

# Zilog Java Oscilloscope (ZJO)

Brad Lawrence

CS 339

4/25/2006

## Design Overview

This project focuses on creating a Java based Oscilloscope using the Zilog (ZJO) Z8Encore microcontroller to read an analog input, and then output that input onto a USB FIFO device provided by DLP Design. The microcontroller is responsible for interfacing with the USB FIFO device and sending three byte data values to a java application deployed on a pc. The Z8Encore microcontroller is also responsible for receiving a one byte control data and turning on its LED array to indicate that the PC interface wishes to have the input signal scaled. The scaling mechanism is not implemented, because an external reference voltage was not in the scope of this project.

The java graphical user interface (GUI) is responsible for receiving the signal data, and displaying it in a graphical form. It also allows the user to dynamically set the voltage ranges, time range, and a trigger event.

The proposed requirements that the project must meet are:

- Z8Encore Interact with PC through the use of the USB FIFO device.
- Implement a data schema for representing voltages in three bytes of data.
- Be able to view specific user defined time increments.
- Be able to view specific user defined voltage ranges.
- Be able to set a trigger voltage with a time frame.
- Be able to send a control byte to indicate the user wishes to scale data.

## Hardware Modules

This project consists of two hardware modules, the Z8Encore development kit and the DLP Design USB-245-M device. The basic hardware diagram is shown below.

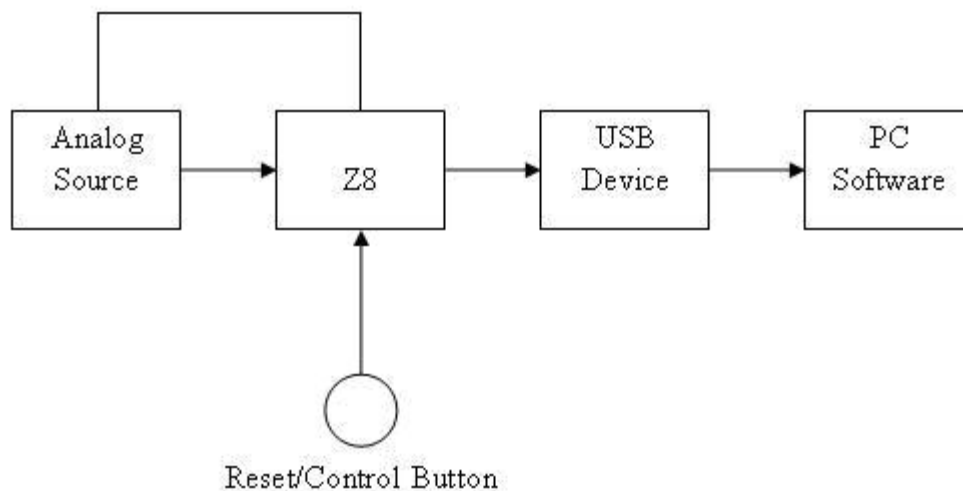


Figure 1

## *Z8Encore Development Kit*

The Z8Encore development kit has several areas of interest for this product including the internal analog to digital converter (ADC), three multi purpose timers, and general purpose input output pins (GPIO).

First of all, the oscilloscope needs an input source. The multi purpose timers of the Z8Encore kit are a suitable input source. The timers can be set to output a PWM signal on one of the Z8Encore GPIO pins by using the alternate function of the pin. For the purposes of this project, the timer was set up to output a PWM voltage with a frequency of 1000 Hertz and a 40 percent duty cycle.

The internal ADC is the most critical component of the Z8Encore development kit for this particular project since it is used to convert the PWM wave into a digital representation. The internal ADC used by this project has outputs a 10 bit binary number representing the voltage based on the internal voltage reference generator. An external voltage reference generator could have been used to allow for more flexibility, but that is outside the scope of this project. In particular, this ADC can sample at a rate of 256 cycles, and since the Z8Encore has a clock speed of 18,432,000 the sample rate will be 72,000 Hertz. Therefore, the sample rate will be around 0.000014 seconds. This sample rate is important to note for timing diagrams later on in the project. It also gives us the Nyquist frequency for the Java oscilloscope which is half the sampling rate. So the 32 Kilo Hertz is max frequency that the ZJO can read effectively.

Finally the GPIO pins facilitate the connection of the Z8Encore development kit and the DLP Design USB FIFO device. The usage of the GPIO pins can be broken down into eight reserved for Z8Encore to USB data transfer only, four reserved for USB device control, and one for the input signal. The connections will be detailed in the connecting hardware section.

## *DLP Design USB-245-M device*

The USB device is a 24 pin device with 8 pins directly responsible for data. The functions of the 24 pins are listed below.

- Pin 1 is the board ID. Low for the 245M model.
- Pins 2, 5, and 7 are ground pins.
- Pin 3 is the external reset pin. Can be tied to VCC if not required to be externally reset.
- Pin 4 is the Reset Out pin. This pin is High if the module is in the process of a reset.
- Pin 6 is a 3.3v output pin that could be used for an external device.
- Pin 8 is the sleep output pin. This pin is high during a USB suspend.
- Pin 9 is the send/wake-up pin.
  - When in USB suspend, a positive edge on this pin causes a wake-up
  - When in active mode, a positive edge on this pin causes data to be sent regardless of how many bytes are in the transmit buffer.
- Pin 10 is a 3.0v to 5.25v UART interface pins. Can be connected to VCC of target, or the external VCC pin.
- Pin 11 is the external VCC pin. Used to supply main power. 4.4v to 5.25v.

