Fall 2013

Communication protocols for integrated fiber optic sensor interrogator

Tameem Ahmed Khan

Department:

COMMUNICATION PROTOCOLS FOR

INTEGRATED FIBER OPTIC SENSOR INTERROGATOR

by

TAMEEM AHMED KHAN

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER ENGINEERING

2013

Approved by

Dr. Hai Xiao (Advisor)
Dr. Maciej Zawodniok
Dr. Minsu Choi

ABSTRACT

Fiber optic sensors have gained rapid recognition in sensing environmental factors such as strain, stress, acoustics, pressure, temperature among emerging other sensing applications. Major advantages of fiber based systems are small size, light weight, multiplex-ability of sensors on a single fiber, and immunity to Radio Frequency (RF) interference. However, single most major disadvantage of such technology is that traditionally the interrogation systems are typically bulky and heavy lab only equipment. Embed-ability of sensors into different structures and environments requires that the whole systems (sensors + interrogator) be small, modular and portable.

This thesis presents such a custom Fiber optic Sensor Interrogator (FSI) based on a Low-power TMS320C6748 applications Processor (DSP) and Virtex®-5 FPGA hardware. Specific objectives for such FSI include; first, to develop a management module that could compare and set FLASH parameters for LASER Controller, second to develop robust code for data collection through a DRP2E Photo Detector, third to establish communication protocols including Serial, USB, Ethernet and Wireless (both on FSI and PC end).

ACKNOWLEDGMENTS

I am extremely thankful to my advisor Dr. Hai Xiao for his support and the knowledge imparted by him, which not just helped in my thesis but also in my overall development as a student. I would also like to thank him for supporting my work with graduate research assistantship.

I thank Dr. Maciej Zawodniok and Dr. Minsu Choi in having trust in me for timely completion of this work and serving on my committee.

I am thankful to my team at Photonics Technology Lab who time to time taught me basics and advanced topics in laser physics and extended their support in understanding the existing lab equipment.

I also thank my wife Adviya Saba Khan for her support on the personal end without which it would have been things would have been different for me. I also thank my mother Arifa Begum for support she extended by joining hands with us in our endeavors.

Finally, I would like to thank my fellow graduate students who always gave a helping hand when I needed one.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

Page

NOMENCLATURE

Symbol	Description
FBG	Fiber Bragg Grating
FPGA	Field-Programmable Gate Array
DSP	Digital Signal Processor
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
FIFO	First In First Out
IIR	Infinite Impulse Response

1. INTRODUCTION

With increasing demand on the sensor based monitoring for structural integrity sensing, harsh chemical sensing or any other on-site sensing system, the necessity for a compact modular yet cost / power effective sensor interrogator and communications systems is becoming indispensible.

Electrical sensors have for decades been the default mechanism for measuring various phenomena. Despite their simple structure, these sensors have inherent limitations such as transmission loss, power and susceptibility to electromagnetic interference (EMI) that make their usage challenging or impractical in many applications. Fiber-optic sensing is an excellent solution to these challenges, using light rather than electricity and standard optical fiber in place of copper wire.

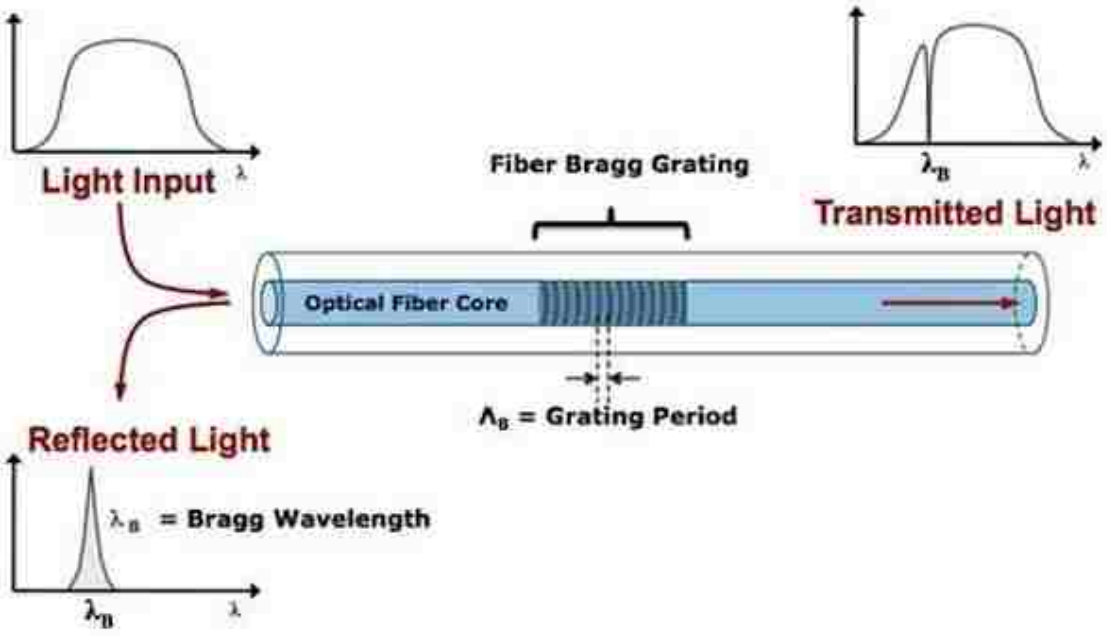


Figure 1.1 Structure of the fiber optical sensor [2]

Fiber optical sensing overcomes many of the challenges commonly found with electrical sensing by using light (laser light in this project) rather than electricity and standard optical fiber in place of copper wire. Optical fibers and FBG optical sensors are nonconductive, electrically passive, and immune to EMI.

Interrogation with a high-power tunable laser enables measurements that can run over long distances with little or no loss in signal integrity hence making them best suited for remote sensing. Also, unlike electrical sensing systems, each fiber optical channel has the capability to measure multiple FBG sensors, hence reducing the size, weight, and complexity of the measurement system.

Further, lack of flexibility limits the system's ability to meet many structural test and monitoring application needs. This project realizes a solution for FSI which is small in size, robust in communication and to an extent scalable.

2. SETUP

2.1. HARDWARE

The hardware design includes multi layered board layout. The first level is the DC power distribution board. Second level is the Logic / Signal processing (including FPGA, Flash, PROM, COM, USB, Ethernet and WiFi mega modules). Third level onwards a maximum of four laser driver boards can be connected. Each laser board will have one laser chip and maximum of two photo diodes that can detect the transmitted or reflected light.

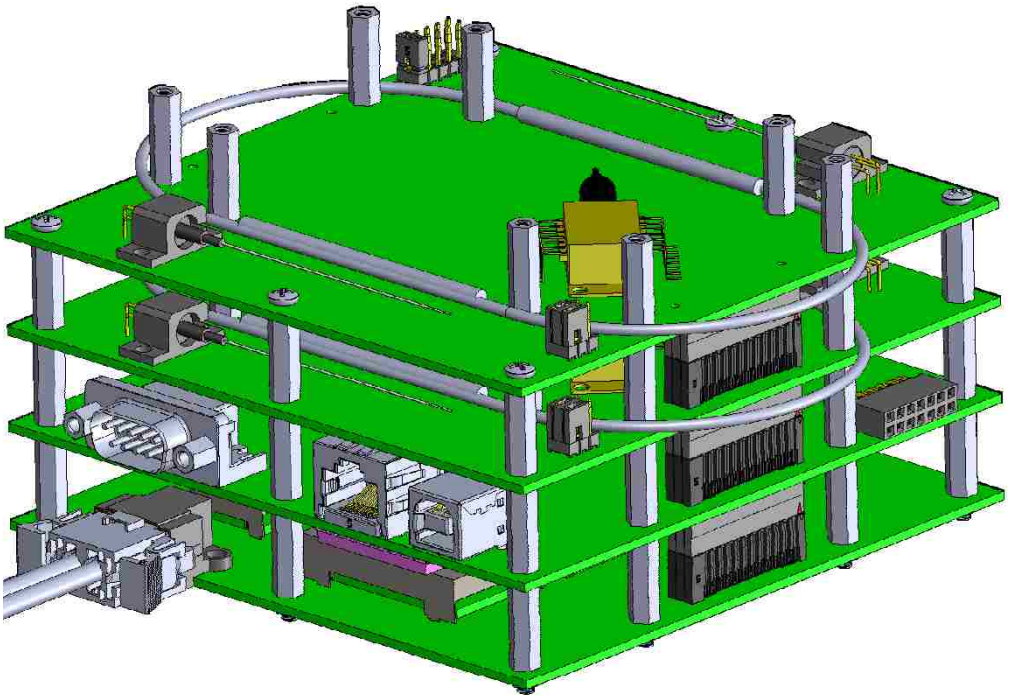


Figure 2.1 Multi layered board layout

Below two layouts show details on the parts of the board layout discussed throughout this report that participate in communication.

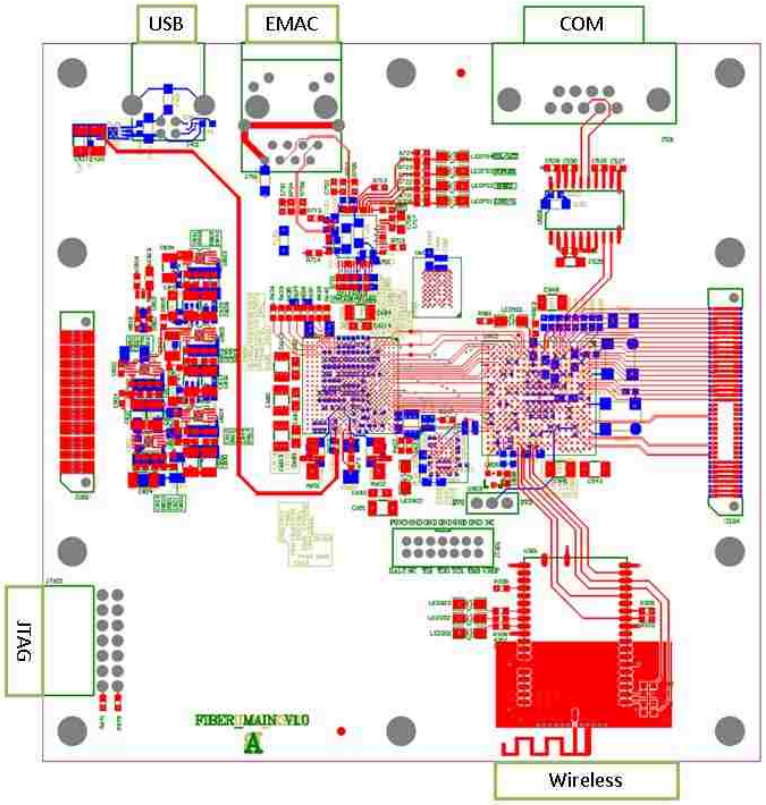


Figure 2.2 PCB layout of main board

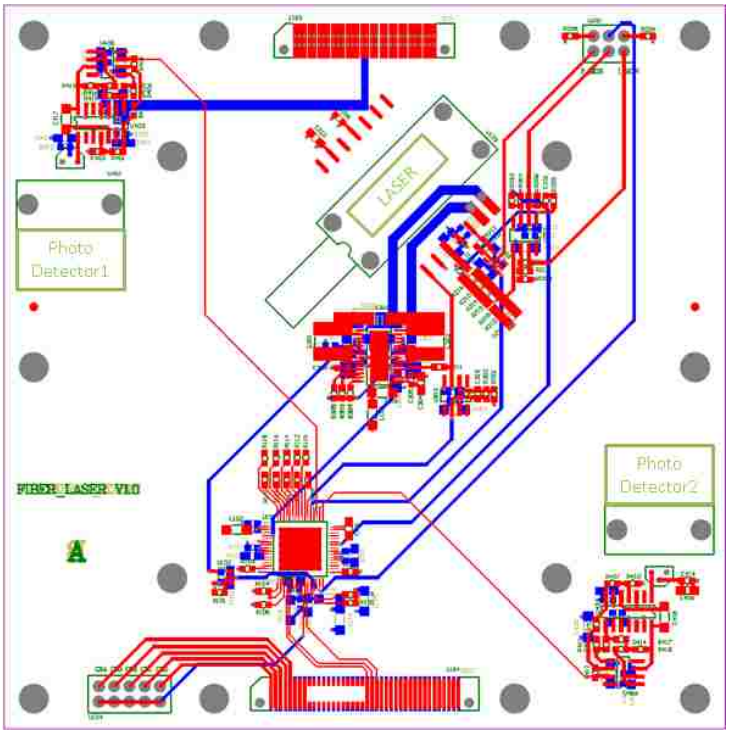


Figure 2.3 PCB layout of LASER board

a. Laser module

The laser module D2570H [8] used has following properties:

- ITU wavelengths available from 1528.77 nm —1610.06 nm
- Temperature tunable for precise wavelength selection

b. Photo detector

The photo detector DRP2EX - 43T2 [7] has following properties

- Bias voltage: 5.0 V
- Wavelength: 1520nm – 1570nm

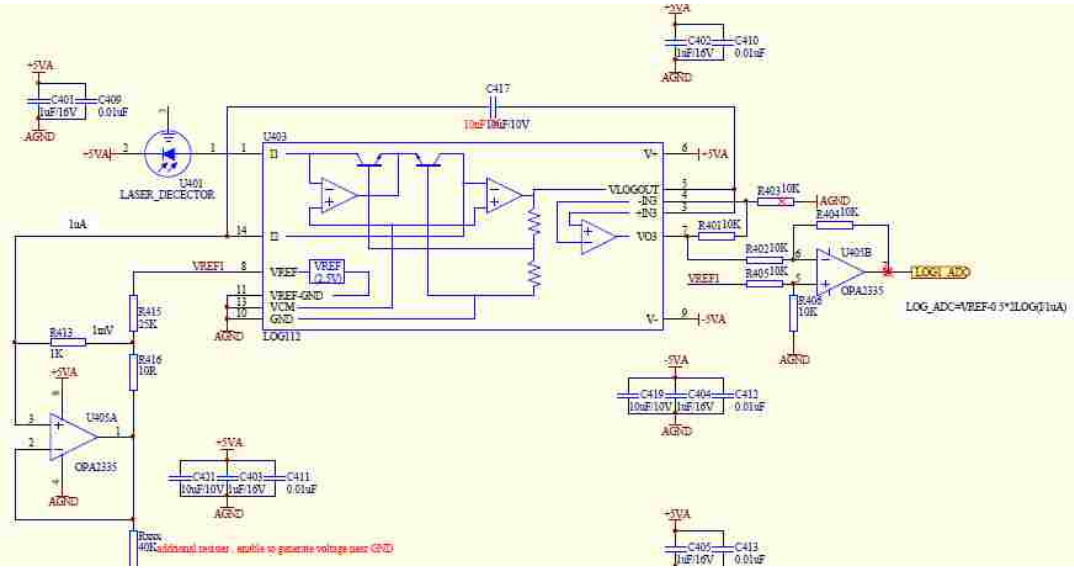


Figure 2.4 Photo detector schematic

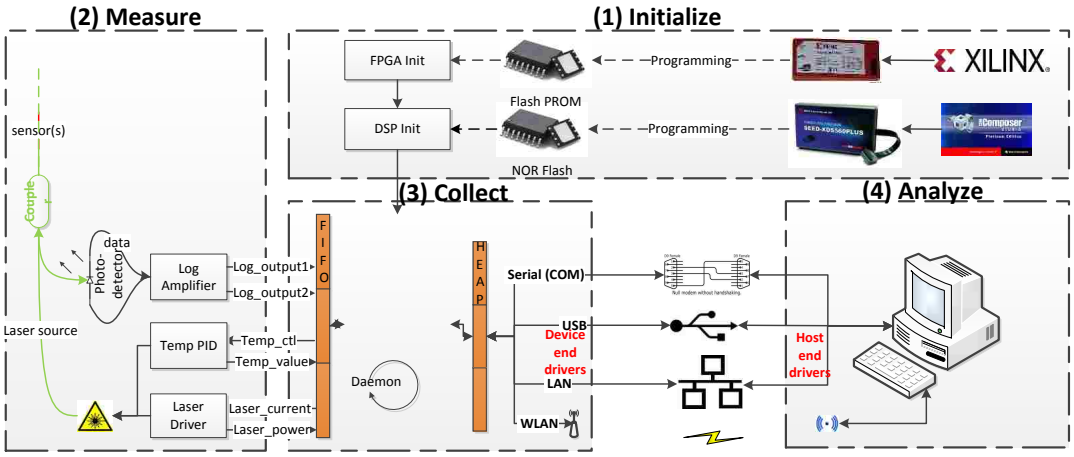


Figure 2.5 Hybrid view of proposed FSI

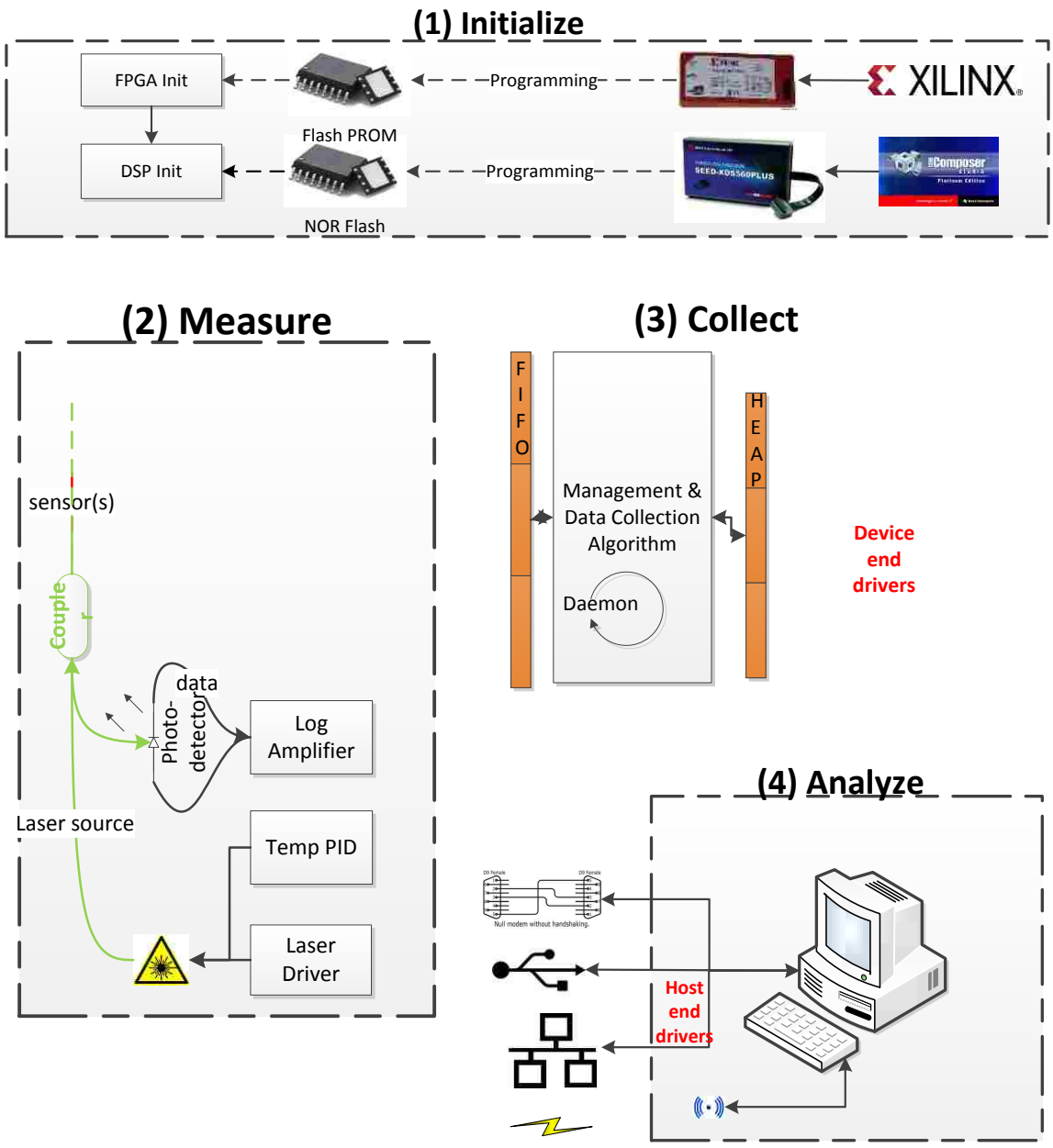


Figure 2.6 Modular view of proposed FSI

2.2. SOFTWARE

Below figure depicts the communication flow of the FSI which will be the main focus of this thesis. The code is developed using Code Composer Studio v3.3 (for DSP) and Xilinx iMPACT 12.3 (for FPGA).

