

JTAG Test Access Port Controller for Device Interconnect

Group # D12

Chittaranjan Samantaray (01D07028)

Saurabh Bhandari (01D07029)

Ankit Jain (01D07030)

Supervisor: Prof. M. C. Chandorkar

ABSTRACT:

The project involves communication between an embedded test bus controller (eTBC) and a device having JTAG port and further programming it using its JTAG port. This enables us to take advantage of the latest emulation technology and use it in any device compliant to IEEE Std. 1149.1 target. For this purpose, an eTBC was used to master an IEEE Std. 1149.1 test access port (TAP) under the control of a microcontroller host. The microcontroller is embedded on a USB based controller. Thus, one can program the microcontroller by giving commands through the USB port of a host computer which then sends signals to the eTBC. Conceptually, an eTBC operates like a memory-mapped or I/O-mapped peripheral to a microcontroller to achieve the desired communication.

We were thus able to program the CPLD using its JTAG port. Codes have been written and hardware developed which enables us to program the CPLD once its 'jedec' file is available. The project can further be extended to program any JTAG compliant device by developing the software specific to that device.

CONTENTS

1. Introduction	1
2. Design Aspects	2
2.1 Block Diagram	2
2.2 Various Components of the Circuit	3
2.2.1 Power Unit	3
2.2.2 Oscillator Circuit	4
2.2.3 TUSB3210 (Host for eTBC)	5
2.2.4 Embedded Test Bus Controller (SN74LVT8980A)	5
2.2.5 JTAG Header	6
2.2.6 RS-232 Port	6
2.2.7 MAX3232	6
2.2.8 LEDs	7

3. PCB Layout	8
3.1 Schematic Layout	8
3.2 Board Layout	9
3.3 Board Photograph	10
3.4 Other Available Headers	11
3.3.1 EMU Header	11
3.3.2 General 8-Pin Header	11
3.3.3 TRST Header	11
3.3.4 General 12-PIN Header	11
3.4.5 Reset Header	11
3.4.6 Common Ground Pin	12
3.5 List of Components	12
4. Functioning of TUSB3210 and eTBC	13
4.1 Functioning of TUSB3210	13
4.1.1 Pin Description	13
4.1.2 Functional Block Diagram	14
4.1.3 TUSB Operations	15
4.2 Functioning of eTBC	17
4.2.1 Pin Description	17
4.2.2 eTBC Operations	18
5 Programming of the Target Device	22
5.1 Firmware Programming Flow	22
5.1.1 Downloading the Application Code into the Device	22
5.1.2 Application Code	23
5.1.3 Interrupt Subroutines	23
5.1.4 RS232 Port for Debugging	24
5.2 eTBC Functioning	25
5.2.1 Read/Write Access	25
5.2.2 Register Description	26
5.3 Programming of the CPLD	27
5.3.1 Application Firmware Code	28
5.3.2 Host Side Programming	29
5.3.3 Erasing the CPLD	30
5.3.4 Verifying the CPLD	31
6 Conclusions and Future Scope	32
References	33
Appendix	34
Appendix A : The send.c Program	34
Appendix B : Using RS-232 for readback	39

1. INTRODUCTION

As the dimension of devices shrunk and surface-mount packing technology improved, it became increasingly complex to test devices. To surmount this problem, JTAG (Joint Test Action Group) devised a specification for performing boundary-scan hardware testing at the IC level. This resulted in IEEE 1149.1, a standard that established the details of access to any chip with a JTAG port.

The specification JTAG devised uses boundary-scan technology, which enables extensive debugging and diagnostics on a system through a small number of dedicated test pins. Signals are scanned into and out of the I/O cells of a device serially to control its inputs and test the outputs under various conditions. Today, boundary-scan technology is probably the most popular and widely used design-for-test technique in the industry.

A device is generally programmed using the RS-232 port or the parallel port (printer port) of a computer which is quite slow. Through the use of a JTAG port, the in system programming of devices is much faster. Though the communication between eTBC and devices is serial, that between the eTBC and microcontroller occurs with a parallel 8-bit efficiency and is hence faster. We use a USB based controller which has an embedded 8052 based microcontroller and this microcontroller is programmed using USB commands from a host computer. The USB port is more versatile and provides capability of connecting to multiple devices. Further, it is very fast. Thus, we can use USB to program many devices simultaneously.

The basic module of our system consists of an eTBC receiving instructions from a microcontroller and sending appropriate signals to a device having JTAG port. The system needs a regulated power supply, because, device characterizers run for hours together and a battery or a local power storage device cannot provide supply for such a long time consistently. Here, we power the system using the power supplied by the USB port.

The microcontroller is programmed through a USB and sends instructions to the eTBC. The TAP (Test Access Port) state generator operates under the control of an executing command to generate the test sequences to move the connected target device from one stable state to another, to capture and scan test data into/out of target devices and to operate built-in test modes of the target device.

2. DESIGN ASPECTS

The circuitry of target device contains an interface through which instructions and data are communicated. Required test features define a Test Access Port (TAP) to allow access to test circuitry, instruction register for interpreting commands, boundary registers for implementing device I/O tests and bypass register for implementing scan path. The TAP interface consists of the four signals, viz. TMS, TCK, TDI, and TDO. Our system aims at sending appropriate TMS sequence to the target device for its programming. This also means that we move the TAP controller from one of its stable state to some other state.

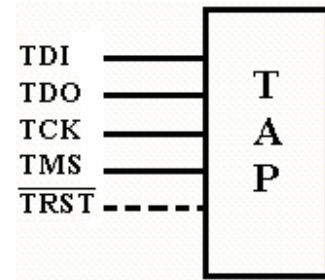


Fig.2.1 Test Access Port

2.1 BLOCK DIAGRAM

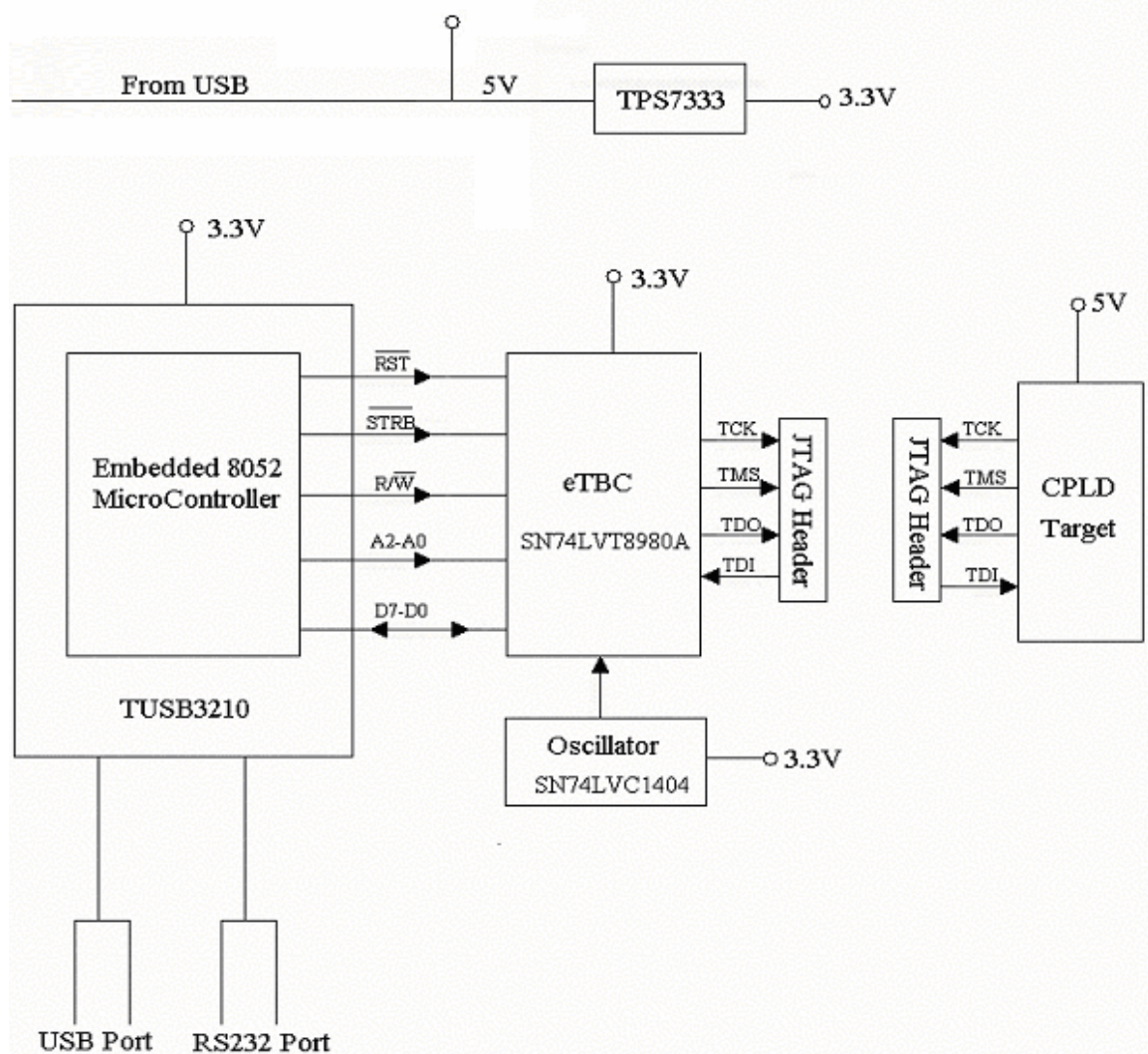


Fig.2.2 Basic Block Diagram

As mentioned earlier, the basic module of our project is an eTBC, which operates under the commands given by a microprocessor that is programmed using USB protocols. All the components in our circuit operate at a V_{cc} of 3.3 V and hence a regulator is also required as indicated in the block diagram shown in Fig.2.2. Our target device (CPLD) operates at 5 V but since the eTBC is 3.3-5 V tolerant this is not a cause of concern. However, the target device has an external power supply, and only the programming signals are exchanged between the programmer and the target device. The target device might be sitting on another system, and need not be taken out for programming.

2.2 VARIOUS COMPONENTS OF CIRCUIT

Our device can be divided into following blocks based on their functionality:

2.2.1 POWER UNIT:

The 5 V supply obtained from the USB is used for powering up the system. This 5 V is used as an input to a regulator, **TPS7333** (Low Dropout Regulator) to get a constant 3.3 V supply which is then used as V_{cc} for all the circuit components.

- i) The system is powered through the USB which supplies a voltage of 5 volts. Also, this can give a maximum current of 500mA. Since our circuit consumes a current far less than this value, we have powered our device through the USB.

- ii) TPS7333 – *To convert 5 V to 3.3 V constant output*

It can source a maximum current of 2 A. This provides V_{cc} for other components in the circuit like USB controller, oscillator and eTBC. The TPS7333, unlike many other low dropout voltage regulators (LDOs), features very low quiescent currents that remain virtually constant even with varying loads. This is stable even at zero loads; no minimum load is required for operation.

As with most LDO regulators, the TPS7333 requires an output capacitor for stability. A low-ESR 10 μF solid-tantalum capacitor connected from the regulator output to ground is sufficient to ensure stability over the full load range.

