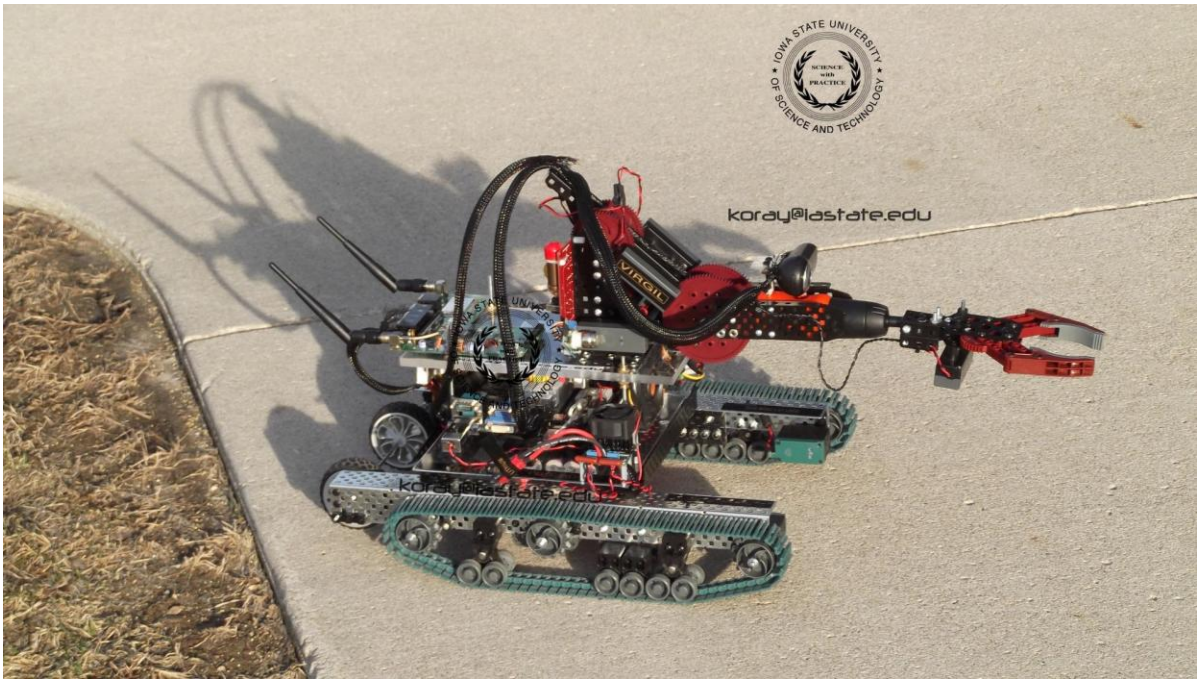
Serial Data Hub

Design Document



Senior Design Group DEC13-13
Steven LeBlanc, Adriana Ceylan, Darin Cleveland, Justin Wheeler
Adviser: Dr. Arun K. Somani
Client: Dr. Koray Celik

Introduction:

Objective:

Our task is to build a serial data hub. The peripheral will be a separate module that connects to a Linux laptop by a USB cable and then connects to sixteen other serial ports on various robotics components of a machine. The hardware logic will be implemented on an Altera Cyclone II FPGA mounted on a PCB board with RS-232 connectors. The FPGA will run firmware with Altera's Nios II soft-core processor. Communication will be handled via an API on a laptop running Ubuntu v11.04.

Document Overview:

In this design document, we will discuss the main requirements and deliverables to our client and advisor. This document will also highlight in lower level detail, the system and subsystem architecture, programs and tools used to create the architecture, and languages used.

Deliverables:

-DE2 board containing an Altera Cyclone II FPGA loaded with a custom Nios II processor and firmware.
-Quartus project files for custom Nios II processor
-Verilog code for custom hardware module
-Firmware code to run on the soft processor
-Interconnect Schematics and System Block Diagrams
-Device API for Ubuntu 11.04 Linux Laptop

System Level Design:

Functional Requirements:

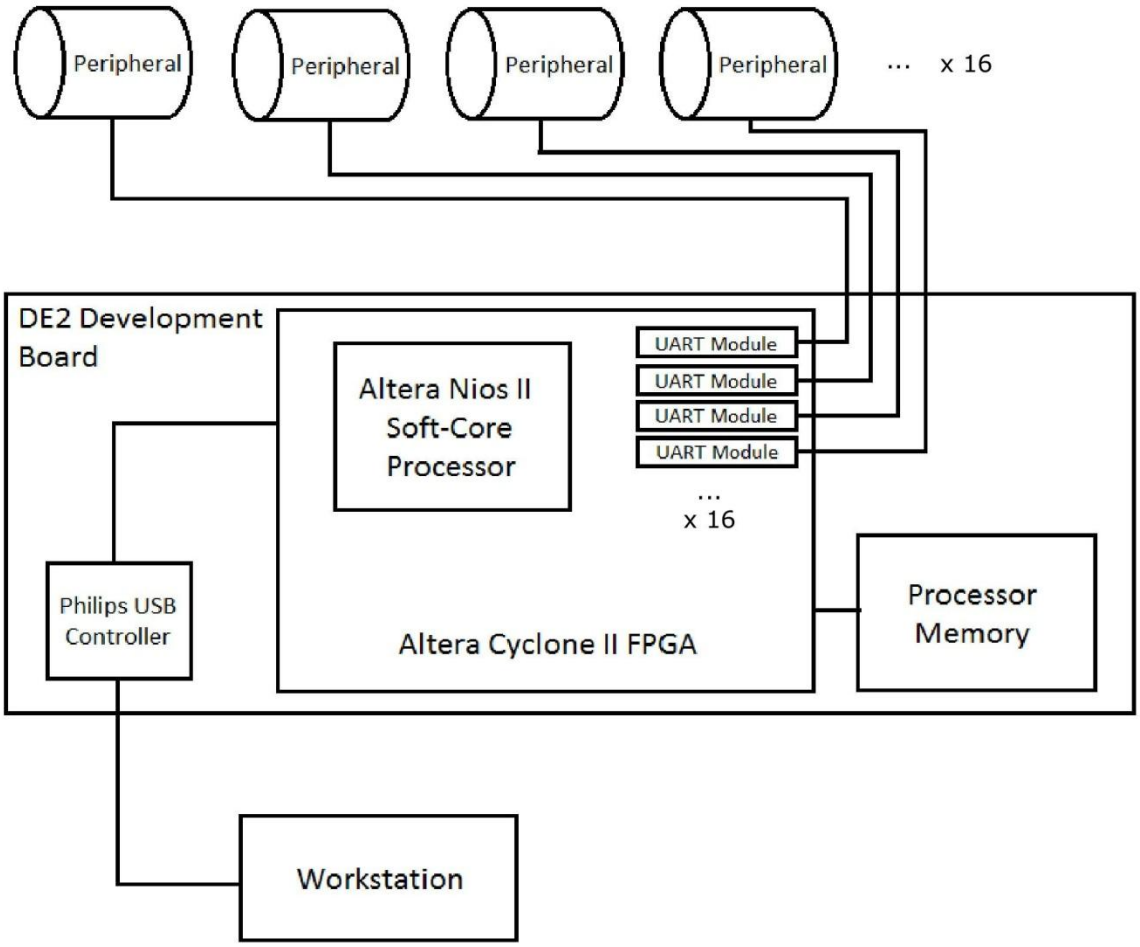
System must provide multiple serial port interfaces to connect robotics peripheral components
System must provide a serial port interface to connect to controlling PC
System must transfer data between PC and component in a way that ensures no data is lost
System must manage 16 to 1 data throughput channels
System must transfer data in a prompt manner to reflect real-time systems
System must be controlled by an API library on the PC