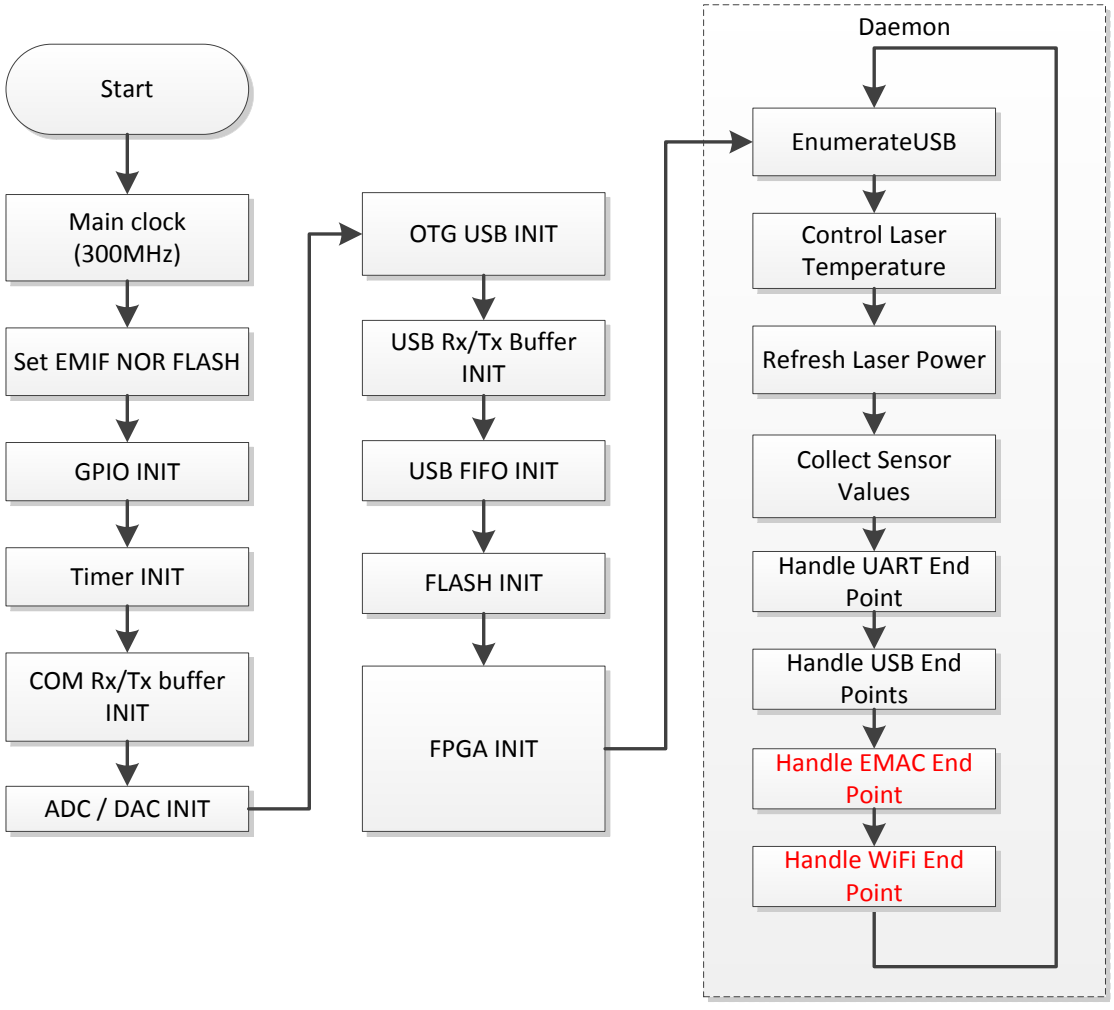


Figure 2.7 Software control flow of proposed FSI

2.2.1. Device End FPGA Setup. Connect Xilinx 12.3 FPGA programmer and start ISE iMPACT (M .70d)

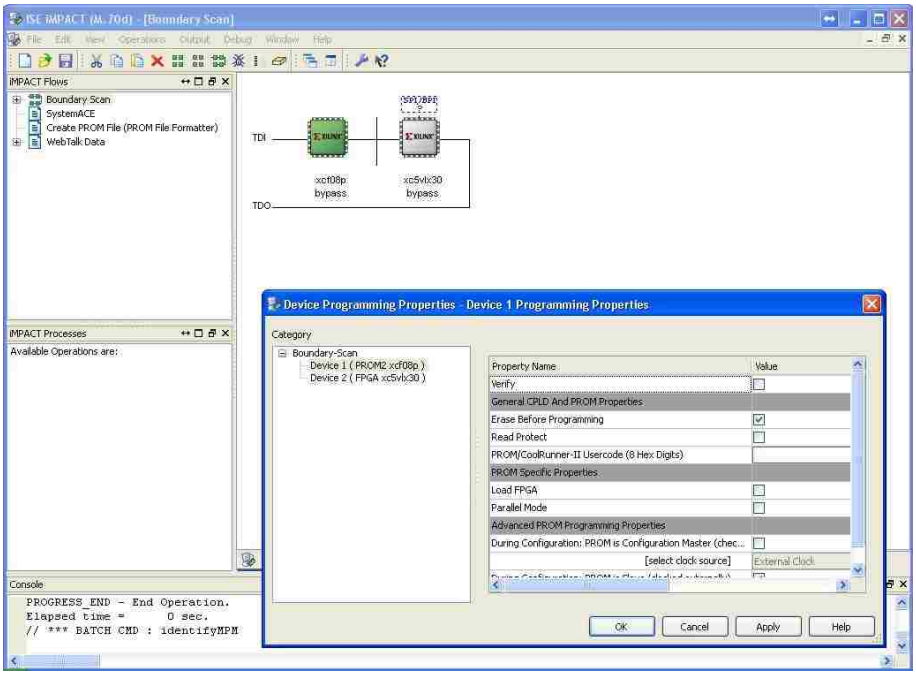


Figure 2.8 Perform a new boundary scan & assign configuration

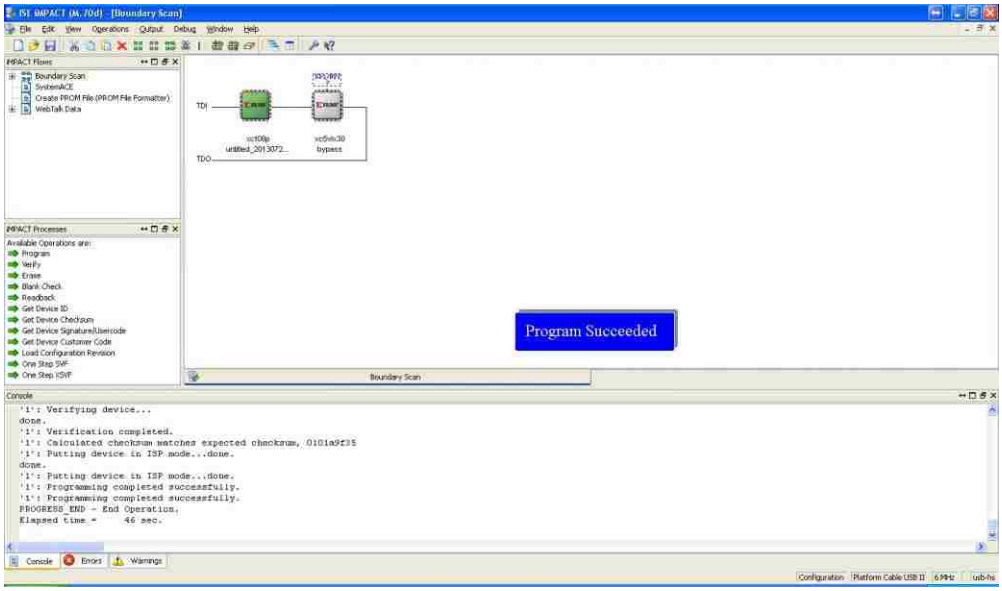
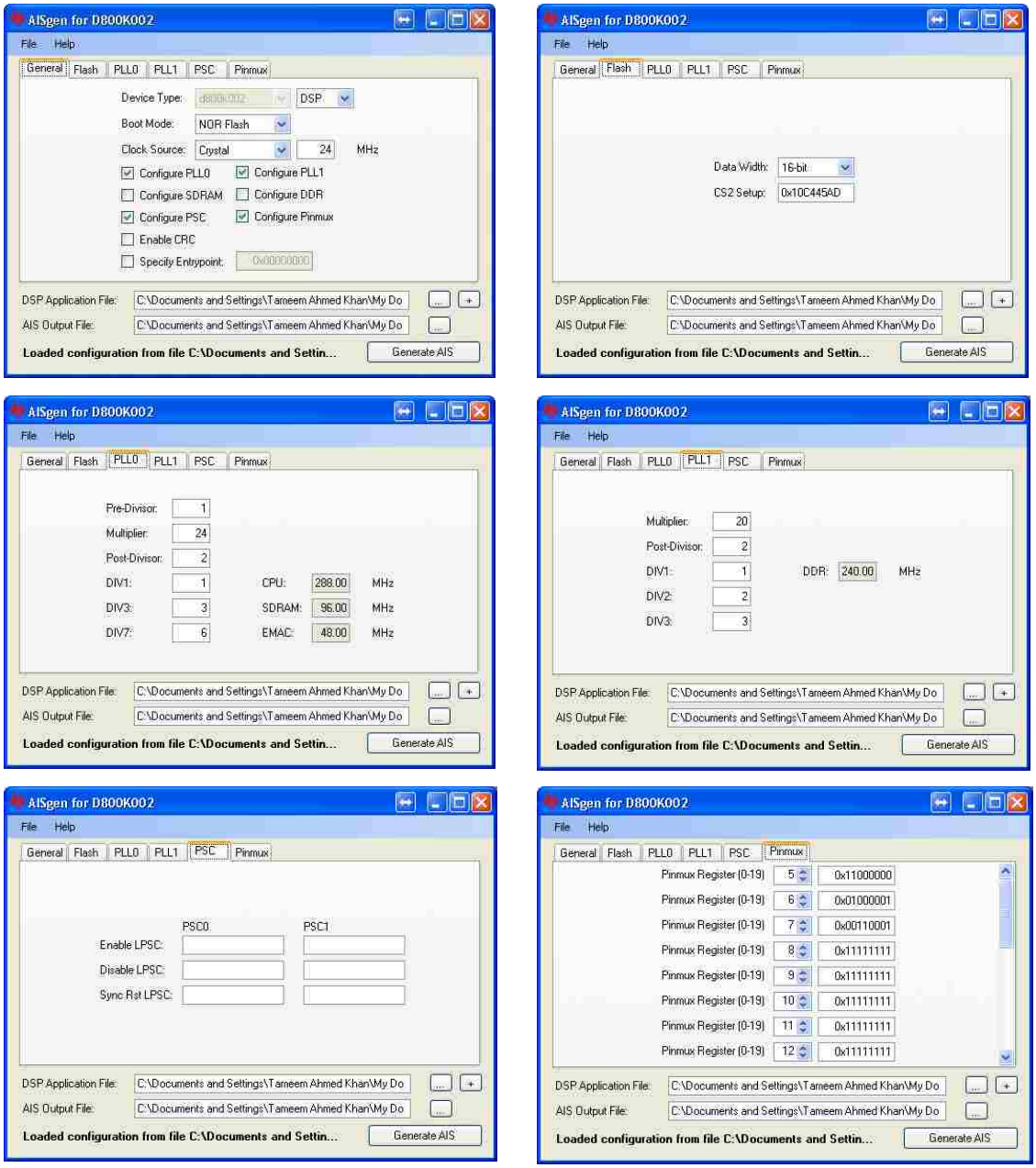


Figure 2.9 Perform FPGA programming

2.2.2. Device End DSP Setup. Open AISgen for D800K002 and perform below steps




```
    PLL0 init done for Core:300MHz, EMIF:100MHz
    -------------------------------------------
Opened AIS file, C:\Documents and Settings\Tameem Ahmed Khan\My Documents\Dropbox\Tams\Work\MST\05_LaserController\DSP\DSP_Firmware.bin
FLASH Programming Complete
```

Figure 2.10 DSP Configuration

2.2.3. PC End COM Interrogation. Make sure that COM port to which FSI is connected is detected through the hardware wizard of the operating system (here COM3). Modify com_4.m if the COM port changes. Below figure shows the default data when LASER board is not connected.

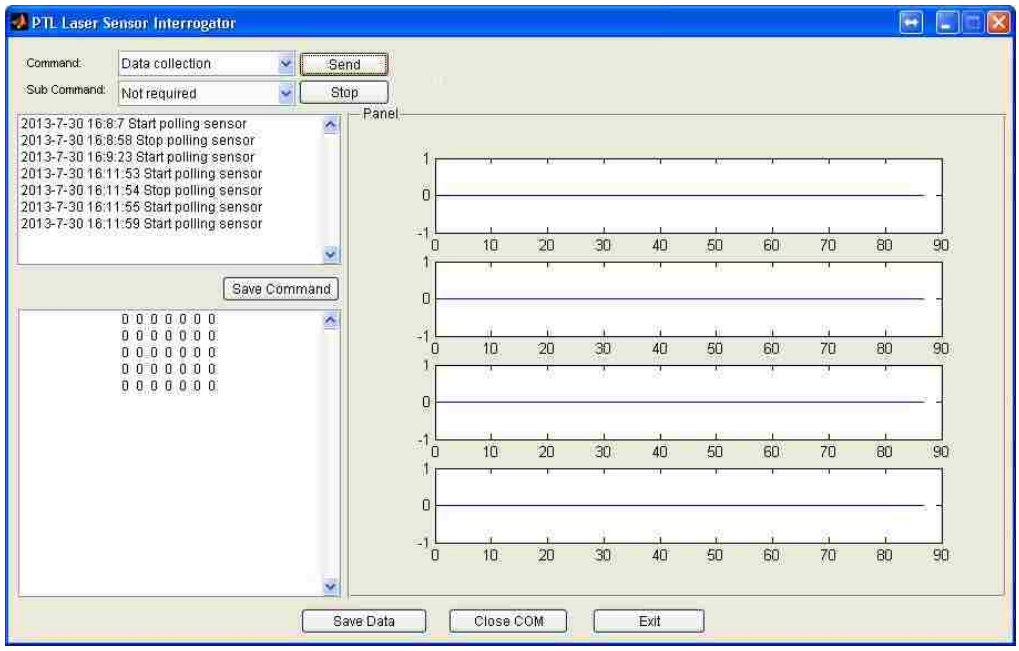


Figure 2.11 FSI connected via COM port

2.2.4. PC End USB Interrogation. Using DriverWizard 3.2.0,

Create a new USB project with:

- Vendor ID: 1b49
- Device GUID: B0655D76-ED59-44B8-B37E-E7EDF175F962

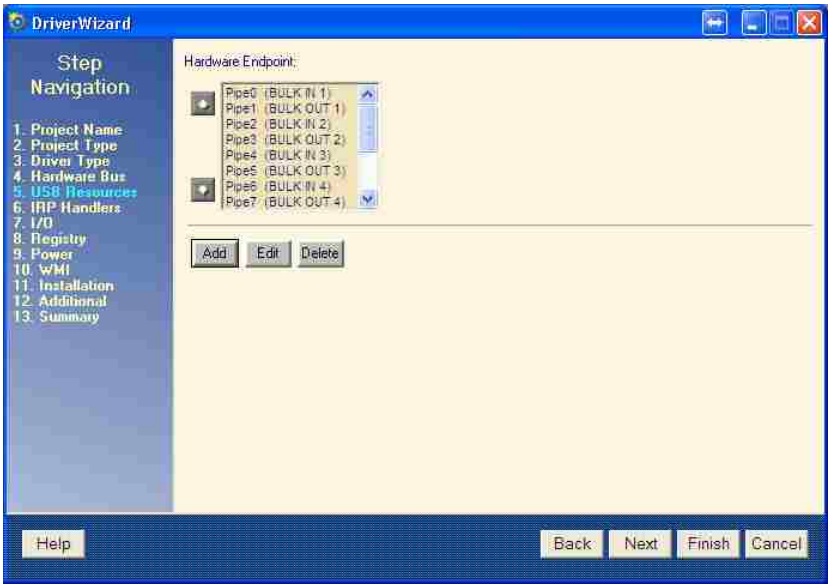


Figure 2.12 Configuring hardware endpoints

Driver Wizard produces custom WDM driver which requires DDK (used 3790.1830) to build into the DLL project. This source code is just bare bones, modify according to the end points and URBs supported by the FSI.

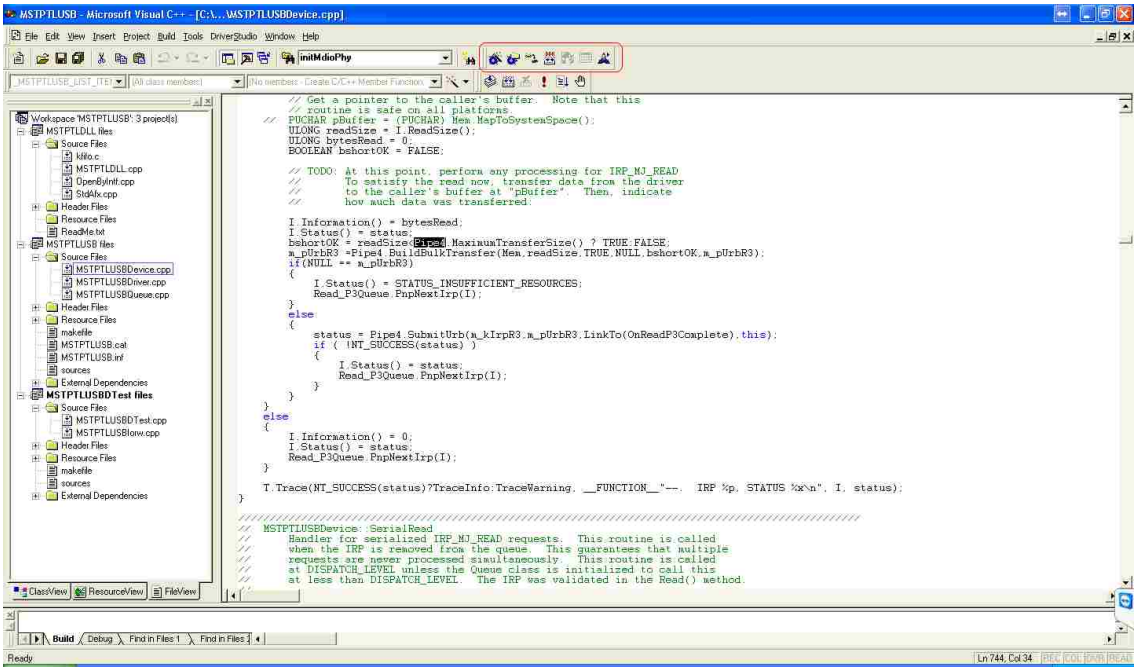


Figure 2.13 Building USB driver project

Use the buttons shown above to build the new driver. The DLL and SYS files generated are the output.

Note: INI and Test project need to be written if PnP functionality is required.

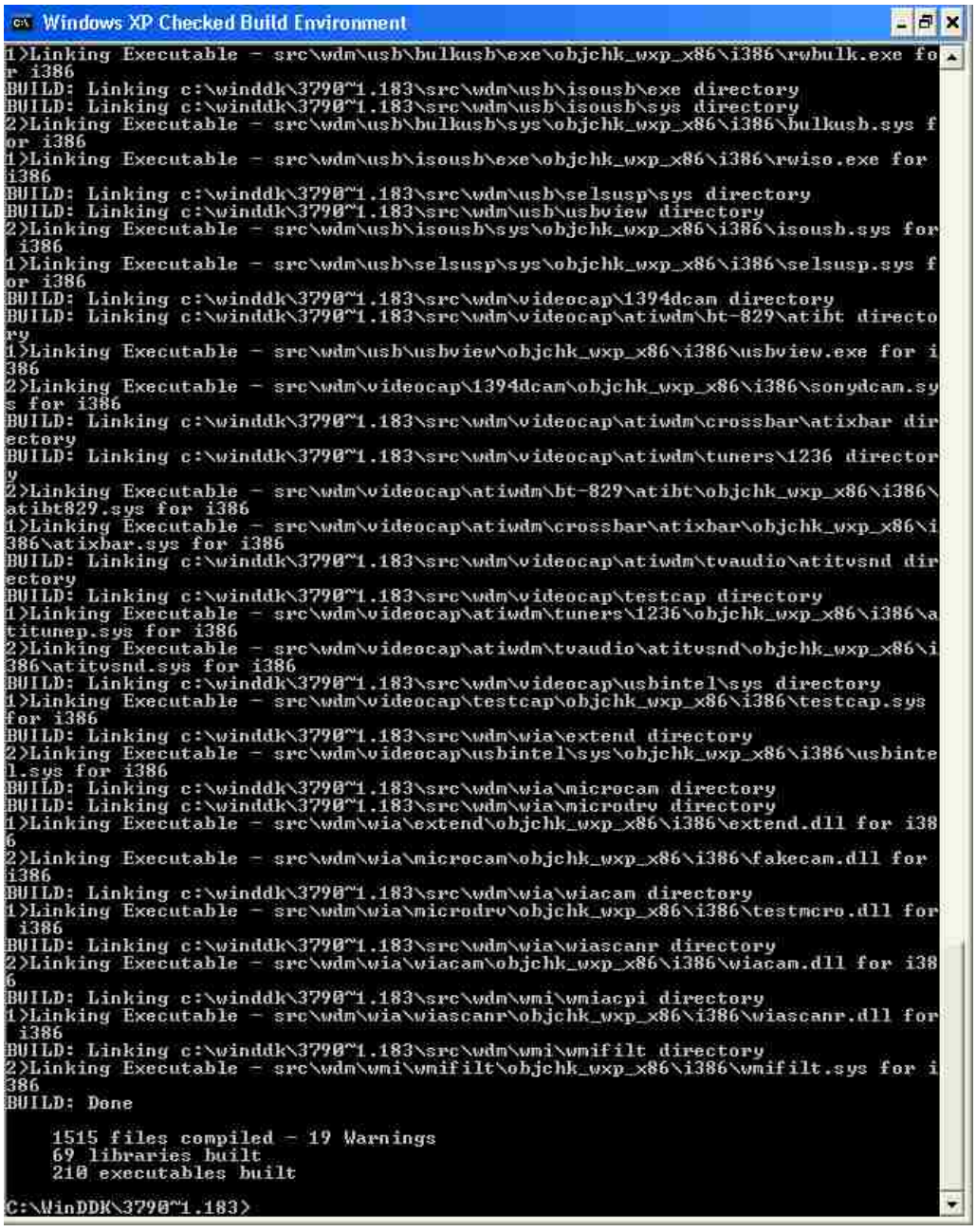


Figure 2.14 Build the DDK

3. SERIAL PORT (COM)

3.1. PURPOSE OF THE PERIPHERAL

The serial COM port is physical interface through which information transfers in or out one bit at a time. This port supports different properties that can be set before the transfer begins.

