properties can be tested with the help of host controller and device controller test-environments. These environments are being delivered as part of controller IP delivery.

4.1.1 Host Controller test environment

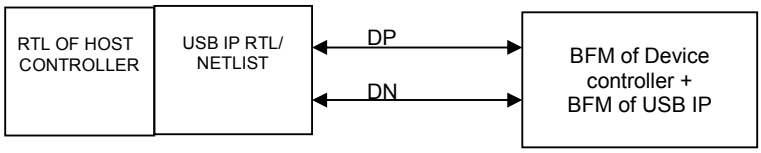


Fig. 4.1 Block Diagram of Host Controller test environment

RTL of Host controller is coming from Synopsys. This block generates all the different USB protocol signals and interacts with UTMI interface of USB2.0 IP.

USB IP RTL is being developed in ST and is being replaced by behavioral model of USB2.0 coming from Synopsys. This RTL/netlist is under test. It generates necessary UTMI signals as well as final data on DP DN line.

BFM of device controller is a bus functional model to interact using DP/ DN. This is placed to complete the system. This model too is coming from Synopsys. It contains approx. 378 test cases. All the test cases can be run in parallel.

4.1.2 Device Controller Environment

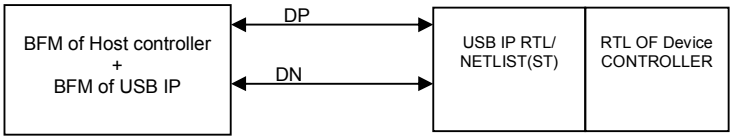


Fig. 4.2 Block Diagram of Device Controller Environment

BFM of host controller is a bus functional model to interact using DP DN. This is placed to complete the system. This model is coming from Synopsys.

USB IP RTL is being developed in ST and is being replaced by behavioral model of USB2.0 coming from Synopsys. This RTL/netlist is under test. It collects information from DP DN line and generates necessary UTMI signals.

RTL of Host controller is coming from Synopsys. This block generates all the different USB protocol signals and interacts with UTMI interface of USB2.0 IP.
It contains approx. 104 test cases. All the test cases can be run in parallel.

4.1.3 USB2PHY Test Suite

Fig. 4.3 Block diagram of ST device Test Suite

USB2PHY test Suite is ST's own work environment. The purpose of this test suite is to verify USB IP RTL/Netlist considering all test conditions including corner cases too.

The advantage of this environment is that the USB IP RTL/Netlist can be verified precisely even for small changes made in the inputs. The purpose of host controller and device controller is served by test-benches.

It consists of 108 test-cases which are to be verified. More test conditions are to be enlisted from the available documents for USB_2.0 specifications and UTMI_1.05 specifications.

4.2 ST device test suite

ST device test suite can be broadly divide into four categories according to the the speed mode in which the conditions are to be checked:

1. HS only mode
2. HSINFS mode
3. FS only mode
4. LS only mode

Different test conditions are enlisted for each mode and tests are generated to verify those conditions.

4.2.1 Test Conditions for Test Cases:

4.2.1.2 Transmission tests

These tests are aimed at testing the mechanisms of stuffing, differential encoding, NRZI signaling and the sequence of operations during transmission under normal conditions and conditions that will lead to corner cases of their implementation.

Various test cases are generated for verifying the behavior of TXValid/TXValidH, TXReady, Linestate, Txcvr select, and Opmode termselect signals.

Test cases also verify all possible combination of data transmission and conditions which generate errors for transmission. The behavior of PHY for different operational modes can also be verified. Delays for the start and end of packets and signals under various conditions can also be verified.

4.2.1.2 Reception tests

These tests are aimed at testing the mechanisms of decoding, data recovery, unstuff and the receive state machine under normal conditions and conditions that will lead to corner cases of their implementation.

Test mentioned in this category verify the status of various receive signals like RXActive, RXValid/RXValidH, RXError linestate, txcvrselect, termselect, opmode, etc. SOP (start of packet) and EOP (end of packet) conditions can also be verified.