Non Functional Requirements:

System shall implement an Altera Nios II soft processor.
System shall use the soft processor to run firmware code on FPGA.
System shall be prototyped on Altera's DE2 Development Board
System shall use USB 2.0 to communicate between the FPGA and the PC API.
System shall use same USB controller as the DE2 Board
System shall have 16 serial ports implementing RS-232
System shall use external memory modules rather than FPGA memory

Design Architecture:

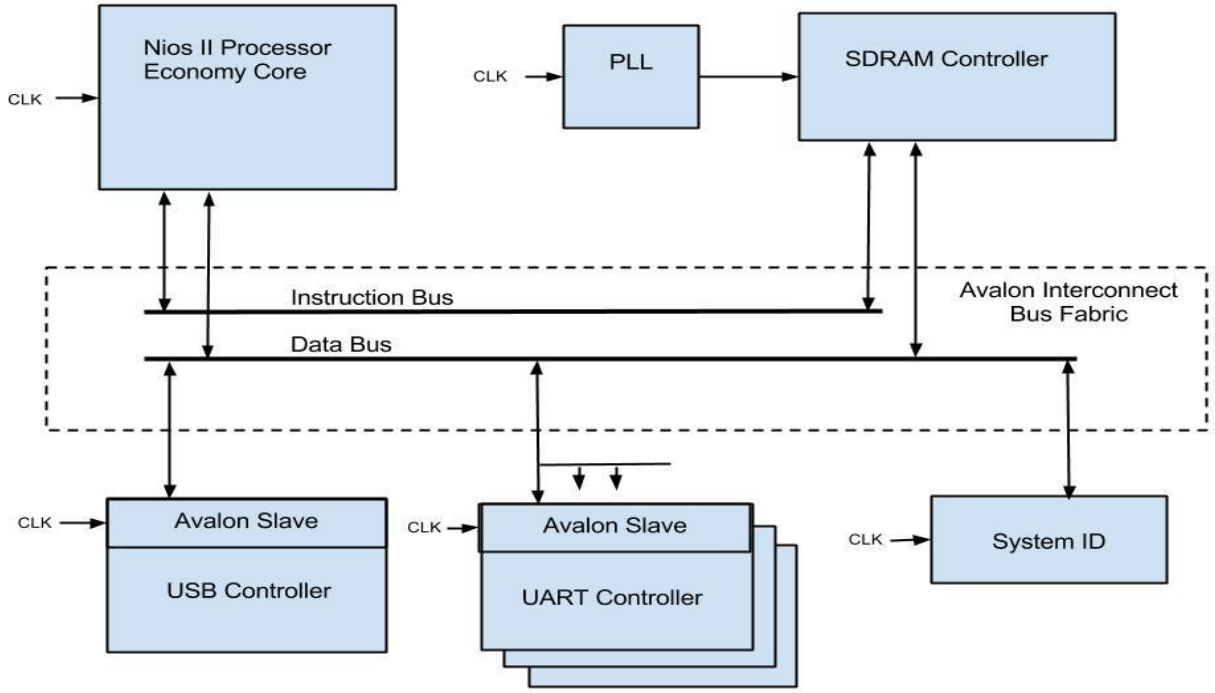
System Architecture:



The system architecture consists of three major structures. The software interface consists of a C++ API that will send commands and data to the system over a USB connection. The software interface has been designed to work on Ubuntu 11.04 Linux. The data being sent over USB will be handled on the other end by the soft processor. Altera's Nios II soft processor will be implemented on Altera's Cyclone II FPGA. From the CPU, data will be passed through the UART structure. The UART structure contains sixteen parallel UARTs that will output the data through 33 serial pins (16 pairs of TX and RX pins and then one pin for a common ground to be shared across all devices).

FPGA Inner System Architecture

On the System Module, a soft processor will be implementing on the Cyclone II. It will feature Altera's Nios II soft processor and various other IP components that are interconnected with Avalon Bus Technology.



Processor

Nios II processor. Economy Version.
Smallest of the three offered by Altera. Does not require a specific license.

Memory

Off-chip Memory. (meaning off of the FPGA chip)
IP cores for an SRAM memory controller. (not SDRAM)

USB IP

Third-party module to interact with on-board USB controller
Adapted to the Avalon Interconnect Fabric

UART IP

Created our own UART IP module
Needed to explicitly make connections to the Avalon Interconnect Fabric

Misc Modules

System ID module. Used for Firmware programming.
System Clock Module. Running at 50 MHz.

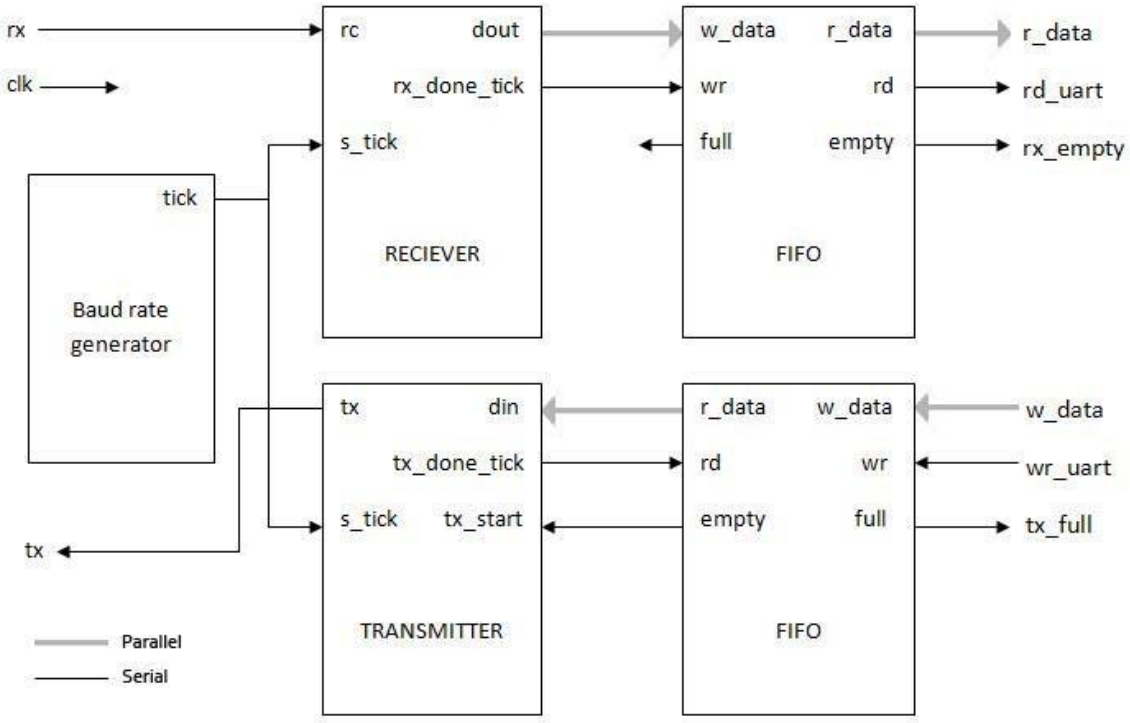
Interaction with USB Controller:

The DE2 board is embedded with a USB Controller that can act as either a USB host or a USB device. The controller is the ISP1362 made by Philips. The controller is operated by writing to its data pins asynchronously. Writing to the data pins will read and write data from the controller's registers. This is the way that controller is designed. A third party VHDL module was found that can do all low-level interaction with the ISP1362. It set up the controller to act as a USB device and to do two-way bulk transfers. We modified the module to conform to the Avalon Bus interface and then connected the module to our system.