3.2. OVERVIEW OF FEATURES

Table 3.1 COM Features

Feature	Value
Speed	**115200**
Frame Size	**44**
Data Bits	**8 bits**
Parity	**Even**

3.3. FUNCTIONAL BLOCK DIAGRAM

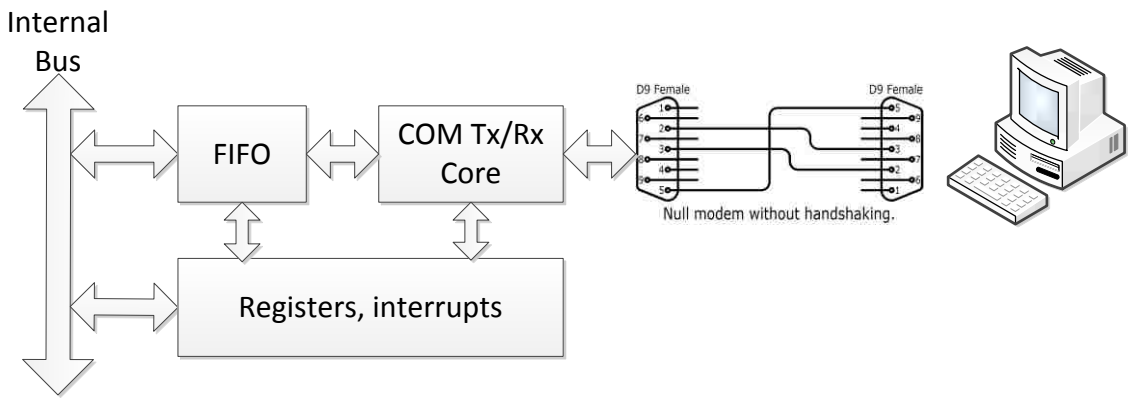


Figure 3.1 COM functional block diagram

Table 3.2 COM Protocol

#	Feature (w.r.t. FSI)	PC-> Peripheral (Hex)	Peripheral->Host (Hex)
1	Download Data to PC	5A,5A,1,1,0,0,0,0,0	A5, A5, 1, 1, data(240B)
2	Upload Current / Power mode to FSI	5A, 5A, 2, Laser#, data(5B)	A5, A5, Status(1B)
3	Upload Temp, PID to FSI	5A, 5A, 3, Laser#, data(5B)	A5, A5, Status(1B)
4	Download Laser parameters to PC	5A, 5A, 11, 11	A5, A5, data (60D)

3.4. FLOW DIAGRAM FOR COM

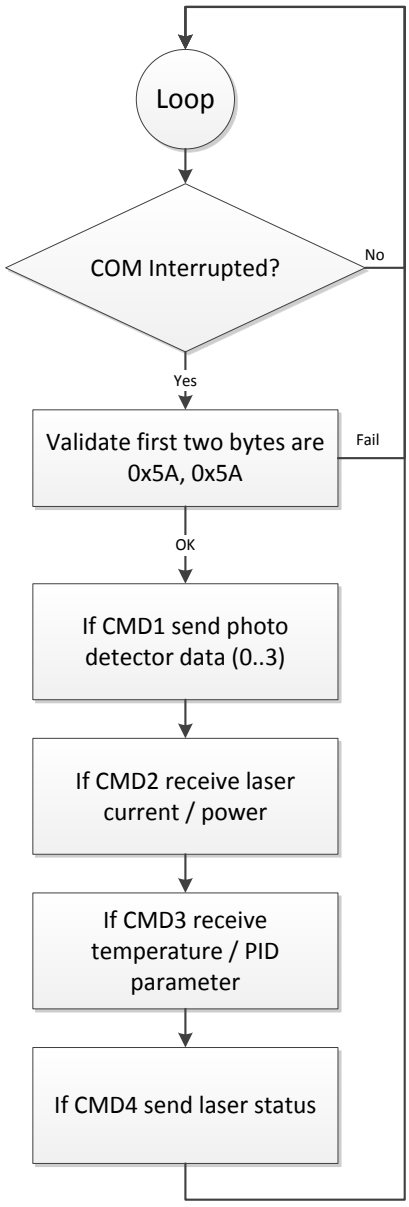


Figure 3.2 COM Flow Diagram

4. UNIVERSAL SERIAL BUS (USB)

4.1. PURPOSE OF THE PERIPHERAL

The USB controller[9] supports data throughput rates up to 480 Mbps. It provides a mechanism for data transfer between USB devices and also supports host negotiation.

4.2. OVERVIEW OF FEATURES

The USB has the following features [5]:

- Supports USB 2.0 peripheral at High Speed (480 Mbps) and Full Speed (12 Mbps)
- Supports USB 2.0 host at High Speed (480 Mbps), Full Speed (12 Mbps), and Low Speed (1.5 Mbps)
- Supports four simultaneous RX and TX endpoints, more can be supported by dynamically switching
- Each endpoint can support all transfer types (control, bulk, interrupt, and isochronous)
- Includes a 4K endpoint FIFO RAM, and supports programmable FIFO sizes
- External 5V power supply for VBUS can be controlled through I2C

Features Not Used

- Supports USB extensions for Session Request (SRP) and Host Negotiation (HNP)
- Includes a DMA controller that supports four TX and four RX DMA channels
- Includes RNDIS mode of DMA for accelerating RNDIS type protocols using short packet termination over USB

Features Not Supported

- The following features are not supported:
- High Bandwidth Isochronous Transfer.
- High Bandwidth Interrupt Transfer.
- Automatic Amalgamation of Bulk Packets (CPPI DMA will indirectly handle this feature and is not supported at the core level).
- Automatic Splitting of Bulk Packets (CPPI DMA will indirectly handle this feature and is not supported at the core level).

4.3. FUNCTIONAL BLOCK DIAGRAM

The USB functional block diagram is shown in figure below.

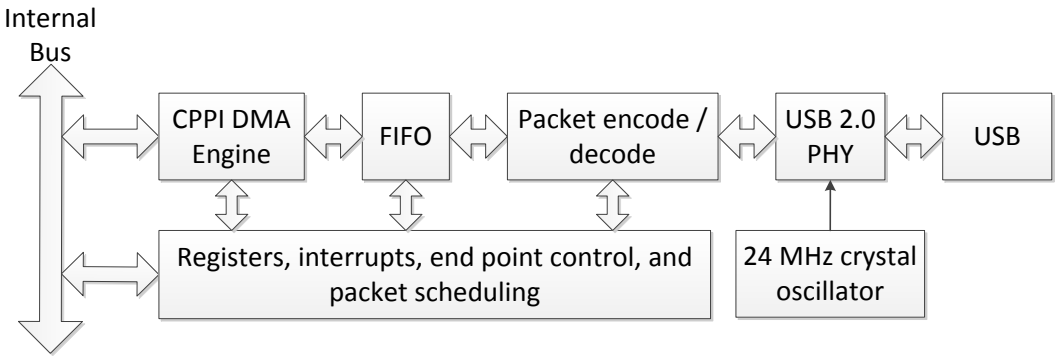


Figure 4.1 USB Functional Block Diagram

4.4. USB CONTROLLER PERIPHERAL MODE OPERATION

The USB controller can be used in a range of different environments. It can be used as either a high-speed or a full-speed USB peripheral device attached to a conventional USB host (such as a PC). It can be used as either host or peripheral device in point-to-point data transfers with another peripheral device - or, if the other device

also contains a Dual-Role Controller, the two devices can switch roles as required. (This second device may be either a high-speed, full-speed or low-speed USB function.) Or the controller can be used as the host to a range of such peripheral devices in a multi-point setup.

Whether the controller expects to behave as a host or as a peripheral device depends on the way the devices are cabled together. Each USB cable has an A end and a B end. If the A end of the cable is plugged into the controller, it will take the role of the Host device and go into host mode. If the B end of the cable is plugged in, the controller will go instead into peripheral mode.

The USB controller interrupts the DSP on completion of the data transfer on any of the endpoints or on detecting reset, resume, suspend, connect, disconnect, or SOF on the bus. When the DSP is interrupted with a USB interrupt, it needs to read the interrupt status register to determine the endpoints that have caused the interrupt and jump to the appropriate routine. If multiple endpoints have caused the interrupt, endpoint 0 should be serviced first, followed by the other endpoints. The suspend interrupt should be serviced last.

4.4.1. Soft Connect. After a reset, the SOFTCONN bit of POWER register (bit 6) is cleared to 0. The controller will therefore appear disconnected until the software has set the SOFTCONN bit to 1. The application software can then choose when to set the PHY into its normal mode. Systems with a lengthy initialization procedure may use this to ensure that initialization is complete and the system is ready to perform enumeration before connecting to the USB. Once the SOFTCONN bit has been set, the software can also simulate a disconnect by clearing this bit to 0.

4.4.2. Entry into Suspend Mode. When operating as a peripheral device, the controller monitors activity on the bus and when no activity has occurred for 3ms, it goes into Suspend mode. If the Suspend interrupt has been enabled, an interrupt will be generated at this time.

At this point, the controller can then be left active (and hence able to detect when resume signaling occurs on the USB), or the application may arrange to disable the controller by stopping its clock. However, the controller will not then be able to detect resume signaling on the USB. As a result, some external hardware will be needed to detect Resume signaling (by monitoring the DM and DP signals), so that the clock to the controller can be restarted.

4.4.3. Resume Signaling. When resume signaling occurs on the bus, first the clock to the controller must be restarted if necessary. Then the controller will automatically exit Suspend mode. If the Resume interrupt is enabled, an interrupt will be generated.

4.4.4. Initiating a Remote Wakeup. If the software wants to initiate a remote wakeup while the controller is in Suspend mode, it should write to the Power register to set the RESUME bit to 1. The software should leave then this bit set for approximately 10 ms (minimum of 2 ms, a maximum of 15 ms) before resetting it to 0.

NOTE: No resume interrupt will be generated when the software initiates a remote wakeup.

4.4.5. Reset Signaling. When reset signaling occurs on the bus, the controller will perform the following actions:

- Sets FADDR register to 0
- Sets INDEX register to 0
- Flushes all endpoint FIFOs
- Clears all control/status registers
- Generates a reset interrupt.

If the HSENA bit in the POWER register (bit 5) was set, the controller also tries to negotiate for high-speed operation. Whether high-speed operation is selected is indicated by HSMODE bit of POWER register (bit 4). When the application software

receives a reset interrupt, it should close any open pipes and wait for bus enumeration to begin.

4.5. PERIPHERAL MODE: CONTROL TRANSACTIONS

Endpoint 0 is the main control endpoint of the core. The software is required to handle all the standard device requests that may be sent or received via endpoint 0. The protocol for these device requests involves different numbers and types of transactions per transfer. To accommodate this, the software needs to take a state machine approach to command decoding and handling.

The Standard Device Requests received by a USB peripheral device can be divided into three categories:

- Zero Data Requests (in which all the information is included in the command),
- Write Requests (in which the command will be followed by additional data), and
- Read Requests (in which the device is required to send data back to the host).

This section looks at the sequence of actions that the software must perform to process these different types of device request.

NOTE: The Setup packet associated with any standard device request should include an 8-byte command. Any setup packet containing a command field of anything other than 8 bytes will be automatically rejected by the controller.

4.5.1. Zero Data Requests. Zero data requests have all their information included in the 8-byte command and require no additional data to be transferred. Examples of Zero Data standard device requests are:

- SET_FEATURE
- CLEAR_FEATURE
- SET_ADDRESS
- SET_CONFIGURATION
- SET_INTERFACE

The sequence of events will begin, as with all requests, when the software receives an endpoint 0 interrupt. The RXPKTRDY bit of PERI_CSR0 (bit 0) will also have been set. The 8-byte command should then be read from the endpoint 0 FIFO, decoded and the appropriate action taken.

For example, if the command is SET_ADDRESS, the 7-bit address value contained in the command should be written to the FADDR register. The PERI_CSR0 register should then be written to set the SERV_RXPKTRDY bit (bit 6) (indicating that the command has been read from the FIFO) and to set the DATAEND bit (bit 3) (indicating that no further data is expected for this request). The interval between setting SERV_RXPKTRDY bit and DATAEND bit should be very small to avoid getting a SetupEnd error condition.