Tests cases also exist for verification of possible data combinations, conditions which lead to receive errors, delays at start and end of packet, interpacket delays and other corner cases.

4.3 ST device test environment

4.3.1. File structure for tests

There are basically three main files:

1 **Test Bench**. This file creates the conditions which are to be applied to design under test. DUT is instantiated in this test bench

2. **Procedure file**: This file contains the procedures that are to be used in test bench to give input and get output from test bench.

3. **Text file**: The input is given through a text file to the test bench. This file serves the purpose of data input file for test bench.

4.4 MODIFICATIONS DONE IN PACKAGE MODIFIED:

4.4.1Procedure LINESTATE_CHECK

This procedure has been changed. Delta delays (wait for 0 ns) has been added before the linestate output and the DP/DN status is compared .

This is done because the comparison of the above two is done at a clock edge and at the same time linestate transition takes place. Hence the simulator sometimes assumes the state of linestate before transition as the present state and sometimes it takes the state after transition as its present state.

4.4.2 Procedure USB_NON_IDLE

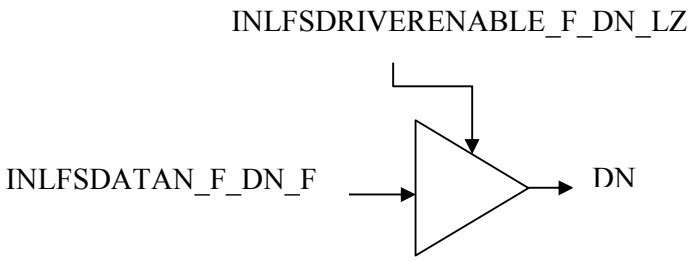
The change in the procedure is made to avoid the detection of start of packet by another procedure USB_DATA_RATE_CHECK from incorrect point.

There is a difference in the electrical states on DN between FS idle and first driven K bit .In FS idle state, DN is high impedance with ground which is modeled as High Z in the vhdl model of the transceiver. In driven K state (first bit of SOP), DN is low impedance to ground and hence is modeled as strong 0. This difference was confusing the existing checking mechanism in the test environment. The checking mechanism too has been changed to make it intelligent enough to recognize this difference and consider the start of packet from correct point.

4.4.3 Procedure USB_DATA_RATE_CHECK.

The change is made for the packet length which was checked by comparing the number of bits * bit period and the actual data packet length from the simulation output. The packet length is changed from number of bits to number of bits -1. And from the simulation waveform the packet length is considered till the 2 SEO's (leaving the J of SEO)

The checking mechanism was also not working correctly for packet length. This was because in simulations, the last J of EOP (appearing as strong 1 on DP and strong 0 on DN) was coming of 77.5 ns. This was followed by DN going to HighZ state. This was happening due to difference in propagation delay of LFSDRIVER that was put in it's model. See diagram below for the 2 timing arcs that are significant in the case of LFSDRIVER.



The problem was coming because INLFSDATAN_F_DN_F (10.5ns) is not equal to INLFSDRIVERENABLE_F_DN_LZ (4.7ns). Hence the last J bit was appearing to be shortened by (10.5 – 4.7 = 5.8 ns).

4.4. 4. Procedure FS_PACKET_LENGTH_CHECK

This procedure is written for changing the packet length measurement till idle state to EOP j state start point. it is required to overlook the error in transmission tests due bit period of j state of EOP as 77.56 ns.

This is used in USB_DATA_RATE_CHECK

4.4.5. Procedure CLOCK_27

For the TOUAREG RTL, there should only be a single clock of 27 MHz which is given as input to PLL block. Hence this has been written

4.4. 6. Procedure CLOCK_VAR

This procedure has been written to verify the RXERROR behavior when the reception clock is to be changed by ppm clock frequency difference. While using this clock it should be made sure that the signal bypass_pllext should be true thus bypassing the clock coming from PLL.

