University of Central Florida
School of Electrical Engineering and Computer Science

A

100

CD Changer Emulator

Senior Design – Group 6
Daniel Gallagher
Ethan Steffens
Tony Trinh

December 9, 2002

Table of Contents

Table of Figures

List of Tables

1 Introduction

Today many MP3 players exist to suit the needs of computer and music enthusiasts around the globe. With the gateway to digital audio, computer components such as hard drives, compact flash cards, and solid state memory cards replace bulky optical drives and discs. Optical drives and/or compact disc (CD) players have moving mechanical parts and may be prone for damage rendering the unit nonfunctional. For the car audio specialist current technologies utilize CDs with MP3s written or burned on the disc for playback in a MP3 compatible in-dash or CD changer unit.

The CD Changer Emulator project is designed to replace the CD changer with a portable laptop hard drive that will play MP3s. In addition, navigational controls of the MP3 player will solely be based upon the head unit's button commands. By merging these two technologies the user of the CD Changer Emulator can have full functionality and control of the MP3 player from the convenience of the head unit's pre-existing buttons.

The knowledge gained from the courses taken at the University of Central Florida, not to mention the knowledge gained by personal research and endeavors allows for the collaborative effort of this group to accomplish this project. The design, development and completion of this project were performed in the facilities of the University of Central Florida and with the supervision of Dr. Fernando Gonzalez.

This documentation includes a motivation for the project, budget report, project milestones, and chapters for specific aspects and components of the CD Changer Emulator. Each aspect and special topics are explained with extensive research and personal knowledge of the subject matter. Certain diagrams are approved and reprinted with permission; other diagrams are from our own design and circuitry layout.

Reprint Permission Information

All pictures, diagrams, and figures used in this document are used with the permission of the author. Below is a listing of some of the most noteworthy.

- Texas Instruments - http://www.ti.com/corp/docs/legal/copyright.htm
- Microchip - http://www.microchip.com/1010/overview/legal/disclaim/index.htm
- Sony - http://www.sony.com/terms.shtml
- Alpine - http://www.alpine1.com/html/termsofuse.htm
- Maxim - http://www.maxim-ic.com/LegalNotice.htm
- AVRFreaks.com - http://www.avrfreaks.com/legal.html

2 *Motivation for the Project*

In today's world digital audio is king. Many technology magazines currently hype up this phenomena and as we are able to fit more storage in a more confined space digital audio will be in the market place for decades to come. Most people enjoy listening to music in their car. Today people increasingly tend to spend more time in their cars in lieu of traffic jams and long road trips. With our project, instead of having a CD Changer with limited amount of discs and music, storing MP3 files on a hard drive allows for expandable amount of listening time for unlimited results. By analyzation of the CD Changer Head Unit's protocol and by programming a Programmable Integrated Circuit (PIC) with software to emulate the responses of the CD changer, essentially this project can be adapted to perform with various types CD head units by only replacing the code in the PIC module. With this in mind allows the greatest convenience for the consumer to buy any off the shelf CD Changer head unit and interface it with a hard drive MP3 player. Our overall motivation was to use the Head Unit controls for MP3 playback off of a hard drive by emulating the CD Changer's response waveforms via a PIC.

3 *Budget Report*

Material	Qty	Price	Cost
Car CD Changer Controller (Head Unit)	1	$300.00	Existing
CD Changer Interface Cable and Connector	1	$ 20.00	$ 20.00
Atmel AVR ATmega161L DIP Chip	1	$ 12.07	$ 12.07
Atmel STK500 AVR Flash Starter Kit	1	$ 79.00	$ 79.00
Microchip PIC16F628 with Programmer	1	$ 20.00	$ 20.00
Microchip PIC16F628 Additional Chip	3	$ 12.00	$ 12.00
VS1001K MP3 Decoder Chip	1	$ 20.00	$ 20.00
VS1001K Piggyback Kit	1	$ 55.00	$ 55.00
Laptop Hard Drive - IDE ATA 2.5"	1	$ 90.00	$ 90.00
IDE Cable 44-Pin	1	$ 8.00	$ 8.00
IDE ATA Interface Connector	1	$ 10.00	$ 10.00
USB Connector - Type B	1	$ 2.50	$ 2.50
FTDI USB Chip	1	$ 8.00	$ 8.00
FTDI USB Prototype Board	1	$ 30.00	$ 30.00
Alps Miniature Push button Switch	2	$ 1.75	$ 3.50
Crystal - 7.3728 MHz	2	$ 2.30	$ 4.60
Crystal - 12.288 MHz	2	$ 2.70	$ 5.40
Crystal - 4.000 MHz	1	$ 1.50	$ 1.50
Logic Level Converter SN74LVC245AN	3	Sample	Sample
Latch - SN74HC573AN	3	Sample	Sample
LM393 - Dual Comparator	3	Sample	Sample
LF353 - JFET Op-Amp	3	Sample	Sample
P-Channel MOSFET - TPS1100D	3	Sample	Sample
2 Input AND Gate - SN74AVT08N	1	Sample	Sample
DC-DC Controller - MAX1626	1	Sample	Sample
RS-232 Transceiver MAX202	1	Sample	Sample
Transil Diode - SM15T15A	1	$ 1.00	$ 1.00
Schottky Diode	1	$ 1.00	$ 1.00
Inductors	1	$ 0.50	$ 0.50
Resistor - Various Values	N/A	$ 10.00	$ 10.00
Capacitor - Various Values and Types	N/A	$ 10.00	$ 10.00
Various Connectors and Headers	N/A	$ 10.00	$ 10.00
Various Discrete Components	N/A	$ 5.00	$ 5.00
PCB Raw Materials	N/A	$ 15.00	$ 15.00
Case and Mounting - Raw Material	N/A	$ 10.00	$ 10.00
Shipping Costs - Various		$ 20.00	$ 20.00
		Total	$462.07

Table 3-1: Budget Report

4 *Project Milestones*

Overview

In this section are project milestones before and during the project. The initial proposed milestones in Section 4.1 are estimates of the project before implementation. Most of the research started during the Summer 2002 semester. Section 4.2 shows the actual milestones of the project for the Fall 2002 semester.

4.1 Initial Proposed Milestones

Week of August 19
- CD Changer Protocol Analysis
- Overlook circuit design
- Get developmental board in evaluation phases

Week of August 26
- CD Changer Protocol Analysis wrap-up
- Simulation of circuit design on the computer
- Loading test programs in the PIC and ATmega161

Weeks of September 2 – September 23
- Programming of the PIC with the Protocol
- Programming the ATmega161 with code to read MP3s and play them
- Interfacing the head unit and the PIC and finding results with I/O communication
- ATmega161 handles MP3 playback and navigation

Week of September 30 – October 14
- The interfacing of the head unit, the PIC, and the ATMEL ATmega161
- Debugging software and tweaking code for optimizations
- To confirm the pin readings and everything in working order

Week of October 28 – November 11
- Final proceedings including board milling and software development
- Debugging and tweaking of the code for final implementation
- Circuit board milling for the final product

Week of November 18 – End of Semester
- Final preparations and testing
- Hardware and Software documentation
- Mock runs for the final presentation
- PowerPoint presentation slides
- Organization of documentation to be turned in

4.2 Actual Milestones

Week of August 19
- CD Changer Protocol Analysis

Week of August 26
- More CD Changer Protocol Analysis
- Determining the best method for capturing pulse widths
- Start of PIC assembly coding

Weeks of September 2 – September 23
- Running test programs for data capture and waveform output
- Waveform output verified
- Designed circuit to allow the difference in voltages of the head unit and PIC operating levels

Week of September 30 – October 14
- Interfacing the head unit and the PIC and finding results with I/O communication
- Using test program for testing serial input and output of the PIC
- Debugging code.
- Switched code from PIC Assembly to PIC Basic Pro for the native pulsin function to measure the low pulse width
- Verified waveform output with the Pic Basic Pro code
- Protocol packet checking
- MP3 Player works

Week of October 28 – November 11
- Attempting to communicate with the head unit and PIC
- Successful communication between the head unit and PIC
- Further coding of the protocol into the PIC for other button packets

Week of November 18 – End of Semester
- Full Project integration
- MP3 Player plays and the PIC communicated with the head unit.
- Presentation scheduled for December 9, 2002

5 *CD Changer Head Unit*

Many CD Changers require a head unit for playback and navigation. Each head unit comes equipped by the manufacturer with navigational buttons and LCD displays. The head unit, dependent upon the particular manufacturer, communicates with the CD Changer through groups of binary digits that constitute a packet of information to be interpreted by the CD Changer. This protocol is specific to the type of head unit and the manufacturer of the device.

5.1 Protocol

The protocol is the means by which devices communicate with each other. By being able to interpret the packets the CD Changer emulator can determine what function to perform whether it is play, pause, next, previous, or stop. Each of these commands has a specific command format. Here is a list of protocol packets considered. Some of these protocols are open to the public. However some protocols are proprietary and require reverse engineering of the command packets in order for the CD Changer Emulator to work.

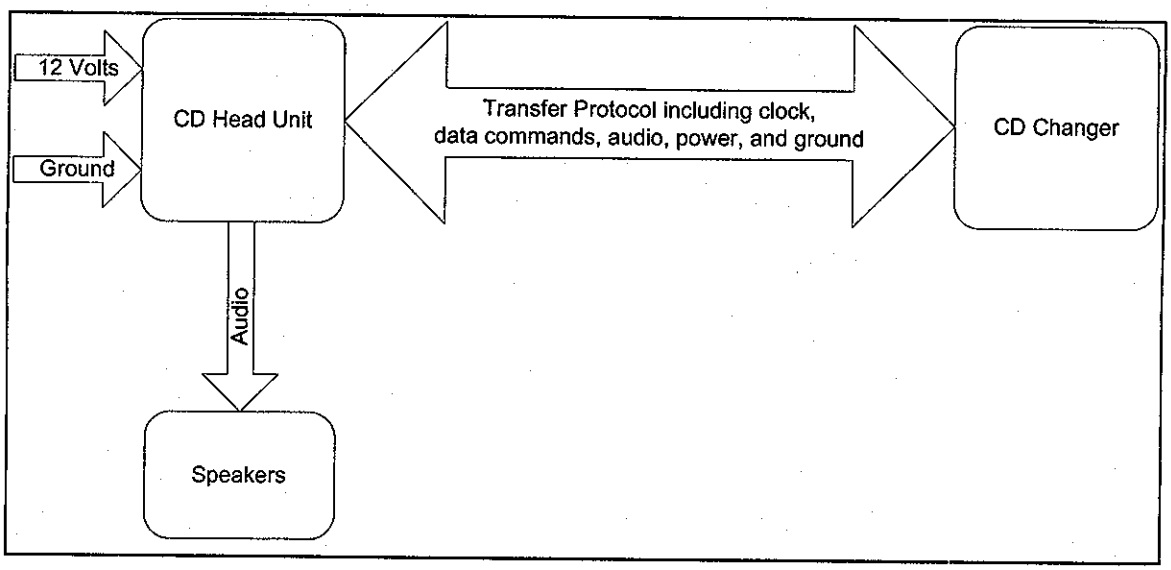


Figure 5-1: Typical communications with a CD Head Unit and CD Changer

5.1.1 Other Protocol Considerations

Here is a list of other protocol considerations and some type of explanation of how the protocol is interpreted and used by the head unit as well as the CD Changer.

5.1.1.1 Sony Unilink

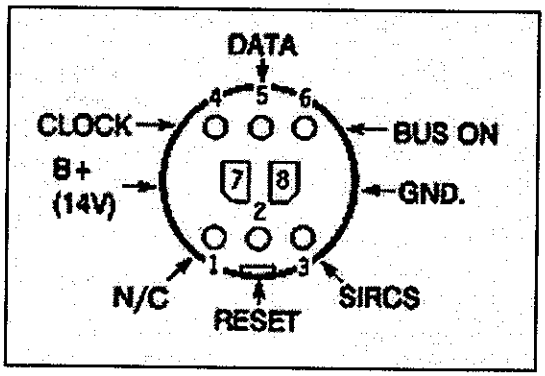


Figure 5-2: Sony Unilink Data Cable

The Sony Unilink uses two cables, one for the bus and one to pass the audio signal. The layout of the pins for the bus cable is shown in Figure 5-2.

5.1.1.1.1 Slave and Master Communication

When the BUS ON line is LOW the master goes into sleep mode and no communication occurs between the slave and master. The slave can wake the master up by forcing the DATA line HIGH which causes the master to activate the BUS ON line. In normal mode there are four states: idle, initial link, time polling, and request polling. In the idle state there is only a square wave on the DATA line; no data activity occurs. In initial link state the master polls the slave to detect what group type it is and then assigns it an address. The master does this to each slave it finds. Every 600ms the master switches to the time polling state and polls the slave to ensure that nothing has changed, if it has then it updates accordingly. Request polling occurs when the slave needs to send data.

5.1.1.1.2 Timing and Packet Description

Figure 5-3 shows a detailed timing diagram of the data bus. All packets are separated by at least a 50ms gap. When sending bytes out, the clock is normally LOW for 8us and HIGH for 8us. If no bytes have been sent then the clock line will remain LOW. Because of the 50ms delay after the last byte of data you can assume that any blank space over 25 ms marks the end of the packet. As shown in Figure 5.3 there are three data words: short – 6 bytes, middle – 11 bytes, and long – 16 bytes.

Each short word has a receiver address, transmitter address, first byte of the command identifier, subcommand id or first data byte, and parity byte for error checking. The middle words have 4 additional data bytes with an extra parity byte and the longs words have additional 9 data bytes with an extra parity byte. The receiving address device, RAD, and the transmitting address device, TAD, are always built from two nibbles. The higher nibble specifies the device group and the lower nibble specifies the device id that is assigned by the master, usually the head unit, during the initial link mode. Table 5-1 shows some commands for the Sony Unilink.