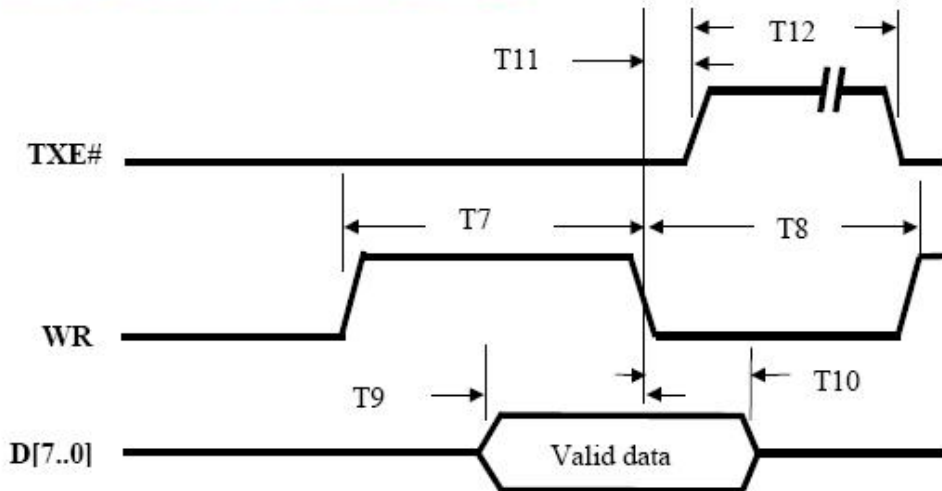
- Pin 12 is the Power from the USB port.
- Pin 13 is the RXF output pin.
  - When low there is data to be read.
- Pin 14 is the TXE output pin.
  - When high data is being transmitted, so do not attempt to transmit while this pin is high.
- Pin 15 is the WR in pin.
  - When this pin is taken from high to low, data is taken from the data pins (pins 17 – 24), and written to the transmit buffer.
  - Data is sent to the host PC after a timeout (16 mS).
- Pin 16 is the RD in pin.
  - When this pin is taken from high to low, takes the data pins to the current byte in the receive buffer.
  - Taking RD back to high returns the data pins to high, and prepares the next byte in the receive buffer (if there is one available).

Other more generic specifications for the USB device are listed below.

- Send / Receive Data over USB at up to 1 MB per second
- 384 byte FIFO Transmit buffer
- 128 byte FIFO receive buffer
- USB 1.1 and USB 2.0 compatible
- Virtual COM Port Drivers available from FTDI
- D2XX ( USB Direct Drivers + DLL S/W Interface )

Each specification above could have had an effect on the overall performance of the ZJO. However in most cases, the limiting factor was the ADC of the Z8Encore that really limited the specification of the ZJO. For instance, the 8 million bits per second data transfer was not a factor when considering that the max sampling rate was 72,000 hertz. The timing diagram below along with its corresponding table show that the USB module could transmit at much faster speed than 72,000 bytes a second.

## DLP-USB1 TIMING DIAGRAM – FIFO WRITE CYCLE



Time	Description	Min	Max	Unit
T7	WR Active Pulse Width	50		ns
T8	WR to WR Pre-Charge Time	50		ns
T9	Data Setup Time before WR inactive		20	ns
T10	Data Hold Time from WR inactive	10		ns
T11	WR Inactive to TXE#	5	25	ns
T12	TXE inactive after RD cycle	80		Ns

As you can see the transmit time will be in the nanoseconds range. One can ignore the data setup and data hold times and concentrate on the WR active pulse width and the WR inactive to TXE# because WR and TXE# are the control pins, and the data is written while they are being pulled low or taken high. The WR to WR pre-charge time is also being done synchronously with the TXE inactive after RD cycle so in total we can add the WR Active Pulse Width section, the WR Inactive to TXE# section, and the TXE inactive after RD cycle to come up with a maximum value of 155 nanoseconds.

The specification for the project determines that three bytes of data will be transmitted for each sampling of voltage. Therefore it is necessary to have 465 nanoseconds of transmit time after the sampling. Since the sampling is 0.000014 seconds, there will be enough time in between samplings for the transmission of the three bytes.

One place that the Z8Encore was not a factor was the 384-byte FIFO Transmit buffer. The default timeout transmission (as per the description of pin 15 / WR) is 16 mS. The ZJO transmits at a rate of 0.000014 seconds, which is around 1,140 bytes per 16 mS. This would be a real problem if the SND/WK-UP pin (pin 9) did not exist. Therefore the ZJO will have to use the SND/WK-UP pin (description of usage is in the software module section).

The read timing diagram could also have been included in this description of the USB FIFO device, but the ZJO did not use the reading portion of the USB FIFO device because there was no need to read at high rates. The specification of the ZJO project only required one control byte to be read at a time, and that control byte was generated through the user input. The user input was clicking a button on the GUI; therefore again the reading portion of the ZJO was seen as a non-factor.