When the host moves to the status stage of the request, a second endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software. The second interrupt is just a confirmation that the request completed successfully. For SET_ADDRESS command, the address should be set in FADDR register only after the status stage interrupt is received.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the PERI_CSR0 register should be written to set the SERV_RXPKTRDY bit (bit 6) and to set the SENDSTALL bit (bit 5). When the host moves to the status stage of the request, the controller will send a STALL to tell the host that the request was not executed. A second endpoint 0 interrupt will be generated and the SENTSTALL bit (bit 2 of PERI_CSR0) will be set.

If the host sends more data after the DATAEND bit has been set, then the controller will send a STALL. An endpoint 0 interrupt will be generated and the SENTSTALL bit (bit 2 of PERI_CSR0) will be set.

NOTE: DMA is not supported for endpoint 0, so the command should be read by accessing the endpoint 0 FIFO register.

4.5.2. Write Requests. Write requests involve an additional packet (or packets) of data being sent from the host after the 8-byte command. An example of a Write standard device request is: SET_DESCRIPTOR.

The sequence of events will begin, as with all requests, when the software receives an endpoint 0 interrupt. The RXPKTRDY bit of PERI_CSR0 will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded.

As with a zero data request, the PERI_CSR0 register should then be written to set the SERV_RXPKTRDY bit (bit 6) (indicating that the command has been read from the FIFO) but in this case the DATAEND bit (bit 3) should not be set (indicating that more data is expected).

When a second endpoint 0 interrupt is received, the PERI_CSR0 register should be read to check the endpoint status. The RXPKTRDY bit of PERI_CSR0 should be set to indicate that a data packet has been received. The COUNT0 register should then be read to determine the size of this data packet. The data packet can then be read from the endpoint 0 FIFO.

If the length of the data associated with the request (indicated by the wLength field in the command) is greater than the maximum packet size for endpoint 0, further data packets will be sent. In this case, PERI_CSR0 should be written to set the SERV_RXPKTRDY bit, but the DATAEND bit should not be set.

When all the expected data packets have been received, the PERI_CSR0 register should be written to set the SERV_RXPKTRDY bit and to set the DATAEND bit (indicating that no more data is expected).

When the host moves to the status stage of the request, another endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software, the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the PERI_CSR0 register should be written to set the SERV_RXPKTRDY bit (bit 6) and to set the SENDSTALL bit (bit 5). When the host sends more data, the controller will send a STALL to tell the host that the request was not executed. An endpoint 0 interrupt will be generated and the SENTSTALL bit of PERI_CSR0 (bit 2) will be set.

If the host sends more data after the DATAEND has been set, then the controller will send a STALL. An endpoint 0 interrupt will be generated and the SENTSTALL bit of PERI_CSR0 (bit 2) will be set.

4.5.3. Read Requests. Read requests have a packet (or packets) of data sent from the function to the host after the 8-byte command. Examples of Read Standard Device Requests are:

- GET_CONFIGURATION
- GET_INTERFACE
- GET_DESCRIPTOR
- GET_STATUS
- SYNCH_FRAME

The sequence of events will begin, as with all requests, when the software receives an endpoint 0 interrupt. The RXPKTRDY bit of PERI_CSR0 (bit 0) will also have been set. The 8-byte command should then be read from the endpoint 0 FIFO and decoded. The PERI_CSR0 register should then be written to set the SERV_RXPKTRDY bit (bit 6) (indicating that the command has read from the FIFO).

The data to be sent to the host should then be written to the endpoint 0 FIFO. If the data to be sent is greater than the maximum packet size for endpoint 0, only the maximum packet size should be written to the FIFO. The PERI_CSR0 register should then be written to set the TXPKTRDY bit (bit 1) (indicating that there is a packet in the FIFO to be sent). When the packet has been sent to the host, another endpoint 0 interrupt will be generated and the next data packet can be written to the FIFO.

When the last data packet has been written to the FIFO, the PERI_CSR0 register should be written to set the TXPKTRDY bit and to set the DATAEND bit (bit 3) (indicating that there is no more data after this packet).

When the host moves to the status stage of the request, another endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software: the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the PERI_CSR0 register should be written to set the SERV_RXPKTRDY bit (bit 6) and to set the SENDSTALL bit (bit 5). When the host requests data, the controller will send a STALL to tell the host that the request was not executed. An endpoint 0 interrupt will be generated and the SENTSTALL bit of PERI_CSR0 (bit 2) will be set.

If the host requests more data after DATAEND (bit 3) has been set, then the controller will send a STALL. An endpoint 0 interrupt will be generated and the SENTSTALL bit of PERI_CSR0 (bit 2) will be set.

4.5.4. Endpoint 0 States. When the USB controller is operating as a peripheral device, the endpoint 0 control needs three modes – IDLE, TX and RX – corresponding to the different phases of the control transfer and the states endpoint 0 enters for the different phases of the transfer (described in later sections).

The default mode on power-up or reset should be IDLE. RXPKTRDY bit of PERI_CSR0 (bit 0) becoming set when endpoint 0 is in IDLE state indicates a new device request. Once the device request is unloaded from the FIFO, the controller decodes the descriptor to find whether there is a data phase and, if so, the direction of the data phase of the control transfer (in order to set the FIFO direction).See Figure 3.3.

Depending on the direction of the data phase, endpoint 0 goes into either TX state or RX state. If there is no Data phase, endpoint 0 remains in IDLE state to accept the next device request.

The actions that the CPU needs to take at the different phases of the possible transfers (e.g., loading the FIFO, setting TXPKTRDY) are indicated in below figure.
NOTE: The controller changes the FIFO direction, depending on the direction of the data phase independently of the CPU.

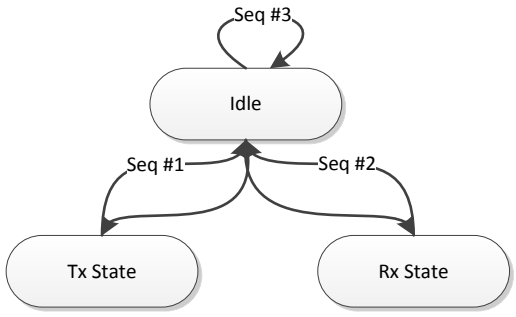


Figure 4.2 CPU Actions at USB Transfer Phases

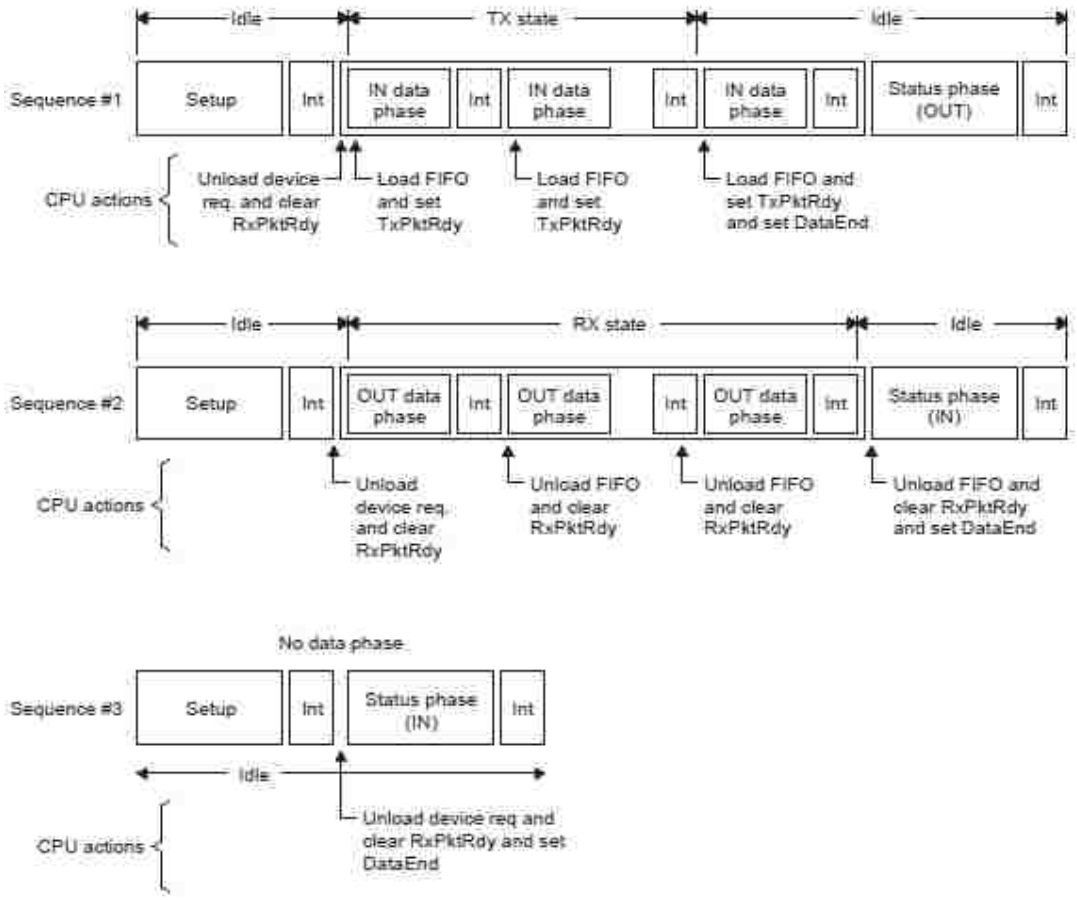


Figure 4.3 USB Sequence of Transfer

4.5.5. Endpoint 0 Service Routine. An Endpoint 0 interrupt is generated when:

- The controller sets the RXPKTRDY bit of PERI_CSR0 (bit 0) after a valid token has been received and data has been written to the FIFO.
- The controller clears the TXPKTRDY bit of PERI_CSR0 (bit 1) after the packet of data in the FIFO has been successfully transmitted to the host.
- The controller sets the SENTSTALL bit of PERI_CSR0 (bit 2) after a control transaction is ended due to a protocol violation.
- The controller sets the SETUPEND bit of PERI_CSR0 (bit 4) because a control transfer has ended before DATAEND (bit 3 of PERI_CSR0) is set.

Whenever the endpoint 0 service routine is entered, the software must first check to see if the current control transfer has been ended due to either a STALL condition or a premature end of control transfer. If the control transfer ends due to a STALL condition, the SENTSTALL bit would be set. If the control transfer ends due to a premature end of control transfer, the SETUPEND bit would be set. In either case, the software should abort processing the current control transfer and set the state to IDLE.

Once the software has determined that the interrupt was not generated by an illegal bus state, the next action taken depends on the endpoint state. Refer below figure for flow of this process.

If endpoint 0 is in IDLE state, the only valid reason an interrupt can be generated is as a result of the controller receiving data from the bus. The service routine must check for this by testing the RXPKTRDY bit of PERI_CSR0 (bit 0). If this bit is set, then the controller has received a SETUP packet. This must be unloaded from the FIFO and decoded to determine the action the controller must take. Depending on the command contained within the SETUP packet, endpoint 0 will enter one of three states:

- If the command is a single packet transaction (SET_ADDRESS, SET_INTERFACE etc.) without any data phase, the endpoint will remain in IDLE state.
- If the command has an OUT data phase (SET_DESCRIPTOR etc.), the endpoint will enter RX state.
- If the command has an IN data phase (GET_DESCRIPTOR etc.), the endpoint will enter TX state.

If the endpoint 0 is in TX state, the interrupt indicates that the core has received an IN token and data from the FIFO has been sent. The software must respond to this either by placing more data in the FIFO if the host is still expecting more data or by setting the DATAEND bit to indicate that the data phase is complete. Once the data

phase of the transaction has been completed, endpoint 0 should be returned to IDLE state to await the next control transaction.

NOTE: All command transactions include a field that indicates the amount of data the host expects to receive or is going to send.

If the endpoint is in RX state, the interrupt indicates that a data packet has been received. The software must respond by unloading the received data from the FIFO. The software must then determine whether it has received all of the expected data. If it has, the software should set the DATAEND bit and return endpoint 0 to IDLE state. If more data is expected, the firmware should set the SERV_RXPKTRDY bit of PERI_CSR0 (bit 6) to indicate that it has read the data in the FIFO and leave the endpoint in RX state.

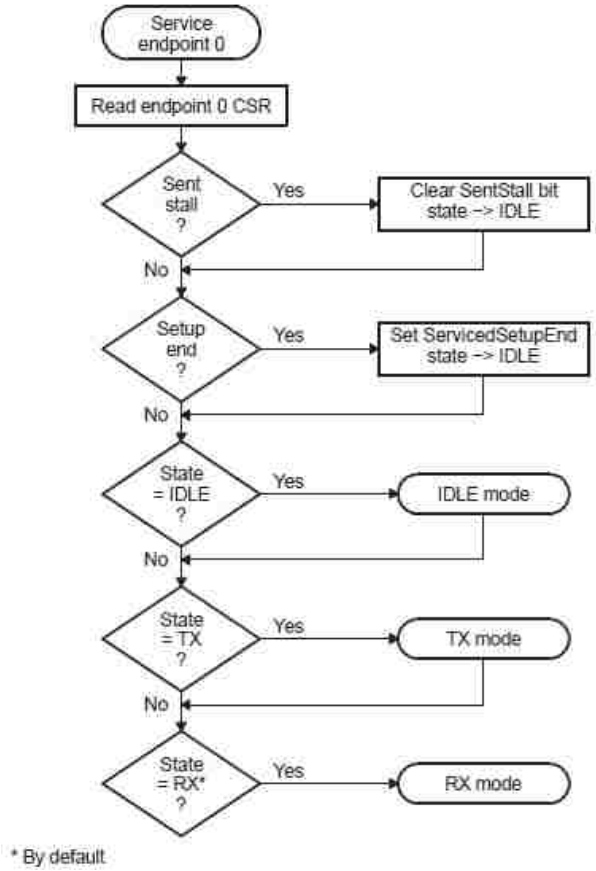


Figure 4.4 USB Service Endpoint 0 Flow Chart

4.5.6. IDLE Mode. IDLE mode is the mode the endpoint 0 control must select at power-on or reset and is the mode to which the endpoint 0 control should return when the RX and TX modes are terminated. It is also the mode in which the SETUP phase of control transfer is handled (as outlined in below figure).