4.4.7. Procedure USB_SYNC_CHECK

A small modification is done for HS mode to avoid the index constraint violation error. The number of zeros was not counted properly hence this error was seen in test_hs_tx _16_2.1.2.2.b.

To avoid the incorrect warning “More zeros in SYNC than allowable” when correct number of zeroes are there in SYNC; the checking condition is changed from 2 bit period to 1.5. This was generated when the as the average bit period which is to be 2082 ps minimum and the bit period is lesser in USB sync .

4.4.8. Procedure USB_TX_END DELAY

The checker is changed as not to verify the TX end delay for FS/LS modes because no specific value is mentioned in the specs.

4.4.9. Procedure USB_RXSTART_DELAY

The checker is changed as not to verify the TX end delay for FS/LS modes because no specific value is mentioned in the specs.

4.4.10. Procedure TESTCONFIG

This has been changed to define bit period for HSINFS mode, TX end delay for hsinfs mode, USB EOP for hsinfs mode as per specs.

4.4.11. Procedure FSLS_rx_inter_packet_delay_check.

This procedure is written to verify the minimum interpacket delay between to receive packets.

4.5 CONDITIONS FOR NEW TEST CASES

New test cases have been written for the following test conditions:

1. Behavior of TX ready in FS only, HSINFS and LS only modes.
2. End delay corner cases for the above mentioned modes
3. Assertion/deassertion of RXactive within specified start and end delays
4. Behavior of RXError under various error conditions
5. Corner cases for start and end delays in HS mode for transmission

6. Corner cases for start and end delays for reception in HS mode.

There are about 60 new test cases for all the mentioned test conditions.

Chapter 5: NC-VHDL SIMULATOR

5.1 Introduction

The NC-VHDL simulator is a VHDL digital logic simulator that combines the high-performance of native compiled code simulation with the accuracy, flexibility, and debugging capabilities of event-driven simulation. The NC-VHDL simulator, like the NC-Verilog simulator, is a native compiled code simulator based on Cadence's Interleaved Native Compiled Code Architecture (INCA). The simulator is fully compliant with the IEEE 1076-1987 and IEEE 1076-1993 standards. The simulator is also compliant with the IEEE 1076.4, 2000VITAL ASIC Modeling Specification.

- Mixed Verilog and VHDL simulations
- Multiple levels (behavioral, RTL, gate)
- Mixed signal (digital, analog) simulations

With INCA, all the supported simulation styles leverage a single high-performance engine. Optimizing compilers for each input language or format create a sequence of instructions that are interleaved to create a single, contiguous code stream. This code stream is effectively a custom-built engine for the specific blend of simulation languages or techniques represented by a particular design.

For example, in a Verilog/VHDL configuration, both the Verilog and VHDL compilers are used to generate code for the Verilog and VHDL portions of the design, respectively. During an elaboration process similar to the linking used in computer programming, the Verilog and VHDL code segments are combined into a single code stream. This single executable, the simulation snapshot, is then directly executed by the host processor. This approach allows completely transparent mixed language and mixed-level simulations. It also lays the foundation for mixed signal simulations.

The following figure illustrates the INCA mixed-language simulation flow:

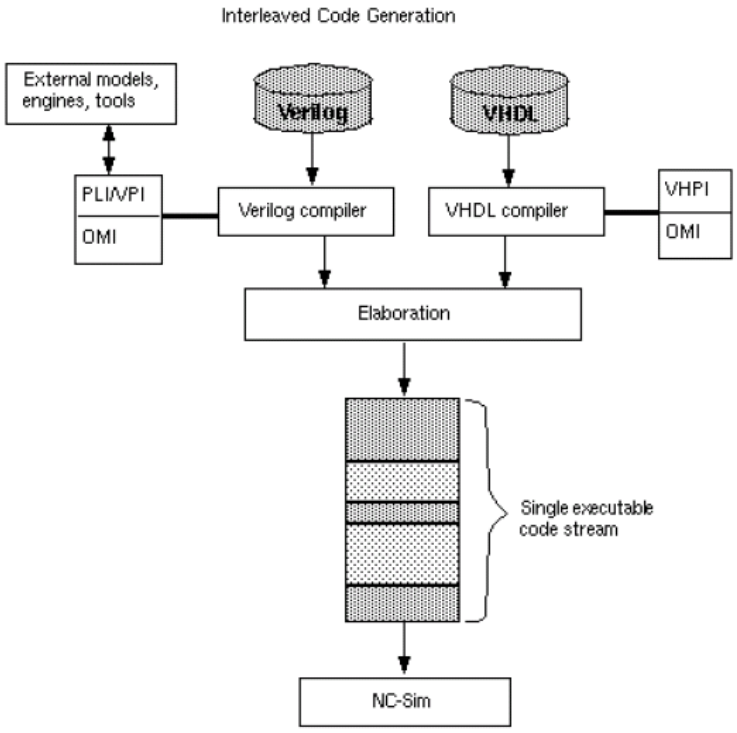


Fig. 5.1 simulation flow

5.2 Setting up Design Environment

In the NC-VHDL simulator, compiled objects and other derived data are stored in libraries. The library structure is organized around a Library.Cell: View approach, where:

- A library relates to a specific design or to a reference library.
- Cells relate to specific building blocks of the design.
- Views relate to different representations of the building blocks.

Each library has a logical name and is represented by a unique directory. When you compile your VHDL source files, the compiled objects and other derived data are stored in a library called the *work* library. All of the internal representations of cells and views that are required by the simulator are contained in a single file stored in the library directory.

No environment setup files are required to run the NC-VHDL simulator. The compiler can simply be invoked to compile VHDL source files, and the compiler will automatically create a default work library called worklib in a directory called INCA_libs, which is under the current directory. All design units are compiled into this library. That it creates automatically is a convenient feature that lets using the simulator quickly. However, this

methodology does not provide with any control over the work library and where it is located. If more control over the libraries into which design units are compiled, is required:

- Create a cds.lib file. This file contains statements that define libraries and that map logical library names to physical directory paths.
- Specify which library is the work library.
- Defining the WORK variable in an hdl.var file. Besides defining which library is the work library, the hdl.var file also can contain definitions of other variables that determine how design environment is configured, control the operation of NC-VHDL tools, and specify the locations of support files and invocation scripts.
- Using the -work command-line option.

If cds.lib and hdl.var file are created, all NC-VHDL tools and utilities that use these files use the same search mechanism to find the files. The first file that is found is used.
A setup.loc file can be created to change the directories to search or to change the order of precedence to use when searching for the cds.lib and hdl.var files.

5.3 Running the NC-VHDL Simulator

The NC-VHDL simulator can be run by executing three tools in succession. Each tool is invoked with its own command line and options. The following information summarizes what each tool does:

5.3.1 Compiling VHDL Source Files with ncvhdl

Ncvhdl analyzes and compiles VHDL source. This tool performs syntactic and static semantic checking on the HDL design unit(s) contained in the VHDL input source file(s) and generates an intermediate representation for each HDL design unit.

All intermediate objects are stored in a single file contained in the library directory. This library database file is called inca.architecture.lib_*version*.pak

After writing or editing VHDL source files, they are to be compiled. The program that is used to analyze and compile VHDL source is called *ncvhdl*.

The *ncvhdl* parser performs syntactic and static semantic checking on the input source files or VHDL design units. If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single packed library database file in the work library directory.

5.3.1.1 Invoking ncvhdl with options and a VHDL source file name(s).

These arguments can occur in any order except that parameters to options must immediately follow the option they modify. Ncvhdl can also be invoked *with* the -unit or -specific unit option and a design unit name.