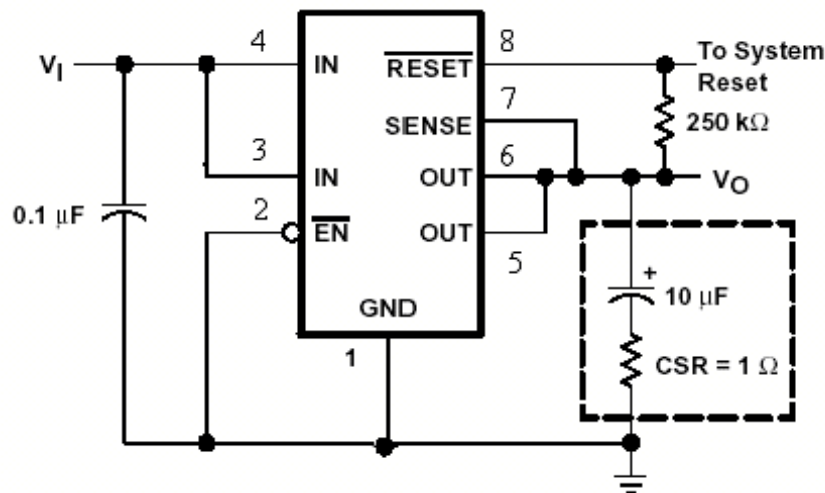


Fig.2.3 TPS7333 Circuit Diagram [1]

2.2.2 OSCILLATOR CIRCUIT

This block of our project aims at generating a square wave input for eTBC at its CLKIN pin. The block consists of two parts, first being the crystal oscillator and the second part consists of its circuitry to provide a square wave of desired frequency. To obtain this, SN74LVC1404 chip is used, which has output drive strength of around 25 mA at 3.3 V operation. Logic diagram for the chip is shown in Fig.2.4. The CTRL input is provided to disable the oscillator circuit to reduce the power consumption. To obtain continuous output at the OSCOUT pin, CTRL signal is kept at a high level. The XIN and XOUT pins are connected to a crystal of frequency 12 MHz. Circuitry involved for the crystal connected between pins XIN and XOUT is shown in Fig.2.5.

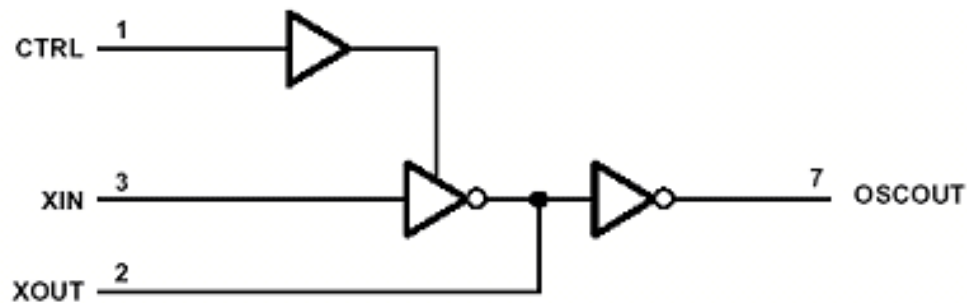


Fig 2.4 Logic Diagram for SN74LVC1404 [2]
A crystal is connected between XIN and XOUT and required clock is obtained at OSCOUT

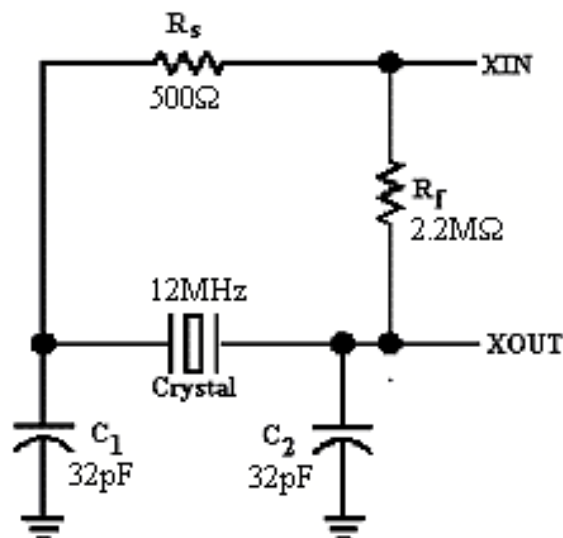


Fig.2.5 Crystal Circuit connected between XIN and XOUT [2]

2.2.3 TUSB3210 (Host for eTBC)

The TUSB3210 is a USB based controller. It has an embedded 8052 based microcontroller and hence provides the designers with much flexibility. Its programmability makes it flexible enough to use for various general USB I/O applications. Using a 12 MHz crystal, the onboard oscillator generates the internal system clocks. The device can be programmed via an inter-IC (I²C) serial interface at power on from an EEPROM, or optionally, the application firmware can be downloaded from a host PC via USB. Its embedded 8052-based microprocessor acts as a host to the eTBC. The microcontroller can be programmed via USB from a host computer and sends appropriate signals to the eTBC. Data transfer between the USB, MCU and external devices are defined by an Endpoint Descriptor Block (EDB). Four input and four output EDBs are present. The RS-232 port is available to monitor 8052 MCU activity for debugging purposes.

It operates at a Vcc of 3.3V. Each port pin of the TUSB can sink around 20mA of current. It has an on-chip PLL which generates 48 MHz from 12 MHz supplied to it by the oscillator.

Embedded 8052 based Microcontroller

The embedded 8052 based microcontroller is a high performance 8-bit microcontroller. It is programmed via the USB port. Its several features include:

- Four ports with 8 pins on each port
- 256-byte internal data memory
- Three 16-bit Timer/Counters
- Upto 48 MHz operation
- I²C support
- All ports have internal pull ups

2.2.4 EMBEDDED TEST BUS CONTROLLER SN74LVT8980A

Though powered at 3.3 V, the TAP interface is fully 5 V tolerant for mastering both 5 V and/or 3.3 V IEEE Std. 1149.1 targets. It provides simple interface to a low-cost 3.3 V microprocessor/microcontroller via 8-bit asynchronous read/write data bus. Further it is easily programmable via scan-level command set and smart TAP control. It generates the TMS sequences required for communication with JTAG compliant targets. The TAP-state generator sources the TMS signal, which sequences the TAP controller of connected IEEE Std. 1149.1-compliant target device.

The eTBC masters all TAP signals required to support one 4 or 5 wire IEEE Std. 1149.1 serial test bus: test clock (TCK), test mode select (TMS), test data input (TDI), test data output (TDO), and test reset (TRST). All such signals can be connected directly to the associated target IEEE Std. 1149.1 devices without need for additional logic or buffering. However, as well as being directly connected, the TMS, TDI, and TDO signals can be connected to distant target IEEE Std. 1149.1 devices via a pipeline, with a retiming delay of up to 15 TCK cycles; the eTBC automatically handles all associated serial-data justification. The exact technical details are explained in section 4.2.

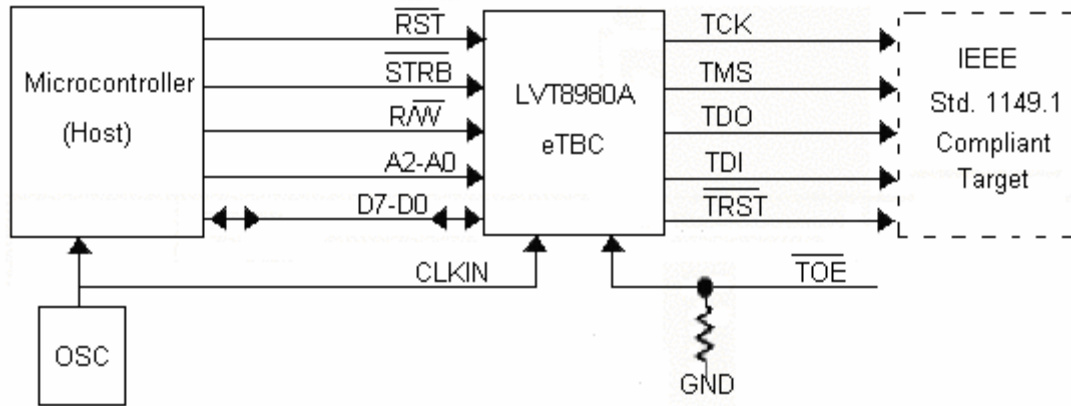


Fig.2.6 Basic Block Diagram of the eTBC application [3]
Microcontroller used is an embedded 8052 microcontroller

2.2.5 JTAG HEADER

The JTAG header used has 4 pins. This is the standard interface used by JTAG emulators to interface to various devices, in our case CPLD (XC9572XL_PC44). The various signals at different pins of the header are mentioned below.

Table 2.1
Signals at different pins of JTAG header

Pin #	Signal
1	TDO
2	TDI
3	TMS
4	TCK

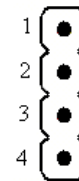


Fig.2.7 JTAG header
Signal at various pins is given in Table 2.1

2.2.6 RS-232 Port

The RS-232 port is available on board to monitor 8052 MCU activity for debugging purposes. It helps in communicating with the MCU via the serial port of computer and hence makes debugging easy. The signals propagating through RS232 are buffered using MAX3232 for better signal levels. The RS232 port connection can be broken up using jumpers provided on the PCB. There are jumpers to break the connection from the MCU as well as to disable the power supply of MAX3232.

2.2.7 MAX3232

The MAX3232 device consists of two line drivers, two line receivers, and a dual charge-pump circuit. The device provides the electrical interface between an asynchronous communication controller and the serialport connector. The charge pump and four small external capacitors allow operation from a single 3 V to 5.5 V supply. The devices operate at data signaling rates up to 250 kbit/s and maximum of 30 V/ μ s driver output

slew rate. It has a low supply current of 300 μA and requires four external capacitors of 0.1 μF .

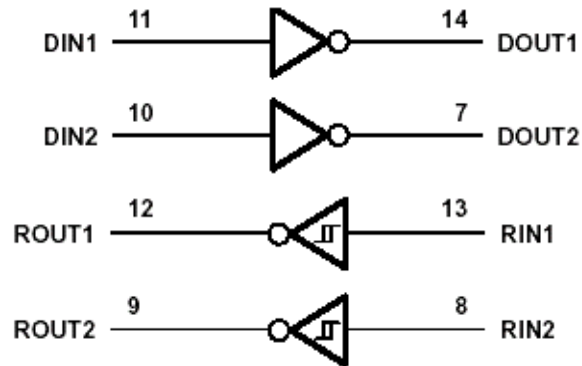


Fig.2.8 Logic Diagram for MAX3232 (Positive Logic) [4]

2.2.8 LEDs

Five LEDs have been provided on the board. Four of them represent pins 0, 1, 2 and 3 of port 2 of the embedded 8052 based microcontroller and hence provide a good way for analysis of a problem and interpretation of results. An additional LED has been used as a power-on indicator. The circuitry used to implement LEDs is shown in Fig.2.9. The port pins can sink upto 8mA each.