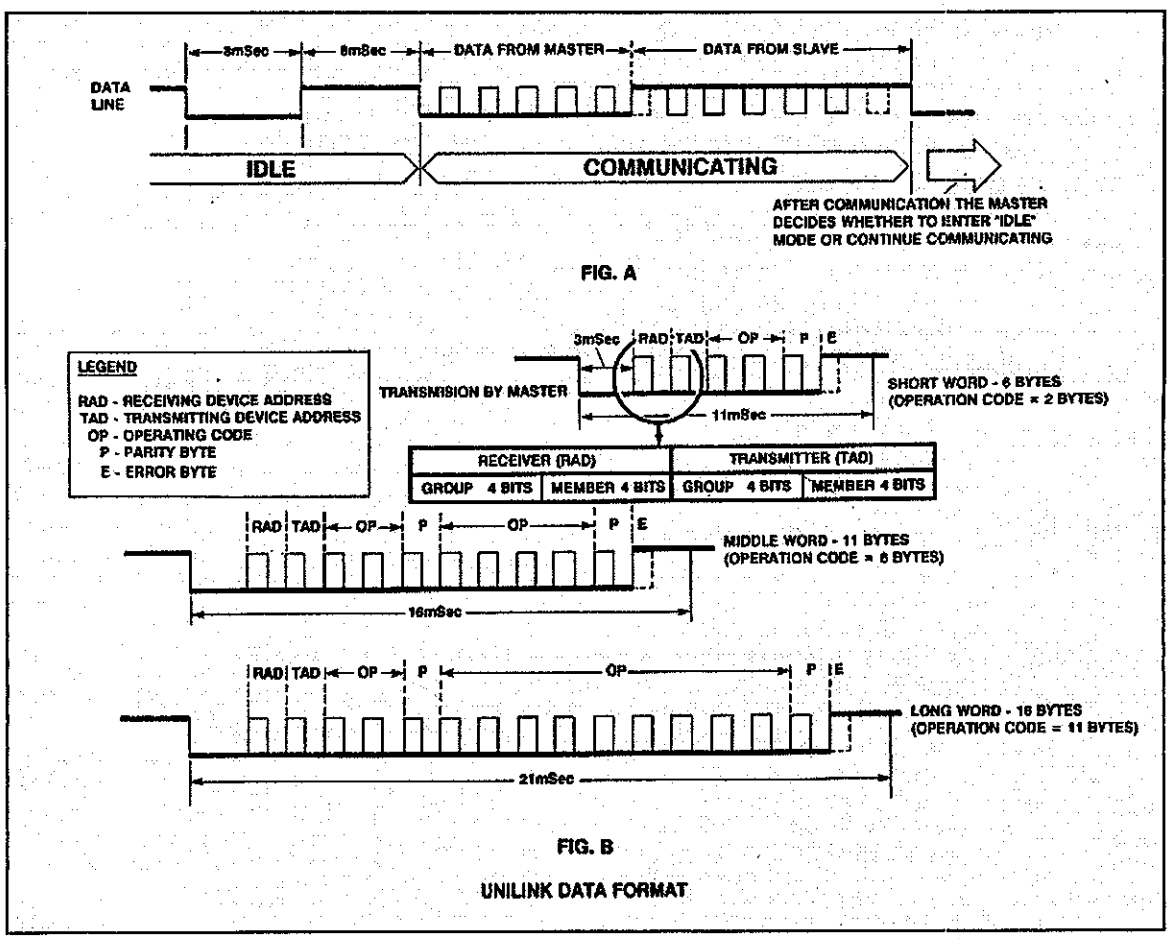


Figure 5-3: Timing and Packet Description for the Sony Unilink

Length	CMD1	CMD2	Command Meaning
Short	0x00		State report (from slave)
		0x00	-Playing
		0x20	-Changed CD
		0x21	-Seeking
		0x40	-Changing CD
		0x80	-Idle
		0xC0	-Ejecting
Short	0x01		Requests (from master)
		0x00	-Re-initialize bus
		0x02	-Checks status of slave (initial link)
		0x03	-This command is always after 0x02
		0x11	-Time Polling End.
		0x12	-Time Poll (answered by slave with 0x00 short word)
		0x13	-Request Time Poll (playing position or receive frequency)
Short	0x02		Configuration things (frommaster)
		0x01	-Tell slave about assigned ID number
		0x44	-Initialize tuner
Short	0x08		
		0x00	- Key Off (this command is sent when the key is released.
Middle	0x8e		Magazine/Cartridge information (from changer)
		0x40	- Slot in magazine is empty
		0x80	- A magazine was put into the changer
		0xc0	- No magazine

Table 5-1: Command Packet Structure for the Sony Unilink

5.1.1.2 Kenwood's Protocol

Kenwood's protocol uses only one cable to interface between the head unit and the CD changer. Unlike the Sony Unilink data is sent using various pins.

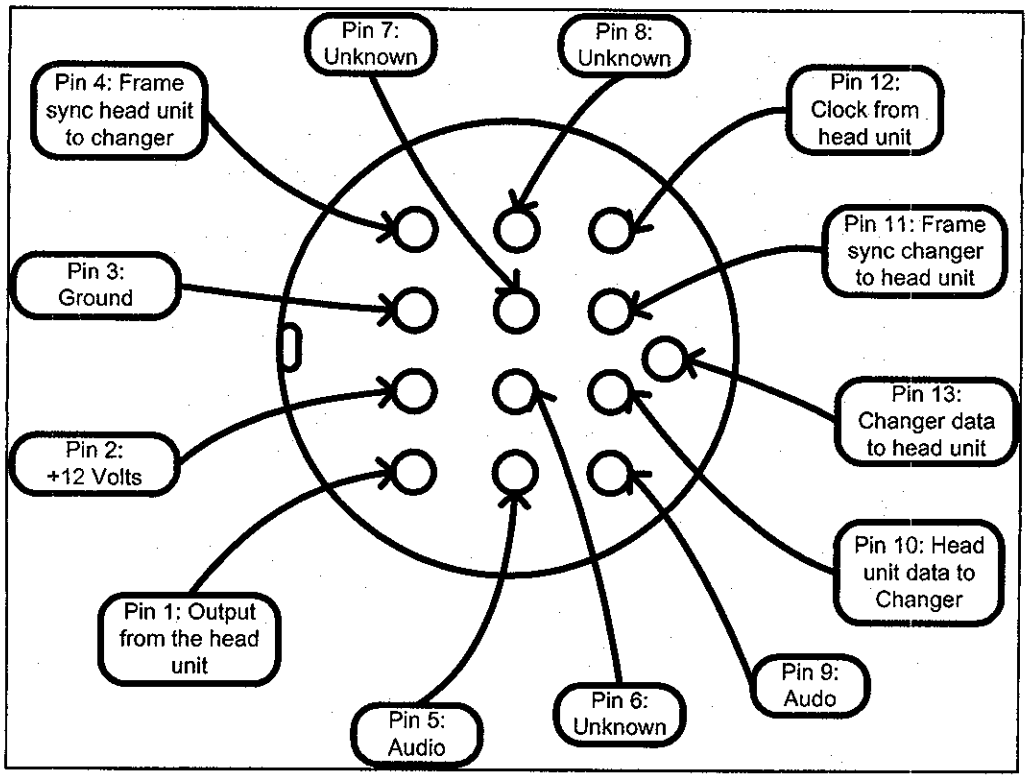


Figure 5-4: Kenwood Data Cable

5.1.1.2.1 Description of the Pins

There are 13 pins on the connector. Shown in Figure 5-4 are the pin outs of the cable. Pin 1 allows the head unit to switch on the CD changer by forcing the pin to 5 volts. Pin 2 supplies the 12 Volts of power to CD changer and pin 3 is used as a ground. Pin 4 is used to ensure the packets sent to the CD changer from the head unit are accepted. It does this by going LOW thereby telling the CD changer that the radio wants to send data. When the CD changer is ready to receive it, the changer drops pin 11 LOW and the data is transferred across pin 10. There is something similar for the CD changer. The changer drops pin 11 LOW first then the head unit drops pin 4 and data is received across pin 13. Pin 9 and 5 are used to transfer the audio from changer and the head unit uses pin 12 to supply a clock with an 8 μs period. The duty cycle of the clock is 50%.

5.1.1.2.2 Packet Description

Each packet consists of a 4-byte packet header and the data. The first 2 bytes of the packet header are used for the address of the receiver. The third and fourth bytes contain the address of the sender. The fifth byte contains the size of the data. The bytes thereafter vary according to what command or information is being sent.

5.1.2 Chosen Project Protocol

Our senior design project uses the Alpine M-Bus Protocol. This is an ***undocumented*** protocol and with some research has not been open to the public or reversed engineered to the best of our knowledge.

5.1.2.1 Alpine M-bus Protocol

This protocol uses only one data pin to communicate between the head unit and the CD changer. Power, audio, clock, and data commands such as fast forward are transmitted through this cable. No information could be found on the details of the packet information or communication between the head unit and changer.

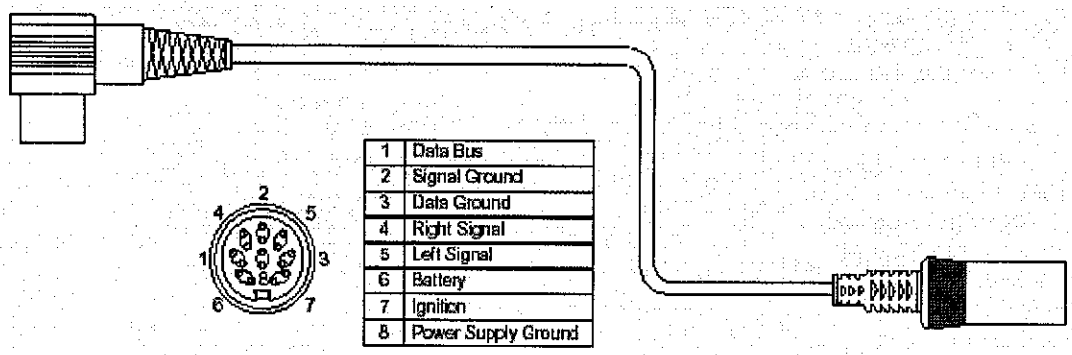


Figure 5-5: M-Bus Data Cable

5.1.2.1.1 The "Right" Protocol

Each protocol is propriety therefore we needed both a CD head unit and a CD changer of the same brand. Several factors affected our decision-making as to which brand we should use. They included cost of parts, ease of use and understanding of the protocol, and the availability of information existing on that protocol. Because we have a low budget we needed to obtain our head unit and changer at a low cost, essentially zero. This narrowed our choices tremendously. We had access to a Sony, Kenwood, and Alpine. The Kenwood was not chosen because of compatibility issues. Therefore the Sony was chosen over the Alpine because of the vast information that could be found on the Sony Unilink.

This protocol was used in our design up until there were circumstances out of our control and caused us to no longer have access to the Sony head unit or the CD changer. Thus the Alpine M-bus protocol was our only option. This protocol must be decoded entirely by us considering that no information can be found relating to this protocol.

5.1.2.1.2 Analysis of the M-Bus Protocol

To analyze the M-bus protocol we used the Tektronix TLA715 logic analyzer which enabled us to obtain a detailed log of how the CD changer and CD head unit interact. Because the protocol used only two pins, data and ground, and there was no clock, the communication was asynchronous and serial. The control signal, data pin, was active low and worked at 12 volts with very low current. To transfer a 0 or 1 the protocol relies on pulse widths. A pulse width of 709 microseconds symbolized a 0 while a pulse width of 1.21 milliseconds symbolized a 1. Once this was understood the recorded samples from the logic analyzer were "decoded" and compared with each other to determine the packet structure. After extensive analyzation several aspects of the protocol became clear. The CD changer controlled the LCD screen of the CD head unit. The packet size was variable. The first nibble in every packet defined who was speaking. This was pertinent because there is only one data pin. Therefore both master and slave need to talk on it. In most cases the second and third nibbles contained the command information. Also the last nibble is always a checksum to ensure the packet is received correctly. For example the waveform you see on the following page gives an example of the protocol. The many other fuctions that were implemented can be seen in the appendix under the Log of M-bus Protocol.