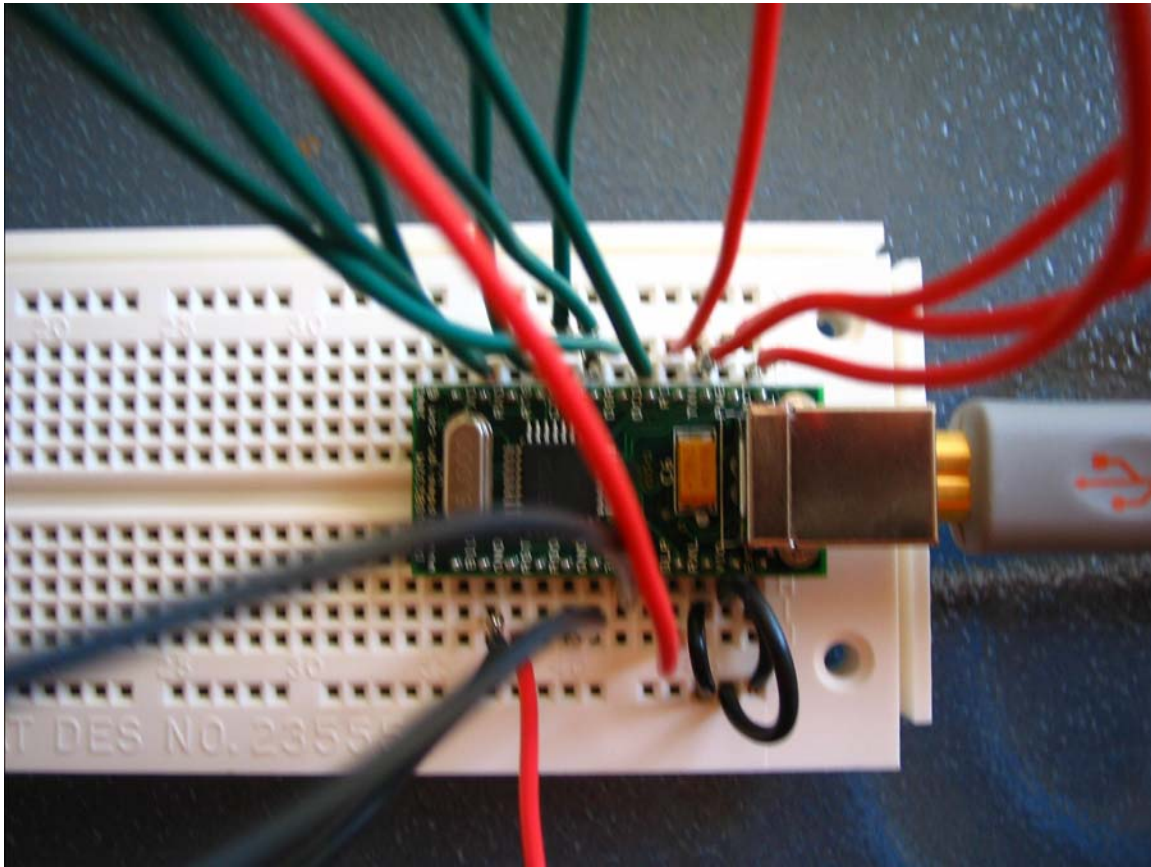
A detailed schematic of the USB FIFO device is provided in Appendix A.

## Hardware Interfacing

The USB to Z8Encore interface is fairly simple. The pins and connections for the USB to the Z8Encore microcontroller are listed in the following table, and the figure below the table displays a view of the wiring of the USB controller.

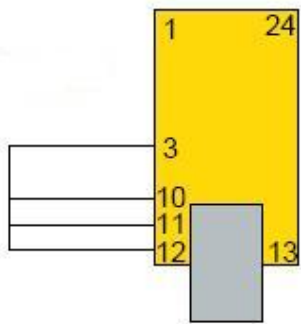
<b>Pin Number / Name</b>	<b>Z8Encore Connection Pin</b>
Pin 1 / Board ID	Not Connected
Pin 2 / Ground	Connected to Ground
Pin 3 / External Reset	Connected to VCC (no external reset required by the ZJO)
Pin 4 / Reset Out	Not Connected
Pin 5 / Ground	Connected to Ground
Pin 6 / 3.3V Out	Not Connected
Pin 7 / Ground	Connected to Ground
Pin 8 / Sleep output	Not Connected
Pin 9 / SND/WK-UP	Connected to PD5
Pin 10 / UART	Not Connected to Z8 (this pin is jumped together on the breadboard of the USB device so power is drawn from main power of USB)
Pin 11 / External VCC Pin	Not Connected to Z8 (pin is jumped to the power from USB port so that the USB device draws power from the USB port)
Pin 12 / Power from USB Port	Not Connected to Z8 (pin is jumped together with pins 10 and 11 to allow for default configuration of drawing power from the USB port)
Pin 13 / RXF Output pin	Connected to PD0
Pin 14 / TXE Output pin	Connected to PD1
Pin 15 / WR input pin	Connected to PD2
Pin 16 / RD input pin	Connected to PD4
Pin 17 / Data pin 7	Connected to PF7
Pin 18 / Data pin 6	Connected to PF6
Pin 19 / Data pin 5	Connected to PF5
Pin 20 / Data pin 4	Connected to PF4
Pin 21 / Data pin 3	Connected to PF3
Pin 22 / Data pin 2	Connected to PF2

Pin 23 / Data pin 1	Connected to PF1
Pin 24 / Data pin 0	Connected to PF0



The description of the pins that are connected to the Z8Encore will be described in the software section of this summary. Therefore, this portion will focus on pins that were not connected to the Z8Encore board.

The first pin was the board id, and since the software of the ZJO did not specify the use of any particular board id, and the drivers for the USB FIFO device were provided by FTDI, the board ID is not needed in this application and so it is left unconnected. The next unconnected pin is the reset out pin. The USB-FIFO device is only reset on power up of the device itself. There was no reason for the ZJO software to programmatically reset the USB FIFO device, and so it was left unconnected. The third pin unconnected to the Z8Encore was the 3.3V out pin. There was no need to have a device powered by the USB device so it was left unconnected. The next unconnected pin was the SLEEP out pin, and since there was no reason to have the USB device in sleep mode, it was also not connected. Finally the three pins that determine power were not connected to the Z8Encore device, because the USB device can be configured to have the USB port power the USB device. All one has to do to configure the USB device to be powered by the USB port is to jump these three pins (pin 10, 11, and 12) together as shown in the diagram below.