Firmware:

The firmware is written in C, and will be programmed on top of the Nios II processor. The firmware will receive messages from both the computer via USB, and the motors and sensors via 16 UARTs. Once it receives the message from the USB it takes the assigned header, port number from the message and sends the command to the respective UART and on to the motor. When it receives a message from the UARTs and motors, the firmware sends a response message back to the USB indicating the command was completed. Each message is sent and received in 1 byte segments.

UART Architecture:



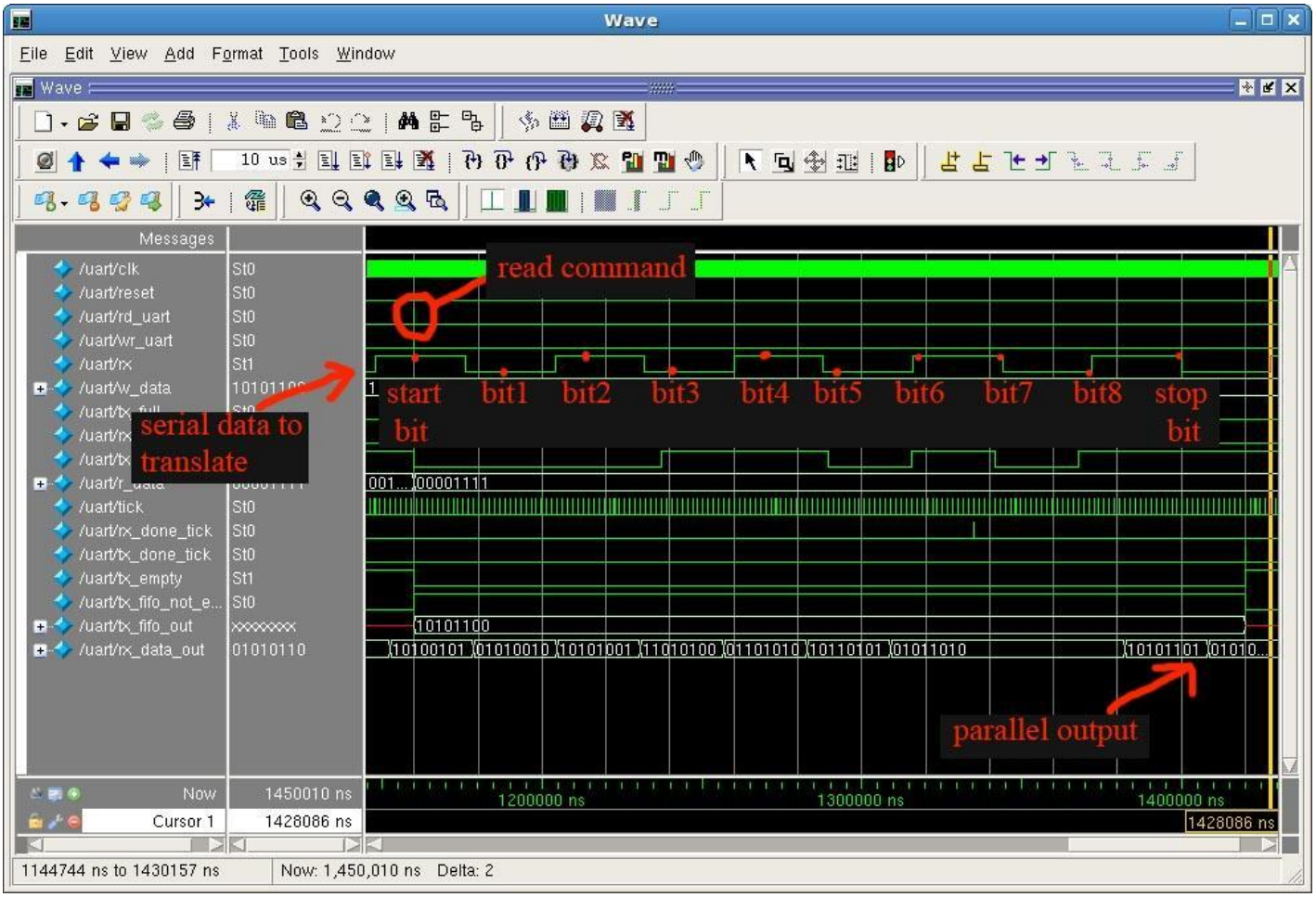
The UART (Universal Asynchronous Receiver and Transmitter) is a circuit that takes parallel data and converts it to be sent through a serial line, and vice-versa. This is a crucial component to the project, as it facilitates communication between the FPGA and the motor/sensors it's supposed to interface with. The UART's two primary components are a transmitter and a receiver. The transmitter is a specialized shift register which takes in the parallel data and shifts it out across the serial line bit by bit, following a timing pattern 16 times greater than the clock, when triggered. The receiver does the reverse: when triggered, it takes in serial data (polling at a timing pattern 16 times greater than the clock), stores the bits as they file in, and then sends them out together on the parallel line. Two FIFOs are integrated into the UART and used as buffers. This is in case data is send/received faster than the UART can handle, and will assure steady data movement and reduce the risk of losing that data.

Our UART code was modeled from “FPGA Prototyping by Verilog Examples Xilinx Spartan -3rd Version” by Pong P. Chu. The textbook was recommended by our client, as UARTs are commonly used pieces of code and time would be spent more efficiently copying an existing version instead of creating one from scratch.