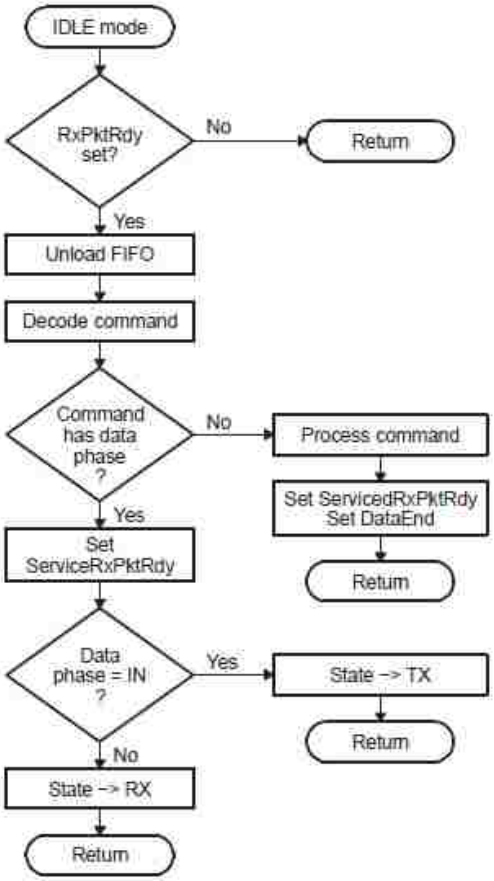


Figure 4.5 USB IDLE Mode Flow Chart

4.5.7. TX Mode. When the endpoint is in TX state all arriving IN tokens need to be treated as part of a data phase until the required amount of data has been sent to the host. If either a SETUP or an OUT token is received while the endpoint is in the TX state, this will cause a SetupEnd condition to occur as the core expects only IN tokens (See below figure).

Three events can cause TX mode to be terminated before the expected amount of data has been sent:

- The host sends an invalid token causing a SETUPEND condition (bit 4 of PERI_CSR0 set).
- The software sends a packet containing less than the maximum packet size for endpoint 0.
- The software sends an empty data packet.

Until the transaction is terminated, the software simply needs to load the FIFO when it receives an interrupt that indicates a packet has been sent from the FIFO. (An interrupt is generated when TXPKTRDY is cleared.)

When the software forces the termination of a transfer (by sending a short or empty data packet), it should set the DATAEND bit of PERI_CSR0 (bit 3) to indicate to the core that the data phase is complete and that the core should next receive an acknowledge packet.

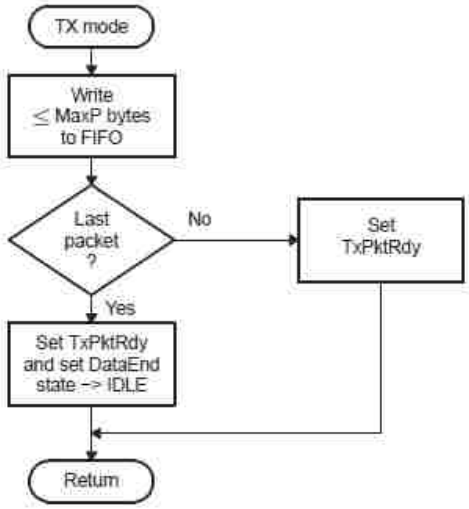


Figure 4.6 USB TX Mode Flow Chart

4.5.8. RX Mode. In RX mode, all arriving data should be treated as part of a data phase until the expected amount of data has been received. If either a SETUP or an IN token is received while the endpoint is in RX state, a SetupEnd condition will occur as the controller expects only OUT tokens.

Three events can cause RX mode to be terminated before the expected amount of data has been received as shown in below figure:

The host sends an invalid token causing a SETUPEND condition (setting bit 4 of PERI_CSR0). The host sends a packet which contains less than the maximum packet size for endpoint 0.

The host sends an empty data packet. Until the transaction is terminated, the software unloads the FIFO when it receives an interrupt that indicates new data has arrived (setting RXPKTRDY bit of PERI_CSR0) and to clear RXPKTRDY by setting the SERV_RXPKTRDY bit of PERI_CSR0 (bit 6).

When the software detects the termination of a transfer (by receiving either the expected amount of data or an empty data packet), it should set the DATAEND bit (bit 3 of PERI_CSR0) to indicate to the controller that the data phase is complete and that the core should receive an acknowledge packet next.

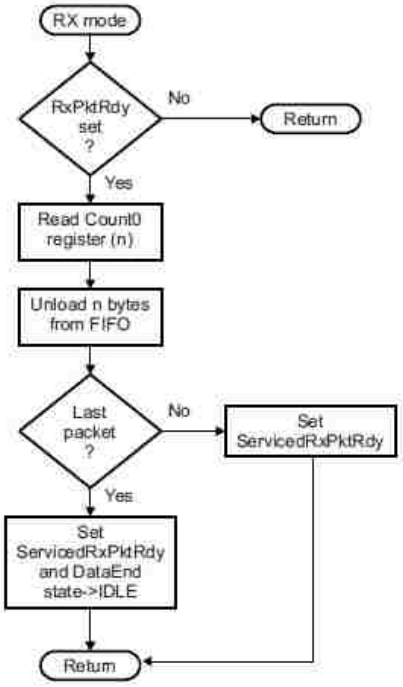


Figure 4.7 USB RX Mode Flow Chart

4.5.9. Error Handling. A control transfer may be aborted due to a protocol error on the USB, the host prematurely ending the transfer, or if the software wishes to abort the transfer (e.g., because it cannot process the command).

The controller automatically detects protocol errors and sends a STALL packet to the host under the following conditions:

- The host sends more data during the OUT Data phase of a write request than was specified in the command. This condition is detected when the host sends an OUT token after the DATAEND bit (bit 3 of PERI_CSR0) has been set.
- The host requests more data during the IN Data phase of a read request than was specified in the command. This condition is detected when the host sends an IN token after the DATAEND bit in the PERI_CSR0 register has been set.
- The host sends more than Max Packet Size data bytes in an OUT data packet.

- The host sends a non-zero length DATA1 packet during the STATUS phase of a read request.
- When the controller has sent the STALL packet, it sets the SENTSTALL bit (bit 2 of PERI_CSR0) and generates an interrupt. When the software receives an endpoint 0 interrupt with the SENTSTALL bit set, it should abort the current transfer, clear the SENTSTALL bit, and return to the IDLE state.
- If the host prematurely ends a transfer by entering the STATUS phase before all the data for the request has been transferred, or by sending a new SETUP packet before completing the current transfer, then the SETUPEND bit (bit 4 of PERI_CSR0) will be set and an endpoint 0 interrupt generated. When the software receives an endpoint 0 interrupt with the SETUPEND bit set, it should abort the current transfer, set the SERV_SETUPEND bit (bit 7 of PERI_CSR0), and return to the IDLE state. If the RXPKTRDY bit (bit 0 of PERI_CSR0) is set this indicates that the host has sent another SETUP packet and the software should then process this command.
- If the software wants to abort the current transfer, because it cannot process the command or has some other internal error, then it should set the SENDSTALL bit (bit 5 of PERI_CSR0). The controller will then send a STALL packet to the host, set the SENTSTALL bit (bit 2 of PERI_CSR0) and generate an endpoint 0 interrupt.

Note: FSI USB implementation currently does not support Bulk Transactions, Interrupt Transactions, Isochronous Transactions

5. ETHERNET

5.1. EMAC

The EMAC module [10] (in figure below) interfaces to the outside world through the Media Independent Interface (MII) and/or Reduced Media Independent Interface (RMII). The interface between the EMAC module and the system core is provided through the EMAC control module.

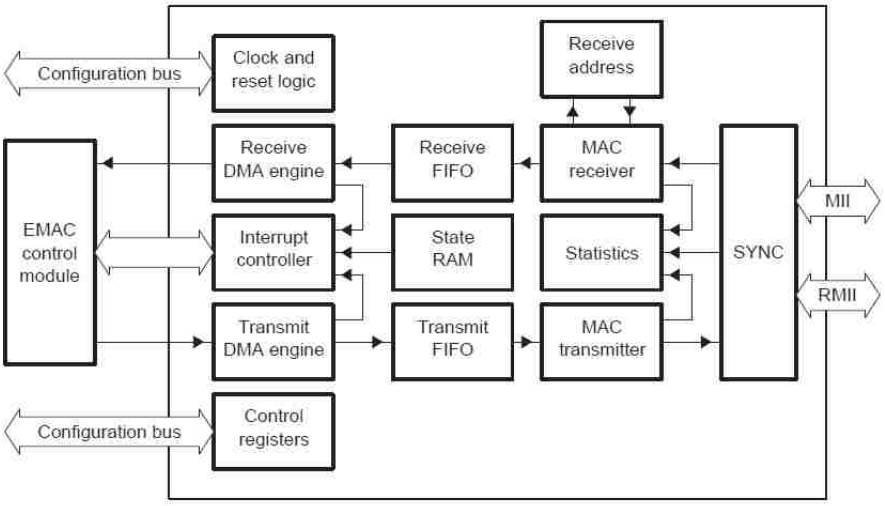


Figure 5.1 EMAC Module Block Diagram

5.1.1. Purpose of the Peripheral. The EMAC module is used to move data between the device and another host connected to the same network, in compliance with the Ethernet protocol.

5.1.2. Features. The EMAC/MDIO has the following features:

- Synchronous 10/100 Mbps operation.
- Standard Media Independent Interface (MII) and/or Reduced Media Independent Interface (RMII) to physical layer device (PHY).
- EMAC acts as DMA master to either internal or external device memory space.
- Eight receive channels with VLAN tag discrimination for receive quality-of-service (QOS) support.
- Eight transmit channels with round-robin or fixed priority for transmit quality-of-service (QOS) support.
- Ether-Stats and 802.3-Stats statistics gathering.
- Transmit CRC generation selectable on a per channel basis.
- Broadcast frames selection for reception on a single channel.
- Multicast frames selection for reception on a single channel.
- Promiscuous receive mode frames selection for reception on a single channel (all frames, all good frames, short frames, error frames).
- Hardware flow control.
- 8k-byte local EMAC descriptor memory that allows the peripheral to operate on descriptors without affecting the CPU. The descriptor memory holds enough information to transfer up to 512 Ethernet packets without CPU intervention. (This memory is also known as CPPI RAM.)
- Programmable interrupt logic permits the software driver to restrict the generation of back-to-back interrupts, which allows more work to be performed in a single call to the interrupt service routine.

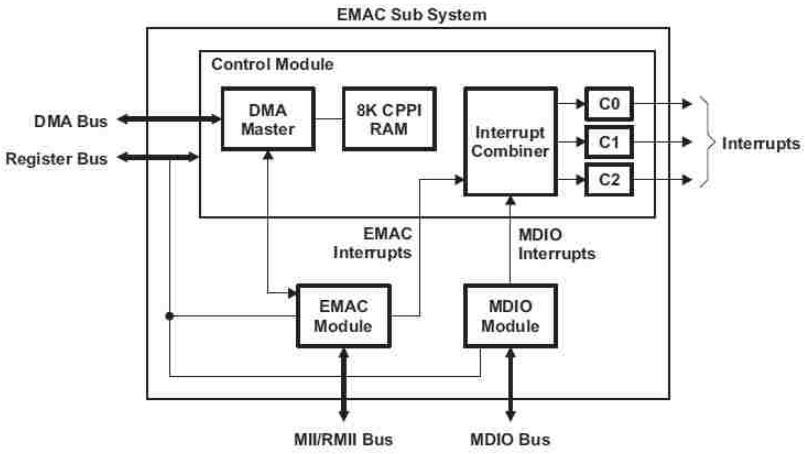


Figure 5.2 EMAC and MDIO Block Diagram

Above figure shows the three main functional modules of the EMAC/MDIO peripheral: EMAC control module, EMAC module, MDIO module

The EMAC control module is the main interface between the device core processor to the EMAC and MDIO modules. The EMAC control module controls device interrupts and incorporates an 8k-byte internal RAM to hold EMAC buffer descriptors (also known as CPPI RAM).

The MDIO module implements the 802.3 serial management interface to interrogate and control up to 32 Ethernet PHYs connected to the device by using a shared two-wire bus. Host software uses the MDIO module to configure the auto-negotiation parameters of each PHY attached to the EMAC, retrieve the negotiation results, and configure required parameters in the EMAC module for correct operation. The module is designed to allow almost transparent operation of the MDIO interface, with very little maintenance from the core processor.

The EMAC module provides an efficient interface between the processor and the network. The EMAC on this device supports 10Base-T (10 Mbits/sec) and 100BaseTX (100 Mbits/sec), half-duplex and full-duplex mode, and hardware flow control and quality-of-service (QOS) support.

- The DMA bus connection from the EMAC control module allows the EMAC module to read and write both internal and external memory through the DMA memory transfer controller.
- The EMAC control, EMAC, and MDIO modules all have control registers. These registers are memory-mapped into device memory space via the device configuration bus. Along with these registers, the control module's internal CPPI RAM is mapped into this same range.
- The EMAC and MDIO interrupts are combined into four interrupt signals within the control module. Three configurable interrupt cores within the control module receive all four interrupt signals from the combiner and submit interrupt requests to the CPU.

5.1.3. Memory Map. The EMAC peripheral includes internal memory that is used to hold buffer descriptions of the Ethernet packets to be received and transmitted. This internal RAM is 2K × 32 bits in size. Data can be written to and read from the EMAC internal memory by either the EMAC or the CPU. It is used to store buffer descriptors that are 4-words (16-bytes) deep. This 8K local memory holds enough information to transfer up to 512 Ethernet packets without CPU intervention. This EMAC RAM is also referred to as the CPPI buffer descriptor memory because it complies with the Communications Port Programming Interface (CPPI) v3.0 standard.

The packet buffer descriptors can also be placed in other on- and off-chip memories such as L2 and EMIF. There are some tradeoffs in terms of cache performance and throughput when descriptors are placed in the system memory, versus when they are placed in the EMAC's internal memory. In general, the EMAC throughput is better when the descriptors are placed in the local EMAC CPPI RAM.