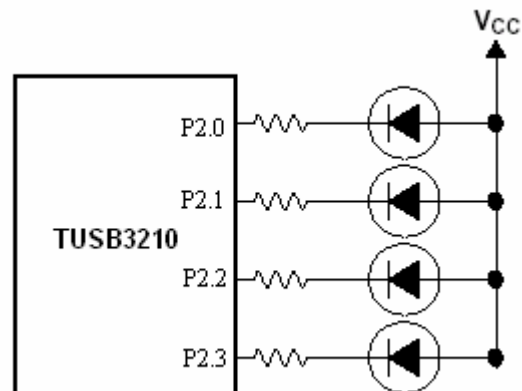


Fig.2.9 LED Connections on PCB

3.1 SCHEMATIC LAYOUT



3.2 BOARD LAYOUT

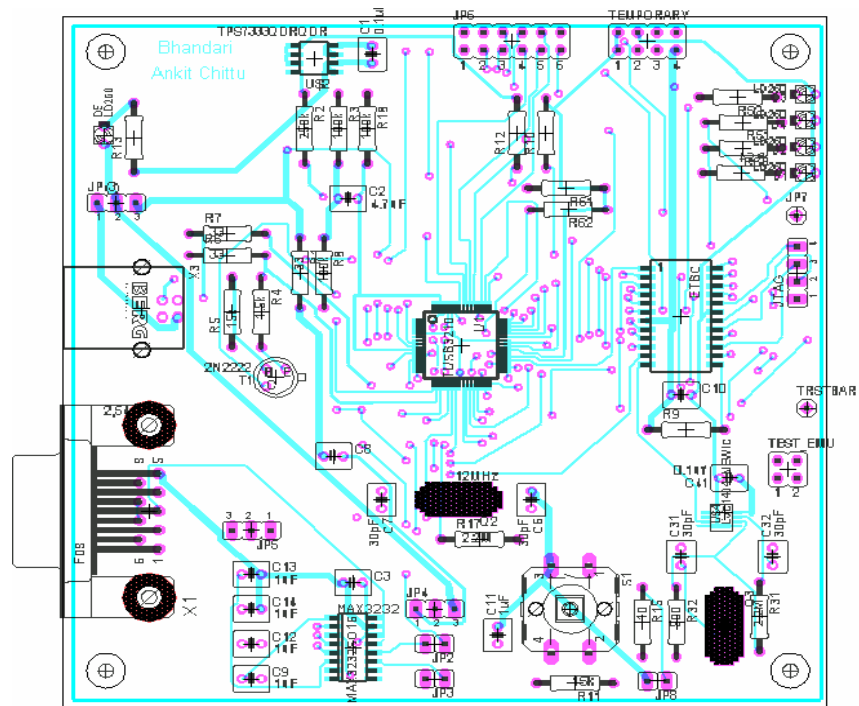


Fig 3.2 Top view of the PCB

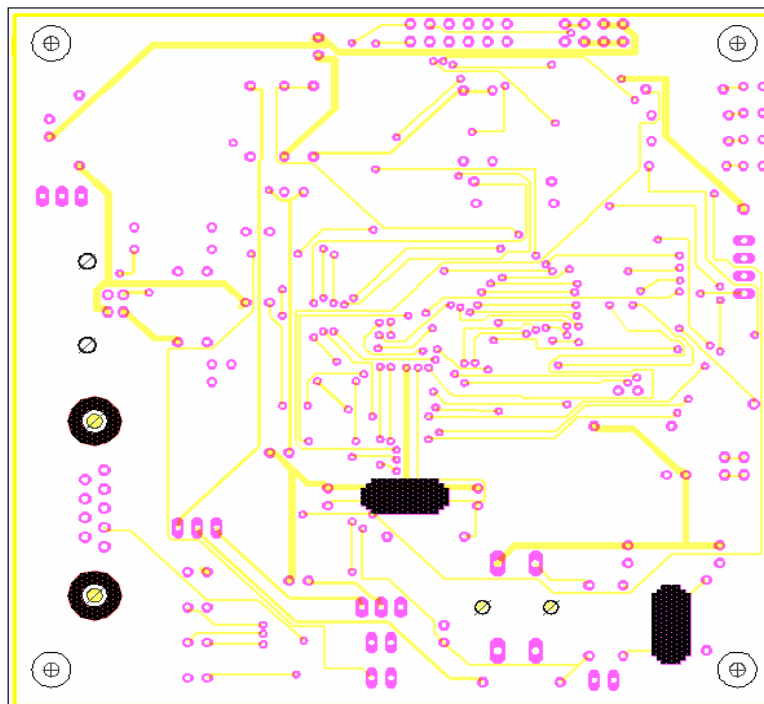
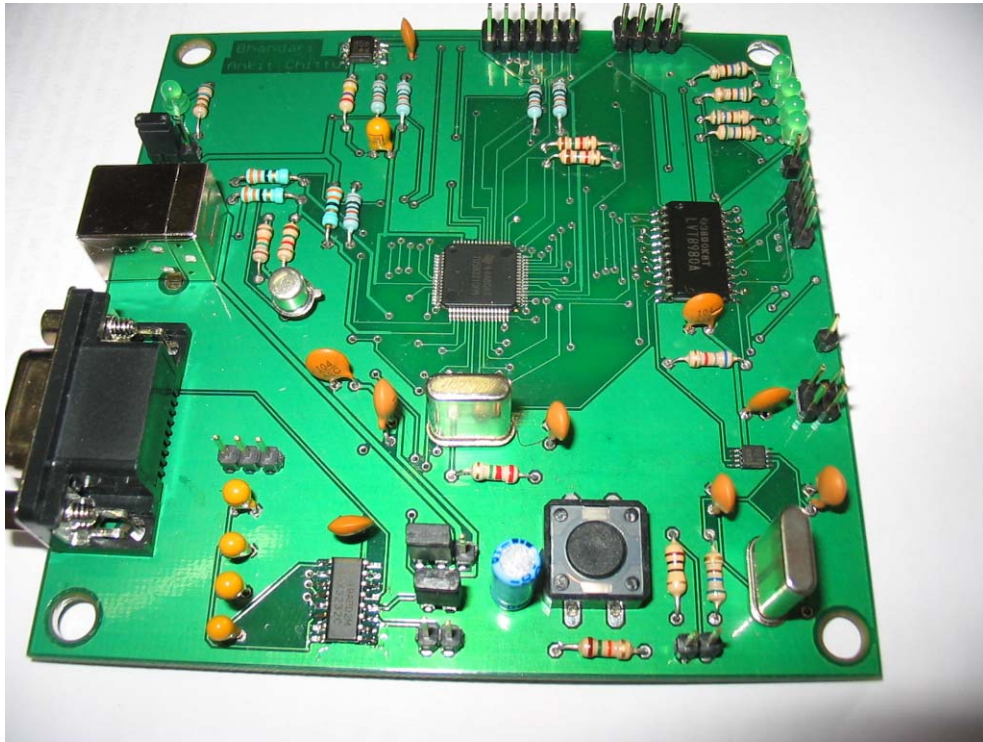
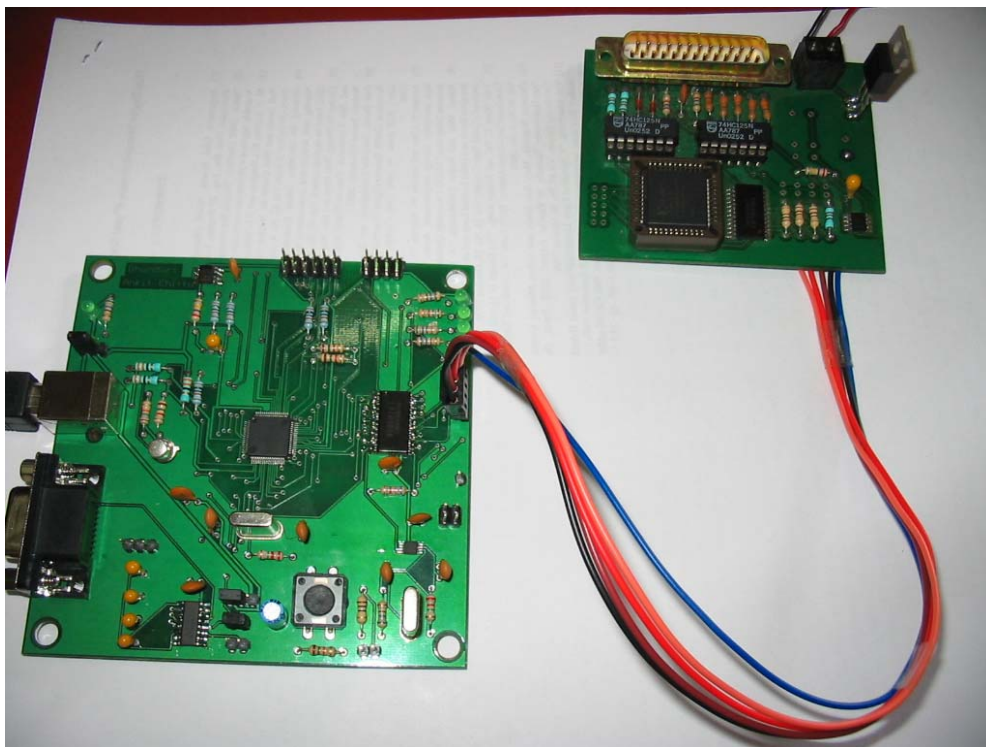


Fig 3.3 Bottom view of the PCB

3.3 PCB PHOTOGRAPH



A photograph of the PCB (JTAG Programmer)



A photograph of the JTAG Programmer along with target device

3.4 OTHER AVAILABLE HEADERS

3.4.1 EMU Header

EMU0 and EMU1 pins are required if the target device is a TI device. It is used when multiple devices are to be connected in the target chain of eTBC. So jumpers for these two pins have been provided externally on our board to make them high/low as and when required.



Fig 3.4 EMU header

3.4.2 General 8-PIN Header

We have an 8-pin header 3 pins of which are Gnd, 3 pins are 5V and remaining 2 pins are connected to 3.3V. These may be useful in future and are also useful for debugging purposes.

3.4.3 TRST Header

In addition to the above two, a jumper to give $\overline{\text{TRST}}$ is also provided for future use. Like the EMU, this header is required for a TI device and is not used while working with the CPLD.

3.4.4 General 12-PIN Header

There is also a 12 pin header on PCB which has its various pins connected to the ports of embedded microcontroller. These pins can also be used for debugging purpose. The configuration of the pins is shown in the table below.

3.4.5 Reset Header

A header for resetting the device has been provided in addition to the reset switch already available on board. This has been done from future considerations wherein the completely boxed device has a reset switch connected to this header.

Table 3.1
Signals at 12 Pin Header

Pin #	Signal	Pin #	Signal
1	P1.3	7	P3.2
2	P1.4	8	P3.3
3	P2.4	9	P3.4
4	P2.5	10	P3.5
5	P2.6	11	P3.6
6	P2.7	12	Not Connected

3.4.6 Common Ground PIN

A ground pin has been provided adjacent to the JTAG header for providing common ground to both the target device and the programmer.

3.5 LIST OF COMPONENTS:

1. TUSB3210
2. SN74LVT8980A
3. SN74LVC1404
4. TPS7333
5. MAX3232S016
6. Bridge Rectifier
7. LM7805CV
8. Resistances: 2.2M Ω , 220k Ω , 100 k Ω , 15 k Ω , 4.7k Ω , 1k Ω , 560 Ω
9. Capacitors: 473 μ F, 10 μ F, 1 μ F, 0.1 μ F, 33pf, 473nF
10. 12 MHz Crystal Oscillators
11. 9V Transformer
12. USB Port
13. RS232 Port
14. Push-Button Switch
15. Jumpers
16. LEDs

4. FUNCTIONING OF TUSB3210 AND EMBEDDED TEST BUS CONTROLLER

4.1 FUNCTIONING OF TUSB3210

The TUSB3210 is a USB based controller. It has an embedded 8052 based microcontroller which acts as a host to the eTBC. The microcontroller can be programmed via USB from a host computer and sends appropriate signals to the eTBC. This chapter gives a brief description of TUSB. For details, one has to take a look at the datasheet of TUSB3210 from the TI website as mentioned in the reference [5].

4.1.1 PIN DESCRIPTION

A brief description of the functionalities of the various pins of the TUSB3210 is given in table 4.1.