Examples:

```
% ncvhdl -messages foo.vhd;        # foo.vhd is a source file
% ncvhdl -unit worklib.reg4;       # reg4 is an HDL design unit
```

Ncvhdl treats each command-line argument that is not an option or a parameter to an option as a filename. For each filename, it first tries to open the file as specified. If this fails, each file extension specified with the VHDL_SUFFIX variable in the hdl.var is appended to the name and *ncvhdl* tries to open the file. The default file extensions are .vhd and .vhdl. If all suffixes are exhausted, *ncvhdl* generates an error.

5.3.1.2 Library. Cell

Design units are compiled into a Library.Cell: View.

- The library name is the logical name of the work library into which the design units have been compiled. If you have not created cds.lib and hdl.var setup files, the compiler automatically creates a default work library called worklib. This library is created in a directory called INCA_libs, which is under the current directory (that is, the physical location of the default work library is ./INCA_libs/worklib). If you have created a cds.lib file, the library name is either the logical library name used with the -work command-line option or the library defined in the hdl.var file with the WORK variable.

- The cell and view names depend on the kind of VHDL design unit parsed.they are as shown in the following figure.

Design Unit	Cell Name	View Name
Entity	Entity name	`entity`
Architecture	Corresponding entity name	Architecture name
Package	Package name	`package`
Package body	Corresponding package name	`body`
Configuration	Configuration name	`configuration`

Fig. 5.2 Table for Cell and View Names

For example, if the work library is worklib, and you compile a source file that contains an entity called drink machine, the entity is compiled into worklib.drink_machine:entity. If the source file contains an architecture called rtl for entity drink_machine, the architecture is compiled into worklib.drink_machine:rtl. The following figure illustrates the *ncvhdl* process flow:

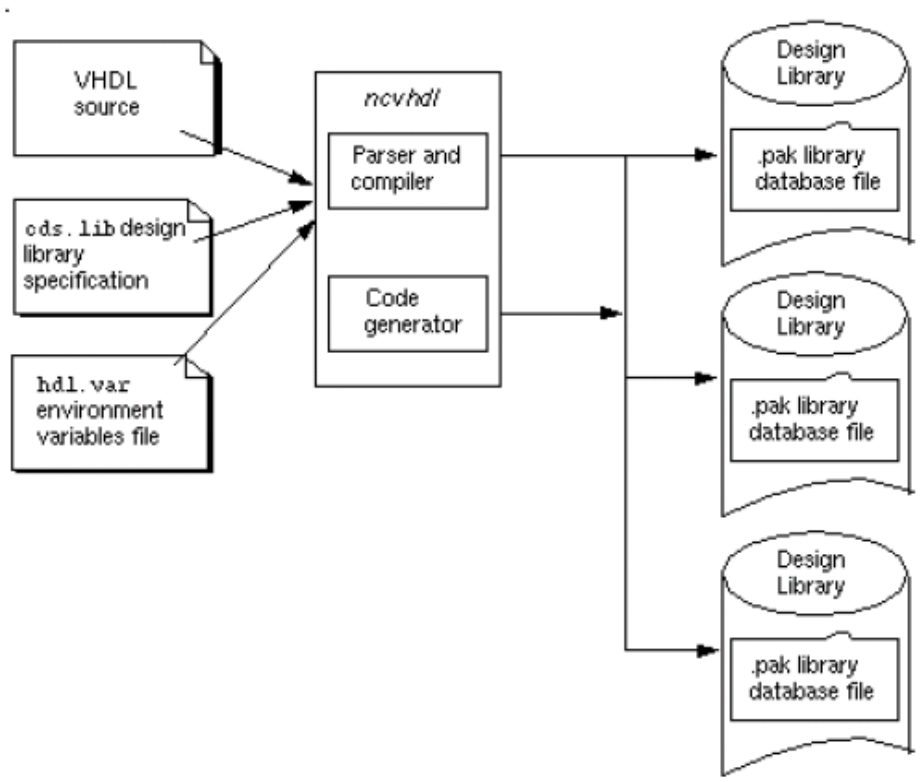


Fig 5.3 NCVHDL Command Syntax

Command-line options can be entered in uppercase or lowercase, and can be abbreviated to the shortest unique string, indicated here with capital letters.