**Basic Bus-Powered  
5V System**

The configuration for the ZJO does not exactly mirror the figure to the left because pin 3 is not also jumped together with the three power pins. The reset pin (pin 3) is set to the VCC of the Z8Encore. Looking back on it now, the figure is probably a better configuration than the current configuration of the ZJO because the reset pin is tied directly to the power. Having the reset pin

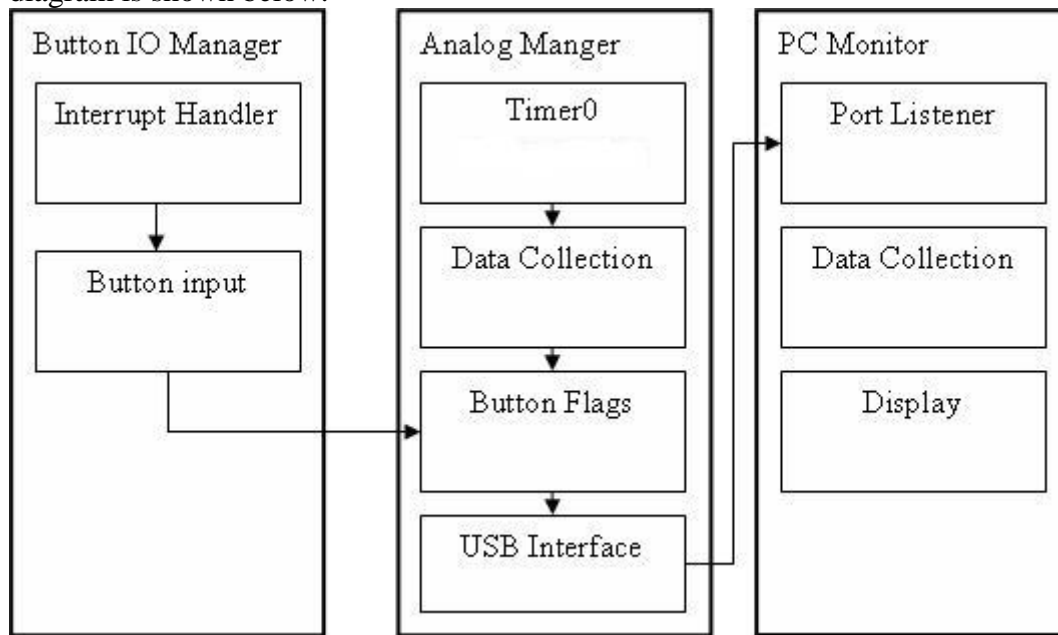
connected to two different power sources may cause problems down the line, but none were noticed during the development of the ZJO system.

*Part List*

- DLP Design USB FIFO Device – Cost \$25.00
- Zilog Z8Encore Development Kit – Cost unknown
- Standard Breadboard – Cost \$12.00
- Wiring – Cost unknown

## Software

The software portion of this project is divided into two sets of code, the C based Z8Encore to USB interface and the Java based GUI. A final software design overview diagram is shown below.



The Button IO Manager and the Analog Manger represent the C based Z8Encore to USB interface, and the PC Monitor represents the Java application that has a front end GUI.

*The C based Z8Encore to USB interface*

The Button IO Manager while designed as a separate module from the Analog module was not really large enough to be physically separate from the overall Z8Encore

to USB Interface software module. The Button IO Manager really just involves setting up an interrupt to occur on the SW1 button on the Z8Encore so that the Button flag will be set. Once the button flag has been sent, data is then forwarded to the USB device. The internal ADC of the Z8Encore is always collecting data, but it is not always being sent because of the button flag.

The Timer0 module in the Analog Manager represents the internally generated input source for the ZJO. To fulfill the specifications of the design overview, the timer0 of the Z8Encore is set up to output a pulse width modulated signal with a 40 percent duty cycle. To complete the setup, a wire connected to the timer0 output pin, pin PA1 to the input pin of the ZJO, which was PB0.

The data collection module in the Analog Manager is implemented through the setup of the internal ADC of the Z8Encore. The setup is geared around the input signal being placed onto pin PB0. The continuous sampling by the ADC occurs at a rate of 256 cycles as described in the hardware description section. An interrupt is generated upon completion of the ADC conversion.

Finally the USB interface is implemented through the connection of the data pins on the USB device to the PF0 through PF7 pins on the Z8Encore. The USB interface is a set of methods such as `waitForReadyToXmit()`, `writeChar()`, `void read_pin_F(void)`, `void init_input_portF_write(void)`, and `void init_input_portF_read(void)` that are detailed as follows:

- `waitForReadyToXmit()` – waits for PD1 (connected to TXE) to be low which indicates that the USB device is read to transmit.
- `init_input_portF_write(void)` – sets all of the PF pins (connected to data pins) as output pins.
- `writeChar()` – sets PFOUT to be the data to output, then cycles PD2 (connected to WR), which causes the data to be transmitted.
- `init_input_portF_read(void)` – sets all of the PF pins (connected to data pins) as input pins.
- `read_pin_F(void)` – reads PF pins, and then turns on all LED lights if they were off or turns them all off if they were on.

Whenever the ADC interrupt occurs in the data collection module, a `readData()` method is called to read the data and convert it into voltage first. Then after the data is converted into voltage, the data collection module converts the data into a specified data format of three bytes. The first byte is always a control byte to indicate that the data is starting after this byte. The ZJO has a small data design flaw in it. The control byte is the hex value 0x30, which is also a valid value for data. So that if the data happened to exactly be the control byte, the entire data stream could be corrupted. The data scheme is a simple scheme in which the two byte value is interpreted as an unsigned short integer value. The voltage value is multiplied by 100 and then 32768 (the value interpreted as 0) is added to the voltage. So a value of 3.33V would be multiplied by 100 to get 333, and then added to 32768 to get a value of 33101. The min and max of the data are listed in the table below along with the voltage 0 to show how the data scheme works.