Table 4.1
Terminal Functions of TUSB3210

TERMINAL	DESCRIPTION
1.8VDD	1.8V. When VREN is high, 1.8V must be applied externally to provide current for the core during suspend.
DM	Differential data-minus USB
DP	Differential data-plus USB
P0[0:7] P1[0:7] P2[0:7]	General-purpose I/O port bits 0–7, Schmitt-trigger input, 100mA active pullup, open-drain output.
P3.0/RXD	P3.0: General-purpose I/O port 3 bit 0, Schmitt-trigger input, 100mA active pullup, open-drain output. RXD: Can be used as a UART interface.
P3.1/TXD	P3.1: General-purpose I/O port 3 bit 1, Schmitt-trigger input, 100mA active pullup, open-drain output. TXD: Can be used as a UART interface.
P3.2	General-purpose I/O port 3 bit 2, Schmitt-trigger input, 100mA active pullup, open-drain output; INT0 only used internally.
P3.3	General-purpose I/O port 3 bit 3, Schmitt-trigger input, 100mA active pullup, open-drain output; may support INT1 input.
P3.[4:7]	General-purpose I/O port 3 bits 4–7, Schmitt-trigger input, 100mA active pullup and open-drain output.
PUR	Pullup resistor connection pin (3-state).
RST	Controller master reset signal, Schmitt-trigger input, 100mA active pullup.
S2	General-purpose input, This input has no internal pullup.
S3	General-purpose input. This input has no internal pullup.
SCL	Serial clock I ² C; push-pull output.
SDA	Serial data I ² C; open-drain output.
SUSP	Suspend status signal: suspended (HIGH); unsuspended (LOW)

VREN	Voltage regulator enable: enable active-LOW; disable active-HIGH
X1	12-MHz crystal input
X2	12-MHz crystal output
Vcc	Power supply input, 3.3V typical.

4.1.2 FUNCTIONAL BLOCK DIAGRAM

The USB-based controller (TUSB3210) has an embedded 8052-based microcontroller which acts as a generic host to the eTBC. Its programmability makes it flexible enough to be used for other general-purpose input/output applications. The device can also be programmed via an I²C interface. The functional block diagram is given in Fig.4.1

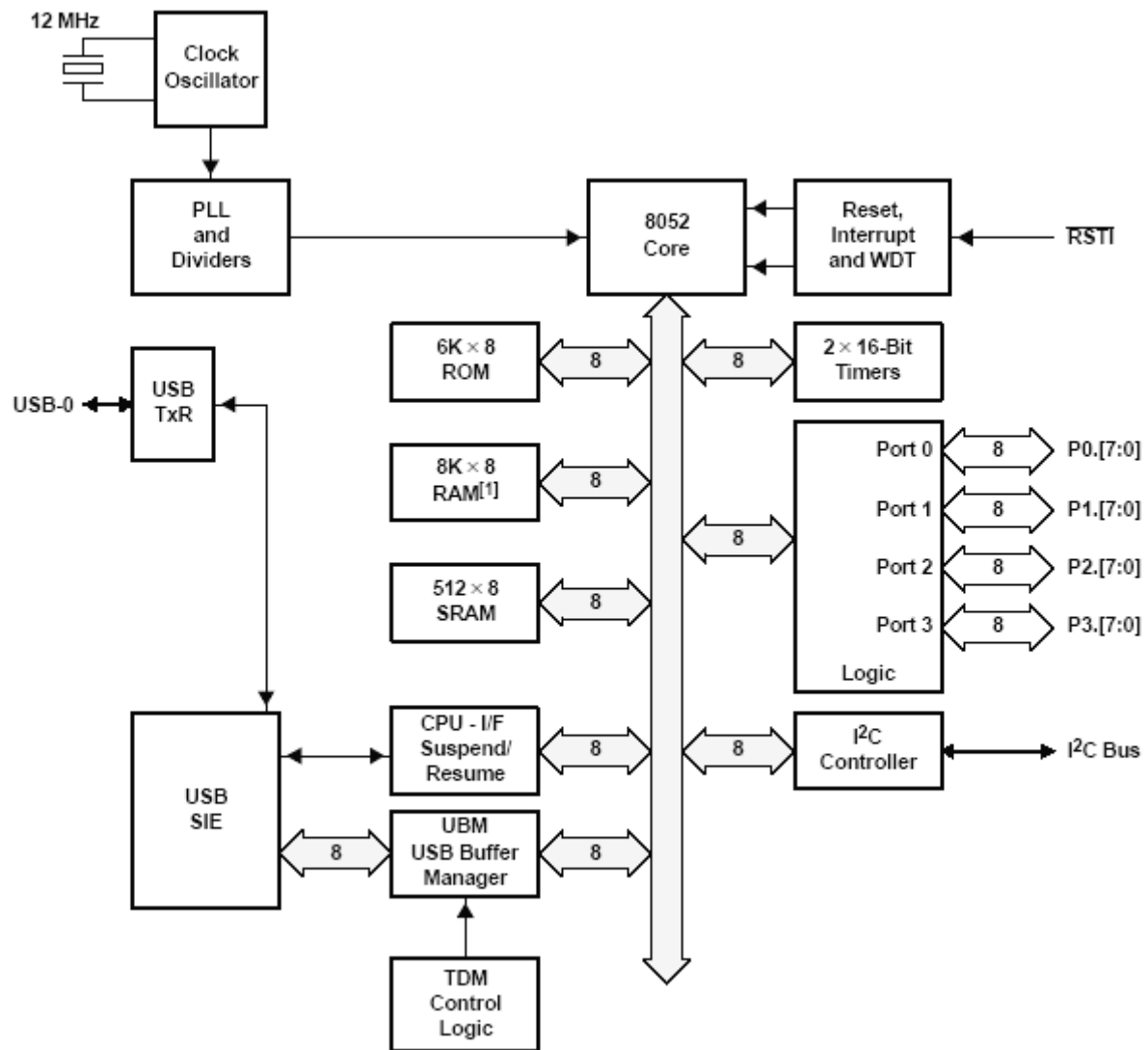
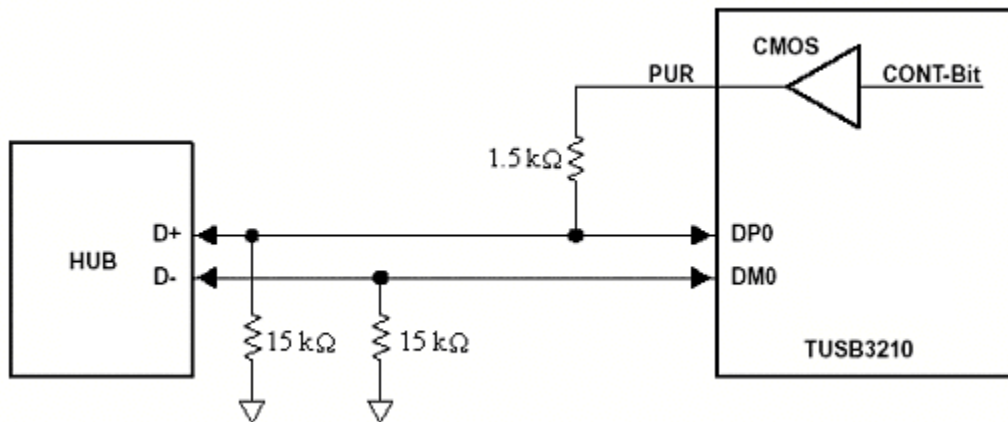


Fig. 4.1 The functional block diagram of the TUSB3210 [5]

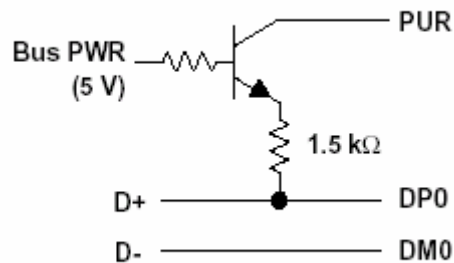
4.1.3 TUSB OPERATIONS

a) Pullup Resistor Connect/Disconnect

After reading firmware into RAM, the TUSB3210 can re-enumerate using the new firmware (no need to physically disconnect and reconnect the cable). Fig.4.2 shows an equivalent circuit implementation for Connect and Disconnect from a USB upstream port. When the CONT bit in the USBCTL register is 1, the CMOS driver sources V_{cc} to the pullup resistor (PUR pin) presenting a normal connect condition to the USB hub (high speed). When the CONT bit is 0, the PUR pin is driven low. In this state, the $1.5\text{ k}\Omega$ resistor is connected to GND, resulting in device disconnection state. The PUR driver is a CMOS driver that can provide $V_{dd}-0.1\text{ V}$ minimum at 8 mA of source current.



(a)



(b)

Fig. 4.2 Pullup Resistor Connect/Disconnect Circuit [5]

b) RESET

There are three requirements for the reset signal timing. First, the minimum reset pulse duration is $100\text{ }\mu\text{s}$. At power up, this time is measured from the time the power ramps up to 90% of the nominal V_{cc} until the reset signal exceeds 1.2 V . The second requirement is that the clock must be valid during the last $60\text{ }\mu\text{s}$ of the reset window. The third requirement is that, according to the USB specification, the device must be ready to

respond to the host within 100 ms. This means that within the 100 ms window, the device must come out of reset, load any pertinent data from the I²C EEPROM device, and transfer execution to the application firmware, if any is present.

These requirements are depicted in Fig.4.3 below. When using a 12 MHz crystal or the 48 MHz oscillator, the clock signal may take several milliseconds to ramp up and become valid after power up. Therefore, the reset window may need to be elongated up to 10 ms or more to ensure that there is a 60 μ s overlap with a valid clock.

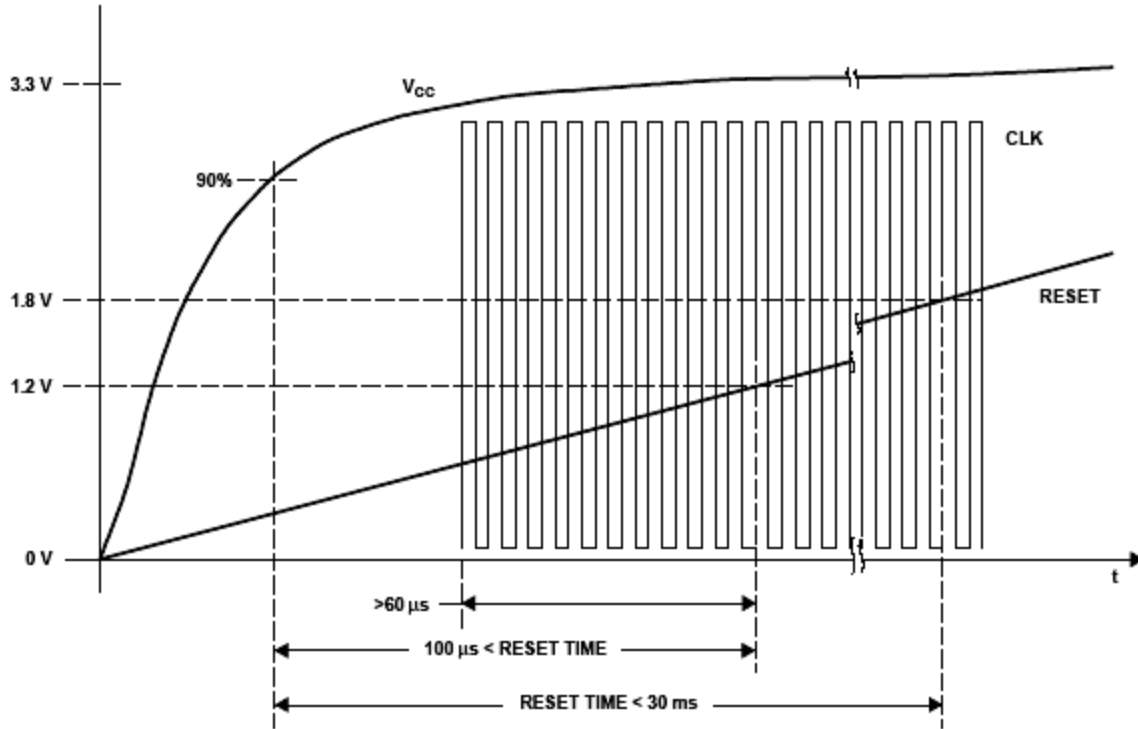


Fig 4.3 Requirements of the reset signal

Implementation of reset

To implement the reset signal, the circuitry and properties of TPS7333 voltage regulator have been exploited. The $\overline{\text{RESET}}$ output of the TPS7333 initiates a reset in microcomputer and microprocessor systems in the event of an undervoltage condition. An internal comparator in the TPS7333 monitors the output voltage of the regulator to detect an undervoltage condition on the regulated output voltage. If that occurs, the $\overline{\text{RESET}}$ output turns on, taking the $\overline{\text{RESET}}$ signal low. It stays low for the duration of the undervoltage condition. Once the undervoltage condition ceases, a 200 ms timeout begins. At the completion of the 200 ms delay, $\overline{\text{RESET}}$ goes high. Fig. 4.4 explains the implementation. The reset pin of TUSB is connected to a circuitry consisting of a reset switch. This circuitry can be supplied a V_{CC} of 3.3 V or directly short to V_{res} of TPS7333 by changing the position of jumper JP5.