ncvhdl [*options*] *filename* [*filename* ...]

ncvhdl [*options*] { -specificunit [*lib*.]*cell*[:*view*] | -unit [*lib*.]*cell*[:*view*] }

5.3.2 Elaborating the Design with ncelab

Before model can be simulated, the design hierarchy defining the model must be elaborated. The tool used for elaborating the design is called *ncelab*. *Ncelab* is a language-independent elaborator. It constructs a design hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot file. The snapshot is the representation of design that the simulator uses to run the simulation.

Ncelab can be invoked with command-line options and arguments. The options and arguments can be specified in any order, but parameters to options must immediately follow the options they modify.

The argument to the ncelab command can be:

- The Library.Cell: View name(s) of the top-level HDL design unit(s). Design units specified on the command line cannot be instantiated in the design. The top-level unit(s) specified on the command line can be:
  - One VHDL top-level unit.
  - One or more Verilog top-level units.
  - One VHDL unit and one or more Verilog units.

Syntax:

% ncelab [*options*] [*Lib*.] *Cell* [: *View*]...

The cell (top-level unit name) must be specified. The library if a top-level unit with the same name exists in more than one library must be specified. If there are multiple views (Verilog views or VHDL architectures) of the top-level unit(s), the easiest, and recommended, thing to do is to specify the view on the command line.

ncelab uses the following rules to resolve the reference to the top-level design unit if the view is not specified:

a. Search the library defined with the WORK variable in the hdl.var file. If one view of the cell exists in that library, use that view. For Verilog, generate an error message if more than one view exists. For VHDL, use the most-recently analyzed architecture.

b. If the WORK variable is not defined in the hdl.var file, search the libraries defined in the cds.lib file. If one view of the cell exists in the libraries, use that view. For Verilog, generate an error message if more than one view exists. For VHDL, use the most-recently analyzed architecture.

- The name of a Verilog configuration.

Syntax:

% ncelab [*options*] -libmap *library_map_file configuration_name*

If you are running the NC-Verilog simulator using the single-step invocation method (*ncverilog*), use the following command:

% ncverilog [*options*] +libmap+*library_map_file* +nctop+*top_level_unit* *source_files*

- The name of a 5.x configuration.

Syntax:

% ncelab [*options*] [*Lib*.]*Cell*[:*View*]

Elaboration produces a simulation snapshot. The snapshot is also a Lib.Cell:View. Assuming that the -snapshot option was not used to explicitly name the snapshot, the snapshot is named:

- *Library*—the name of the library where the top-level unit on the *ncelab* command line was found. If more than one Verilog top-level module is specified on the command line, the *Library* is the name of the library where the first top-level module listed on the command line was found.
- *Cell*—the name of the top-level unit on the *ncelab* command line. If more than one Verilog top-level module is specified on the command line, the *Cell* is the name of the first top-level module listed on the command line.
- *View*—the view name that was specified for the first top-level design unit on the *ncelab* command line or (if a view was not specified) the name of the view that was used as a result of the rules that *ncelab* uses to resolve references to top-level units given on the *ncelab* command line.

The following figure illustrates the *ncelab* process.