5.1.4. Media Independent Interface (MII) Connections.

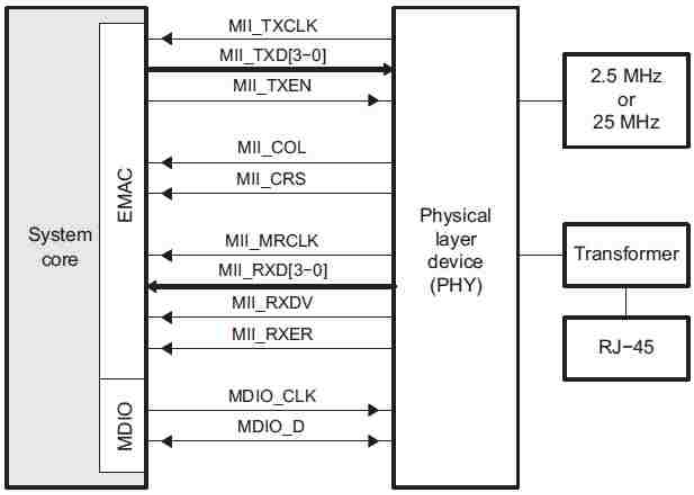


Figure 5.3 Ethernet Configuration—MII Connection

Note: Reduced Media Independent Interface (RMII) Connections is not required if MII is supported

5.1.5. Ethernet Frame Format.

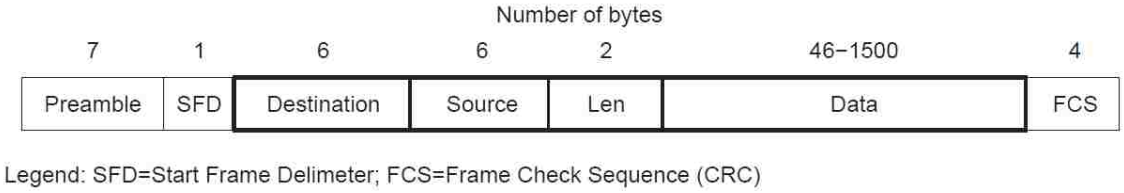


Figure 5.4 Frame Format for Ethernet

5.1.6. Packet Buffer Descriptors. The buffer descriptor is a central part of the EMAC module and is how the application software describes Ethernet packets to be sent and empty buffers to be filled with incoming packet data. The basic descriptor format is shown in figure below

Word Offset	Bit Fields 31 — 16	15 — 0
0	Next Descriptor Pointer	
1	Buffer Pointer	
2	Buffer Offset	Buffer Length
3	Flags	Packet Length

Figure 5.5 Basic Ethernet Descriptor Format

5.1.7. Transmit and Receive Descriptor Queues. The lists used by the EMAC are maintained by the application software through the use of the head descriptor pointer registers (HDP). Since the EMAC supports eight channels for transmit and receive, there are eight head descriptor pointer registers for both transmit and receive. They are:

- TXnHDP - Transmit Channel n DMA Head Descriptor Pointer Register
- RXnHDP - Receive Channel n DMA Head Descriptor Pointer Register

After an EMAC reset and before enabling the EMAC for send and receive, all 16 head descriptor pointer registers must be initialized to 0.

5.1.8. Transmit and Receive EMAC Interrupts. The EMAC synchronizes descriptor list processing through the use of interrupts to the software application. The interrupts are controlled by the application using the interrupt masks, global interrupt enable, and the completion pointer register (CP). The CP is also called the interrupt acknowledge register.

5.2. MDIO MODULE

The MDIO module is used to manage up to 32 physical layer (PHY) devices connected to the Ethernet Media Access Controller (EMAC). The device supports a single PHY being connected to the EMAC at any given time. The MDIO module is designed to allow almost transparent operation of the MDIO interface with little maintenance from the CPU.

The MDIO module continuously polls 32 MDIO addresses in order to enumerate all PHY devices in the system. Once a PHY device has been detected, the MDIO module reads the MDIO PHY link status register (LINK) to monitor the PHY link state. Link change events are stored in the MDIO module, which can interrupt the CPU. This storing of the events allows the CPU to poll the link status of the PHY device without continuously performing MDIO module accesses. However, when the CPU must access the MDIO module for configuration and negotiation, the MDIO module performs the MDIO read or write operation independent of the CPU. This independent operation allows the processor to poll for completion or interrupt the CPU once the operation has completed.

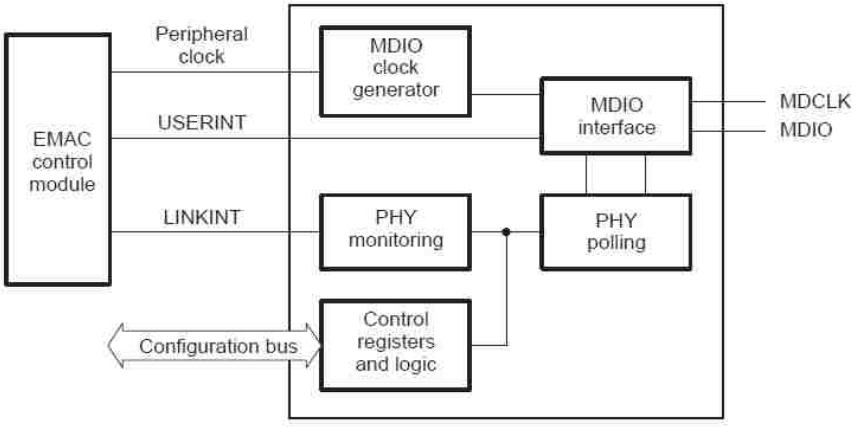


Figure 5.6 MDIO Module Block Diagram

5.2.1. Initializing the MDIO Module. The following steps are performed by the application software or device driver to initialize the MDIO device:

1. Configure the PREAMBLE and CLKDIV bits in the MDIO control register (CONTROL).
2. Enable the MDIO module by setting the ENABLE bit in CONTROL.
3. The MDIO PHY alive status register (ALIVE) can be read in polling fashion until a PHY connected to the system responded, and the MDIO PHY link status register (LINK) can determine whether this PHY already has a link.
4. Setup the appropriate PHY addresses in the MDIO user PHY select register (USERPHYSELn), and set the LINKINTENB bit to enable a link change event interrupt if desirable.
5. If an interrupt on general MDIO register access is desired, set the corresponding bit in the MDIO user command complete interrupt mask set register (USERINTMASKSET) to use the MDIO user access register (USERACCESSn). Since only one PHY is used in this device, the application software can use one USERACCESSn to trigger a completion interrupt; the other USERACCESSn is not setup.

6. RESULTS

6.1. SERIAL PORT / COM

6.1.1. User Interface. A simple COM interface using MATLAB shows UI with four menu options

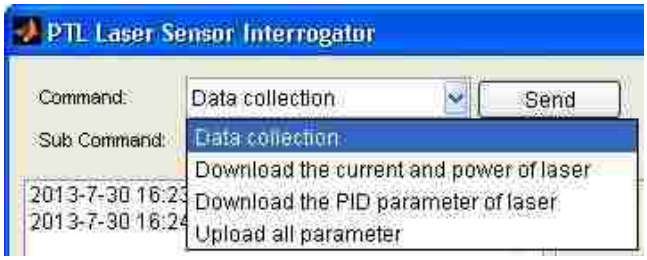


Figure 6.1 Sample UI Menu

6.1.2. Data Transfer. When Laser board is not connected, FSI detects the disconnection and transmits default zero data as shown in below figure.

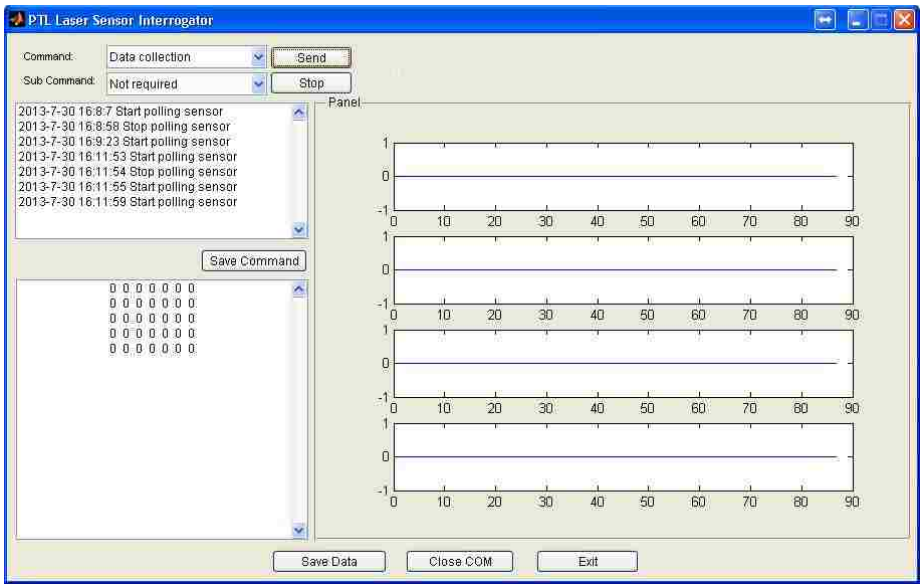


Figure 6.2 COM data to PC without Laser board connected

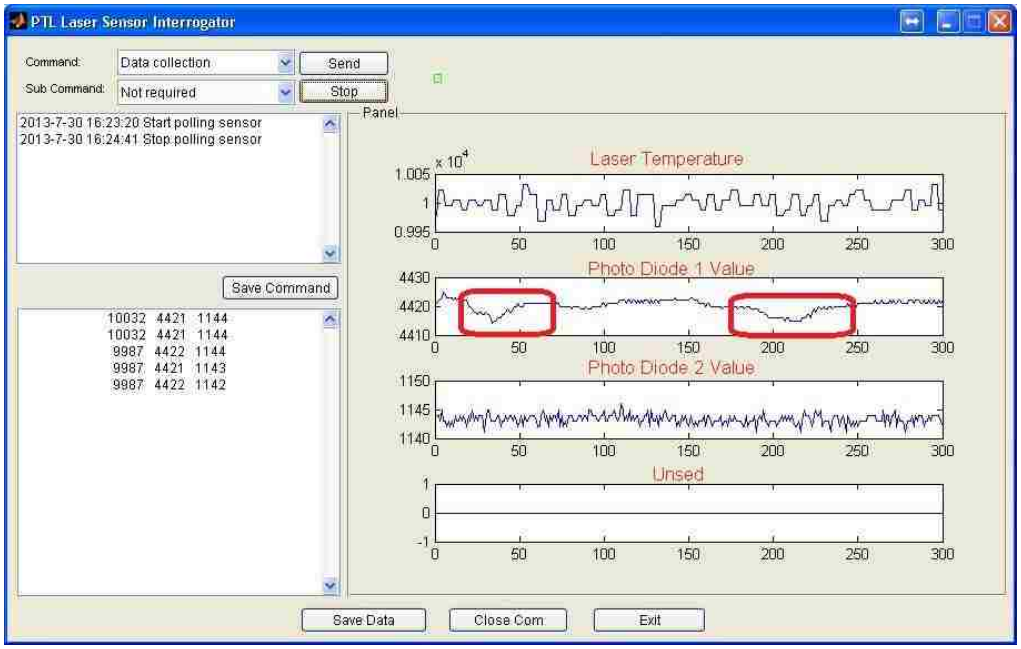


Figure 6.3 COM data to PC when Laser board connected

Whereas, above figure shows the measured laser temperature, Photo diode 1 and Photo diode 2 sensor data. The data is multiplied by 1000 and floating point is truncated for speeding up transfer.

The two valleys correspond to shining of white light to cleaved open end of plain fiber optic cable connected to photo diode 1. IIR is applied to remove any noise before the data is sent, however similar filters on the PC end can be applied depending on the noise filter tolerance level of the end application.

6.2. UNIVERSAL SERIAL BUS (USB)

6.2.1. USB Driver. Windows side DDK build for USB end driver is verified by a small test program that can identify this specific connected device (based on manufacturer id).

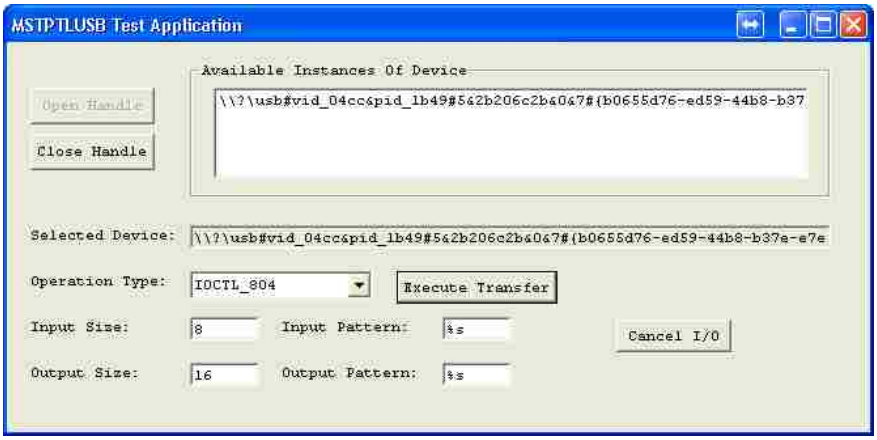


Figure 6.4 PC side USB DLL Test

6.2.2. Data Transfer. FSI USB protocol has four different channels through which data can be exchanged. In this implementation we use first channel dedicatedly for parameter download and updates. Second channel is used for high speed (16B, 64B and 512 B) data transfer per burst.

As we discussed in section 3.2 Overview of Features, this implementation can support all control, bulk, interrupt, and isochronous. However we use interrupt type of transfer only. This decision is based on the type of data we need to transfer. For bulkier data transfers the implementation can be tweaked to confirm bulk or isochronous modes. Tests recorded an average transfer rate of 3.8MBps @ 512 bytes buffer.