Integer Value	Hex Value	Voltage Value
65535	0xFFFF	327.67V
32768	0x8000	0.0V

0	0x0000	-327.68V
---	--------	----------

So as you can see the data could be the exact value that the control character is, which could turn into a real problem. So far, the problem has not shown up yet, but the ZJO also has not had a wide range of inputs being used since the ZJO is simply reading a PWM wave generated by the internal timer.

### *The Java Framework and APIs*

The Java based GUI can be broken down into two basic sets of classes. The framework classes are the base abstract interfaces that allow for polymorphism within the ZJO, and the implementation classes of those interfaces.

Before any specific classes developed within the ZJO GUI can be discussed, one must first detail the different open source APIs used during development. The two java APIs used by the ZJO are the JFreeChart API and the RXTX implementation of the javax comm. API.

### *The JFreeChart API*

The JFreeChart API is an open source java API that allows for easy creation of charts. There are numerous different types of charts that are available for use including pie charts, bar charts, and finally line charts. In case of the ZJO, the JfreeChart provides an easy medium to display 2D representations of voltages through the use of its idea of a line chart.

The manner in which a line chart is created, is a fairly simple matter. The JFreeChart API has its own representation of a what it labels as a series. To create a series, one simply adds to a series through a method that has two doubles as parameters. The first double represents the value of the X axis while the second double represents the value of the Y axis in a line chart. Once a series is created, it can be added to a collection of series because each line chart could have more than one series. In the case of the ZJO, only one series will exist which is the graph of the voltage.

After the collection is created, the collection can then be passed through a Factory class that will return a chart depending upon the method called. Once the chart is returned, the chart can then either be written out to a java.awt.Graphics2D object, or it can be written as a java.awt.image.BufferedImage. A description of attaching this image to the GUI will be described in the later GUI section.

### *The RXTX API*

The RXTX implementation of the javax.comm. API is a set of open source classes that allow for the connection of Java to a Serial or parallel port. The RXTX implementation was used because Sun stopped supporting the Windows implementation of the javax.comm. API. Basically this API and its implementing classes allows for the easy access of a Serial port in Java. The serial port API allows for event driven access of the data, but the Parallel port API does not. One can suppose that the event driven interface was not implemented for Parallel ports because parallel ports were mainly used as output sources (especially in the case of printers) and did not need complicated read interfaces to inform the application that the parallel port device had sent data.

The event driven architecture has an interface that extends the `EventListener` java interface so that a class such as the `CommPortReader` of the ZJO will implement that interface and then in some sense be notified of an event. The paradigm is very similar to an interrupt driven architecture.

### *The Java Based Framework*

The four abstract interfaces that comprise the ZJO framework are the `ChartCreatorIF`, `CommReaderIF`, `OscilloscopeFrameIF`, and the `WaveDataIF`, and their responsibilities are detailed below:

- `ChartCreatorIF` – Responsible for generating an instance of the `JfreeChart` class.
- `CommReaderIF` – Responsible for reading data from the Serial Port. Also responsible for returning an instance of the Serial Port it is connected to.
- `OscilloscopeFrameIF` – Responsible for updating the graphics of the ZJO.
- `WaveDataIF` – Responsible for being containers of the data and the time scale requested from the user.

In general this framework allows for extensibility of the ZJO without changes to its core source. For instance, if the rate at which the `CommReaderIF` was too slow for someone. A developer could throw out or change the implementation class of the `CommReaderIF` and simply replace it, as long as it still implemented each of the methods listed in the `CommReaderIF`, the other classes in the ZJO would not be affected.

The idea of frameworks and abstracting out interfaces is essential to the reusability of code. The open closed principle states that software should be open to extension but closed to change. This principle is one of the core foundations of object oriented analysis and design.

### *The Java Implementation classes*

One can easily determine the implementation classes of the ZJO project. There exists one concrete class for each of the abstract interfaces in the framework. Each of these concrete classes is responsible for the functions described in the interfaces above. This particular document will not detail all of the components of the framework. Instead this section documents some of the key functional areas of the implementation classes.

The `CommReader` class is the concrete class for the `CommReaderIF`. It is responsible for also implementing an event listener interface. The only required method of the `SerialEventListener` is `serialEvent(SerialEvent e)`. This method is basically one big case statement that cases on the different types of `SerialEvent` types of the `e` in parameter. The only case that the ZJO is interested in is the `DATA_AVAILABLE` case, which indicates if the input stream from the USB 245-G device has data available to be read. Once this portion is hit, a while loop is entered. The key is reading while there are at least six bytes available. Remember that the ZJO sends data in a three-byte structure. The reason for the three byte structure is that a control byte is needed. This while loop checks for control bytes and skips over them, while getting the last known valid data. The last known valid data index is then used to take any remaining bytes of the data from the six byte input and store them in a class variable. Upon read of the next six bytes, the ZJO will append the read six bytes onto the extra byte buffer from the previous read. So in this manner, the `CommReader` class ensures that the data is valid data as long as the control character is not being hit by the current voltage being read in.