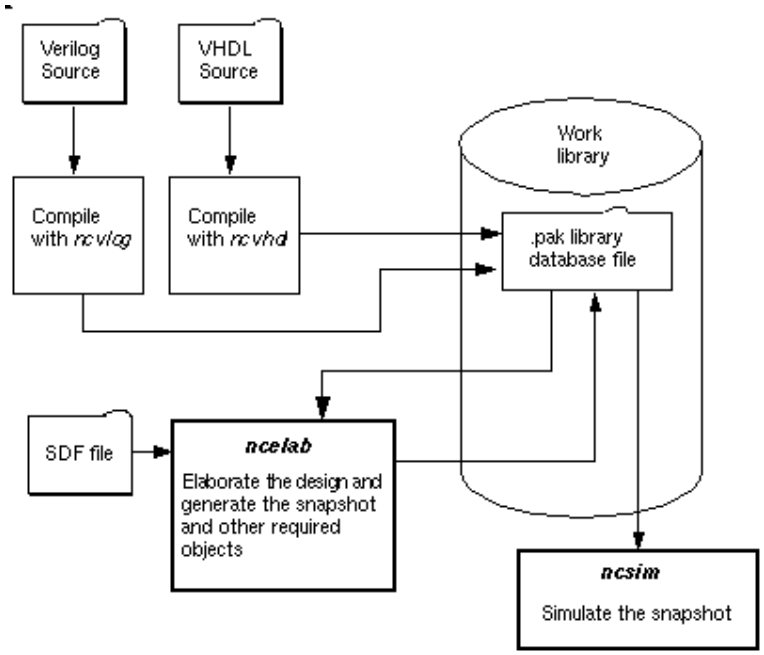


Fig 5.4 Ncelab Process

5.4 Simulating Design with ncsim

After the design is compiled and elaborated, the simulator, *ncsim* can be invoked. This tool simulates Verilog and VHDL using the compiled-code streams to execute the dynamic behavior of the design. *Ncsim* loads the snapshot as its primary input. It then loads other intermediate objects referenced by the snapshot. In the case of interactive debugging, HDL source files and script files also may be loaded. Other data files may be loaded as demanded by the model being simulated (via $read* tasks or Textio). The outputs of simulation are controlled by the model or debugger. These outputs can include result files generated by the model, Simulation History Manager (SHM) databases, or Value Change Dump (VCD) files. The following figure illustrates the *ncsim* process flow.

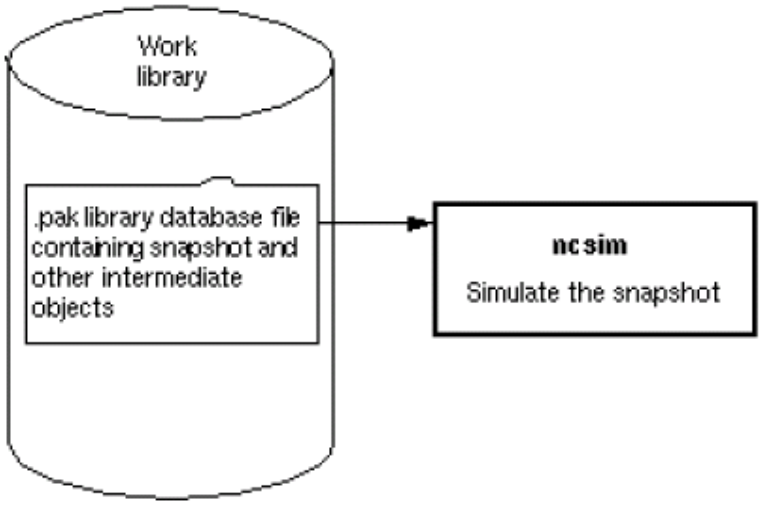


Fig 5.5 Ncsim process

Ncsim can be invoked with options and a snapshot name specified in Lib.Cell: View notation. The options and the snapshot argument can occur in any order except that parameters to options must immediately follow the option they modify. Only one snapshot can be specified.

The syntax for invoking the simulator is:

% ncsim [*options*] [*Lib*.]*Cell*[:*View*]

- the cell must be specified .
- If a snapshot with the same name exists in more than one library, the easiest and recommended) thing to do is to specify the library on the command line.
- If there are multiple views that contain snapshots, the easiest (and recommended) thing to do is to specify the view on the command line.

Chapter 6: CONCLUSION AND FUTURE-SCOPE

Conclusion: Verification is one of the major aspects of testing of design. It is considered to be such an important issue that seventy percent of the testing phase of designing is dedicated to verification only.