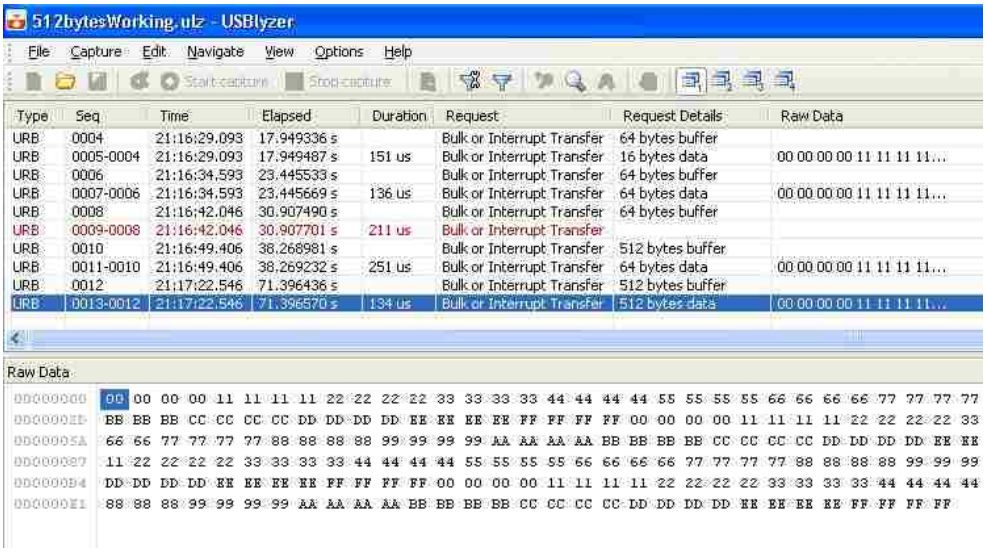


Figure 6.5 USB Data Transactions

6.3. ETHERNET

Some issues were discovered in Ethernet MDIO and debugging is in progress. The test results for Ethernet will be included in future work.

6.4. WIRELESS

Wireless module is not yet tested and will be included in future work.

APPENDIX - CODE OVERVIEW

Table 6.1 Code layout for Flash Utility

File Name	Description
\05_LaserController\burn\main.c (LOC : 102)	
Main()	Initialize 300MHz PLL, Initialize NOR Flash, Burn DSP_Firmware.bin onto the flash
\05_LaserController\burn \evmomapl138_nor.c (LOC : 149)	
WriteFlash()/ReadFlash()	Writes/Reads to/from NOR flash

Table 6.2 Code layout for User Interface (COM)

File Name	Description
\05_LaserController\com\com_4.m (LOC : 517)	
com_4()	Initialize COM3, with even parity and prepare UI com_4.fig
\05_LaserController\com\UARTTx.m (LOC : 9) & SendCommand.m (LOC : 62)	
UARTTx / SendCommand	Transmit data over COM3 to FSI
\05_LaserController\com\UARTRx.m (LOC : 135)	
UARTRx	Receive data over COM3 to FSI
\05_LaserController\com\Blink.m (LOC : 10)	
Blink	Blink image to show activity

Table 6.3 Code layout for DSP Program

File Name / Description	
\05_LaserController\dsp\init.c (LOC : 183)	
OUT_INIT	initialize external communication buffer
ADC_DAC_INIT, ADC_DAC_CONFIG	Initialize / Config ADC/DAC Chip
FLASH_INIT	Initialize Flash memory
FPGA_INT_INIT	Initialize FPGA interrupt
\05_LaserController\dsp\func_pid.c (LOC : 107)	
ControlLaserTemp	Control laser temperature
CLT_GetCurrTemp	Get current laser temperature
CLT_ControlLaserPower	Get current laser power
CLT_DAC2TECControl	Generate dac to TEC control
\05_LaserController\dsp\func_out.c (LOC : 163)	
UARTRxData	
HandleUARTEndPoint	
UARTTxData	
\05_LaserController\dsp\func_flash.c (LOC : 181)	
flash_store	Store downloaded parameter to flash
NOR_read, NOR_write, writeFlash, readFlash, NOR_sector_erase, NOR_block_erase, NOR_chip_erase, NOR_master_erase	Handle NOR flash for erase, read, write
\05_LaserController\dsp\main.c (LOC : 153)	

TEST_mii_loopback	Test MII loopback mode
main()	Initialize Flash, NOR, COM, USB, Ethernet and Wireless modules, Deamonize algorithms to control, collect and serve request over COM, USB, Ethernet and Wireless
\05_LaserController\dsp\func_adc.c (LOC : 296)	
gsLaser[]	One element each for connected active LASER board
CollectSensorValues()	process data from photo diode
IIR_filter()	lowpass filter algothrm
update_laser_driver()	update current or power to board
RefreshLaserPower()	refresh laser power when it is current driven
\05_LaserController\dsp\tms320c6748_usb_enum_desc.c (LOC : 186)	
gDevDesc, gConfDesc, gEPDesc, gLangDesc	Default values of end point descriptors
\05_LaserController\dsp\tms320c6748_usb_enum.c (LOC : 309)	
InitializeOTGUSB	Initialize OTG USB, IdleMode,
EnumerateUSB	Enumerate USB capabilities
IdleMode	Handle Idle mode state machine
USBSendDescriptors	Transmit descriptors when doing handshake
USBTxData	Transmit USB data
SetupEndPointFIFOs	Establish FIFOs for Tx/Rx
\05_LaserController\dsp\tms320c6748_timer.c (LOC : 81)	
Timer_int	Configure TIMER0 for interrupts and TIMER1 for calculating delays
Timer_delay	Cause delay
\05_LaserController\dsp\tms320c6748_psc.c (LOC : 181)	
EVMOMAPL138_pinmuxConfig	configures the desired pin mux register
EVMOMAPL138_lpscTransition	Cause transition
PSC0_LPSC_enable,	Manage LPSC

PSC1_LPSC_enable, PSC1_LPSC_force, PSC_All_On_Full_EVM, PSC1_LPSC_SyncReset	
\05_LaserController\dsp\tms320c6748_led.c (LOC : 193)	
LED_init, LED_isOn, LED_turnOn, LED_turnOff, LED_toggle, convertLedToI2CGpio	Manage LED
\05_LaserController\dsp\tms320c6748_isr.c (LOC : 146)	
Interrupt_disable_int, Interrupt_int, TIMER0_IR	Enable / Disable interrupts
OUT_int	Prepare the output buffer for COM
ADC_10k_int	ADC timer interrupt
\05_LaserController\dsp\tms320c6748_usb_proc.c (LOC : 109)	
HandleUSBEndPoints	Respond to USB interrupt requests
\05_LaserController\dsp\tms320c6748_emac.c (LOC : 769)	
initMdioPhy	Initialize MDIO PHY interface
isLinkActive	Check if link is active
phyRegRead, phyRegWrite	Read / Write to PHY register
EMAC_init	Initialize the EMAC interface
EMAC_rxPacket	Entry point for receiving packet over EMAC
EMAC_txPacket	Entry point for sending packet over EMAC
EMAC_phyPowerOn, EMAC_phyPowerDown	Power on / off PHY
EMAC_phyEnterLoopb	Enable / Disable loopback for testing purposes

ack, EMAC_phyExitLoopback	
\05_LaserController\dsp\msg_data_proc.c (LOC : 154)	
RefreshMFIFO, RefreshDFIFO, RefreshRxBuffer	Refresh data structures to sync values from external interfaces
USBRxData	Receive USB data
USBFIFOInit, USBRxBufferInit	Initialize FIFO for Rx/Tx USB data
\05_LaserController\DSP\tms320c6748_emif_norflash.c (LOC : 56)	
EMIFA_NAND_PINMUX / EMIFA_NOR_PINMUX	Initialize the NOR and NAND flashes
\05_LaserController\DSP\tms320c6748_gpio.c (LOC : 105)	
GPIO_INT, GPIO_SET_IR_INT, GPIO_SET_TRIG	Initialize the GPIO

Table 6.4 Code layout for USB WDK Driver

File Name	Description
\USB\dll\kfifo.c (LOC: 343) - A simple kernel FIFO implementation	
\USB\dll\MSTPTLDLL.cpp (LOC: 554) - Defines the entry point for the DLL application	
init_dev	Initilaize the DEV handler
valid_dev	Validate the DEV handler
MST_USBDev_Open	Open driver handler
MST_USBDev_Close	Close driver handler
MST_WriteP1	Write to Pipe1
MST_ReadP1	Read from Pipe1
MST_WriteP2	Write to Pipe2
MST_ReadP2	Read from Pipe2
MST_WriteP3	Write to Pipe3
MST_ReadP3	Read from Pipe3
MST_WriteP4	Write to Pipe4
MST_ReadP4	Read from Pipe4
\USB\driver\MSTPTLUSBDevice.cpp (LOC: 2754) - functions to handle requests (IRPs) from the system	
MSTPTLUSBDevice	This is the constructor for the class
~MSTPTLUSBDevice	This is the destructor for the class
DefaultPnp	This routine just passes the IRP through to the lower device
OnStartDevice	Initialize the hardware device.
OnStopDevice	the stop device IRP
OnRemoveDevice	deleting the device object
OnQueryCapabilities	The Bus driver fills in the device capabilities structure
OnQueryCapabilitiesComplete	This method is called when the IRP is completed to alter the device capabilities reported by the bus driver
TestBusInterface	This routine uses the USB direct client interface to query

	information
OnDevicePowerUp	The bus driver has completed the IRP and this driver can now access the hardware device
OnDeviceSleep	The hardware has yet to be powered down and this driver can now access the hardware device
Create	Dispatch routine for IRP_MJ_CREATE requests
Close	Dispatch routine for IRP_MJ_CLOSE requests
Read	Read requests
Write	Write requests
DeviceControl	Device Control
\USB\driver\MSTPTLUSBDriver.cpp (LOC: 162)	
DriverEntry	This routine is called when the driver is loaded
AddDevice	This routine is called when the system detects a device for which this driver is responsible
Unload	This routine is called when the driver is unloaded
\USB\driver\MSTPTLUSBQueue.cpp (LOC : 313) - This class implements a driver managed queue that serializes IRP	
StartIo	system queuing
MSTPTLUSB.cat / MSTPTLUSB.inf (LOC : 149) - Install information file for MSTPTLUSB Driver (cat is for signed drivers)	
\USB\dtest\MSTPTLUSBDTest.cpp (LOC : 678) - Methods to input / output text in the window and open / call / close correct instance of driver	

BIBLIOGRAPHY

[1] E. A. Mendoza, J. Prohaska, C. Kempen, S. Sun And Y. Esterkin, "Fully Integrated Miniature Multi-Point Fiber Bragg Grating Sensor Interrogator (FBG-TransceiverTM) System for Applications where Size, Weight, and Power are Critical for Operation," *6th European Workshop on Structural Health Monitoring - Poster 7*

[2] Jens-Thomas Wernicke, Rain Byars, Jody Shadden, Christiane Schmoeller, "Production Integration Of Fiber Optical Sensors Embedded In New Rotor Blades For Real Time Loads Feedback," *European Wind Energy Conference 2007 May 9, 2007, Milan, Italy.*

[3] Amardeep Kaur, Sriram Nagarajan, Sudharshan Anandan, Lei Yuan, K. Chandrashekhara, et al. " Embeddable fiber optic strain sensor for structural monitoring," *Proc. SPIE 8692, Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2013, 86921W (April 19, 2013); doi:10.1117/12.2009462; http://dx.doi.org/10.1117/12.2009462*.

[4] Van Hoe B, Lee G, Bosman E, Missinne J, Kalathimekkad S et al., "Ultra small integrated optical fiber sensing system," *Sensors (Basel). 2012;12(9):12052-69. doi: 10.3390/s120912052. Epub 2012 Sep 3*

[5] Alan D. Kersey, "Interrogation and multiplexing techniques for fiber Bragg grating strain sensors," *Proc. SPIE 2071, Distributed and Multiplexed Fiber Optic Sensors III, 30 (December 30, 1993); doi:10.1117/12.165923.*

[6] Hongo, A., Kojima, S. and Komatsuzaki, S. (2005), "Applications of fiber Bragg grating sensors and high-speed interrogation techniques," *Struct. Control Health Monit., 12: 269–282. doi: 10.1002/stc.70*

[7] Bookham Technology, Data sheet for DRP2EX (43T2REV 1 Feb 2003)

[8] Lucent Technologies Microelectronics group, Data Sheet for D2570H (February 2000).

[9] Texas Instruments "TMS320DM644x DMSoC Universal Serial Bus (USB) Controller," (Literature Number: SPRUE35G, June 2010).

[10] Texas Instruments "TMS320C674x/OMAP-L1x Processor Ethernet Media Access Controller (EMAC)/ Management Data Input/Output (MDIO) Module" (Literature Number: SPRUFL5A May 2010).

[11] Texas Instruments "TMS320C6748 DSP Development Kit," (Literature Number: SPRT633 2012).

[12] Xilinx Virtex-5 FPGA Configuration User Guide," (UG191 (v3.11) October 19, 2012)

[13] Rodrigo A. Silva-Muñoz, Roberto A. Lopez-Anido, "Structural health monitoring of marine composite structural joints using embedded fiber Bragg grating strain sensors," *Composite Structures 89 (2009) 224–234*

[14] Hong-Nan Li, Dong-Sheng Li, Gang-Bing Song, Recent applications of fiber optic sensors to health monitoring in civil engineering, Engineering Structures," *Volume 26, Issue 11, September 2004, Pages 1647-1657, ISSN 0141-0296, http://dx.doi.org/10.1016/j.engstruct.2004.05.018*

VITA

Tameem Ahmed Khan was born in Mysore, Karnataka, India on February 17, 1982. He completed his Bachelor of Engineering in Computer Science from Visveswariah Technological University, Belgaum in July 2003. After his undergrad, Tameem worked as a Software Engineer for 7 years at Wipro Technologies, India (deputed at Toshiba, Japan for 3 years), and then as Research Engineer for 1 year at University of Missouri, Columbia, MO.

Because of his interests in sensors and embedded systems, Tameem started his Master of Science in Computer Engineering on a distance program in January 2012 later in June he moved as a full time student at Missouri University of Science and Technology, Rolla. He worked as a research assistant on sensor interfacing projects at Photonics Technology Lab under Dr. Hai Xiao. He received his masters in August, 2013.