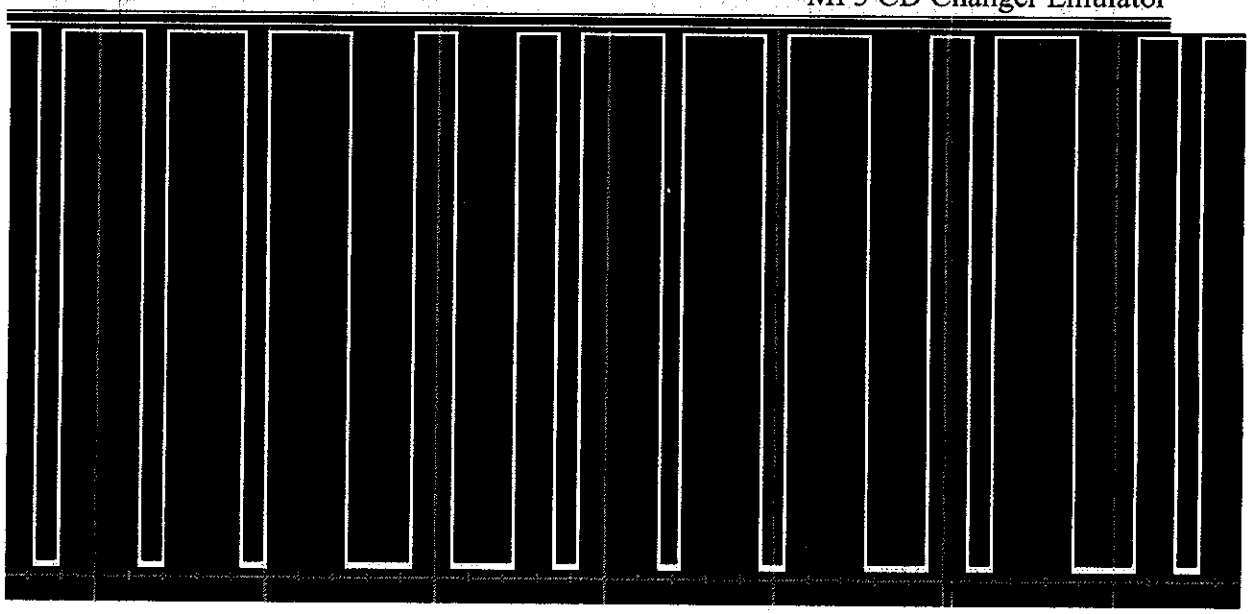


Figure 5-6: Alpine M-Bus Head Unit Polling Command

0001 → CD head unit talking
1000 → Polling command
1010 → Checksum

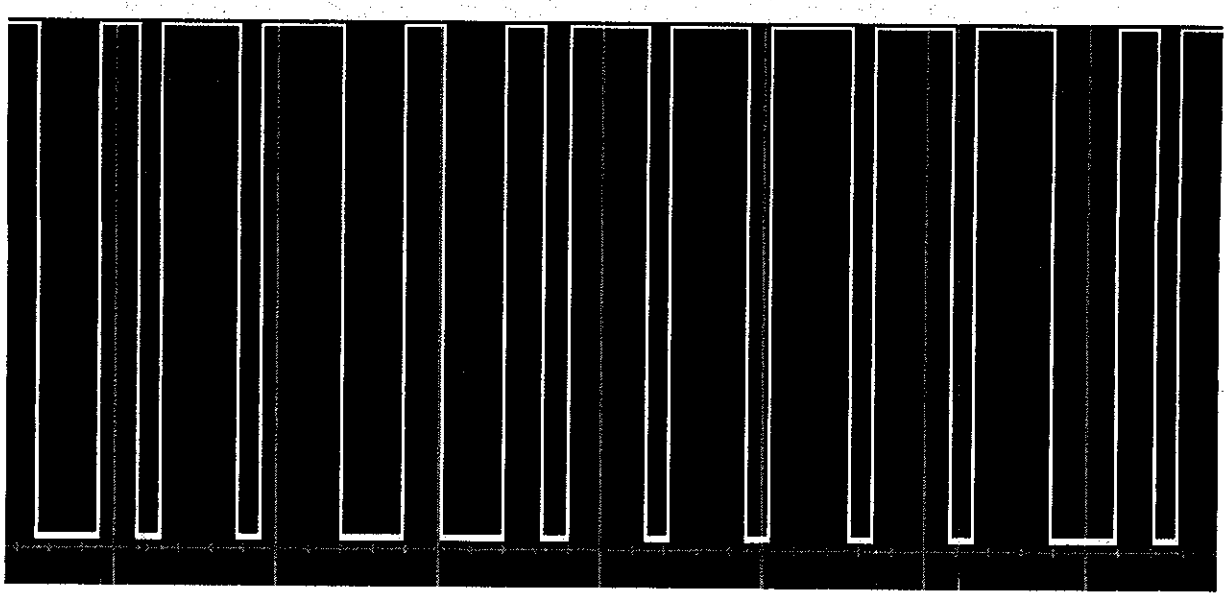


Figure 5-7: Alpine M-Bus CD Changer Polling Command

0001 → CD Changer talking
1000 → Polling command
1010 → Checksum

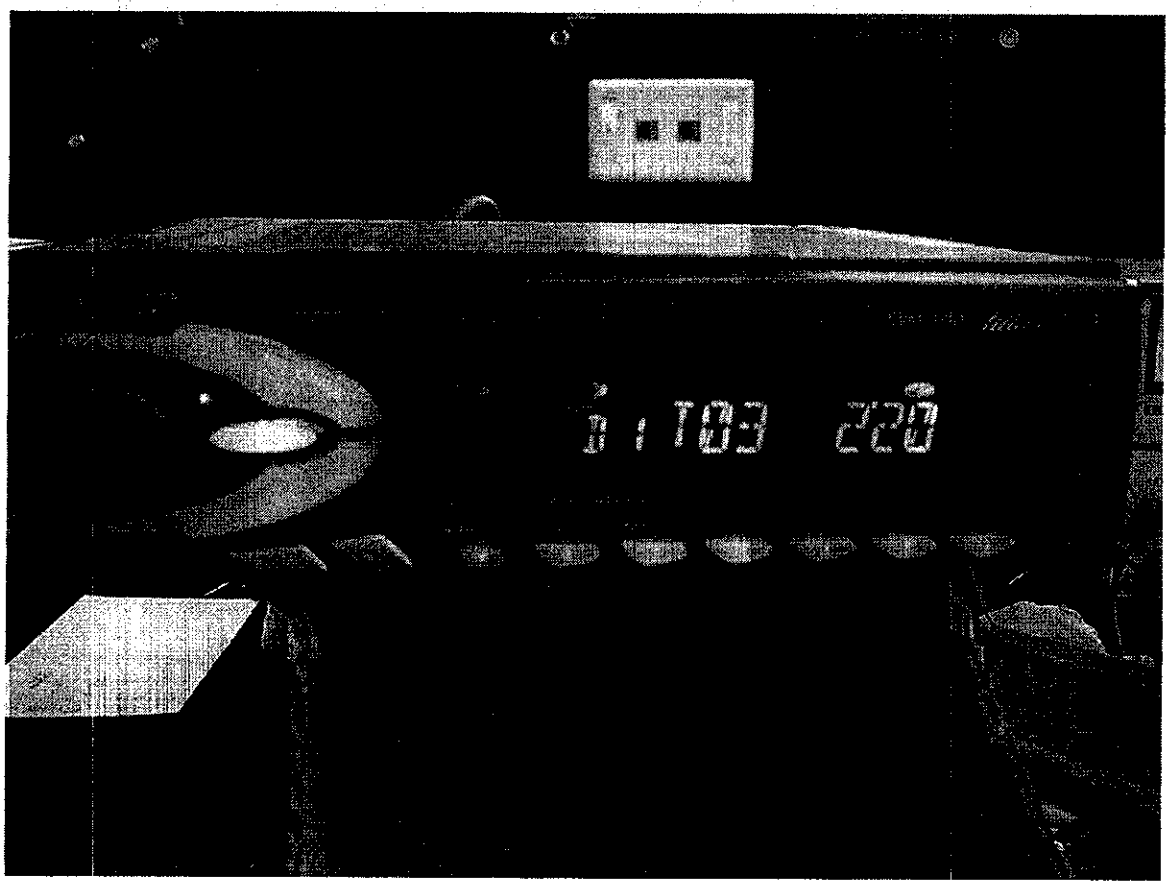


Figure 5-8: The Alpine CD Changer Head Unit

5.2 Other Features

The head unit commands everything from external components to preprocessing the audio for modifications. Head units with CD changer capabilities may have buttons to change discs or may have an up and down arrow. Not every head unit is the same and certain features may allow playback of MP3 audio straight from a CD. Some head units have pre-amplifiers while some contain remote controls for away from the head unit navigation. Certainly implementation of these functions are possible, however some are not needed nor required for the basic functions of a hard drive MP3 player.

6 *Tektronix Logic Analyzer Family – TLA715*

The TLA700 series logic analyzers are high-performance analyzers with digitizing storage oscilloscope (DSO) and pattern generator module. The DSO module enables digital real-time signal sampling with different triggering modes and signaling that allows for the optimal and most informative analysis of any waveform. The logic analyzer software can be downloaded from Tektronix's website and can be installed on any PC-compatible computer running Microsoft Windows. With the downloaded viewer we are able to view the files saved from the TLA715 from the convenience of any computer with out the device connected.

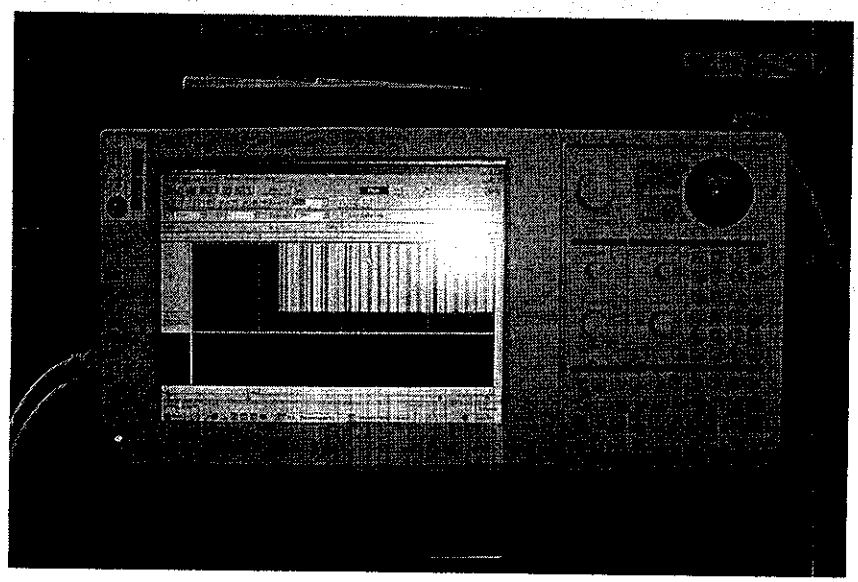


Figure 6-1: Tektronix Logic Analyzer – TLA715

6.1 General Purpose Probes

The Tektronix logic analyzer has its own probes that allow digital sampling of waveforms and other information going into the probes. By using these probes we hooked up it up to the data line coming from the head unit and monitored the pin for waveforms for which we then interpreted as the M-Bus protocol. Figure 6-2 is the probe for the logic analyzer. Figure 6-3 is the end that connects to the logic analyzer. Figure 6-4 shows the end of the many probes able to use for analyzation.

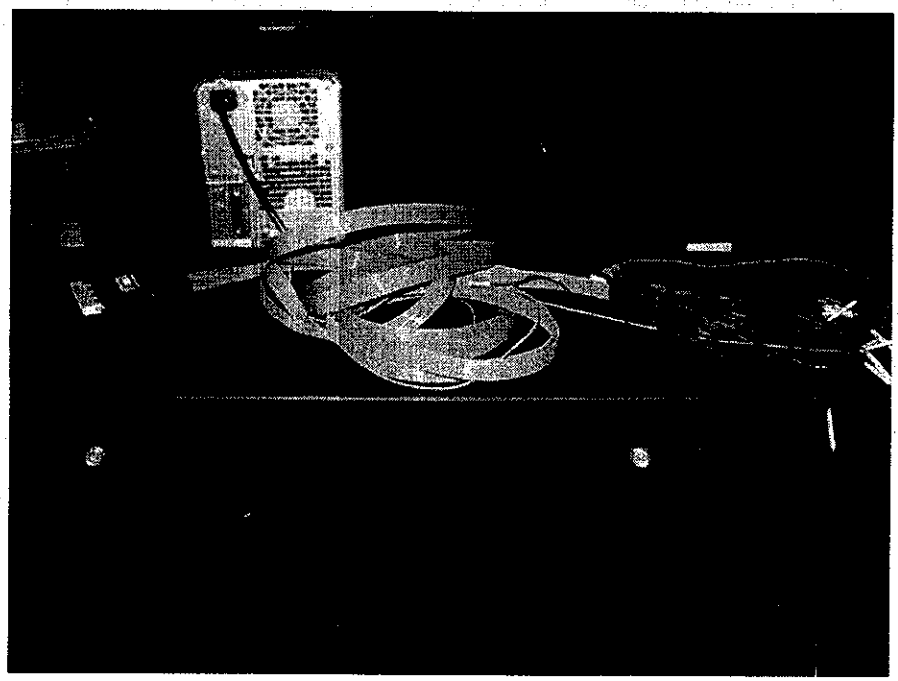


Figure 6-2: Logic Analyzer Probes

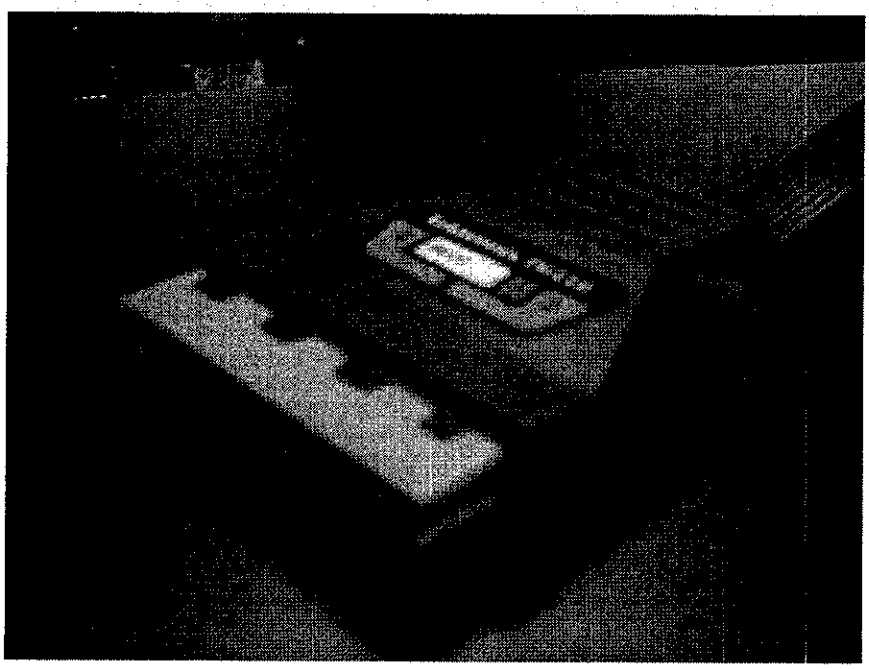


Figure 6-3: Cable connecter to the logic analyzer

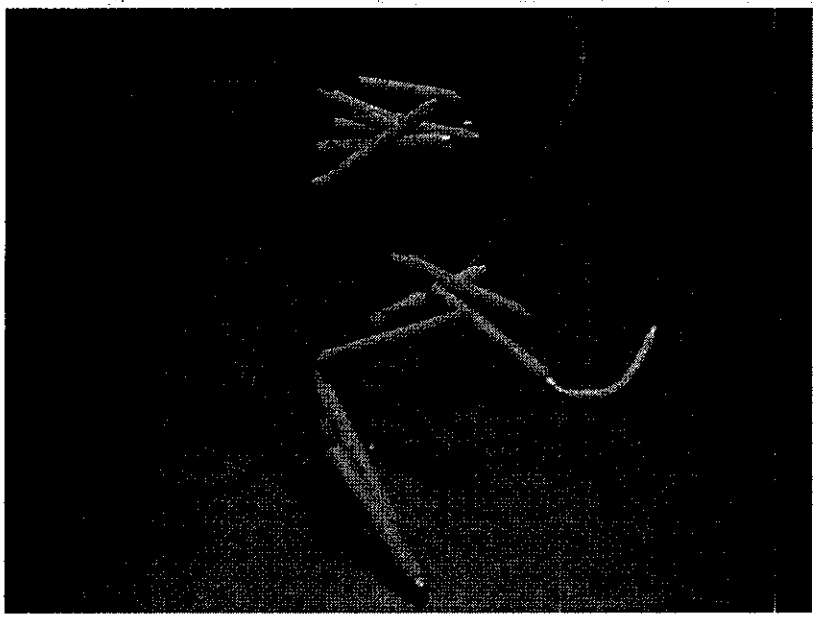


Figure 6-4: Probe connectors for analysis of waveforms

6.2 Logic Analyzer Settings

The system window has certain settings needed for the type of analysis desired. In this window the options to turn on monitoring, system setup, and triggering setup are located here. In addition listing windows and waveform viewer are also included. Below are some screen captures of the various setup screens and windows.