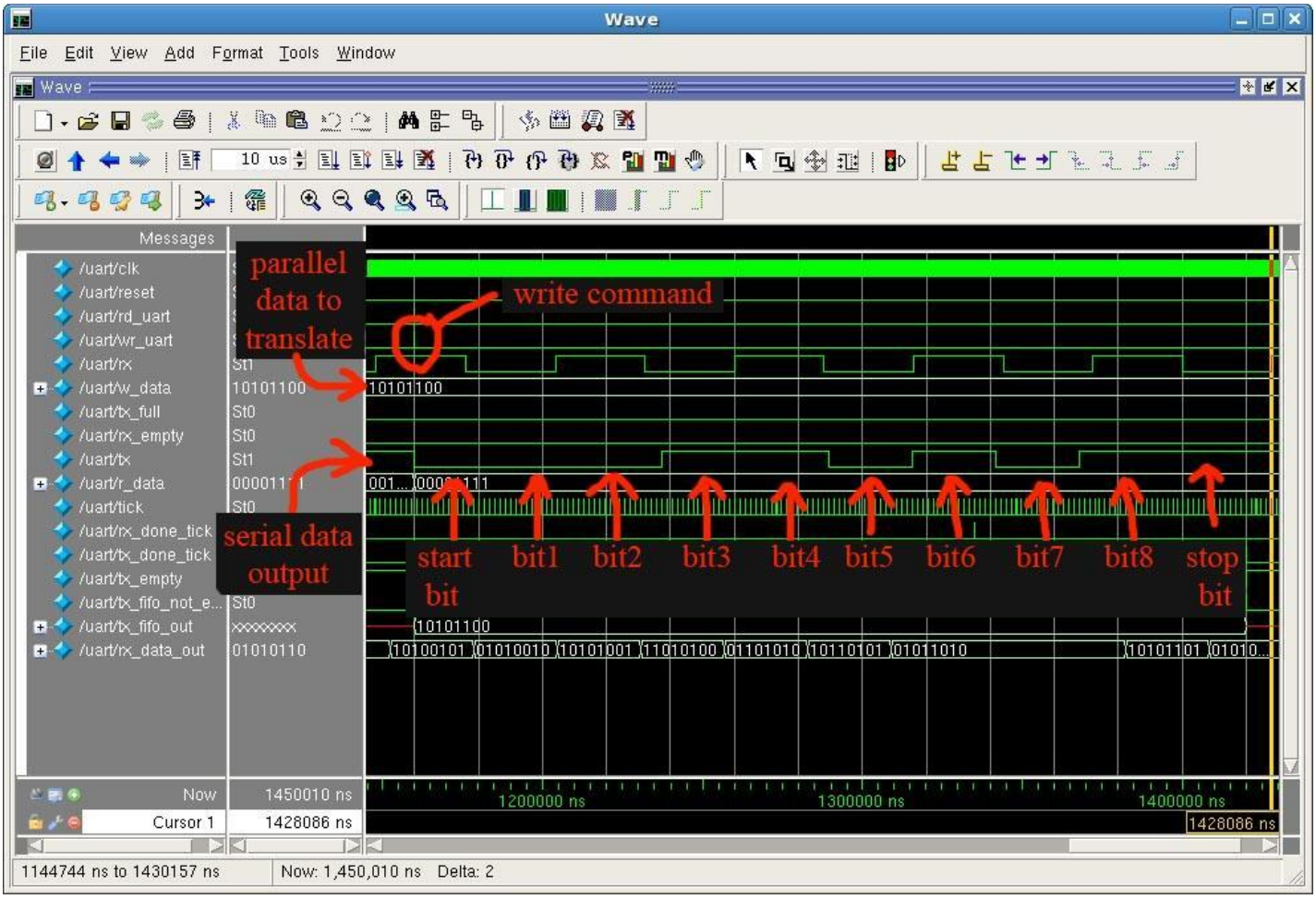
UART Testing

After testing individual modules of the UART, two tests were designed to test the whole UART: one using Modelsim and one implemented on the FPGA. For the first inclusive test, the main module uart.v was simulated with Modelsim. From the Wave page, the simulation's running time was controlled along with the designated input wires. Parallel data was manually fed to the transmitter and after the "write" wire was toggled, the transmitter outputted serial data appropriately. Serial data was then manually generated on the line feeding the receiver, along with a toggling of the "read" wire. After polling the serial data for its proper start bit, eight data bits, and stop bit, the receiver outputted parallel data properly. The UART functioned as intended under ModelSim simulation.

Receiver test:



Transmitter test:



A test was created with the intent of writing the code to the FPGA and testing the UART's functionality on the hardware.

uart_test_serialShort shorted the receiver and transmitter together on their serial sides. It created a module that would echo back a character over the parallel hookups of the UART. Theoretically, if the transmitter had parallel data to send, it would be translated into serial, immediately feed back into the receiver, be retranslated into parallel data, and the receiver would output parallel data. The source of the parallel data was a set of eight switches on the Altera board. Whenever the trigger was toggled, the LEDs switched to match the eight switches. The test was successful for every switch combination tested and as it tested all components of the UART (transmitter, receiver, FIFO, baud rate generator) the UART was deemed fully functional on the Altera board.

See "UART Code" for source code and test code.

Software Architecture:

Overview

We will deliver, as stated above, a device driver and an API that will work on an Ubuntu 11.04 Linux laptop. Our software will take commands from the client's control program and send them off to the FPGA system. Our API will package the commands in our protocol and send them over the USB connection. Our FPGA system will receive them and will forward the packets accordingly. Our protocol is compliant with our client's current 40-bit commands as well as allowing flexibility to use alternative command and data structures.

Detailed Description

Our API contains several functions that help a Linux computer communicate with a USB device that the user defines in the source code. Before using the library, if the user is using a different device from the one that our client is using, they need only change some of the #defines in our source code to give the correct vendor ID, product ID, and the endpoints of the device they are using in order to get the library to work for them.

When the user wishes to establish a communication channel between the Linux computer and the USB device, they call initialize_connection() and save the libusb_session_info struct that is returned by this method. This struct contains two other structs, a libusb_device_handle which will serve as an argument for most of the other functions in the API, and a libusb_context, which is used in the libusb_close_connection() function for cleanup. The initialize_connection() method works by first initializing the libusb library for use on the current system, then by calling libusb_get_device_list() in order to get a pointer to an array of libusb_device structs that describe all the USB devices that are currently attached to the Linux computer. The function then checks each of these libusb_devices to see if any of them match the vendor ID and product ID of the desired device. If so, then it calls libusb_open() on that device and gives the user a libusb_device_handle() to use. After that, it does a bit of housekeeping, claiming an interface on the device to use and detaching the active kernel driver if present so it doesn't get in the way of what the user wishes to do. Finally, it returns a struct containing the libusb_context and the libusb_device_handle structs.

Once the libusb_device_handle has been obtained, the user can call the other functions in the API, all of which take a libusb_device_handle as one of their arguments. At this point these include send_motor_command() and query_motor_info(), each of which take in several arguments defining which motor or sensor they wish to communicate with and what data or command they wish to send to the motor/sensor. The functions then convert that information into the proper sequence of bytes to send to the motor/sensor, and then send that sequence to the motor/sensor using libusb_interrupt_transfer(), which takes in a pointer to an array of bytes to be sent_received, a libusb_device_handle, the number of bytes to be transferred, a pointer to a location to place the actual number of bytes that were transferred, the endpoint of the device to use (which determines whether the function is *sending* or *receiving* bytes, and a number defining how long to wait for the bytes to be transferred before giving up (a 0 means that the function should never give up). In the same way, if the command sent is one that will prompt output to be sent back to the Linux computer from the device, the user simply calls

libusb_interrupt_transfer() again, but with a different endpoint selected, one that tells the function to receive data, rather than send it. The received data is placed in the location defined by the pointer argument that also defines where the data to be sent is located when libusb_interrupt_transfer() is being used to send data, where it can be used by the user.

When the user is done communicating with the device and wants to cut off communication, they call the close_connection() function. This function takes as an argument the same struct that was returned by initialize_connection() back when the connection was established. It releases the device interface, closes the libusb_device_handle, and exits the libusb_context session. This essentially serves as the cleanup/garbage collection function.

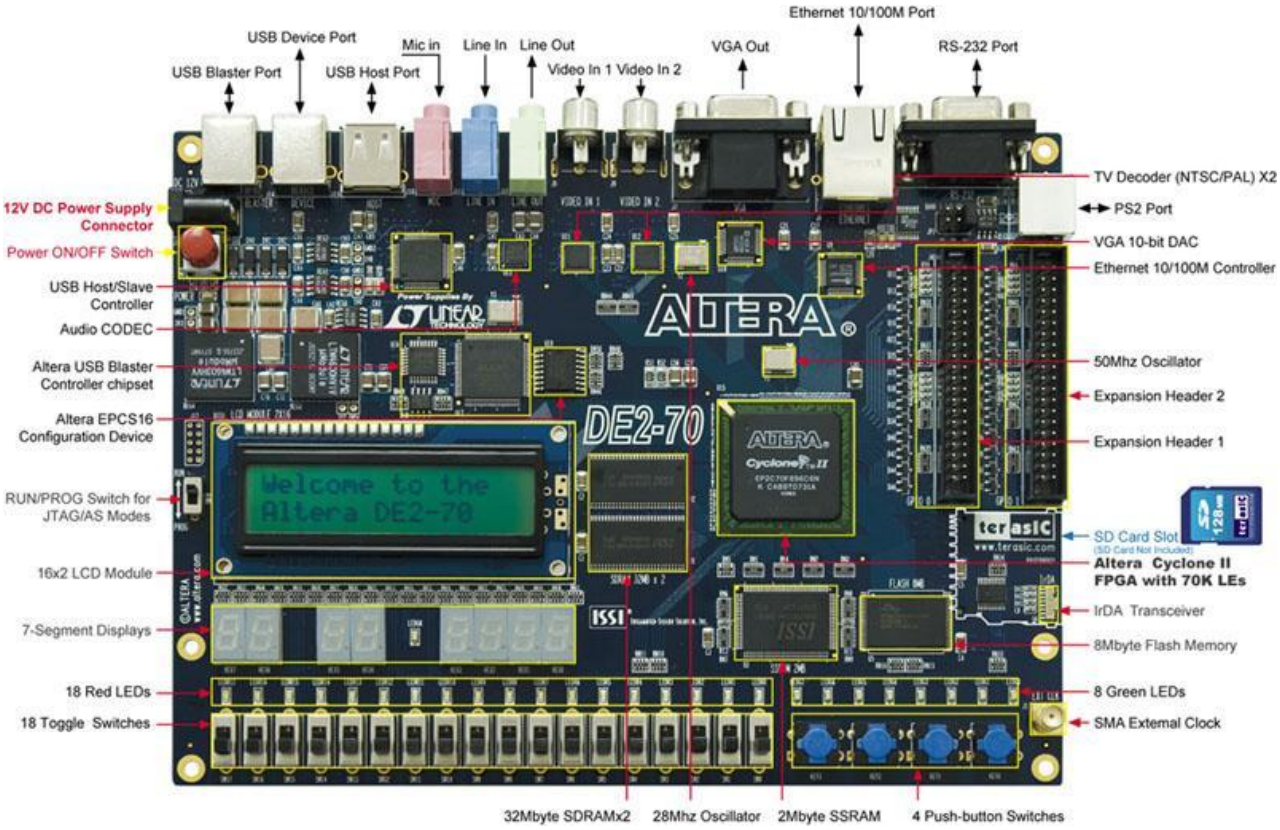
System Integration:

Qsys is a system integration tool part of the Quartus II library of software. It and Nios II build tools will be predominantly used to generate the code to be loaded onto the FPGA and run the Serial Data Hub. Qsys is used to “wire” together predefined IO hardware modules (the CPU, memory, clock, pio ports, USB controller, and the UART we created) to create a hierarchy that will allow the system to function. That gets compiled and Quartus II references it. Using Nios II Software Build Tools for Eclipse (referencing the files generated by Qsys), we program the main.c file which will dictate how inputs and outputs are handled between hardware components. A Board Support Package and reference files are generated from this, and act as a pseudo-operating system. Quartus II takes all these files and compiles them into code to be uploaded to the DE2 board’s FPGA and run.

Tools Used:

Hardware:

-Altera Cyclone II FPGA development board (DE2)



The DE2 box includes:

- The 8 x 6 inch DE2 board with a Cyclone II EP2C35 (672-pin package) FPGA
- 9V AC/DC adaptor
- USB cable
- Plexiglas cover for the DE2 board
- Installation guide

Specifications:

DE2 Board Information	
Feature	**Description**
FPGA	• Cyclone II EP2C35F672C6 with EPCS16 16-Mbit serial configuration device
I/O Interfaces	• Built-in USB-Blaster for FPGA configuration • Line In/Out, Microphone In (24-bit Audio CODEC) • Video Out (VGA 10-bit DAC) • Video In (NTSC/PAL/Multi-format) • RS232 • Infrared port • PS/2 mouse or keyboard port • 10/100 Ethernet • USB 2.0 (type A and type B) • Expansion headers (two 40-pin headers)
Memory	• 8 MB SDRAM, 512 KB SRAM, 4 MB Flash • SD memory card slot
Displays	• Eight 7-segment displays • 16 x 2 LCD display
Switches and LEDs	• 18 toggle switches • 18 red LEDs • 9 green LEDs • Four debounced pushbutton switches
Clocks	• 50 MHz clock • 27 MHz clock • External SMA clock input

(http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html)

-Laptop with Linux Ubuntu v11.04

Software:

Altera Quartus II:

A licensed Altera software tool used for analysis and synthesis of hardware descriptive language (HDL) design. This program enables the developer to compile their design, perform timing analysis, simulate designs and configure the target device.

ModelSim:

A licensed software tool that help simulates, compiles, and debugs VHDL or Verilog designs without loading the design on a FPGA.

MegaCore Library:

A licensed library for the Altera Quartus II software. This library contains a list of licensed Altera cores. The Nios II embedded soft processor is located in this library.

Code:

High Level API code:

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <stdlib.h>
#include "libusb.h"
/* If you are NOT connecting to an Altera USB-Blaster cable, you will
 * need to find the correct IDs and endpoints for your device and enter them in the .cpp file.
#define VENDOR_ID 0x09fb
#define PRODUCT_ID 0x6001
#define DEVICE_INTERFACE 0
#define OUT_ENDPOINT 2
#define IN_ENDPOINT 0x86
/* If these numbers are different for your device, feel free to change them accordingly in the .cpp file.
#define NUMBER_OF_MOTORS 16
#define MAXIMUM_SPEED 255*/

/* This struct is initialized and returned by initialize_connection().
 * It contains two parameters: a libusb_context and a libusb_device_handle.
 * These are necessary because initialize_connection must return both of those values
 * as close_connection() needs both of them to fully clean up everything.*/
struct libusb_session_info{
 libusb_context *ctx;
 libusb_device_handle *dev_handle;
};

/* Sets up a connection with the device defined by all the #defines above.
 * When you are finished with the connection, please call close_connection()
 * for garbage collection. */
libusb_session_info initialize_connection();
/* Takes in the motor number, direction, and speed that the command should contain,
 * as well as the libusb_device_handle* that denotes which device the command is to
 * be sent to, converts that info to the command format that the device will accept,
 * and sends the command to the device.*/
int send_motor_command(int motor_num, int direction, int speed, libusb_device_handle *dev_handle);
unsigned char* query_motor_info(int motor_num, libusb_device_handle *dev_handle);
/* Sends a user-defined array of bytes to the device, along with the motor number that it is to be sent to. */
int send_custom_command(int motor_num, unsigned char *data, int num_bytes, libusb_device_handle *dev_handle);
/* Sends a user-defined array of bytes to the device, along with the motor number that it is to be sent to.
 * Also receives an array of bytes fromt the device in return, and placed it in a user-defines location.
 * User will need to provide the number of bytes to expect. May remove this limitation later in order to
 * give more flexibility.*/
int send_custom_command_and_receive_data(int motor_num, unsigned char *data_out, int num_bytes_out, unsigned char
*data_in, int num_bytes_in, libusb_device_handle *dev_handle);