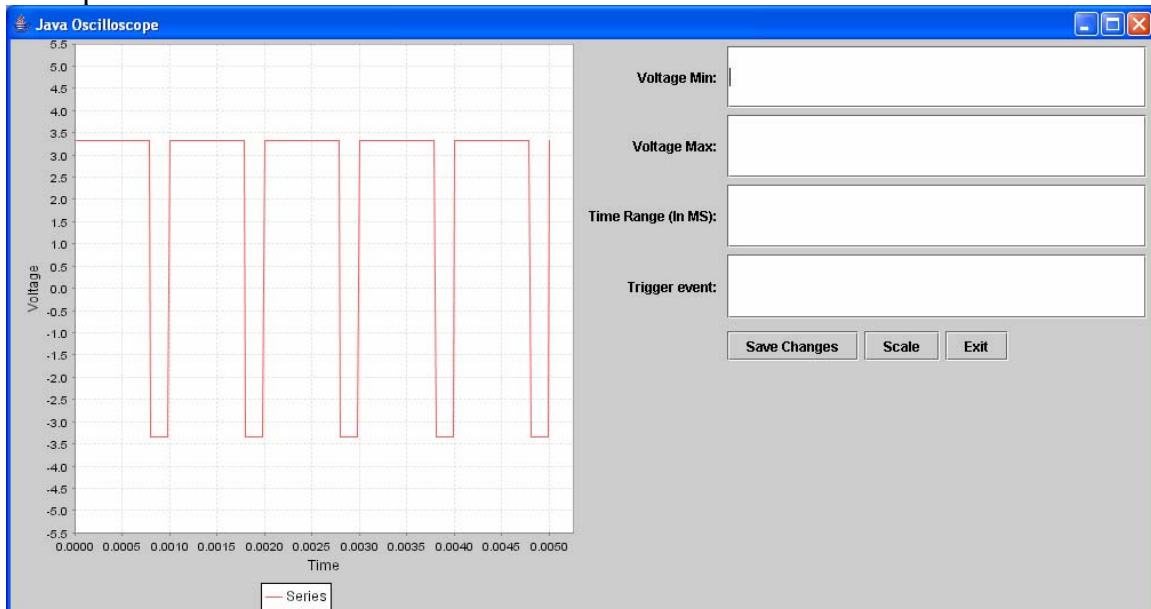
Another interesting portion of the code is the manner in which the data is updated through the GUI. The GUI implements the abstract interface OscilloscopeFrameIF. This interface has one method, which passes the data as an array of integers. The CommReaderIF also must have an addListener method that takes an OscilloscopeFrameIF as its parameter. So first the CommReader will add the OscilloscopeFrame to its listener list. Then the CommReader will notify all of its listeners of a data update through the update method in the OscilloscopeFrameIF. Notice that the CommReader keeps a list of listeners so that the ZJO could have multiple GUIs implementing the OscilloscopeFrameIF “listening” to the same signal if that was desired.

Once the update method in the OscilloscopeFrameIF is called, the OscilloscopeFrame class will call the ChartCreator class to generate a chart with the new data from CommReader. The chart will then be updated on the screen. One area that the ZJO could be improved was to call updateListeners more often. As of now the ZJO GUI waits until the user inputted time range is reached.

The other classes in the ZJO GUI are not as interesting. The implementing class of the WaveData is simply a data holder class. There are a couple other helper classes such as a class that facilitates the creation of an XYSeries JfreeChart object. The XYSeries class is a representation of a graph in the JfreeChart API. There is a simple for loop within the SeriesCalculator class that generates data based upon the voltage integer array and increments of the time range.

### *The GUI Manual*

This section of the project details a basic how-to for the GUI of the ZJO. Since the ZJO GUI is fairly simple, this screenshot of the ZJO will be followed by a short description of each of its fields and buttons.



The basic function of any oscilloscope is viewing the voltage wave. The graph in the left pane of the ZJO shows the current wave with the parameters entered. The ZJO starts up with this default wave displayed.

A short description of the four fields and three buttons is as follows:

- Voltage Min Text Field – Enter the desired voltage min in this field.
- Voltage Max Text Field – Enter the desired voltage max in this field.
- Time Range (in MS) Text Field – Enter the desired range of the wave.
- Trigger Event (in volts) – This field causes a trigger event on the value entered. The current wave value will be compared with the value entered in this field. After the first compare, the ZJO will wait for these events:
  - If the wave value was greater than the trigger value, then the ZJO will wait until a value is less than the trigger value. After the trigger event occurs the ZJO will measure up until the time range and then recheck for the next event.
  - If the wave value was less than the trigger value, then the ZJO will wait until a value is greater than the trigger value.
- Save Changes Button – This button attempts to save the data in the text fields and update the displays. Basic checks for empty values and non-number values are made. One specific check is that if a Trigger Event is inputted, then a Time Range must also be inputted.
- Scale Button – This button sends a control character to the Z8Encore. It is intended for future enhancements of the overall ZJO system.
- Exit Button – This button causes the system to exit.

## Future Additions and Changes

There are many things that can be improved upon with the current state of the ZJO system. Future aspirations for the ZJO can be broken down into enhancements and bug fixes.

There are several small bug fixes that I believe could be handled. The list of bugs that have been recorded for the ZJO are as follows.

- The fact that the ZJO starts up with a default wave instead of a zero voltage read is a bug that could easily be fixed.
- Once a wave is stopped, the ZJO still seems to read the last wave read. This occurs because data is not cleared from the WaveData object in the ZJO.
- Large time ranges tend to cause the ZJO to seem to hang for longer than the expected time ranges. I think this is caused by the long for loop that is used to create an XYSeries. More investigation is needed to determine if the for loop is really the cause.
- Data seems to be dropped occasionally. The dropped data could be caused the ZJO wires being crossed. Investigation with wires better shielded could be fruitful.