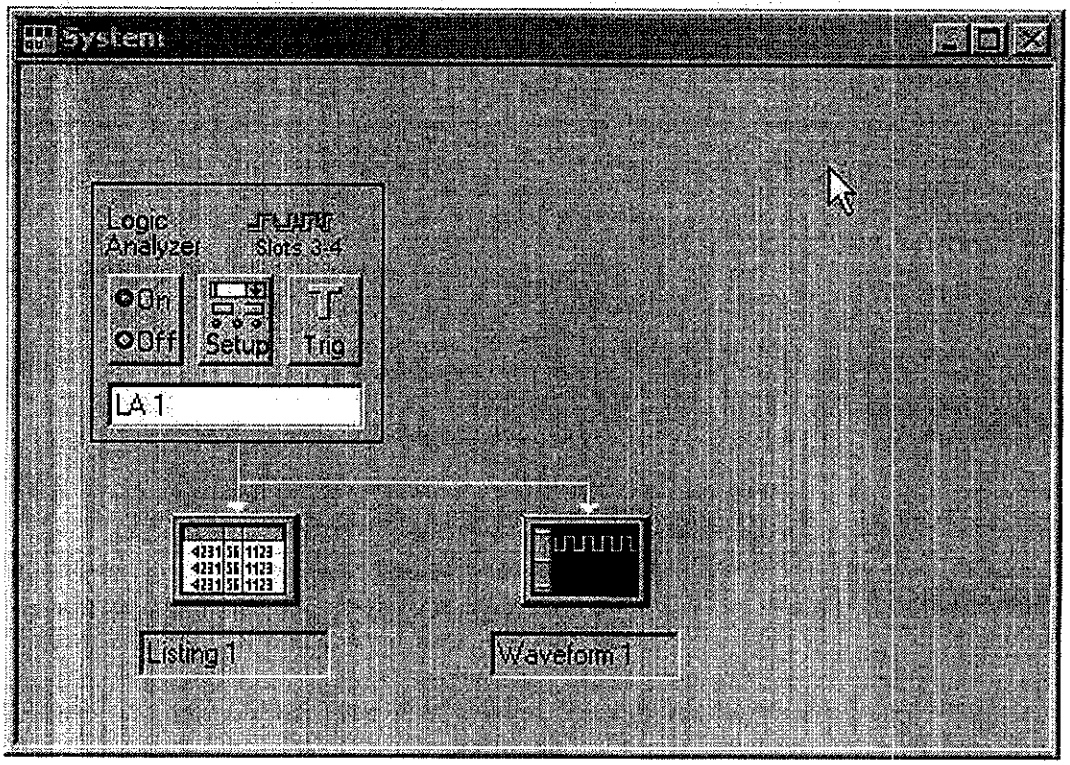


Figure 6-5: System setup window

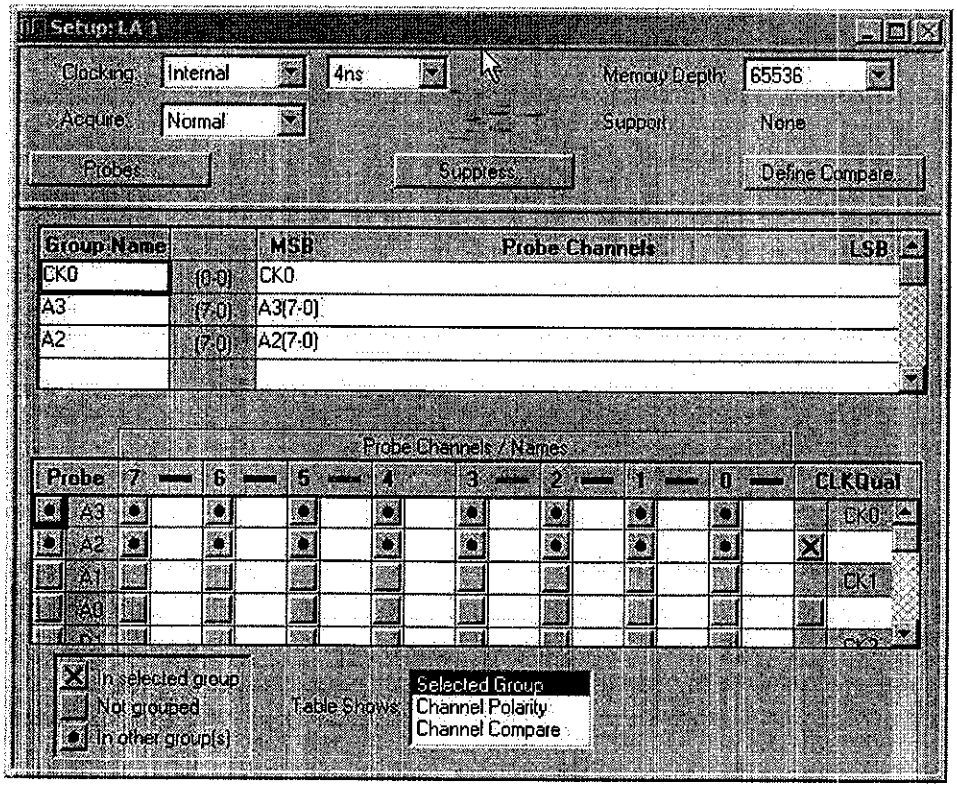


Figure 6-6: Probe setup window

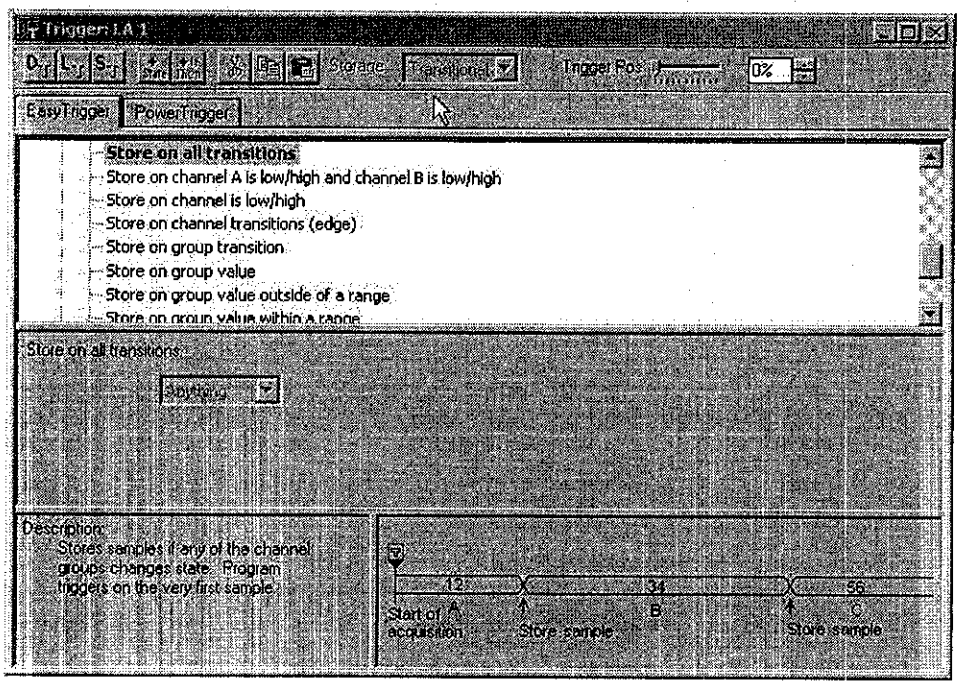


Figure 6-7: Triggering setup window

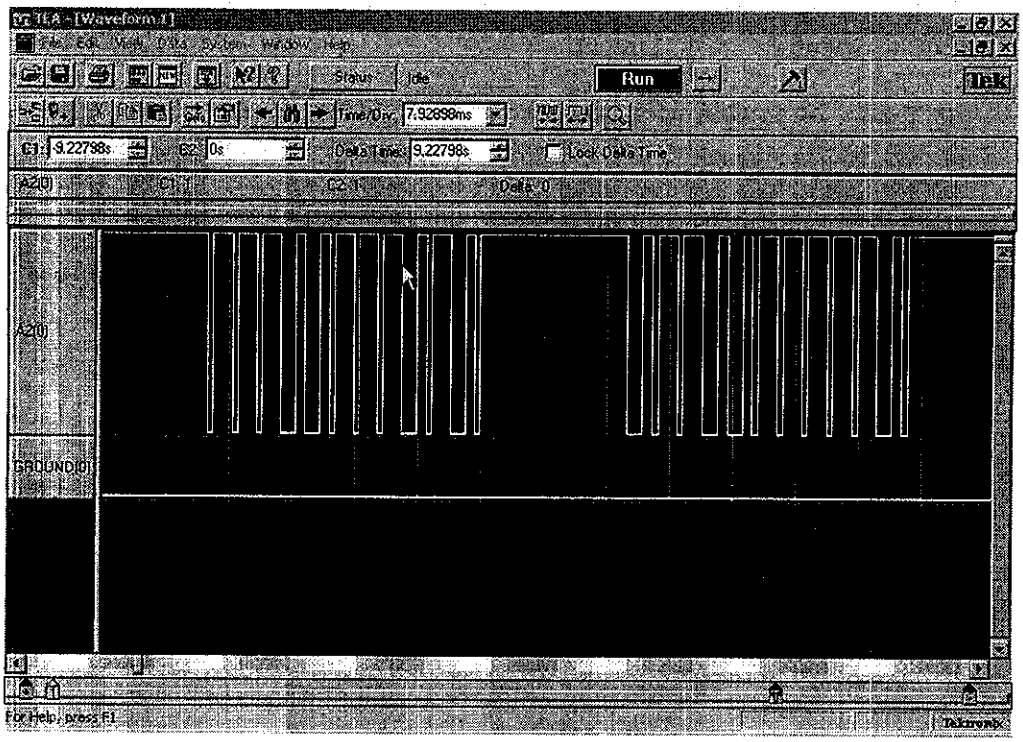


Figure 6-8: Waveform window

6.3 Triggering Setup

Knowing the triggering for the probes required extra settings necessary for the correct readings of the data pin. The triggering is a series of events when to the logic analyzer should begin capturing. The trigger program filters acquired data only for the settings that the user inputs. We had our system triggering set up for

- Store on all transitions
- Store on channel A/B when low/high
- Store on channel low/high
- Store on channel transitions
- Store on group transitions (edge)

7 *MPEG Layer III (MP3)*

The current standard for digital audio is the MPEG Layer III or MP3 format. This chapter will briefly describe what MP3 is, the process of encoding and decoding MP3, the benefits of using the MP3 audio codec for this project, and two different types of decoder chips.

7.1 What is MP3?

MP3 is am audio compression format that allows a song to be compressed by a factor of 10 to 14 without much loss of audio quality. Therefore a 32 MB song in regular CD format would compress to only 3 MB and results in near CD-quality sound. The compression algorithm of MP3 was developed by research on acoustics and human ear physiology. Basically when an MP3 audio file is created it removes certain sounds that:

- The human ear cannot hear
- Certain sounds that the human ear can hear better than others
- Two sounds but one is louder and the other is significantly softer.

By eliminating sounds as mentioned above only near CD-quality sound is created due to the certain elements removed.

7.2 Creating a MP3 File

The process of creating an MP3 file is not a painstaking one. Ripper software allows the CD audio to be read from the CD and stored on your computer if so desired. The ease of ripper software today can encode MP3 files "on the fly" meaning the audio stream is compressed and encoded into an MP3 file as fast as your CD-ROM can read it. With such technologies and programs any audio file can be made into a MP3 file for playback on any computer music player or MP3 compatible stereo deck.

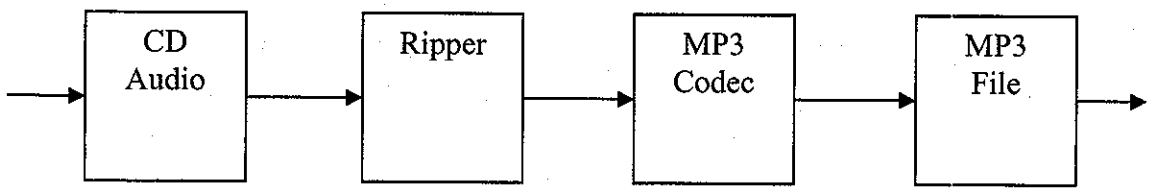


Figure 7-1: Creation of a MP3 file

7.3 Why MP3?

For the obvious reasons why MP3 is the best choice for digital audio for our project is that it allows a solid compression scheme and allows end-users to rip their own music. It enables the consumer to encode and fill the MP3 player with as many songs dependent upon the amount of hard drive space available on the hard drive. With this convenience a user can easily transfer every CD he/she ever owned onto the MP3 hard drive and take the collection wherever desired.

For technical reasons we choose this audio compression format because there were many MP3 decoder chips available in the market that allows simple circuitry and made MP3 playback much easier to work with. Some chips already come with a Digital to Audio Converter (DAC) making the design with one less component.

7.4 MP3 Decoder Chips

There are numerous solutions available for hardware MP3 decoding now available on the market. We explored many of these methods before settling on our final one. Each method included a different chip combination. The chip combination included the decoder chip and a Digital to Analog Converter (DAC). Initially, our design consisted of the STA013 from ST Microelectronics and a separate DAC chip. Although this method is described briefly in the content proceeding, we ultimately abandoned this design. The reason for abandoning this popular design for MP3 players was in favor of the single chip solution; namely VLSI's VS1001.