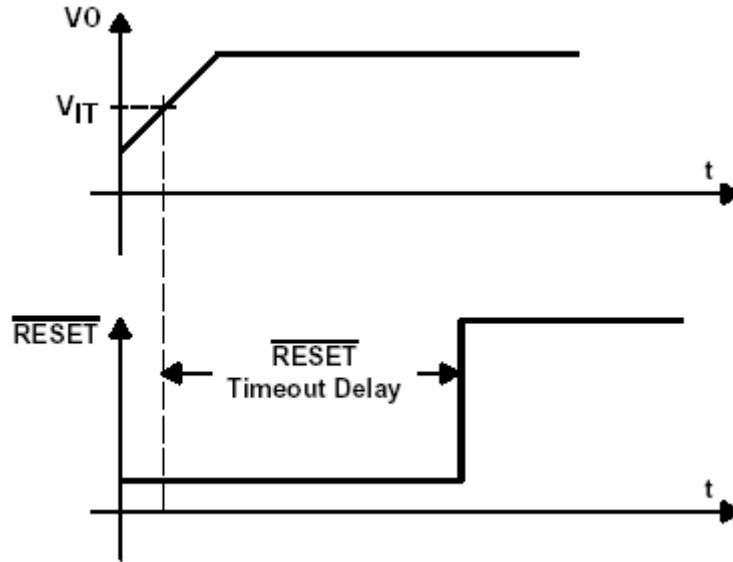


Fig. 4.4 Resetbar waveforms corresponding to TPS7333 [1]

c) SCL/SDA

SCL represents I²C serial clock while SDC represents I²C serial data and is an open drain output. In our case, we aren't using the EEPROM and hence to maintain the functionality they are connected to 3.3 V via 1.5 k Ω resistor.

4.2 FUNCTIONING OF eTBC

The eTBC sources/receives all signals required to master IEEE Std.1149.1 compliant devices viz. TCK, TMS, TDI, and TDO. The eTBC can also source optional TRST signal. Additionally, the eTBC implements high drive output buffers, allowing it to interface directly to targets without need for buffering or any other logic. This subsection gives a brief description of eTBC. For details, please refer its data sheet [2].

4.2.1 PIN DESCRIPTION

A brief description of functionalities of various terminals is given in the table 4.2.

Table 4.2
Pin description of eTBC

Terminal	Description
A2-A0	Address inputs A2-A0 form the 3 bit address bus that interfaces the eTBC to its microcontroller host. These inputs directly index the eTBC register to be accessed.
CLKIN	Clock input. CLKIN is the system clock input for the eTBC. Most operations of the eTBC are synchronous to CLKIN. Internally, the CLKIN signal is divided by a programmable divisor to generate TCK.
D7-D0	Data inputs/outputs. D7-D0 form the 8-bit bidirectional data bus that

	interfaces the eTBC to its microcontroller host.
RDY	Ready output. RDY is used to indicate to the microcontroller host whether or not the eTBC is ready to service the access (read or write) operation that currently is being requested.
$\overline{\text{RST}}$	Reset input. RST is used to initiate asynchronous reset of the eTBC.
$\text{R}/\overline{\text{W}}$	Read/write select. R/W is used by the microprocessor/microcontroller host to instruct the eTBC as to whether it is to perform read access (R/W high) or write access (R/W low).
$\overline{\text{STRB}}$	Read/write strobe. STRB is used by the microcontroller host to instruct the eTBC to initiate (STRB negative edge) or terminate/conclude (STRB positive edge) an access (read or write) operation.
TCK	Test clock. TCK transmits the TCK signal required by the eTBC IEEE Std. 1149.1 target(s). All operations of the TAP are synchronous to TCK.
TDI	Test data input. TDI receives the TDI signal output by the eTBC IEEE Std. 1149.1 target(s). It is the serial input for shifting test data from the target(s); it is sampled on the rising edge of TCK and is expected to be transferred from the target(s) on the falling edge of TCK.
TDO	Test data output. TDO transmits the TDO signal required by the eTBC IEEE Std. 1149.1 target(s). It is the serial output for shifting test data to the target(s); it is transferred on the falling edge of TCK and is sampled in the target on the rising edge of TCK.
TMS	Test mode select. TMS transmits the TMS signal required by the eTBC IEEE Std. 1149.1 target(s). It is the one control signal that directs the next TAP-controller state of the target(s). It is transferred from the eTBC on the falling edge of TCK and is sampled in the target(s) on the rising edge of TCK.
$\overline{\text{TOE}}$	Test-output enable. TOE is the active-low output enable for the eTBC TAP outputs (TCK, TDO, TMS, and TRST).
$\overline{\text{TRST}}$	Test reset. TRST transmits the TRST signal that may be required by some of the eTBC IEEE Std. 1149.1 target(s).

4.2.2 eTBC OPERATIONS

All such tasks as generating TMS sequences, serializing the outgoing bit stream and de-serializing the incoming bit stream are implemented in the eTBC and the microprocessor operates at full 8-bit parallel efficiency and hence test throughput is maximized. The efficiency and flexibility of host is maximized by using eTBC's fully visible status and implementation of ready (RDY) output. RDY goes inactive during a read or write access if the host requested access cannot be performed immediately. Thus, it is used to insert hold or wait states back to the host. When the condition blocking the access clears, the requested access completes. Additionally, all conditions that cause such a blocking are continuously updated in the eTBC status and command registers.

i) TCK GENERATOR

The TCK generator sources the TCK signal required by the IEEE Std.1149.1 target and the eTBC internal test-control logic. The fundamental TCK frequency is produced by division of CLKIN. The divisor is programmable within a range of 1 to 128 in the

While the eTBC is in free-running-TCK mode, if a currently operating command empties or fills a required test data buffer, then the TAP-state generator sources the TMS sequence to move the target device to their Pause-IR or Pause-DR state until the required test-data buffer is serviced appropriately. The TAP controller proceeds through its states based on level of TMS at the rising edge of TCK. There are two main paths through the state diagram, one to access and control the selected data register and one to access and control the instruction register. Only one register can be accessed at any given time.

iii) COMMAND/CONTROL

The command register is used to initiate test operations in the target. Any read to the command register while a command is in progress returns the value written to the command register upon initiation of the command. Once a command finishes, the opcode bit group in the command register is set to null. The opcode bit group (bits 3-0) of the command register determines the test operation to be executed in the target and the end-tap-state bit group (5-4) determines the TAP state in which the target scan chain is left when the command finishes.

Now, upon command initiation test control logic is initialized and TDO and TDI buffers are cleared. So anything that is to be written to these buffers should be written after command initiation. All eTBC commands operate similarly to accomplish test objectives. First, the eTBC generates a TMS sequence to move the target from its current TAP state to a working state. Second, the command is operated in the working state for a number of TCK cycles determined by the value of the counter upon command initiation. Third, the eTBC generates a TMS sequence to move the target scan chain from the working state to the end state specified in the command register. Various commands which describe its operation are mentioned below:

a) *TAP-state movement*

The state-move command operates to generate a TMS sequence to move the target scan chain directly from its current TAP state to the end state specified in the command register.

b) *Run-test command*

The run-test command generates TMS sequences to move the target scan chain from its current TAP state to the Run-Test/Idle state where it remains for a number of TCK cycles determined by the value of the counter upon command initiation. Upon the countdown of the counter to zero, the eTBC generates TMS sequences to move the target scan chain to the end state specified in the command register.

c) *Scan commands*

eTBC commands are provided to perform scan operations to target scan chains. These are classified by the destination of scan data in the target- instruction register, data register- and by the nature /direction of data transfer- input only, output only, etc. The instruction-register (data register) scan commands scan bits to and/or from the instruction (data) register of the target. The eTBC generates a TMS sequence to move the target scan chain from its current TAP state to the Shift-IR TAP state. Data written to the TDO buffer can be driven serially onto the TDO pin and bits received serially at the TDI pin can be stored into the TDI

buffer for reading by the host. For scan commands of input-only class, only the TDI buffer is used to scan data from the target, outgoing scan data is fixed at a high level throughout the scan operation. For scan commands of the output-only class, only the TDO buffer is used to scan data to the target scan chain, incoming TDI data is simply ignored.

Upon the countdown of the counter to zero, the eTBC generates TMS sequences to move the target scan chain to the end state specified in the command register.

5. PROGRAMMING OF THE TARGET DEVICE

5.1. FIRMWARE PROGRAMMING FLOW

After power-on reset, the bootcode copies predefined USB descriptors to the shared RAM. The first USB descriptor is the device descriptor. It describes the embedded function class, vendor ID, product ID, etc. The second USB descriptor is the configuration descriptor, which contains information such as how the device is powered, the number of configurations available, type and number of interfaces, and endpoint descriptors. Once the bootcode finishes copying descriptors, it looks for the EEPROM on the I²C the port. If a valid signature is found, it reads the data type byte. If the data type is application code, it downloads the code to an external data space. Once the code is loaded and the checksum is correct, bootcode releases control to the application code.

After the bootcode updates the hub register and device descriptor, it sets up for a USB transaction and connects itself to the USB. It remains there until the host drivers download the application code. Once complete, it disconnects from the USB and releases control to the application code.

5.1.1. DOWNLOADING THE APPLICATION CODE ONTO THE DEVICE

To download a file/data to the device, we need to pack the data in a proper format, so that the bootcode residing on its ROM can read the data, and load our specific application code. We need to make sure that the number of bytes that can be transmitted are less than 8192 bytes (8k memory). Apart from that, when sending firmware directly to endpoint 1, one should take care that

- the first two bytes specify the length (LSB first);
- the third byte specifies the checksum
- the firmware follows
- each packet can be up to 255 bytes

A small piece of code for downloading the firmware to TUSB3210 is reproduced here:

```
FILE *in;
unsigned char firmware[8195];
len = fread (&firmware[3],1,8192,in); //reading data
fclose (in);
/* Store the length */
firmware[0] = (unsigned char)(len & 0xff);
firmware[1] = (unsigned char)((len & 0xff00) >> 8);
/* Calculate the checksum */
firmware[2] = 0;
for(i=0;i<len;i++)
    firmware[2] += firmware[3+i];
/* Send the header and data to endpoint 1 in 255-byte chunks. */
len += 3;
for(i=0;i<len;i+=255)
    if((retval=usb_bulk_write(tusb,1,&firmware[i],min(255,len-
i),500))<0)
        die ("Error downloading firmware");
}
```


5.1.2. APPLICATION CODE

The application code consists of various interrupt subroutines, which can be exploited to serve our purpose. The firmware should initialize all the configuration registers, as well as give the definitions of all the interrupt service routines. The application code programs the embedded microcontroller as per its requirements. Simple instructions can be mentioned in the application code. One needs to set the appropriate values of pull-up registers, register locations, etc., if required.