/* Sends the byte 0xAE to the device, returns 0 if it receives 0xAF, returns 1 at all other times.
 * Only of use to our particuar project for testing. Delete after testing complete. */
int basic_test(libusb_device_handle *dev_handle);
/* Closes the connection associated with session_info. Once called, you will need to call
 * initialize_connection again if you wish to communicate more with the device. */
int close_connection(libusb_session_info session_info);
```

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <stdlib.h>
#include "libusb.h"
#ifndef ROBOCOMM
#define ROBOCOMM
#include "robocomm.h"
#endif
/* If you are NOT connecting to an Altera USB-Blaster cable, you will
 * need to find the correct IDs and endpoints for your device and enter them here. */
#define VENDOR_ID 0x09fb
#define PRODUCT_ID 0x6001
#define DEVICE_INTERFACE 0
#define OUT_ENDPOINT 2
#define IN_ENDPOINT 0x86
/* If these numbers are different for your device, feel free to change them accordingly.*/
#define NUMBER_OF_MOTORS 16
#define MAXIMUM_SPEED 255
#define COMMAND_LENGTH_BYTES 5
#define QUERY_LENGTH_BYTES 3
#define MOTOR_INFO_LENGTH_BYTES 5
#define COMMAND_START_BYTE 0xAB
#define QUERY_START_BYTE 0xAC
#define CUSTOM_START_BYTE 0xAD
#define MESSAGE_END_BYTE 0x13


/* Sets up a connection with the device defined by all the #defines above.
 * When you are finished with the connection, please call close_connection()
 * for garbage collection. */
libusb_session_info initialize_connection(){
 libusb_device **devs; /* used to retrieve device list */
 libusb_context *ctx = NULL; /* a libusb session */
 libusb_device_handle *dev_handle;
 struct libusb_device_descriptor desc; /*  */
 struct libusb_config_descriptor *config;
 libusb_session_info session_info;
 session_info.ctx = ctx;
 session_info.dev_handle = dev_handle;
 int r;
 ssize_t cnt, i;
 int device_found = 0; /* 1 if correct device is found, 0 otherwise */
 r = libusb_init(&ctx); /* initializes the library at last */
 if(r < 0){
   printf("Init Error\n"); /* if there's an error */
   exit(1);
 }
 libusb_set_debug(ctx, 3);
 cnt = libusb_get_device_list(ctx, &devs);
 if(cnt < 0){
   printf("Get Device Error\n"); /* if there's an error */
   exit(1);
 }
 for(i = 0; i < cnt; i++){
   /* Check if the device's vendor and product ID match what we want */
   r = libusb_get_device_descriptor(devs[i], &desc); /* gets i-th device descriptor                * and places it in &desc */
```

```
  if(r < 0){
        printf("Couldn't get device descriptor\n");
        exit(1);
  }
  r = libusb_get_config_descriptor(devs[i], 0, &config);
  if(r < 0){
        printf("Couldn't get config descriptor\n");
        exit(1);
  }
  if(desc.idVendor == VENDOR_ID && desc.idProduct == PRODUCT_ID){
        device_found = 1;
        libusb_open(devs[i], &dev_handle);
        break;
  }
 }
 if(device_found == 0){
  printf("Error, correct device not found\n");
  exit(1);
 }

 if(dev_handle == NULL){
  printf("Unable to open device\n");
  exit(1);
 }
 else{printf("Device Opened!\n");}
 libusb_free_device_list(devs, 1);
 if(libusb_kernel_driver_active(dev_handle, 0) == 1){
  printf("Active Kernel Driver Present\n");
  if(libusb_detach_kernel_driver(dev_handle, 0) == 1){
        printf("Detached Kernel Driver\n");
  }
 }
 r = libusb_claim_interface(dev_handle, DEVICE_INTERFACE);
 if(r < 0){
  printf("Couldn't claim interface\n");
  exit(1);
 }
 printf("Interface claimed!\n");
 return session_info;
}
/* Takes in the motor number, direction, and speed that the command should contain,
 * as well as the libusb_device_handle* that denotes which device the command is to
 * be sent to, converts that info to the command format that the device will accept,
 * and sends the command to the device.*/
int send_motor_command(int motor_num, int direction, int speed, libusb_device_handle *dev_handle){
 unsigned char *command;
 int bytes_transferred;
 int r;
 /* CATCHING INVALID INPUT */
 if(motor_num > NUMBER_OF_MOTORS
  | motor_num < 0){
  printf("Invalid motor number: %d\n", motor_num);
  exit(1);}
 if(direction < 0
  | direction > 1){
  printf("Invalid direction: %d\n", direction);
  exit(1);}
 if(speed < 0
  | speed > MAXIMUM_SPEED){
  printf("Invalid speed: %d\n", speed);
  exit(1);}
```

```
 /* MAKING THE COMMAND */
 command[0] = COMMAND_START_BYTE;
 command[1] = motor_num;
 if(direction == 0){
   command[2] = 0xEF;
 }
 else if(direction == 1){
   command[2] = 0xFF;
 }
 command[3] = speed;
 command[4] = MESSAGE_END_BYTE;

 /* SENDING THE COMMAND */
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), command,
COMMAND_LENGTH_BYTES, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == COMMAND_LENGTH_BYTES){
   printf("Writing successful!\n");
 }
 else{printf("Write Error\n");}

 return 0;
}

/* Sends a query to the device requesting all current information on a
 * particular motor. Returns a pointer to an array containing the information. */
unsigned char* query_motor_info(int motor_num, libusb_device_handle *dev_handle){
 unsigned char *motor_info;
 unsigned char *query;
 int bytes_transferred;
 int r;
 /* Catch invalid motor number */
 if(motor_num > NUMBER_OF_MOTORS
   | motor_num < 0){
   printf("Invalid motor number: %d\n", motor_num);
   exit(1);
 }
 query[0] = QUERY_START_BYTE;
 query[1] = motor_num;
 query[2] = MESSAGE_END_BYTE;
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), query, QUERY_LENGTH_BYTES,
&bytes_transferred, 0);
 if(r == 0 && bytes_transferred == QUERY_LENGTH_BYTES){
   printf("Data Send successful!\n");
 }
 else{
   printf("Data Send Error\n");
 }
 r = libusb_interrupt_transfer(dev_handle, (IN_ENDPOINT | LIBUSB_ENDPOINT_IN), motor_info,
MOTOR_INFO_LENGTH_BYTES, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == MOTOR_INFO_LENGTH_BYTES){
   printf("Data Receive successful!\n");
 }
 else{printf("Data Receive Error\n");}
 return motor_info;
}
int send_custom_command(int motor_num, unsigned char *data, int num_bytes, libusb_device_handle *dev_handle){
 int bytes_transferred;
 int r;
```

```
 unsigned char *header;
 header[0] = CUSTOM_START_BYTE;
 header[1] = motor_num;
 header[2] = num_bytes;
 if(motor_num > NUMBER_OF_MOTORS
   | motor_num < 0){
   printf("Invalid motor number: %d\n", motor_num);
   exit(1);
 }
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), header, 3, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == 3){
   printf("Header Send successful!\n");
 }
 else{
   printf("Header Send Error\n");
 }
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), data, num_bytes, &bytes_transferred,
0);
 if(r == 0 && bytes_transferred == num_bytes){
   printf("Data Send successful!\n");
 }
 else{
   printf("Data Send Error\n");
 }

 return 0;
}
int send_custom_command_and_receive_data(int motor_num, unsigned char *data_out, int num_bytes_out, unsigned char
*data_in, int num_bytes_in, libusb_device_handle *dev_handle){
 int bytes_transferred;
 int r;
 unsigned char *header;
 header[0] = CUSTOM_START_BYTE;
 header[1] = motor_num;
 header[2] = num_bytes_out;
 if(motor_num > NUMBER_OF_MOTORS
   | motor_num < 0){
   printf("Invalid motor number: %d\n", motor_num);
   exit(1);
 }
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), header, 3, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == 3){
   printf("Header Send successful!\n");
 }
 else{
   printf("Header Send Error\n");
 }
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), data_out, num_bytes_out,
&bytes_transferred, 0);
 if(r == 0 && bytes_transferred == num_bytes_out){
   printf("Data Send successful!\n");
 }
 else{
   printf("Data Send Error\n");
 }

 r = libusb_interrupt_transfer(dev_handle, (IN_ENDPOINT | LIBUSB_ENDPOINT_IN), data_in, num_bytes_in, &bytes_transferred,
0);
 if(r == 0 && bytes_transferred == num_bytes_out){
   printf("Data Receive successful!\n");
```

```
 }
 else{
   printf("Data Receive Error\n");
 }
 return 0;
}
int basic_test(libusb_device_handle *dev_handle){
 int bytes_transferred;
 int r;
 unsigned char *send;
 unsigned char *receive;
 send[0] = 0xAE;
 r = libusb_interrupt_transfer(dev_handle, (OUT_ENDPOINT | LIBUSB_ENDPOINT_OUT), send, 1, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == 1){
   printf("Send successful!\n");
 }
 else{
   printf("Send Error\n");
 }
 r = libusb_interrupt_transfer(dev_handle, (IN_ENDPOINT | LIBUSB_ENDPOINT_IN), receive, 1, &bytes_transferred, 0);
 if(r == 0 && bytes_transferred == 1){
   printf("Receive successful!\n");
 }
 else{
   printf("Receive Error\n");
 }
 if(receive[0] == 0xAF){
   printf("Correct byte received!\n");
   return 0;
 }
 else{
   printf("Incorrect byte received\n");
   return 1;
 }
}

/* Closes the connection associated with session_info. Once called, you will need to call
 * initialize_connection again if you wish to communicate more with the device. */
int close_connection(libusb_session_info session_info){
 int r;
 r = libusb_release_interface(session_info.dev_handle, DEVICE_INTERFACE);
 if(r != 0){
   printf("Cannot release Interface\n");
 }
 printf("Released Interface\n");
 libusb_close(session_info.dev_handle);
 libusb_exit(session_info.ctx);
 return 0;
}
```