7.4.1 STA013 MP3 Decoder

The STA013 is a very good and widely used chip from ST Microelectronics. STA013 is distributed by popular distributors such as Mouser Electronics, an internet retailer, therefore this chip is very easy to acquire. This contributes to the chips popularity in projects and hobby type applications. This chip receives its input and outputs using serial communication. The output of the STA013 is digital and requires an additional chip to function for our application. This chip needed is a DAC and outputs the analog music signal. The design using this chip combination was abandoned in favor of a single chip solution

7.4.1.1 STA013 Description

The STA013 is a fully integrated high flexibility MPEG Layer III Audio Decoder, capable of decoding Layer III compressed elementary streams, as specified in MPEG 1 and MPEG 2 ISO standards. The device decodes also elementary streams compressed by using low sampling rates, as specified by MPEG2.5. STA013 receives the input data through a Serial Input Interface. The decoded signal is a stereo, mono, or dual channel digital output that can be sent directly to a D/A converter, by the PCM Output Interface. This interface is software programmable to adapt the STA013 digital output to the most common DACs architectures used on the market.

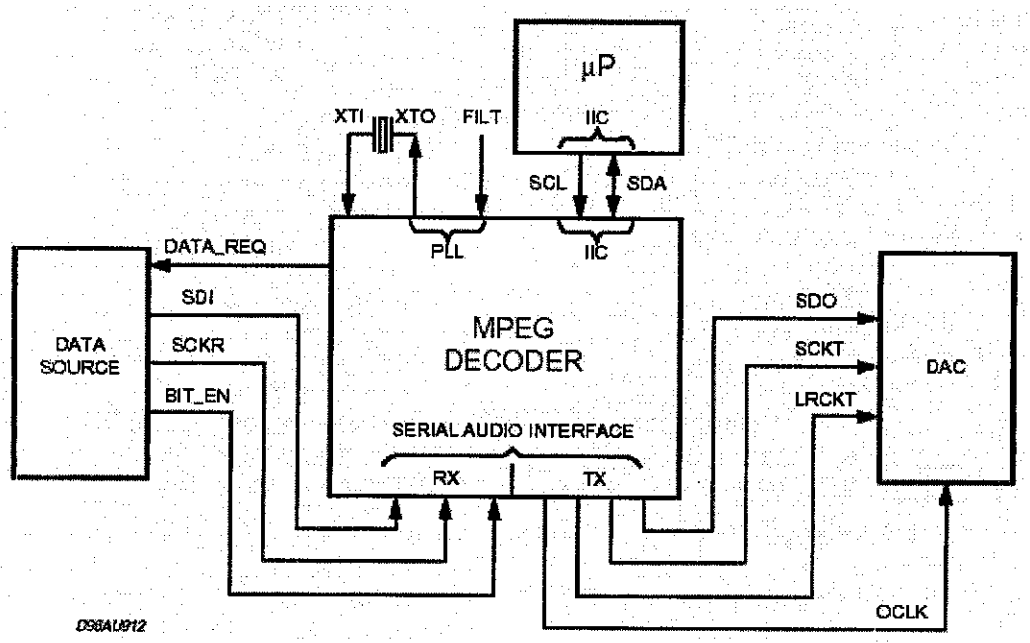


Figure 7-2: Block Diagram for the STA013

7.4.1.2 Controlling the STA013

The functions of the STA013 are controlled by the microprocessor using the I²C communication standard. I²C is a 2-wire protocol, created by Philips where one line acts as a clock and the other data. This protocol is a master slave type communication where the STA013 is always the slave. The STA013 also requires initialization by I²C communication. This is done using a routine provided by ST in the file p02_0609.bin and will not operate unless properly initialized.

7.4.1.3 Multimedia Mode

This mode of the STA013 automatically detects the bitrate of your MP3 data, and gives you a signal requesting more data. This would have been the configuration we would have used. The MP3's sample rate (32, 44.1, 48 kHz) is also automatically detected, and the correct clock and data waveforms are created for the DAC. This clock signal is sent to the DAC through the OCLK pin of the STA013.

7.4.1.4 Connecting the STA013

The majority of the pins on the STA013 simply connect to Vcc (+3V) or Ground. Five pins serve as configuration pins for the chip, Crystal (XTAL) and capacitor connections for example. The rest of the pins receive data or commands or output to the DAC. The DAC output pins consist of a clock (OCLK) and stereo channel data. MP3 input is performed using 3 pins, SDI (data), SCKR (clock), and DATA_REQ (ready). This will connect to the MP3 data source. The chip is controlled by the microcontroller using I^2C through 2 pins, SDA and SCL These pins use the I^2C protocol to send commands and query parameters, to initialize the chip and to control it while it's decoding data. Another important pin is the DATA_REQ pin. This tells the microcontroller that it is ready for more data. The program will read wait for this signal before sending more data. The specific details and remaining for connecting this chip are found in the data sheet from ST Microelectronics.

7.4.2 VLSI VS1001 Decoder Chip

Shortly after learning numerous details of the STA013 we were introduced to the VS1001 decoder chip. The VS1001 is a newer decoder chip from the Finnish manufacturer VLSI Solution that was announced on October 31, 2001. This chip is a complete MP3 decoder, including a DA-converter and headphone amplifier. This is a single chip simplifies the design from two or more chips. This not only reduces the cost of the device, but also power consumption. The manufacturer states that typical current consumption of digital decoding function and analog interface with 30-ohm load is about 17mA. VS1001 decodes all MPEG 1 & 2 layers I, II, and III files as well as MPEG 2.5 layer III extension files with all sample rates and bit rates. Variable bit rate and PCM input are also supported. This would make our player capable of playing virtually any MP3 format with ease. A block diagram from VLSI is shown in Figure 7-2.

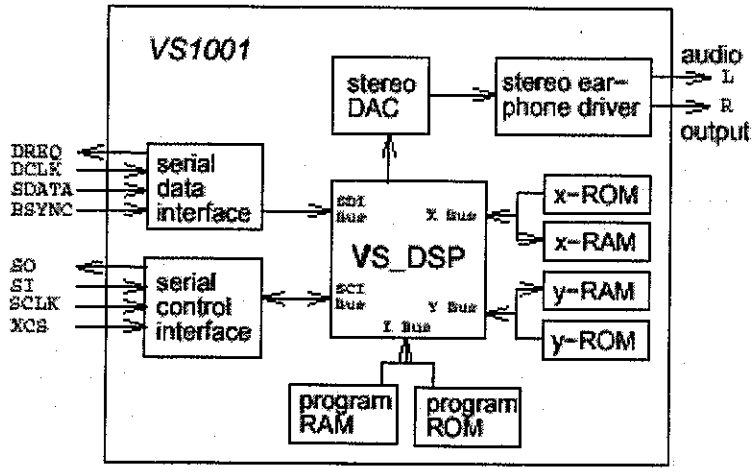


Figure 7-3: VS1001 Block Diagram

7.4.2.1 Using the VS1001

Like the STA013 this chip is also a system slave device. The VS1001 is controlled using a serial control interface by the microprocessor. The VS1001 receives its input bit stream through a serial input bus. This is shown in the block diagram (Figure 7-2). The MP3 stream enters the buffer of the VS1001 that implements two user-transparent buffers: an input bit stream buffer, and an audio output buffer. The length of the input bit stream buffer is 16384 bits. This produced a buffer of 128 ms at a constant 128 kbit/s bit-rate. The output is passed through a stereo earphone driver. This will be used as a pre-amp for transmission back to the head unit. There is also a volume control that will not be utilized since the music signal will pass through an amplifier (head unit) after it leaves this device. The optimum volume will be found through experimentation then coded for this value as a constant. The VS1001 also incorporates basic DSP functions including bass/treble enhancer.

VS1001 Clock Frequency

The clock of the VS1001 chip is selected by an external crystal. The crystal is connected in parallel across the pins XT1 and XT2. The crystal is placed in standard configuration utilizing two equal capacitor values. The manufacturer's data sheet specifies a crystal rated at 12.288 MHz. This crystal frequency causes the chip to have a maximum sample rate of 48 KHz. The clock rate also determines the maximum bitrate possible for decoding. This information can be found in the VS1001 datasheet. The value that we will use will be the 12.288MHz chip and has already been acquired from an online retailer. These pins can also be used as a hardware reset by grounding them with XRESET. A software reset also exists.

ESD Protection

The VS1001 datasheet includes an application note concerning ESD protection. High electrical charges may harm the analog outputs of the chip. This is a fundamental limitation of CMOS technology and requires external protection circuitry. The image shown in figure 3 is taken from the VS1001 datasheet and shows how to protect the chip against strong electronic shocks. It is apparent that a capacitor and resistor serve as a protection circuit.

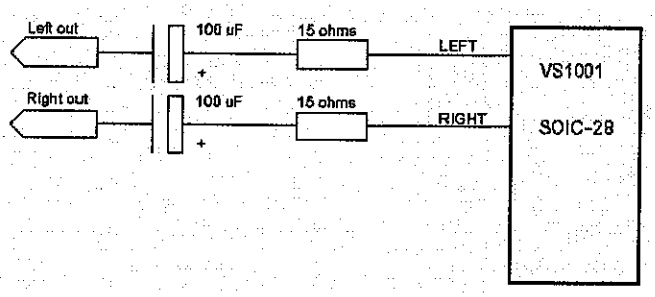


Figure 7-4: Block Diagram of ESD Protection

Logic Level Conversion

The VS1001 uses a 3.3V power supply and operates under 3.3V logic levels. In order for this chip to communicate properly with the 5V logic used by our microprocessor a logic level converter must be used. We will be using the SN74LCV245 from. One reason for using this chip is due to the fact that we were able to order a number of these chips as free samples from TI which made it readily available to us. The chip can take inputs from 3.3V or 5V devices. This feature allows the device to be used as a translator in a mixed 3.3V/5V system environment. The pins from the port on the microcontroller are connected are passed straight through the logic level converter to the control and data pins of the VS1001. An additional voltage regulator will be used in

Typical Connection Diagram

The datasheet contains a typical connection diagram for using the VS1001 chip. This diagram contains the details that were just recently discussed. The ESD Protection, standard crystal configuration, microcontroller port interfacing, voltage regulator circuitry are all depicted in this diagram. Our intent is to model our design circuitry after this diagram and perfect it by testing in the lab. The fore mentioned diagram, copied from the datasheet, is shown in figure 4. Since the VS1001 uses the SPI communication protocol, we will connect it to the microcontroller port which is capable of SPI communication as shown in the connection diagram.

Output Connection

The VS1001 has an internal headphone driver capable of driving a 30Ω load. Although this can be connected to headphones for prototyping and development, the output channels will be connected to the Left and Right signals on the CD changer cable. This can also be connected to an amplifier for use outside the vehicle.

7.4.2.2 VS1001 Piggyback Kit

In order to aid in the prototyping process, we purchased a kit for a piggyback board. This kit functions as the model discussed previously. The VS1001 Piggyback board is a small development board for testing and experimenting with the MP3 decoder chip and can be purchased at Jelu.se, which is linked from the VLSI homepage. The board includes all the components needed to get the VS1001 to work with a microprocessor and is modeled after the typical connection diagram. This board includes the VS1001 chip, crystal oscillator, a logic level converter, a 3.3V regulator, and all the discrete components needed for operation. The board was purchased as a surface mount device PCB and assembled as part of the project by Daniel. All input and output pins are connected to an onboard header for interfacing. Assembling this board was quite a task on its own. The size of the board is about 1.5 inch square and the solder pads were in the dimension of mils. The header pins will correspond to the port on the micro controller and the analog music signal output. Figure 7-4 shows a basic connection diagram for the VS1001.