We have used bulk transfer of data from host to the device, employing the interrupts provided to utilize microcontroller processing power. Interrupt transfer involves halting the microcontroller, while that interrupt service routine is being processed. As mentioned earlier, TUSB3210 supports 4 endpoint interrupts, but we have used only endpoint1 to transfer data to and from the device.

5.1.3. INTERRUPT SERVICE ROUTINES

After the hardware receives an IN token on IEPl, it either transmits an NAK or data to the host depending on whether the hardware has data or not. If the hardware transmits data to the host, it generates an interrupt to the MCU at the end of data transmission. Therefore, the application firmware senses that data has been sent and it updates the IN buffer with new data. At the end of the transfer we need to reset the NAK bit, and load the value of Input EndPoint Byte Count appropriately for further read operation. *USB_bulk_read on the host generates this interrupt, and thus indicates the completion of transfer from the device.*

If the hardware receives an OUT token on OEPl, it either transmits an NAK, if the NAK bit is not cleared or generates an interrupt to the application firmware. The NAK bit and the interrupts can be cleverly used to perform appropriate operations on receiving of data. *USB_bulk_write generates this interrupt to the hardware, thus initiating the transfer of data.* The host program, in order to send the application code also utilizes USB_bulk_write to download firmware to the device.

```
void OEPlInterruptHandler(void) {
//our packing scheme assumes that the first two bytes send
//contain the length of packet while third byte is the check sum
BYTE bSize, bTemp, bCode;
if(bOEPlRecieveLength == 0) {
    bOEPlRecieveLength = pbOEPlXBufferAddress[0];
    bOEPlRecieveLength += ((int)pbOEPlXBufferAddress[1]<<8);
    bSize = tOutputEndPointDescriptorBlock[0].bEPBCTX &
    EPBCT_BYTECNT_MASK;
    for(bTemp=3; bTemp<bSize; bTemp++) {
        bCode = pbOEPlXBufferAddress[bTemp];
        abOEPlSaveBuffer[bOEPlSaveBufferPosition] = bCode;
        bOEPlSaveBufferPosition++;
    }
}
else
{
    bSize = tOutputEndPointDescriptorBlock[0].bEPBCTX &
    EPBCT_BYTECNT_MASK;
    for(bTemp=0; bTemp<bSize; bTemp++) {
```

```

        bCode = pbOEPlXBufferAddress[bTemp];
        abOEPlSaveBuffer[bOEPlSaveBufferPosition] = bCode;
        bOEPlSaveBufferPosition++;
    }
}

//process the data that has came

tOutputEndPointDescriptorBlock[0].bEPBCTX = 0x00;
//allowing next data to come in
bOEPlSaveBufferPosition=0x00;
return; //clear the interrupt
}

```

On similar lines, we can define the Input End Point1 Interrupt.

```

void IEP1InterruptHandler() {
    //code to be performed after data has been sent to the host
    tInputEndPointDescriptorBlock[0].bEPBCTX=0x04;
    //also specifies the number of bytes for next transfer
    return;
}

```

5.1.4. RS-232 PORT FOR DEBUGGING

Apart from the four LED's provided in the hardware, we can also use RS232 port to monitor the 8052 operations. One of the first steps that the application code must do is to initialize the RS-232 port along with other system initialization processes. The firmware should define functions like rs232PutChar, send_buffers(), etc. which can transfer data to the RS-232 port. The RS-232 port can then be constantly polled for any new data. The three jumpers provided can disconnect the RS-232 completely from the circuit. One needs to set the PUR3_0, PUR3_1, P3_0, P3_1 bits high for proper transfer through the RS-232 port. Host program for reading the data coming from RS-232 is provided in the appendix. On the device side, RS-232 can be programmed by invoking the following function:

```

void rs232Initialization(void) {
    // take care of TMOD, SCON setting, because Timer 0&1 use them
    together
    TMOD &= 0x0f; // Mask Timer 1 high nibble
    TMOD |= 0x20; // Set Timer 1 to mode 2 (AUTO RELOAD)
    SCON = 0x40; // Set serial port for mode 1

    // 11.0592MHz uses & SMOD = 0, 24MHz or above uses & SMOD = 1
    PCON = 0x80; // Set SMOD = 1
    TH1 = RS232_BAUDRATE; // reference header
    TR1 = 1; // enable Timer 1
    TI = 1; // Set Transmit Interrupt flag 1 to transmit ready
    RI = 0; // Set Receive Interrupt flag 0 to receive ready
}

```

5.2. eTBC FUNCTIONING

As mentioned earlier, the function of eTBC is to master an IEEE Std. 1149.1 (JTAG) test access port (TAP) under the command of a host microcontroller, which happens to be embedded on TUSB3210. Conceptually, the eTBCs operate as a simple 8-bit memory or I/O-mapped peripherals to a microcontroller (host). High level commands and parallel data are passed to/from the eTBCs via its generic host interface, which includes an 8-bit bidirectional data bus (D7–D0) and a 3-bit address bus (A2–A0). These high level commands can move the TAP controller from one state to another, performing appropriate scan commands in that state. Read/write select (R/W) and strobe (STRB) signals are implemented so that the critical host-interface timing is independent of the CLKIN period. An asynchronous ready (RDY) indicator is provided to hold off, or insert wait states into a host read/write cycle when the eTBCs cannot respond immediately to the requested read/write operation. The pin mapping of our eTBC (SN74LVT8980A) is shown in Fig. 5.1.

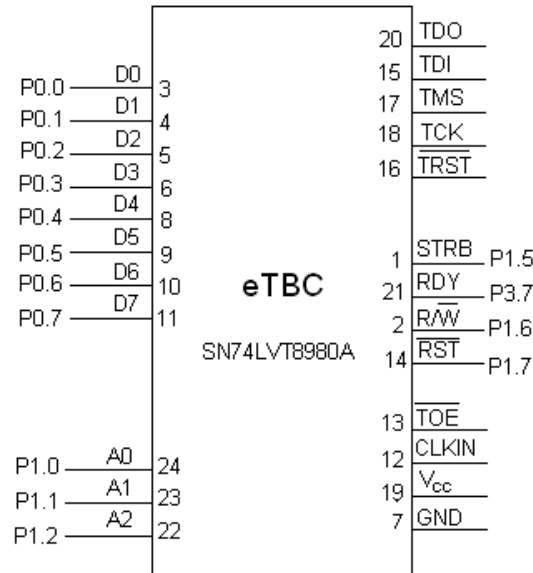


Fig.5.1 Mapping of the pins in SN74LVT8980A

5.2.1 READ/WRITE ACCESS

Interfacing a microcontroller and an eTBC involves Read/Write (R/W), Strobe (STRB) and Ready (RDY) for handshaking. The read/write select (R/W) controls the direction of data flow in the bidirectional data bus. For read access (R/W high), the eTBC data bus outputs are made active on the falling edge of STRB to drive the data in the eTBC register that is selected by address (A2–A0). For write access (R/W low), the eTBC data outputs remain at high impedance, independent of STRB. The register selected by the address (A2–A0) inputs latches the values from the data bus on the rising edge of STRB. The following pieces of code show the read and write operations of eTBC

```

P0=0xff; //input port
setread();
P1_5=0;
temporary=P0;
P1_5=1;

P0=temporary; //output port
setwrite();
P1_5=0;
P1_5=1;

```

5.2.2. REGISTER DESCRIPTION

The address pins (A2-A0) directly index the eTBC's eight read/write registers. Each address pin can be addressed to select any of the registers.

Table 5.1
Register summary

Address	Register	Host Access
000	Configuration A	R/W
001	Configuration B	R/W
010	Status	R
011	Command	R/W
100	TDO buffer	R/W
101	TDI buffer	R
110	Counter	R/W
111	Discrete Control	R/W

A brief overview of the registers available is given below (details are available in [3]):

1. Configuration A:

The eTBC operates under the influence of configuration A register by selecting the modes like automatic/free-running-TCK, automatic/gated-TCK, or discrete-control modes. The register is fully readable at all times and is fully writeable except when an eTBC command is in progress.

2. Configuration B

This register allows software control of the TCK output frequency based on a division of CLKIN input. Moreover, retiming delays can be introduced, which can be used to de-skew the TAP signals to target scan chains that are electrically distant (due to cabling delays, etc).

3. Status

The status of the eTBC is reported fully and updated continuously in the status register. The TDI-buffer status & TDO-buffer status report the readiness of TDI buffer to a host read and that of the TDO buffer to a host write.

4. Command

Apart from providing the software reset, this register can also be used to issue high level commands by the host to cause the eTBC to generate the TMS sequences necessary to move the test bus from any stable TAP-controller state to any other such stable state.

5. TDO buffer

The TDO buffer is the 4×8 -bit-parallel-to-serial FIFO that accepts scan data from the host in 8-bit-parallel format and serializes it onto the TDO pin during scan operations. Scan data is expected to be transferred from the host in least-significant-byte-first order to meet IEEE Std. 1149.1 requirements for LSB-first scan order. Any partial byte to be written should be justified to D0. The TDO buffer is cleared upon command initiation, so no scan data should be written to the TDO buffer before writing a scan command to the command register.

6. TDI buffer

The TDI buffer is the serial to 4×8 -bit-parallel FIFO that serially receives data at the TDI pin and makes it available in 8-bit-parallel format for reading by the host. Scan data is expected to be transferred from the IEEE Std. 1149.1 targets in LSB-first order and is made available for host read in least-significant-byte-first order. The last data available for host read during a scan command may be a partial byte, in which case it is justified to D0.

7. Counter

The counter register, while only 8 bits wide like any other eTBC register, provides read/write access to the full 32-bit eTBC counter. Writes to the counter register are accomplished by four complete host access cycles.

8. Discrete control.

The discrete-control register is used to program the state of the TAP outputs (TCK, TDO, TMS and TRST) and to poll the state of the TAP input (TDI) when the eTBC is in its discrete-control mode.

5.3. PROGRAMMING OF CPLD

All the IEEE Std. 1149.1 target devices have JTAG compatibility. Such devices can be programmed or erased by accessing their appropriate registers and then sending the required commands. A simple JTAG programming will involve the following steps:

1. Check whether the device is connected
2. Load the bypass register
3. Enable in system programming (ISP Enable)
4. Erase the device
5. Load the program
6. Verify (if required)
7. Disable in system programming (ISP Exit)

We successfully programmed an XC9572XL_PC44 CPLD using its JTAG port. Complex Programmable Logic Device (CPLD) provides a simple low density programmable logic solution. These logic devices are characterized by an architecture offering high speed, predictable timing and simple software. The in-system programming of CPLD requires an understanding of their BSDL files. All IEEE Std. 1149.1 target devices support boundary scan logic and have a *boundary scan description file (BSDL)*. These BSDL files contain the opcode for instruction registers like IDCODE (Identification Code), ISPEN (ISP Enable), etc. The data and instruction registers of these devices can be accessed by proper movement in the TAP state machine, and a data register scan is done to read back or write into that data register.