UART code:

```
module uart
 #( //Default setting:
        //192,00 baud, 8 data birts, 1 stop bit, 2'2 FIFO
        parameter DBIT = 8, //# data bits
        SB_TICK = 16, //#ticks for stop bits,
                //16/24/32 for 1/1.5/2 bits
        DVSR = 163, //baud rate divisor
                //DVSR = 50M/(16*baud rate)
        DVSR_BIT = 8, //# bits of DVSR
        FIFO_W = 2 //#addr bits of FIFO
                //#words in FIFO = 2'FIFO_W
 )
 (
        input wire clk, reset,
        input wire rd_uart, wr_uart, rx,
        input wire [7:0] w_data,
        output wire tx_full, rx_empty, tx,
        output wire [7:0] r_data
 );

 //signal declaration
 wire tick, rx_done_tick, tx_done_tick;
 wire tx_empty, tx_fifo_not_empty;
 wire [7:0] tx_fifo_out, rx_data_out;

 //body
 mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
        (.clk(clk), .reset(reset), .q(), .max_tick(tick));

 uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
        (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
        .rx_done_tick(rx_done_tick), .dout(rx_data_out));

 fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
        (.clk(clk), .reset(reset), .rd(rd_uart),
        .wr(rx_done_tick), .w_data(rx_data_out),
        .empty(rx_empty), .full(), .r_data(r_data));

 fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
        (.clk(clk), .reset(reset), .rd(tx_done_tick),
        .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
        .full(tx_full), .r_data(tx_fifo_out));

 uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
        (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
        .s_tick(tick), .din(tx_fifo_out),
        .tx_done_tick(tx_done_tick), .tx(tx));

 assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

```
module uart_tx
#(
  parameter DBIT = 8,
  SB_TICK = 16
)
(
  input wire clk, reset,
  input wire tx_start, s_tick,
  input wire [7:0] din,
  output reg tx_done_tick,
  output wire tx
);

localparam [1:0]
  idle = 2'b00,
  start = 2'b01,
  data = 2'b10,
  stop = 2'b11;

reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
reg tx_reg, tx_next;

always @(posedge clk, posedge reset)
  if (reset)
              begin
              state_reg <= idle;
              s_reg <= 0;
              n_reg <= 0;
              b_reg <= 0;
              tx_reg <= 1'b1;
              end
  else
              begin
              state_reg <= state_next;
              s_reg <= s_next;
              n_reg <= n_next;
              b_reg <= b_next;
              tx_reg <= tx_next;
              end

always @*
begin
  state_next = state_reg;
  tx_done_tick = 1'b0;
  s_next = s_reg;
  n_next = n_reg;
  b_next = b_reg;
  tx_next = tx_reg ;
  case (state_reg)
              idle:
              begin
              tx_next = 1'b1;
              if (tx_start)
              begin
              state_next = start;
```

```
            s_next = 0;
            b_next = din;
            end
            end

            start:
            begin
            tx_next = 1'b0;
            if (s_tick)
            if (s_reg==15)
            begin
            state_next = data;
            s_next = 0;
            n_next = 0;
            end
            else
            s_next = s_reg + 1;
            end
            data:
            begin
            tx_next = b_reg[0];
             if (s_tick)
                        if (s_reg==15)
                        begin
                                    s_next = 0;
                                    b_next = b_reg >> 1;
                                    if (n_reg==(DBIT-1))
                                                state_next = stop;
                                    else
                                                n_next = n_reg + 1;
                        end
            else
            s_next = s_reg + 1;
            end

            stop:
            begin
            tx_next = 1'b1;
            if (s_tick)
            if (s_reg==(SB_TICK-1))
            begin
            state_next = idle;
            tx_done_tick = 1'b1;
            end
            else
            s_next = s_reg + 1;
            end
            endcase
  end
  assign tx = tx_reg;

endmodule
```

```
module uart_rx
#(
   parameter DBIT = 8,
   SB_TICK = 16
)
(
   input wire clk, reset,
   input wire rx, s_tick,
   output reg rx_done_tick,
   output wire [7:0] dout
);

localparam [1:0]
   idle = 2'b00,
   start = 2'b01,
   data = 2'b10,
   stop = 2'b11;

reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;

always @(posedge clk, posedge reset)
   if (reset)
            begin
                     state_reg <=idle;
                     s_reg <= 0;
                     n_reg <= 0;
                     b_reg <= 0;
            end
   else
            begin
                     state_reg <= state_next;
                     s_reg <= s_next;
                     n_reg <= n_next;
                     b_reg <= b_next;
            end

always @*
begin
   state_next = state_reg;
   rx_done_tick = 1'b0;
   s_next = s_reg;
   n_next = n_reg;
   b_next = b_reg;
   case (state_reg)
            idle:
                     if (~rx)
                              begin
                                       state_next = start;
                                       s_next = 0;
                              end
                     start:
                              if (s_tick)
                                       if (s_reg==7)
                                       begin
                                                state_next = data;
                                                s_next = 0;
```

```
                                        n_next = 0;
                                end
                        else
                                s_next = s_reg + 1;
                data:
                        if (s_tick)
                                if(s_reg==15)
                                        begin
                                                s_next = 0;
                                                b_next = {rx, b_reg[7:1]};
                                                if (n_reg==(DBIT-1))
                                                        state_next = stop;
                                                else
                                                        n_next = n_reg + 1;
                                        end
                                else
                                        s_next = s_reg + 1;
                stop:
                        if (s_tick)
                                if (s_reg==(SB_TICK-1))
                                        begin
                                                state_next = idle;
                                                rx_done_tick = 1'b1;
                                        end
                                else
                                        s_next = s_reg + 1;
        endcase
  end
  assign dout = b_reg;
endmodule



module fifo
 #(
 parameter B=8, //number of bits in a word
        W=4  //number of address bits
 )
 (
        input wire clk, reset,
        input wire rd, wr,
        input wire [B-1:0] w_data,
        output wire empty, full,
        output wire [B-1:0] r_data
 );

 //signal declaration
 reg [B-1:0] array_reg [2**W-1:0]; //register array
 reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
 reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
 reg full_reg, empty_reg, full_next, empty_next;
 wire wr_en;

 //body
 //register file write operation
 always @(posedge clk)
        if (wr_en)
        array_reg[w_ptr_reg] <= w_data;
        //register file read operation
```

```
  assign r_data = array_reg[r_ptr_reg];
  //wire enabled only when fifo is not full
  assign wr_en = wr & ~full_reg;