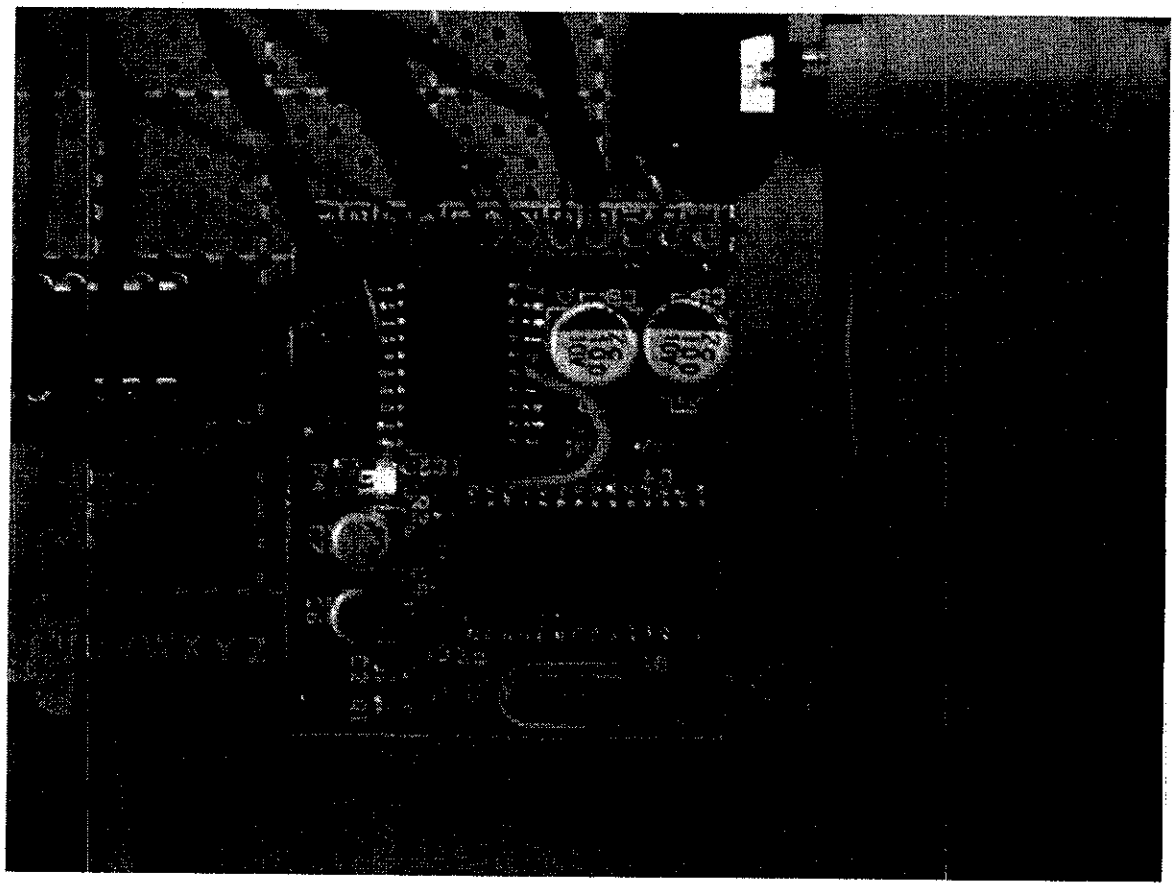


Figure 7-5: VS1001 Piggyback Board

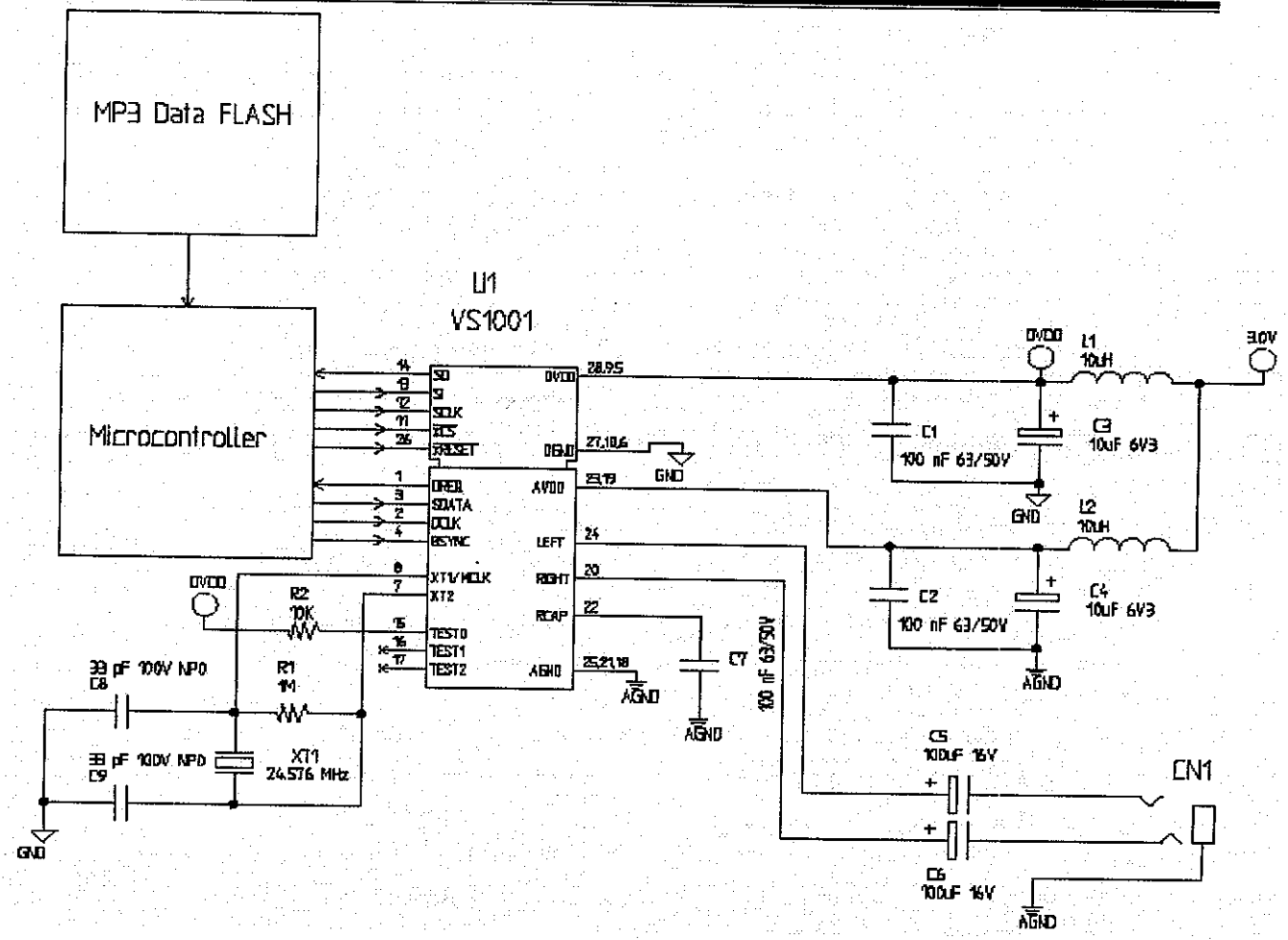


Figure 7-6: VS1001 Basic Connection Diagram

8 *PIC CD Changer Emulator Module*

The programmable integrated circuit (PIC) module for the project is the translator between the CD Changer Head Unit and the Atmel MP3 hard drive player. The programming involved requires either the use of PIC assembly instructions or the use of a higher-level language such as PIC Basic Pro to achieve the same means. A mixture of both languages can be accommodated and perhaps needed in certain run-time processes. This section describes in detail the PIC and the software and programming behind the CD Changer Emulator to communicate successfully with the head unit and the Atmel MP3 player.

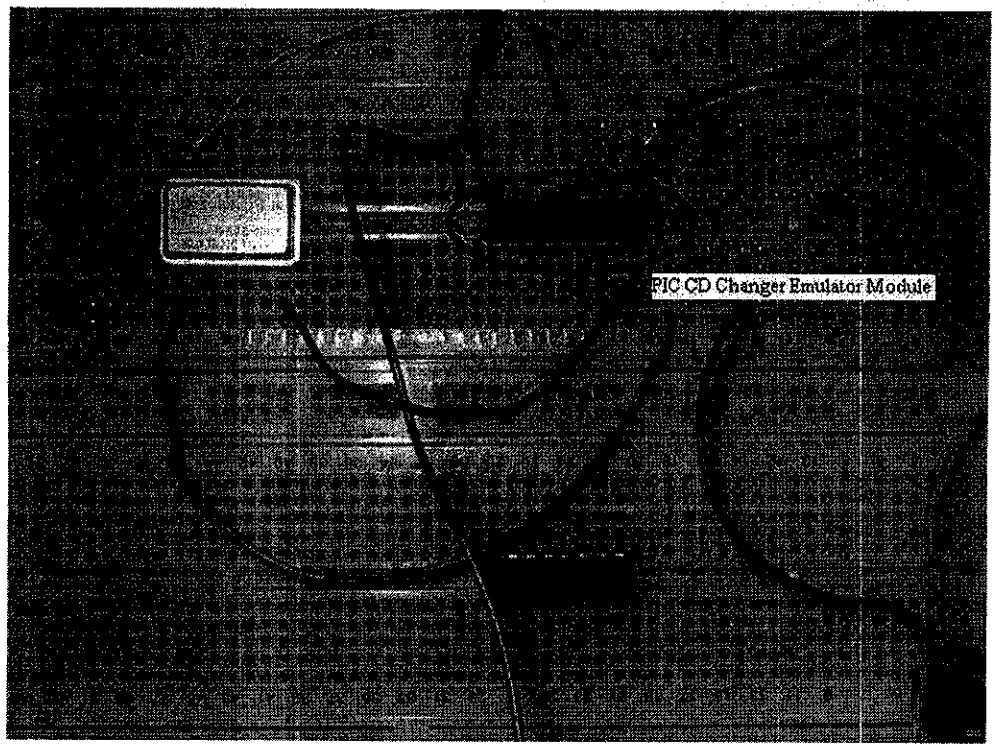


Figure 8-1: PIC16F628 Module

8.1 PIC16F628

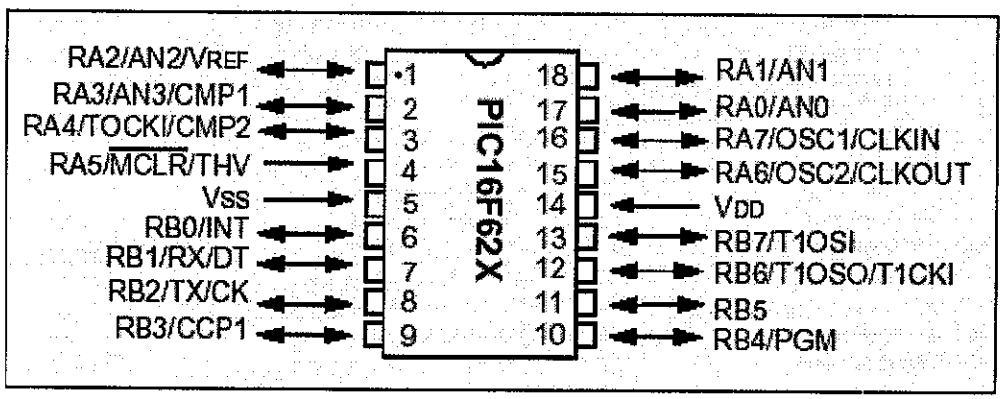


Figure 8-2: PIC16F628 Pin Diagram

The PIC16F628 compared to the PIC16F627 allows more memory for larger flash programs thus more suited for any program flexibility in the future. Programming of the PIC16F628 requires the knowledge of the PIC assembly language that only involves 35 instructions. For this reason we choose to use the PIC16F628 for our programming needs. Some specifications related to the PIC and necessary for the project are listed further in this section.

8.1.1 I/O Ports

The PIC16F628 has two bi-directional ports that are 8-bit wide latches. Although we only needed 4 pins for the project there are certain pins specifically needed to perform specific hardware functions. Although we abandoned an interrupt method for data catching in turn for a polling method, Port B, Pin 0 (RB0) served as the interrupt pin initially. RB1 and RB2 serves as serial receive and transmit respectively. RB3 allows for pulse width measurement in which we needed to determine if a value was binary logic 1 or 0. We also used one pin RA1 for modulating the response waveform. When programming Port A with this particular PIC, Port A is not only for digital I/O but also has internal voltage comparators. Therefore, setting Port A to digital I/O is a must unless we use the internal voltage comparators. For the MP3 Changer Emulator we chose not to use the voltage comparators and set the port up for digital I/O.

8.2 PIC Hardware

PIC hardware includes the different hardware we used while trying to program the PIC. Initially we bought a programmer from a website but soon found out that the quality and the reliability of the programmer was not able to suit our needs. Finally we borrowed the PICSTART Plus Development Programmer in order to complete the project programming.

8.2.1 JDM Programmer

Initially we used a inexpensive PIC programmer. The JDM Programmer from http://www.jdm.homepage.dk/newpic.htm seemed to be a low cost programmer for programming the PIC16F628. After programming a few times on the JDM Programmer it failed to write to the PIC and was rendered useless.

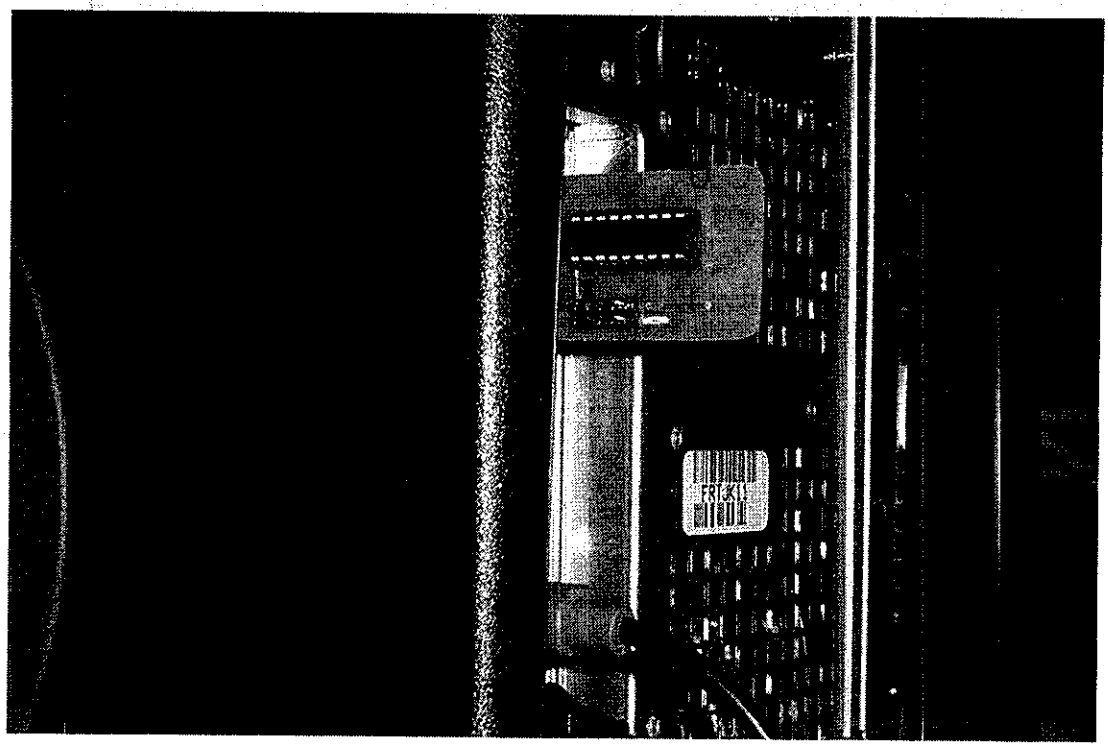


Figure 8-3: Programming with the JDM Programmer and PIC16F628

8.2.2 PICSTART Plus Development Programmer

The PICSTART Plus Development Programmer from Microchip borrowed by Dr. Fernando Gonzalez aided in the completion of the project. This programmer is nicely integrated into the Microchip's MPLAB IDE for programming the PIC when desired.