The data to be programmed into a CPLD is stored in its 'jedec' file. Jedec file shows the location of all the transistors in the array which need to be fused. These locations are displayed by 1's and 0's. We can use our jed2svf program or XILINX impact tool to convert the jed file to svf file. The serial vector format (SVF) files contain the actual data that needs to be sent through the TDO pins of eTBC. The svf file shows the bit stream that will be required to program the CPLD. The svf file also contains the proper order in which these bits must be sent to the target device to program or erase it. This data is used to move the eTBC from one state to another. Every scan operation of CPLD through eTBC requires that the counter is loaded. Next we load the command

register, and finally send the data required for IR scan or DR scan through the TDO pins. We divide the svf file into small chunks which in itself could do a complete scan operation. That is to say, each chunk consists of a counter value, along with the command register opcode and the data (TDO).

5.3.1. APPLICATION FIRMWARE CODE

In the initial part of the firmware the eTBC was loaded with Configuration A and B values. The rest of the data was sent to eTBC in chunks using the interrupt handlers of TUSB. As soon as new data arrives to the TUSB it sets the NAK bit which is reset at the end of the interrupt handler routine. A simple flow chart to describe this process is shown in Fig. 5.2.

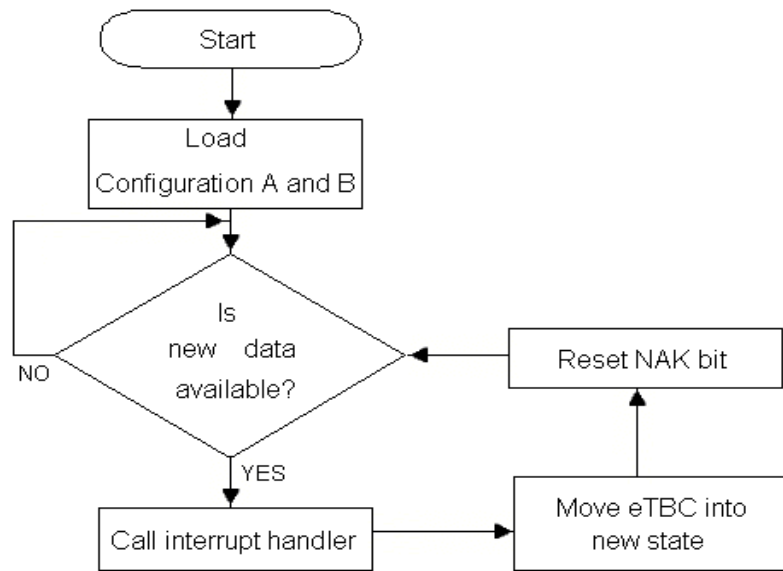


Fig.5.2 A flow chart for the firmware code

The code for the interrupt handler is as below:

```

void OEPlInterruptHandler(void) {
    BYTE bSize, bTemp, bCode;
    int count, tdobytes, delaypresent;
    BYTE temp=0xff;

    if(bOEPlRecieveLength == 0) {
        bOEPlRecieveLength = pbOEPlXBufferAddress[0];
        bOEPlRecieveLength += ((int)pbOEPlXBufferAddress[1]<<8);
        bSize = tOutputEndPointDescriptorBlock[0].bEPBCTX &
        EPBCT_BYTECNT_MASK;
        for(bTemp=3; bTemp<bSize; bTemp++) {
            bCode = pbOEPlXBufferAddress[bTemp];
            abOEPlSaveBuffer[bOEPlSaveBufferPosition] = bCode;
            bOEPlSaveBufferPosition++;
        }
    }
}
  
```

```

    }
else {
    bSize = tOutputEndPointDescriptorBlock[0].bEPBCTX &
    EPBCTX_BYTECNT_MASK;
    for(bTemp=0; bTemp<bSize; bTemp++) {
        bCode = pbOEPlXBufferAddress[bTemp];
        abOEPlSaveBuffer[bOEPlSaveBufferPosition] = bCode;
        bOEPlSaveBufferPosition++;
    }
}

//code for moving the state of TAP controller
//first byte = 0 => Full instruction without delay
//first byte = 1 => Full byte instruction byte with delay
//if first byte = 0
//next 4 bytes counter followed by command register's opcode

count=0;
P2=0xff;
if(abOEPlSaveBuffer[count]==0) {
    P2_3=0;
    count=1;
    tdoBytes=setcounter(count);
    count=count+4;                //6th byte counter
    setcommand(count);
    count++;
    settdo(count,tdoBytes);        //and finally the tdo bytes
    count=count+tdoBytes;
}
else {
    count=1;
    P2_2=0;                        //to show delay loop
    givedelay(count);
    count=count+2;
}
P2_0=~P2_0;
tOutputEndPointDescriptorBlock[0].bEPBCTX = 0x00;
//waits for next packet of instruction
bOEPlSaveBufferPosition=0x00;
return;
}

```

5.3.2. HOST SIDE PROGRAMMING

We need to send the svf file to TUSB, but such huge file cannot be sent all in one piece. Hence, we need to divide the svf file into small pieces. Each piece is then programmed to obtain a chunk of data as shown in table 5.2.

These chunks were made using a *Perl* script which takes a svf file as an input and generates a temporary file. A file called *counter.txt* is also created which contains the size of each chunk. Host side program must read these chunks and send them to the TUSB, which processes it for further use.

Table 5.2
Operation of svf2temp script

Piece of Instruction from svf file	Chunk of data	Explanation
SDR 32 TDI(aabbccdd)	00	00 → not a delay function 01 → a delay function
	20	This gives a 32 bit counter value and it is written starting with the least significant byte
	00	
	00	
	00	
	18	The value to be loaded in command register
	dd	The TDO data to be loaded starting from the least significant byte
	cc	
	bb	
	aa	

The complete send program is provided in the appendix. A piece of send code is produced here:

```

firmposition=0;          //keeps track of the number of bytes sent
for(i=0;i<countlen;i++)  //mycount keeps track of size of each
chunk
{
    if((retval=usb_bulk_write(tusb, 1, &firmware1[firmposition],
mycount[i], 500000)) < 0)
    {
        printf("retval %i\n", i);
        die("Error downloading firmware");
    }
    firmposition+=mycount[i];
}
}

```

After the firmware is loaded in the TUSB, the send program is executed on the host side, which changes the state of the TAP controller accordingly. Complete programming flow is shown in Fig. 5.3

5.3.3. ERASING THE CPLD

To erase the XC9572XL_PC44, the procedure remains the same as explained above. Again we use the svf file generated by either the jed2svf code or by the impact software. Once we have the svf file we convert it into chunks and then send them to the TUSB by the send program. Firmware as well as the send program remains same because they are designed so as to enable the TAP controller to move from one state to another; states being determined by the chunks that the host sends.

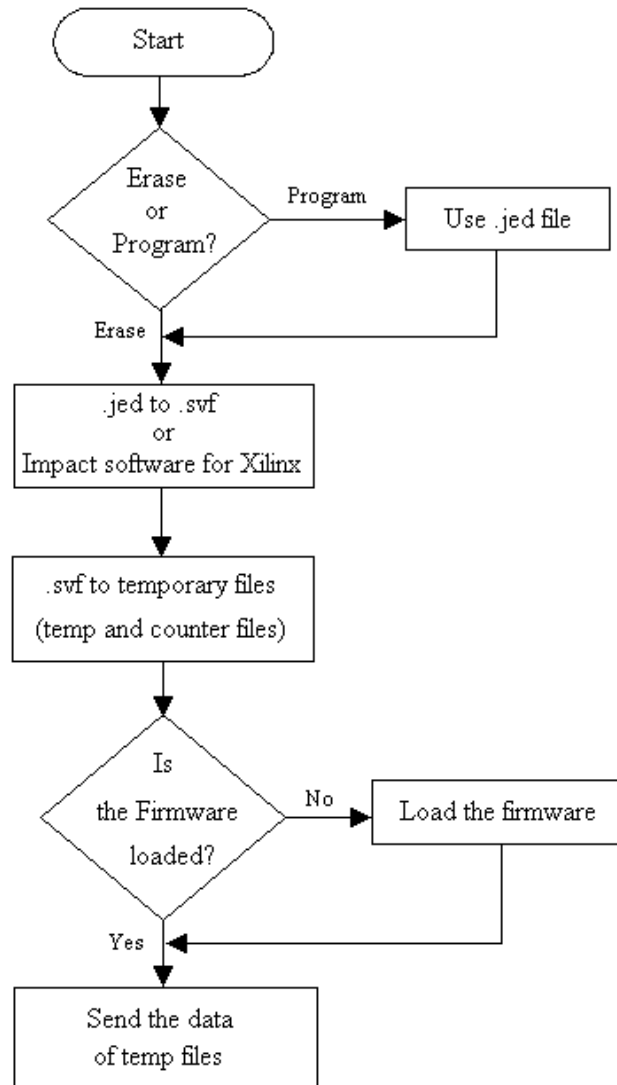


Fig. 5.3 The programming flow followed

5.3.4. VERIFYING THE CPLD

To verify whether the CPLD has been programmed successfully or not, the instruction opcode for the verify instruction is used. It reads back the state of transistors (whether fused or not) at the specified address. The algorithm for this involves loading the bypass register, followed by ISPEN and the verify instruction. In the next step we send the 50 bit configuration register that we used while programming the device. The next input scan returns the state of fuses for the specified location given in the configuration register loaded.

To implement the above scheme, we need to modify the interrupt handler such that it checks the returned value of the configuration register from the device. The send program on the host side should keep sending the chunks of data, each chunk representing 1 scan operation, and consists of a counter value, command register value and the data or instruction for that command.

6. CONCLUSIONS

The project gave a good overview of the communication through the JTAG port. The PCB, as described, was implemented and gave the results expected. We could program as well as erase and verify the program loaded onto CPLD (XC9572XL_PC44). This JTAG programmer can be used to program any IEEE Std. 1149.1 compliant target if the BSDL file of that target is available. The USB interface provided to the device makes it easier to program as well as read back the device configuration registers. Thus our hardware can program devices, removing the requirement of complex, multi purpose and highly expensive JTAG emulators available in the market.

FUTURE SCOPE

The hardware produced can be used to program any IEEE Std. 1149.1 compliant device. All that remains is to write down appropriate software for sending proper instructions to that particular target device, based on its BSDL file. In our case we could develop the software for XC9572XL_PC44 CPLD, using its BSDL file. This software can be used as a basic building block for more complex codes that can program any device. Further, this hardware can be used to program a chain of such devices, but this capability of the hardware is yet to be tested.

REFERENCES

- [1] Texas Instruments, “Data sheet for TPS7333”, <http://www-s.ti.com/sc/ds/tps7333.pdf>
- [2] Texas Instruments, “Data sheet for SN74LVC1404”,
<http://www-s.ti.com/sc/ds/sn74lvc1404.pdf>
- [3] Texas Instruments, “Data Sheet for SN74LVT8980A”,
<http://focus.ti.com/lit/ds/symlink/sn74lvt8980a.pdf>
- [4] Texas Instruments, “Data Sheet for MAX3232”,
<http://focus.ti.com/lit/ds/slls410i/slls410i.pdf>
- [5] To Understand TUSB3210
 - a. Texas Instruments, “TUSB3210 keyboard product development kit”,
<http://www.ti.com/sc/tusb3210kbdpdk>
 - b. Texas Instruments, “ 8052 embedded microcontroller features”,
<http://focus.ti.com/lit/ug/sllu020a/sllu020a.pdf>
 - c. Texas Instruments, “ TUSB3210 bootcode document for USB to general purpose device controller”, <http://focus.ti.com/lit/ug/sllu025a/sllu025a.pdf>
- [6] To Understand JTAG devices and Boundary Scan
 - a. Texas Instruments, “All about JTAG devices”, <http://www.ti.com/sc/jtag>
 - b. Texas Instruments, “BSDL file format”,
<http://www.ti.com/sc/docs/general/logic/lit/bsl.pdf>
- [7] Data sheet for LM7805CV
<http://www.fairchildsemi.com/ds/LM/LM7805.pdf>
- [8] For BSDL files,
 - a. Xilinx, “BSDL files for Xilinx CPLDs”,
<ftp://ftp.xilinx.com/pub/swhelp/bsdl/x9500.zip>
 - b. Actel, “Format of the BSDL files”,
<http://www.actel.com/documents/BSDLformat.pdf>
- [9] For Microcontroller programming
The 8051 Microcontroller: Architecture, Programming and Applications, *Kenneth J Ayala*, Thomas Learning Publications.