  //fifo control logic
  // register for read and write pointers
  always @(posedge clk, posedge reset)
                if (reset)
                begin
                w_ptr_reg <= 0;
                r_ptr_reg <= 0;
                full_reg <= 1'b0;
                empty_reg <= 1'b1;
                end
                else
                begin
                w_ptr_reg <= w_ptr_next;
                r_ptr_reg <= r_ptr_next;
                full_reg <= full_next;
                empty_reg <= empty_next;
                end

//next state logic for read and write pointers
always @*
begin
  //successive pointer values
  w_ptr_succ = w_ptr_reg + 1;
  r_ptr_succ = r_ptr_reg +1;
  //default: keep old values
  w_ptr_next = w_ptr_reg;
  r_ptr_next = r_ptr_reg;
  full_next = full_reg;
  empty_next = empty_reg;
  case ({wr, rd})
                // 2'b00: no op
                2'b01: //read
                if (~empty_reg) //not empty
                begin
                r_ptr_next = r_ptr_succ;
                full_next = 1'b0;
                if (r_ptr_succ==w_ptr_reg)
                empty_next = 1'b1;
                end
                2'b10: //write
                if (~full_reg) //not full
                begin
                w_ptr_next = w_ptr_succ;
                empty_next = 1'b0;
                if (w_ptr_succ==r_ptr_reg)
                full_next = 1'b1;
                end
                2'b11: //write and read
                begin
                w_ptr_next = w_ptr_succ;
                r_ptr_next = r_ptr_succ;
                end
  endcase
end

//output
```

```
assign full = full_reg;
assign empty = empty_reg;

endmodule



module mod_m_counter
#(
parameter N=4,    // number of bits in counter
M=4    // counter modulo
)
(
		input wire clk, reset,
		output wire max_tick,
		output wire [N-1:0] q
		);

// signal declaration
reg    [N-1:0] r_reg;
wire    [N-1:0] r_next;

// body
// register
always @(posedge clk, posedge reset)
if (reset)
r_reg <= 0;
else
r_reg <= r_next;

// next-state logic
assign r_next = (r_reg == (M-1)) ? 0 : r_reg + 1;

// output logic
assign q = r_reg;
assign max_tick = (r_reg == (M-1)) ? 1'b1 : 1'b0;

endmodule




module debounce //used exclusively for testing
 (
		input wire clk, reset,
		input wire sw,
		output reg db_level, db_tick
 );

 //symbolic state delcaration
 localparam [1:0]
		zero = 2'b00,
		wait0 = 2'b01,
		one = 2'b10,
		wait1 = 2'b11;

 //number of counter bits (2'N = 20ns = 40ms)
```

```
localparam N=21;

//signal declaration
reg [N-1:0] q_reg, q_next;
reg [1:0] state_reg, state_next;

//body
//fsmd state & data registers
always @(posedge clk, posedge reset)
        if (reset)
        begin
        state_reg <= zero;
        q_reg <= 0;
        end
        else
        begin
        state_reg <= state_next;
        q_reg <= q_next;
        end

//next_state logic & data path functional units/routing
always @*
begin
        state_next = state_reg;
        q_next = q_reg;
        db_tick = 1'b0;
        case (state_reg)
        zero:
        begin
        db_level = 1'b0;
        if (sw)
        begin
        state_next = wait1;
        q_next = {N{1'b1}}; //load
        end
        end
        wait1:
        begin
        db_level = 1'b0;
        if (sw)
        begin
        q_next = q_reg - 1;
        if (q_next==0)
        begin
        state_next = one;
        db_tick = 1'b1;
        end
        end
        else //sw==0
        state_next = zero;
        end
        one:
        begin
        db_level = 1'b1;
        if (~sw)
        begin
        state_next = wait0;
        q_next = {N{1'b1}};
        end
        end
```

```
            wait0:
            begin
            db_level = 1'b1;
            if (~sw)
            begin
            q_next = q_reg - 1;
            if (q_next==0)
            state_next = zero;
            end
            else //sw==1
            state_next = one;
            end
            default: state_next = zero;
  endcase
end

endmodule



module uart_test_parallelShort
  (
            input wire CLOCK_50, reset,
            inout wire UART_RXD,
            input wire [17:0] SW, //key0 acts as btn, key1 acts as reset.
            inout wire UART_TXD,
  output wire [17:0] LEDR,
            output wire [3:0] an,
            output wire [7:0] sseg, LEDG
 // output wire [7:0] HEX0
  );



//signal delcaration
  wire tx_full, rx_empty, btn_tick, btn_tick2, tick;
  wire [7:0] rec_data, rec_data1;
  wire short, short1;
  reg testerbegin, tester, flag;


  //body
  //instantiate uart
  uart uart_unit
            (.clk(CLOCK_50), .reset(SW[17]), .rd_uart(1),
            .wr_uart(1), .rx(UART_RXD), .w_data(rec_data1),
            .tx_full(tx_full), .rx_empty(rx_empty),
            .r_data(rec_data), .tx(UART_TXD));
  //instantiate debounce circuit
  debounce btn_db_unit
            (.clk(CLOCK_50), .reset(SW[17]), .sw(SW[16]),
            .db_level(), .db_tick(btn_tick));
  // incremented data loops back
  assign rec_data1 = rec_data + 1;
  //assign rec_data1 = SW;
  assign short1 = ~short;
  // LED display
  assign LEDG = rec_data1;

always@(UART_RXD, SW[16], SW[15])
```

```
begin
if (UART_RXD && SW[16] && SW[15])
begin
flag = 1;
end
end
assign LEDR[0] = flag;

  assign an = 4'b1110;
  assign sseg = {1'b1, ~tx_full, 2'b11, ~rx_empty, 3'b111};
endmodule



module uart_test_serialShort
  (
            input wire CLOCK_50, reset,
            inout wire UART_RXD,
            input wire [17:0] SW, //key0 acts as btn, key1 acts as reset.
            inout wire UART_TXD,
  output wire [17:0] LEDR,
            output wire [3:0] an,
            output wire [7:0] sseg, LEDG
  );



//signal delcaration
  wire tx_full, rx_empty, btn_tick, btn_tick2, tick;
  wire [7:0] rec_data, rec_data1;
  wire short, short1;
  reg testerbegin, tester;

  assign rec_data = SW;
  assign LEDR = rec_data;
  //body
  //instantiate uart
  uart uart_unit
            (.clk(CLOCK_50), .reset(SW[17]), .rd_uart(1),
            .wr_uart(1), .rx(short), .w_data(rec_data),
            .tx_full(tx_full), .rx_empty(rx_empty),
            .r_data(rec_data1), .tx(short));
  //instantiate debounce circuit
  debounce btn_db_unit
            (.clk(CLOCK_50), .reset(SW[17]), .sw(SW[16]),
            .db_level(), .db_tick(btn_tick));
  // incremented data loops back
  //assign rec_data1 = rec_data + 1;
  //assign rec_data1 = 8'b11001100;
  assign short1 = ~short;
  // LED display
  assign LEDG = rec_data1;

  assign an = 4'b1110;
  assign sseg = {1'b1, ~tx_full, 2'b11, ~rx_empty, 3'b111};
endmodule
```

```
//For anyone using the UART, it is important to note that a single Write command is triggered for every clock
// period the write signal is high, and it takes 5,216 clock cycles for the UART to output one bit of serial data, of
// which it will output ten bits (one start, one end, and ten data bits) for every write command.
// For testing purposes, you will have to zoom out A LOT to see the data being created, outputted.
// For testing, same goes for the other direction. You will have to feed it serial data at 5,216 clock cycles a bit.
// It takes 163 clock cycles to trigger a FIFO tick. It takes 32 FIFO ticks to send 1 bit of serial. 32 x 163 = 5,216
```