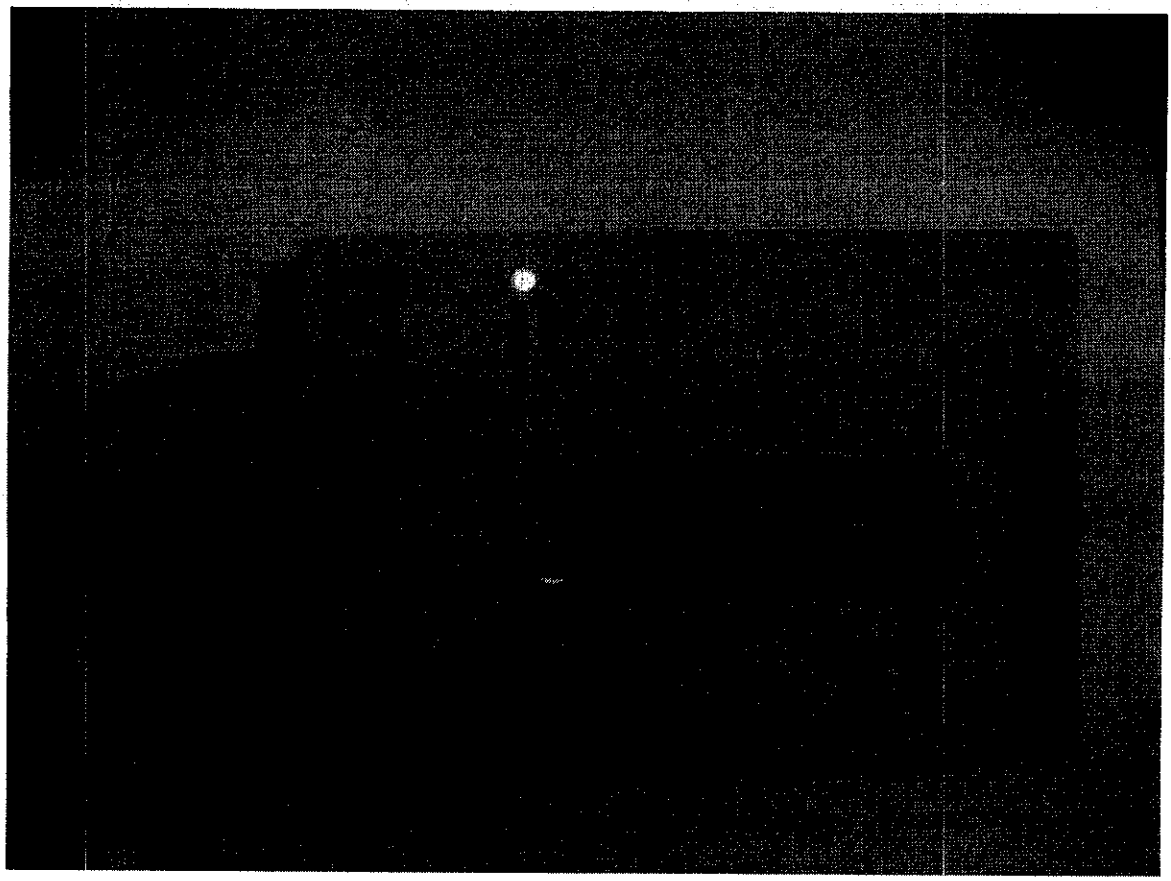


Figure 8-4: PICSTART Plus Development Programmer

8.3 PIC Software

Here are certain software packages used for programming, assembling, and debugging the PIC.

8.3.1 IC-Prog

IC-Prog is the program used for uploading assembled HEX code to the JDM Programmer.

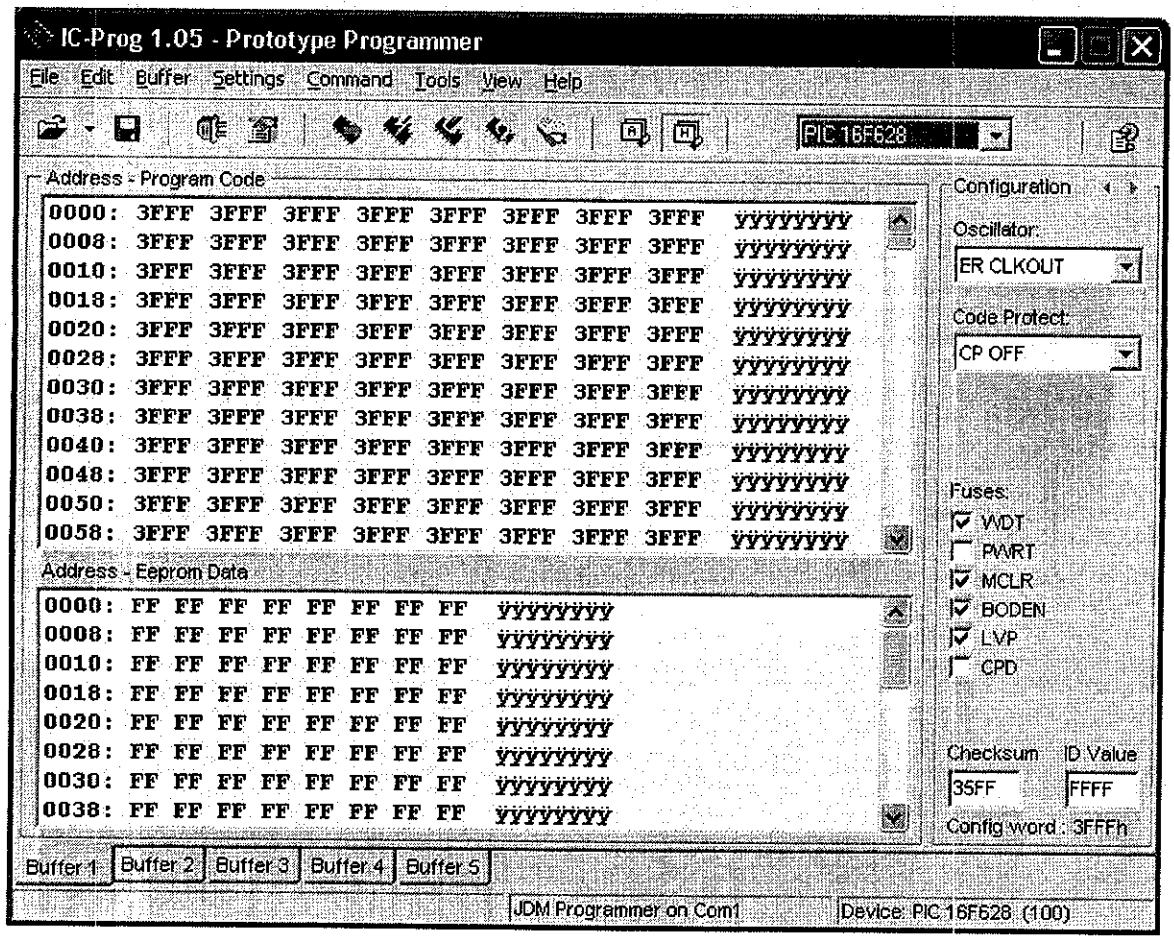


Figure 8-5: IC-Prog software to program the JDM Programmer

8.3.2 Microchip MPLAB IDE

Microchip provides a software integrated development environment (IDE) for the end-user and engineer to use for PIC programming purposes. Figure 8-6 shows a screen shot of the MPLAB IDE.

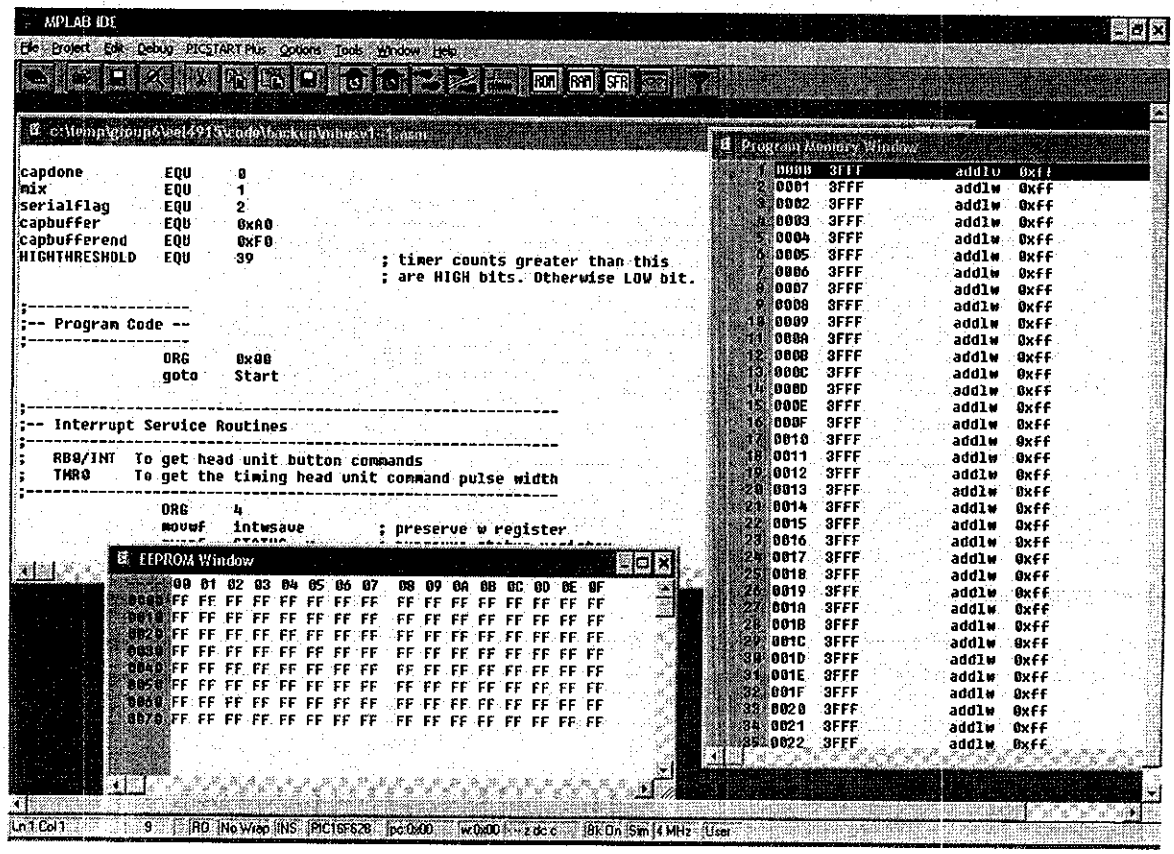


Figure 8-6: Screen Capture of the MPLAB IDE

Here simulation of the PIC processes and other watch windows can be displayed for debugging purposes. From here we are able to compile and load the generated HEX file from the assembly code straight from the IDE.

8.3.3 MPASM Assembler

In cases that the Microchip MPLAB IDE was not used the MPASM was needed to assemble our program for upload to the PIC. Figure 8-7 shows a screen capture of the MPASM program for assembling any PIC code written in assembly.

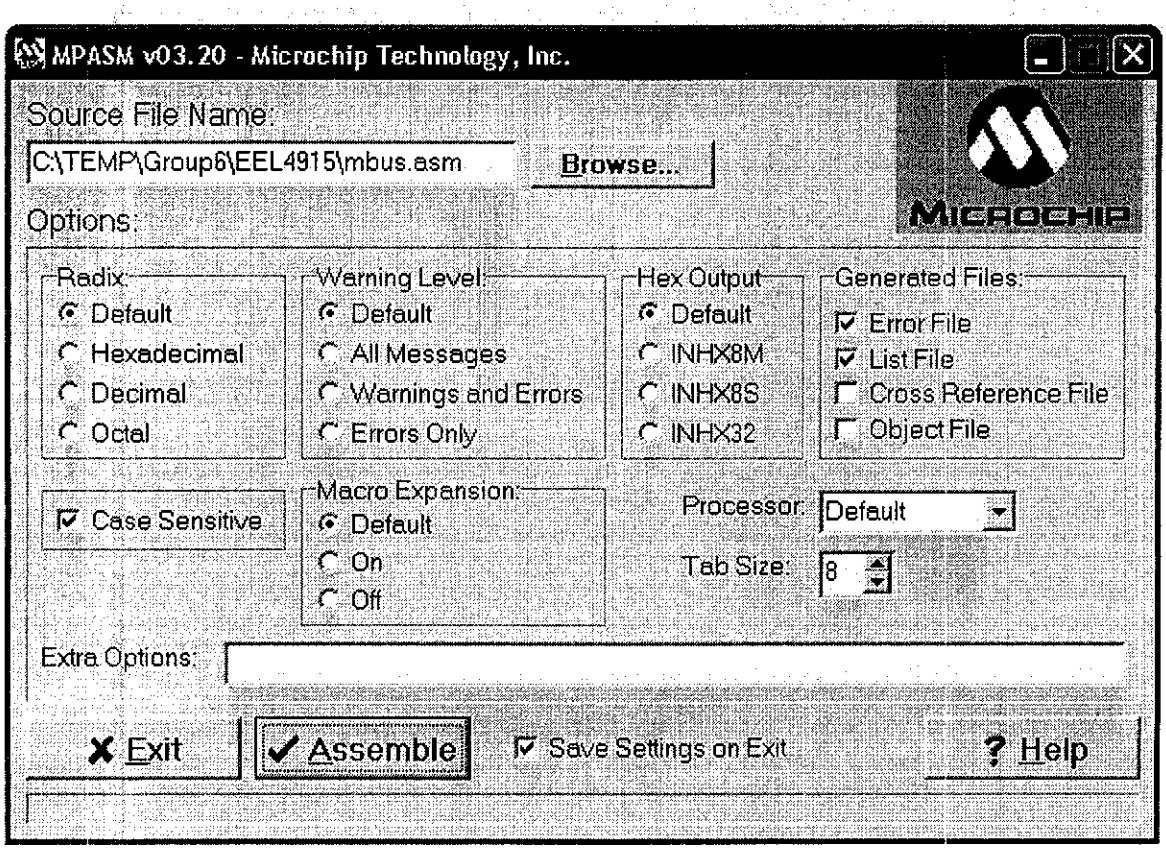


Figure 8-7: Screen Capture of MPASM

8.3.4 PIC Basic Pro

PIC Basic Pro allows PIC programming with a high-level language. With assembly instructions and branches, PIC Basic Pro allows for a more intuitive way to program instead of purely using assembly instructions designated by the particular PIC. PIC Basic Pro can be found from microEngineering Labs, Inc.

8.3.5 CodeDesigner

CodeDesigner is a software suite that is an IDE for PIC programming using PIC Basic Pro. The IDE has syntax highlighting and capabilities to program a PIC through the PICSTART Plus Developmental Programmer through the use of macros and the MPLAB IDE.