APPENDIX

Appendix A: The send program to transmit chunks of data from the host to TUSB

This program reads the file supplied as an argument and sends the read bytes after packing them into chunks. The size of each chunk is read from the counter.txt file. The code is written in line with the bootloader program available on TI website. To compile the code, libusb should be installed on the host PC and reference for its library should be mentioned while making the executable from this code.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <usb.h>
#include "tusb.h"
#ifndef min
#define min(a,b) (((a)<(b))? (a):(b))
#endif
#define die(x) do { perror(x); return 1; } while(0)

int chartoint(unsigned char input)
{
    int returnvalue;
    returnvalue=input-48;
    if(returnvalue<10)
        return returnvalue;
    else
        return (returnvalue-39);
}

int convert(int b,int a)
{
    int returnvalue;
    a=a<<4;
    returnvalue=b|a;
    return returnvalue;
}

int main(int argc, char *argv[])
{
    FILE *in;
    FILE *cf;
    int I,len,retval,finallen,countlen,countlentemp,firmposition;
    unsigned char firmware[8195];
    unsigned char firmware1[8195];
    unsigned char mycount[8195]; //this part imposes
    unsigned char mycount1[8195]; //a constraint on the size of
    unsigned char temp2,temp1; //file to be transmitted
    if(argc!=2) {
        fprintf(stderr,"Usage: %s filename\n",*argv);
        fprintf(stderr,"Sends chunks of data\n");
        return 1;
    }

    if((in=fopen(argv[1],"rb"))==NULL)
```

```

    die(argv[1]);

if(fstat(fileno(in),&s)<0)
    die(argv[1]);

if((i=tusb_init())!=0) {
    fprintf(stderr,"Error initializing USB\n");
    fclose(in);
    return i;
}

/* When sending data directly to endpoint 1:
   - the first two bytes specify the length (LSB first);
   - the third byte specifies the checksum
   - the data follows
   - each packet can be up to 255 bytes.
*/

/* Read the data */
cf=fopen("counter.txt","r");
countlentemp=fread(&mycount1[0],1,8192,cf);
countlen=0;
for(i=0;i<countlentemp;i++)
{
    if(mycount1[i]!=10)
    {
        if(mycount1[i+1]==10)
            mycount[countlen]=(unsigned char)chartoint(mycount1[i]);
        else
        {
            temp1=(unsigned char)(chartoint(mycount1[i]));
            temp2=(unsigned char)(chartoint(mycount1[i+1]));
            mycount[countlen]=convert(temp2,temp1);
            i++;
        }
        countlen++;
    }
}

len=fread(&firmware[3],1,8192,in);
fclose(in);
if(len==0) {
    fprintf(stderr,"Error: input file is empty!\n");
    return 1;
}
else if(len<0) {
    die("Error reading input file");
}

finallen=3;
for(i=3;i<len+3;i=i++)
{
    if(firmware[i]!=10)
    {
        temp1=(unsigned char)chartoint(firmware[i]);
        temp2=(unsigned char)chartoint(firmware[i+1]);
        firmware1[finallen]=convert(temp2,temp1);
    }
}

```

```

        finallen++;
        i=i+1;
    }
}
/* Store the length */
firmware1[0] = (unsigned char)(((finallen-3) & 0xff));
firmware1[1] = (unsigned char)((((finallen-3) & 0xff00) >> 8));

/* Calculate the checksum */
firmware1[2]=0;
printf("firmware1[0]=%d\n",firmware1[0]);
printf("firmware1[1]=%d\n",firmware1[1]);
for(i=3;i<finallen;i++)
{
    firmware1[2] += firmware1[i];
    printf("i=%d, value=%d\n",i,firmware1[i]);
}

/* Send the header and data to endpoint 1 in 255-byte chunks. */
firmposition=0;
for(i=0;i<countlen;i++)
{
    if(i==0)
    {
        if((retval=usb_bulk_write(tusb, 1, &firmware1[0], mycount[i]+3,
500000)) < 0)
        {
            printf("retval %i\n", i);
            die("Error downloading firmware");
        }
        firmposition=mycount[i]+3;
    }
    else
    {
        if((retval=usb_bulk_write(tusb, 1, &firmware1[firmposition],
mycount[i], 500000)) < 0)
        {
            printf("retval %i\n", i);
            die("Error downloading firmware");
        }
        firmposition+=mycount[i];
    }
    printf("Sent %d\n",firmposition);
}

printf("Wrote %d bytes.  Data send.\n",firmposition);
return 0;
}

```

tusb.c and tusb.h – Other files required while compiling

It contains the various vendor and device specific details of TUSB3210 and is used to initiate the communication with the device.

tusb.h – Header file required by send program

```

#ifndef TUSB_H
#define TUSB_H

```

```

#include <usb.h>

typedef unsigned short u16;
typedef unsigned char u8;
extern usb_dev_handle *tusb;

int tusb_init(void);
void tusb_close(void);

#define BTC_GET_BOOTCODE_STATUS          0x80
#define BTC_EXECUTE_FIRMWARE            0x81
#define BTC_GET_FIRMWARE_REVISION        0x82
#define BTC_PRE_UPDATE_HEADER            0x83
#define BTC_UPDATE_HEADER                0x84
#define BTC_REBOOT                       0x85
#define BTC_FORCE_EXECUTE_FIRMWARE       0x8F
#define BTC_EXTERNAL_MEMORY_READ         0x90
#define BTC_EXTERNAL_MEMORY_WRITE        0x91
#define BTC_I2C_MEMORY_READ              0x92
#define BTC_I2C_MEMORY_WRITE             0x93
#define BTC_INTERNAL_ROM_MEMORY_READ     0x94

#endif /* TUSB_H */

tusb.c file – Header file required by bootloader
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <usb.h>
#include "tusb.h"

#define die(x) do { perror(x); return 1; } while(0)

usb_dev_handle *tusb = NULL;

void close_tusb(void) {
    if(tusb!=NULL)
        usb_close(tusb);
}

int tusb_init(void) {
    struct usb_bus *bus;
    struct usb_device *dev;
    struct usb_device *tusb_dev = NULL;
    int x;

    usb_init();
    if(x = usb_find_busses()) {
        printf("Busses found: %i\n", x);
        // die("Bus scan");
    }
    if(x = usb_find_devices()) {
        printf("Devices found: %i\n", x);
        // die("Device scan");
    }
}

```

```

    for (bus = usb_busses; bus; bus = bus->next) {
        for (dev = bus->devices; dev; dev = dev->next) {
            if(dev->descriptor.idVendor==0x0451 && /* TI */
                (dev->descriptor.idProduct==0x3210 ||
                 dev->descriptor.idProduct == 0x2136)
                ) { /* TUSB2316/3210 */
                if(tusb_dev!=NULL) {
                    fprintf(stderr,"Multiple TUSB2316/3210s found, bailing
out.\n");
                    return 1;
                }
                tusb_dev=dev;
            }
        }
    }
    if(tusb_dev==NULL) {
        fprintf(stderr,"TUSB2316/3210 not found.\n");
        return 1;
    }
    if((tusb=usb_open(tusb_dev))!=NULL) {
        fprintf(stderr,"Error opening USB device.\n");
        return 1;
    }
    /* Setting the configuration and interface isn't really necessary
       since there are only one of each in the boot ROM, but it's a good
       way to make sure we can send data to the device. */

    if(usb_set_configuration(tusb, 1)) {
        perror("Can't set configuration");
        fprintf(stderr,"(make sure you have write access to the
device)\n");
        return 1;
    }
    if(x = usb_claim_interface(tusb, 0))
        printf("Claim interface %i\n", x);
    //die("Can't claim interface");

    return 0;
}

```

Other requirements of send program:

A libusb version-0.1.7 or greater needs to be installed for compiling the send.c file. A make file should also mention the location of libraries required by send program. The make file for compiling send.c will look like this:

```

bootload      :      bootload.c bootload.o tusb.c tusb.o
                  gcc -L/root/libusb-0.1.8/.lib/ -lusb -o bootload
bootload.o tusb.o

tusb.o        :      tusb.c tusb.h
                  gcc -I/usr/local/include -L/root/libusb-0.1.8/.lib/ -
c tusb.c

bootload.o    :      bootload.c
                  gcc -I/usr/local/include -L/root/libusb-0.1.8/.lib/ -
c bootload.c

clean         :      rm myusb.o

```


Appendix B: Using RS-232 as a debuggers resource

A serial interface for reading data from the firmware is provided with the device. Firmware must define functions like `rs232PutChar(char)`, to send data through the RS-232 port. On the host side one can use the `getbuf` program to monitor the inputs coming from RS-232 port. Care must be taken to set the P3_0 and P3_1 pins to high in the firmware before using RS-232 port for debugging purposes.

getbuf.c – File to be compiled on the host side

```
#include <stdio.h>
#include <termios.h>
#include <fcntl.h>
#include <limits.h>
#define BAUDRATE B9600
#define TARGET_PORT "/dev/ttyS0"

int open_port(char *port, speed_t baudrate) {
    int fd;
    char *target_port;
    struct termios options;

    /* Set the target port */

    fd = open(port, O_RDWR | O_NOCTTY);
    if (fd < 0) {perror(port); exit(-1); }

    tcgetattr(fd, &options);

    cfsetospeed(&options, baudrate);
    cfsetispeed(&options, baudrate);

    options.c_cflag &= ~CSIZE; // Mask the character size bits
    options.c_cflag |= CS8; //Select 8 data bits
    options.c_iflag = (IGNPAR | IGNBRK);
    options.c_iflag &= ~(IXON | IXOFF | IXANY);
    options.c_cflag |= (CLOCAL | CREAD | CS8 | CSTOPB);
    options.c_cflag &= ~HUPCL; /* Will not drop DTR on hangup */
    options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    options.c_oflag &= ~OPOST;
    options.c_cc[VMIN] = 0;
    options.c_cc[VTIME] = 1;

    tcsetattr(fd, TCSADRAIN, &options); //changed TCSANOW
    tcflush(fd, TCIOFLUSH);
    return(fd);
}

/* Close the serial port */
void close_port(int fd) {
    tcflush(fd, TCIOFLUSH);
    close(fd);
}

void printbuff() {
    int nread_bytes = 0;
    unsigned char read_array[130];
```

```

int fd;
int i;
struct termios options;

fd = open_port(TARGET_PORT, BAUDRATE);
if(fd<0)
    printf("ERROR\n");

tcgetattr(fd, &options);
options.c_cc[VMIN] = 3;      //minimum number of bytes to read
options.c_cc[VTIME] = 100;   //minimum amount of time to wait
tcsetattr(fd, TCSANOW, &options);
for (i=0;i<128;i++)
    read_array[i]=0xff;
nread_bytes += read(fd, &read_array[0], 128);
for(i=0; i<128; i++)
    printf("%04x\t%02x\n", i+0x80, read_array[i]);
printf("%i\n", nread_bytes);
}

void main() {
    printbuff();
}

```

A simple compilation of this program using gcc will make the executable, which will wait for data from the RS-232 port. On the firmware, functions like rs232PutChar, send_buffers, rs232Initailization, etc. must be defined. A detailed function list used by RS-232 on the device side is given at www.ti.com.