Apart from errors in the ZJO system, there are many ways that the ZJO could be refined. For instance, the trigger event in the ZJO system is fairly rudimentary. Drop down menus with different options for trigger events such as “greater than” or “less than” as options would allow the user to specify trigger events more smoothly and should allow the wave to stay more constant.

One idea that would vastly improve the Java based framework would be to remove the JfreeChart return within the ChartCreatorIF. The purpose of interfaces in java is to abstract implementations of objects so that clients of frameworks do not have to be tied to a particular implementation because a client can create their own implementation. In the case of the ZJO Framework and the ChartCreatorIF, the createXYChart method forces all implementations of ChartCreatorIF to know about JfreeChart because it returns a JfreeChart specific class. To make this framework better, ChartCreatorIF should return a more generic class such as the java.awt.image.BufferedImage class. In this manner, some other client could come along and create an implementation of ChartCreatorIF that returns a BufferedImage and not have to use JfreeChart at all.

A more hardware-based enhancement would be to have an external ADC reading the value of the analog input. As stated earlier in this work, the ZJO is limited by the speed of the internal ADC on the Z8Encore. That speed is around 72,000 samples per second. An external ADC could easily be connected to the Z8Encore and cause an interrupt on one of the GPIO pins available. However, a whole new timing structure as well as data structure may be necessary if this enhancement is made. Remember that the data is currently represented in three bytes. Three bytes is really somewhat inefficient for a value that can be represented in ten bits. It was not inefficient in this case because the ZJO was limited more by the internal ADC of the Z8Encore.

The final thought for extensions includes the scale button represented on the ZJO GUI. There are a myriad of scaling or sampling options that the Z8Encore could perform on the data before it is sent to the ZJO GUI. This scale button is intended for such a use. Perhaps an external voltage reference generator is connected to one of the GPIO pins, and the scale button causes the internal ADC to switch from using the internal voltage reference generator to the external voltage reference generator.

## Problems Encountered

Most of the problems encountered in this project dealt with timing. In the beginning, I was more ambitious than I ended up being. Many of the ZJO tests were at much higher sampling rate speeds than the 72,000 sample rate. At first I was trying to max out the USB device because I was not thinking of the project in an overall sense. I was taking the project piece by piece. I ignored the fact that there was no way possible that the ZJO could transmit at the speeds I was attempting to because of the internal ADC. Eventually I settled on a timing diagram, and finished the testing phase of just sending bytes over to the ZJO GUI. Then when I discovered that all my test sampling rates would not work, I was fairly frustrated, but I built the code so that it was not too difficult to change the rates. In the end it wasn't that big of a problem.

I also had a problem in which my wiring was not shielded so I occasionally crossed wires to disastrous effect. Just before the project presentation was due, I believe I did cross some wires, and the USB device stopped working. Mouser.com and UPS love my next day shipping. I believe I ordered several backup USB adapters. The wires not being shielded also caused data loss at times. If the wires were crossed, data would be all over the place in the oscilloscope GUI.

The trigger even was a little tricky to accomplish also. I ended up having several flags indicating a trigger event and trigger voltage. A temp trigger voltage store had to be

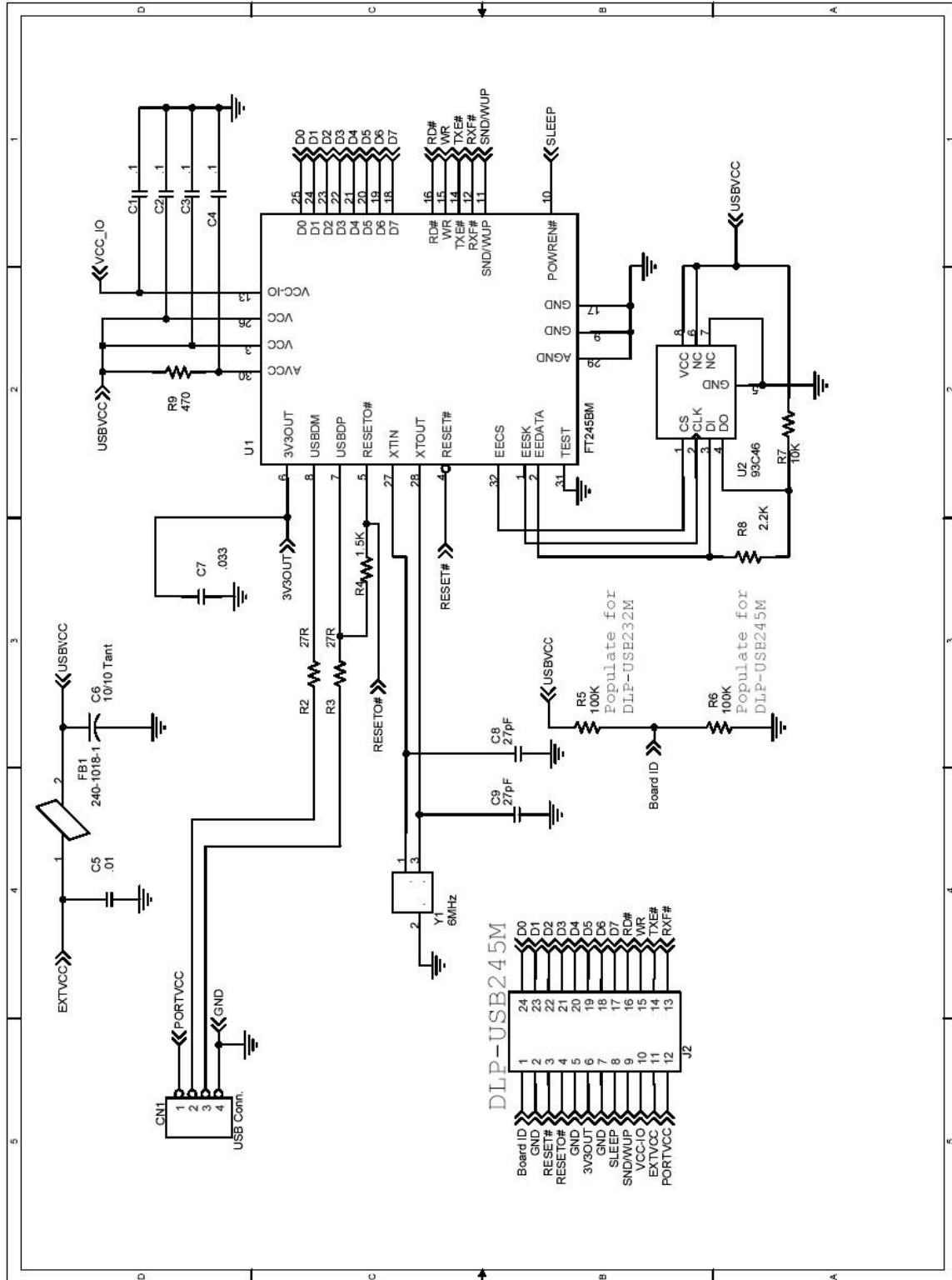
created so that it could be cleared after the first trigger was found, and the data could then be read normally. At first I didn't have this temp, and only a high or low voltage was displayed of the PWM wave because the trigger event would block the data from being read while it the voltage exceeded the trigger event.