Verification of RTL is a very important step in design cycle. If bugs or defects are detected during RTL verification phase; design cycle time, and engineering cost can be reduced. Redesigning is also easier at RTL level.

ST device test environment is an exhaustive environment which checks all possible cases including corner cases.

The purpose of the thesis is to enlist more test conditions and write VHDL code for those test conditions along-with the verification and debugging of existing test environment. By the time of completion of this thesis 70 more test-cases have been written. For these 70 new test conditions and also for existing 108 some new procedures have been written and some existing procedures have been modified

Future-scope: More test-cases are to be generated for more precise verification of USB2PHY. Test-cases for UTMI (USB Transceiver Macro cell Interface) and UTMI plus can be written. It will further help in testing and verification of USB devices.

Code coverage and functional coverage of RTL can also be incorporated.

The same environment can be for post synthesis and post lay-out verification.

REFERENCES

[1]. www.usb.org

[2]. White paper, "*Pros and cons of USB*", from internet

[3]. Compaq Computer Corp., Intel Corp., Microsoft Corp., NEC Corp., "*Universal Serial Bus Specification Revision 2.0"*, 2000.

[4]. Chung-Ping Young Devaney, M.J. Shyh-Chyang Wang, "*Universal serial bus enhances virtual instrument-based distributed power monitoring"*, from proceedings of IEEE transactions on instrumentation and measurement, 2001

[5]. Lynn, K., Micrel Semicond., San Jose, CA, "*Unisversal serial bus(USB) power management*", from proceedings of WESCON,1997

[6]. Depari, A. Ferrari, P. Flammini, A. Marioli, D. Sisinni, E. Taroni, A, "*IEEE1451 smart sensors supporting USB connectivity*", from the proceedings of IEEE Sensors for Industry conference, 2004

[7]. Wnenshenj Luo, "*Self Configuring Network Environment Sensors*", PhD. Confirmation report, University of Queensland, Australia,2005

[8]. J. Axelson, *USB Complete - Everything You Need to Develop Custom USB Peripherals*, Lakeview Research, Madison, 1999.

[9]. Bartolonirej Bebel," design and implementation of a USb to Can bridge for GuRoo project", Phd thesis, university of queensland,Australia,2001

[10]. M. Zerkus, J. Lusher, J. Ward, "USB Primer – Practical Design Guide", Circuit Cellar, http://www.circuitcellar.com (current: 16/6/01).

[11]. Zahariadis, T. Pramataris, K. Zervos, N., "*A comparison of competing broadband in-home technologies",* Electronics and Communication Engineering Journal, 2002

[12] Caldari, M. Conti, M. Crippa, P. Orcioni, S. Sbrega, M. Turchetti, C., "*Object-oriented design methodology applied to the modeling of USB device and bus interface layers*", from the proceedings of circuit and systems, 2002, ISCAS, 2002, IEEE international Symposium, 2002

[13] Philip Koopman , Tridib Chakravarty , " Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks" , 2004 International Conference on Dependable Systems and Networks (DSN'04),2004

[14]. Intel Corporation, "*USB 2.0 transceiver Macrocell Interface (UTMI) Specification"*, 2001

[15]. Pavel kubalik, Jinni Bucek *"FPGA implementation of USB 1.1 device core",* CTU Pregue, 2000

[16]. http://www.xilinx.com/esp/dvt/prof_brdcst/collateral/usb.pdf

[17]. USB in a nutshell, www.beyondlogic.com

[18]. STMicroelectronics, "*Test environment doc*", LIPP, STMicroelectronics, Noida

[19]. STMicroelectronics, "*Usb2PHY_verification plan*", LIPP, STMicroelectronics, Noida, 2001

[20]. J.bhasker, "VHDL primer", Pearson Education, 2003

[21]. Peter J. Ashenden, "the designer's guide to VHDL" , Morgan Kaufmann publishers, California,2001

LIST OF FIGURES