## Conclusion

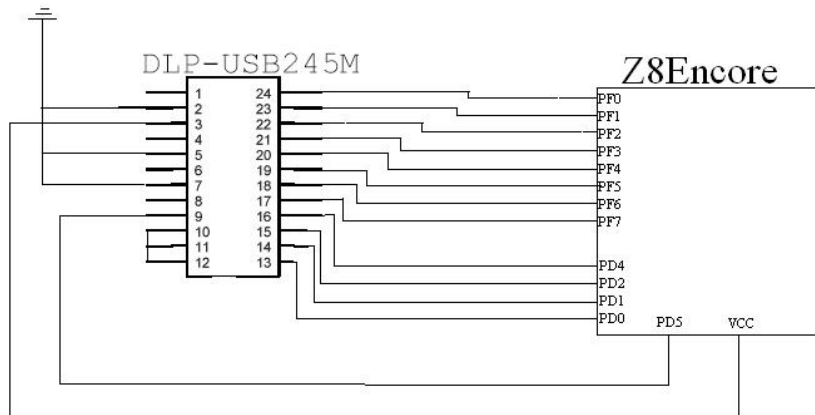
The ZJO Oscilloscope is a cheap alternative to buying an oscilloscope, but it also has limitations such as the low sampling rate of its internal ADC. However, the ZJO GUI does not necessarily tie a user to the hardware implementation of the ZJO. One could easily change the ZJO firmware on the Z8Encore, and then implement another CommReaderIF class to read at the new rates. So the ZJO can be easily upgraded if someone has the desire to do so. Therefore, the ZJO is a viable tool option when considering buying an oscilloscope.

## **Appendix A: Project Pictures**

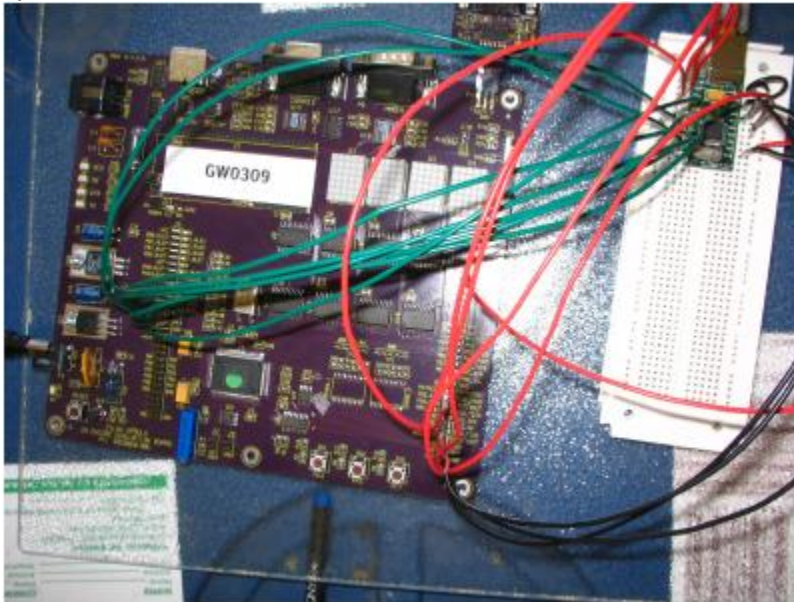
# DLP Design USB245-M Schematic



## System Overview Schematic



## System Picture



## Appendix B: Setup Instructions

The ZJO oscilloscope has no setup instructions for the firmware installed on the Z8Encore, but there are a few instructions for the ZJO GUI. The steps to start the ZJO GUI are as follows:

- Unzip the jfreechart-1.0.1.zip file to a folder.

- Set up an environment variable called JFREECHART to the directory that the jfreechart-1.0.1.zip was unzipped.
- Unzip the rxtx-2.1-7-bins-r2.zip to a directory.
- Copy the RXTXcomm.jar from the unzipped rxtx-2.1-7-bins-r2 directory to the \lib\ext directory of your Java installation.
- Copy the rxtxSerial.dll and rxtxParallel.dll files from the unzipped rxtx-2.1-7-bins-r2 directory to the \jre\bin directory of your Java Installation
- Execute the z8oscope.bat file to start the ZJO GUI.