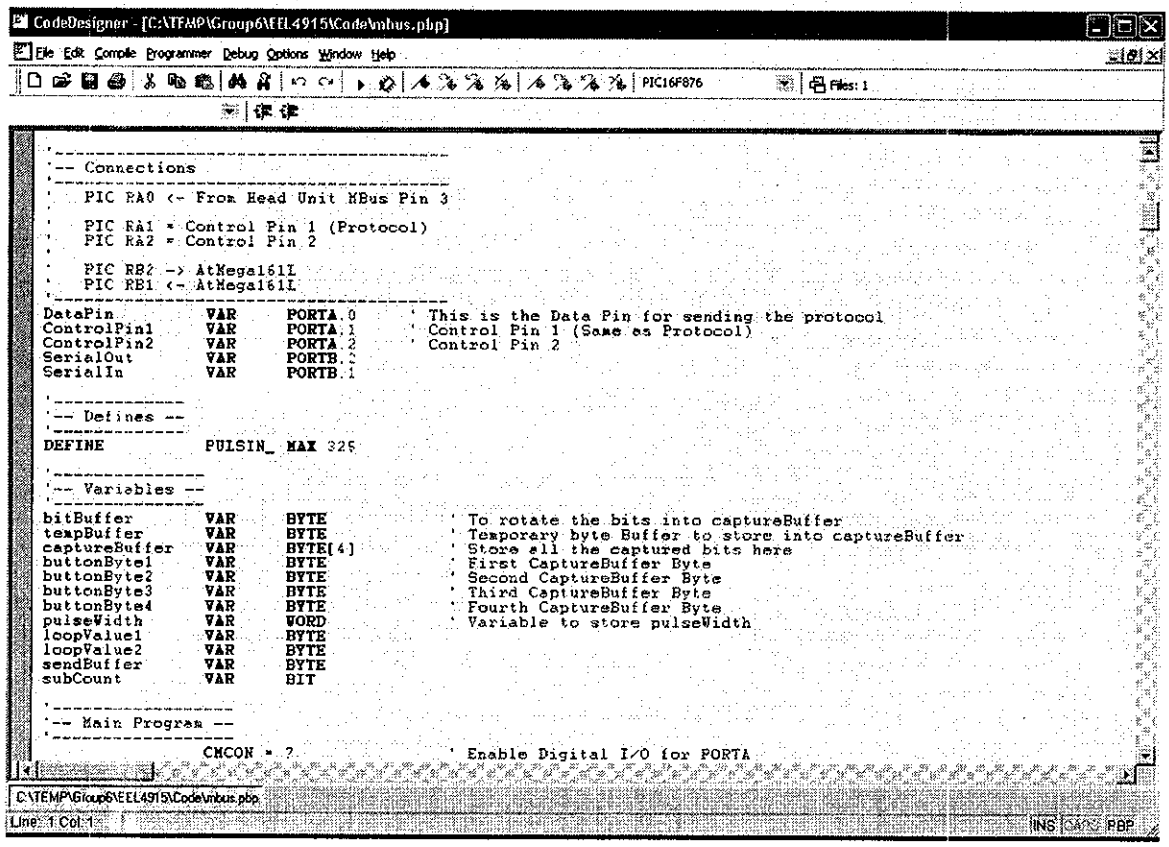


Figure 8-8: Screen Capture of CodeDesigner

8.3.6 Terminal Program by Bray++

This terminal program allows for on the fly changing of the COM Port, Baud Rate, and other serial settings. With this program we can easily debug programs using any type of serial communications. The terminal program has its advantages over HyperTerminal provided by Microsoft due to the on the fly changing of the serial port setup. The program also allows for transmittal of characters.

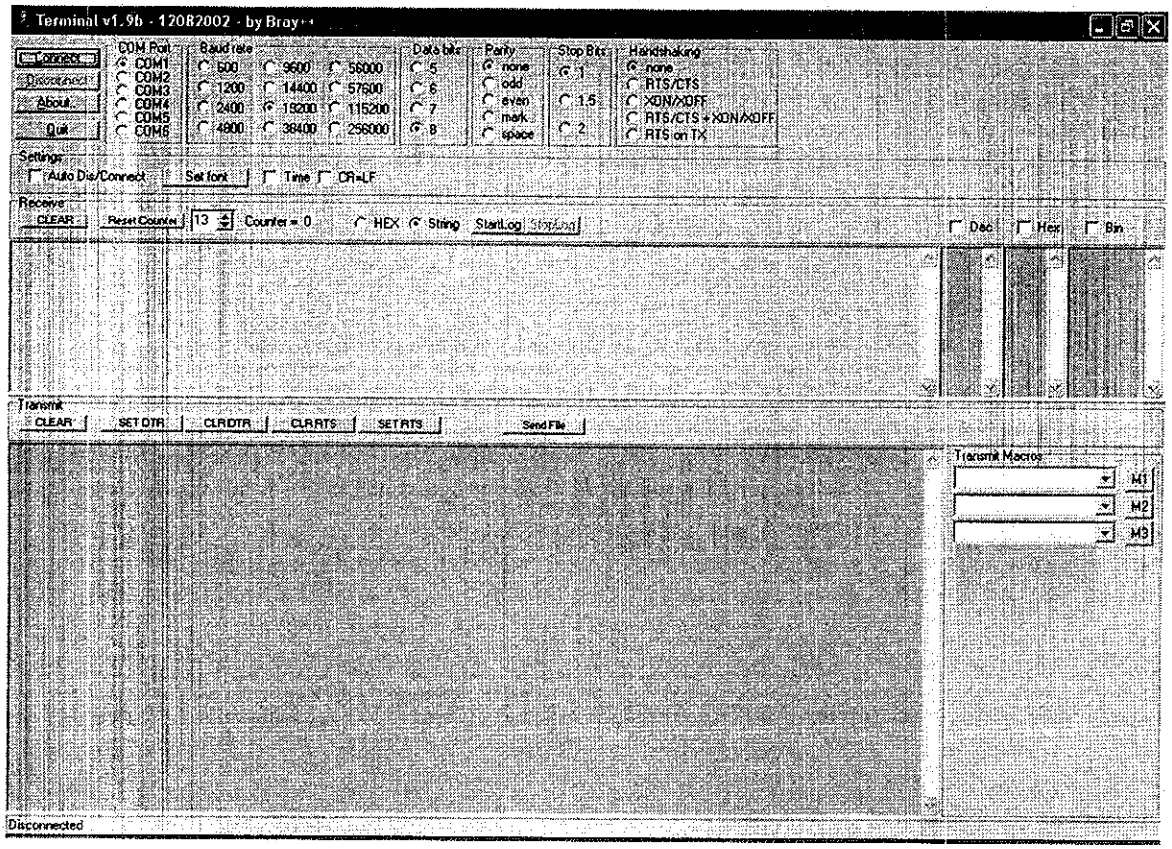


Figure 8-9: Screen Capture of Bray++ Terminal Program

8.4 PIC Programming

PIC16F628 assembly programming requires the knowledge of 35 PIC instructions, however there are other PIC Compilers for high-level languages such as BASIC and C. With this in mind we will venture into the programming practices and the pros and cons of programming in PIC assembly and with higher-level compilers. Basically the focus of this chapter will rely on assembly and PIC Basic Pro approach for programming. Figure 8-10 shows the basic flow chart of the programming needed for the PIC CD Changer Module.

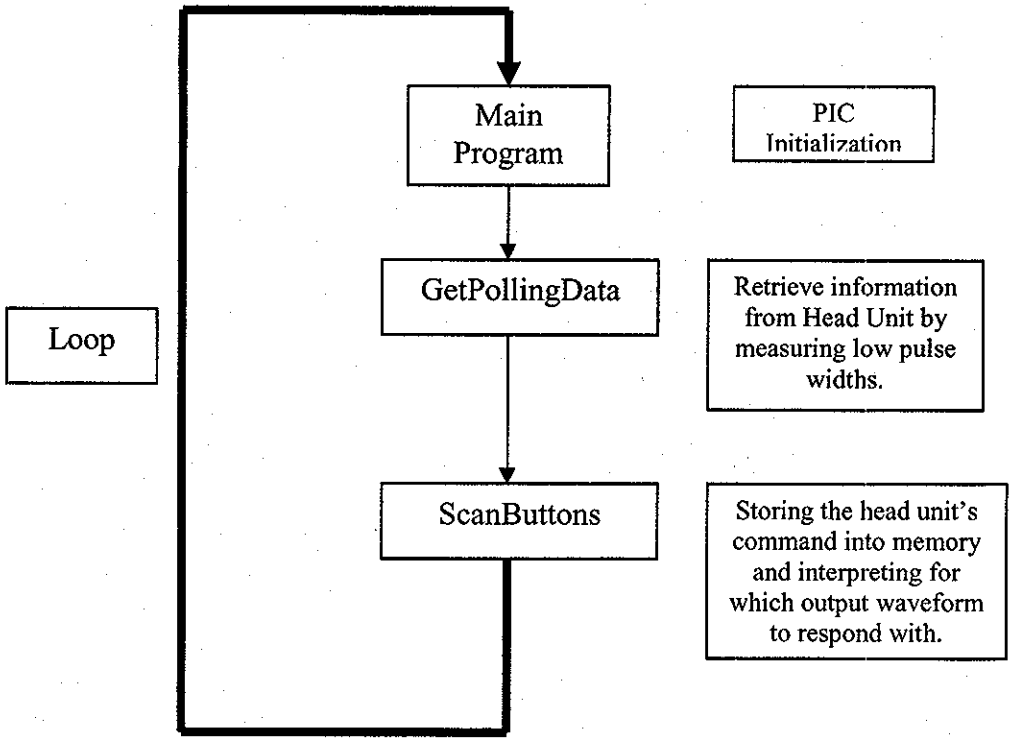


Figure 8-10: Simplified Block Diagram of the PIC CD Changer Module

8.4.1 PIC Assembly Programming

For the CD Changer Emulator we initially started coding the PIC module in pure assembly code with no use of any higher-level instructions. By developing in assembly the code runs faster, however the number of lines of code is lengthy. The learning curve tends to be much steeper when coding in assembly. The particular style of coding sometimes needs a different approach rather than a more straightforward technique commonly found in higher-level languages.

8.4.1.1 PIC Assembly Port Initialization

In PIC Assembly there were many initializations to be considered before going into the main program. If an interrupt method was needed then a bit in the INTCON register needed to be set for RB0 or for other Port B interrupts. Not to mention a global interrupt enable flag needed to be set in order for the PIC to allow any type of interrupt to occur. Another point in particular with the PIC16F628 is that Port A also contains internal voltage comparators. In order to allow digital I/O for Port A the value 0x07 was needed to be set in the CMCON register to turn off the comparators. In addition to set a particular port for input or output the TRISA or TRISB registers need to be set. When a 0 on a TRIS register then the bit is an output and when a 1 is written to the TRIS register than the bit is set for input.

8.4.1.2 Status Flags

In addition to port settings, with assembly programming was consideration of certain flags to be set. This deemed an effective means for the status of an operation yet required massive bookkeeping. When considering branches a simple subtraction of two numbers will allow the Zero flag to be set and one can branch of the zero status bit is set or clear. We needed to use such a technique to detect if the M-Bus protocol was detected in the buffer for the appropriate action.

8.4.1.3 Timing Issues

Since timing is of utmost importance when polling the head unit data and sending responses to the data pin we needed to load an negative value into the register and increment the register to zero. In order to get the timing correct we entered a negative value and incremented the count register.

8.4.1.4 Assembly Programming Problems

With all of these requirements for assembly programming and using various test codes and other methods we could not get the M-Bus protocol detected and needed an alternative to programming in PIC assembly. With this in mind we ported, actually rewrote, our code from assembly to using the PIC Basic Pro compiler.

8.4.2 PIC Basic Pro Programming

The benefits of using a higher-level language instead of using the PIC assembly instructions are greater than gaining faster process time when using the lower-level language. The ease of using a higher level allows a level of programming abstraction that is unmatched when compared to using only assembly instructions. The code is easier to follow and read and only requires a few lines as opposed to many lines in assembly to perform the same action. May of the initialization issues were resolved when switching to PIC Basic Pro due to default initialization settings made by the compiler.

8.4.2.1 Timing Issues

The only worry when using a higher-level language for the PIC CD Changer Module entailed timing for the response and polling data. When we encountered this dilemma we found the correct timing by writing various test codes and monitoring the waveform on the Tektronix TLA715 and compared it with waveforms previously captured by the actual CD Changer. With this we integrated these timing schemes into a complete program and tried to detect the M-Bus protocol from the Head Unit.

8.4.2.2 M-Bus Detection Issues

When initially attempting to detect the M-Bus protocol the newly ported program failed in detecting the M-Bus protocol. When trying to debug the program we could not determine what was happening. However, since the PIC16F628 already contains serial communication capabilities we used the Terminal Program by Bray++ and outputted what the program detected. With this ability to see what the program detected provided the greatest debugging utility. We were able to logically and systematically determine what the program saw and discarded.

The main problem with polling for the M-Bus protocol is that when the head unit sends a command to the data pin the time of when to sample the line for the PIC is unknown. An interrupt method was abandoned due to the problem with the PIC being interrupted continually while processing the buffer for M-Bus protocol detection. Therefore, a

polling method was used. However in order to detect the protocol we needed to find out what the main problem that was inhibiting detection. After many attempts, test code, and analysis a nested for loop was used to detect the head unit identification packet. Once we were able to detect the head unit identification packet the rest of the bits are sampled into a buffer and later used to interpret the command that the head unit attempted to send.

8.4.2.3 Inline Assembly

With higher-level languages the possibility to intermingle assembly or lower-level instructions with high-level language structure allows for the maximum flexibility and empowerment for optimal programming. Certain ways of doing things are actually easier to do in assembly rather than a high-level language like BASIC. When programming the CD Changer PIC module inline assembly allowed certain functions to work at a more efficient speed and seemed to be a more intuitive method than using high-level functions.

Example of Inline Assembly programming within PIC Basic Pro

```
'-------------------------------------------------------
'-- SendByte
'-------------------------------------------------------
'  sends a byte to head unit.
'  load byte to sendBuffer before calling subroutine
'-------------------------------------------------------
SendByte:
                For loopValue1 = 0 to 7
                    Asm
                        rlf         _sendBuffer, 1
                        btfsc   STATUS, C
                        Call    _CntrlPinSet1

                        btfss   STATUS, C
                        Call    _CntrlPinSet0
                    EndAsm
                Next loopValue1

                Return
```

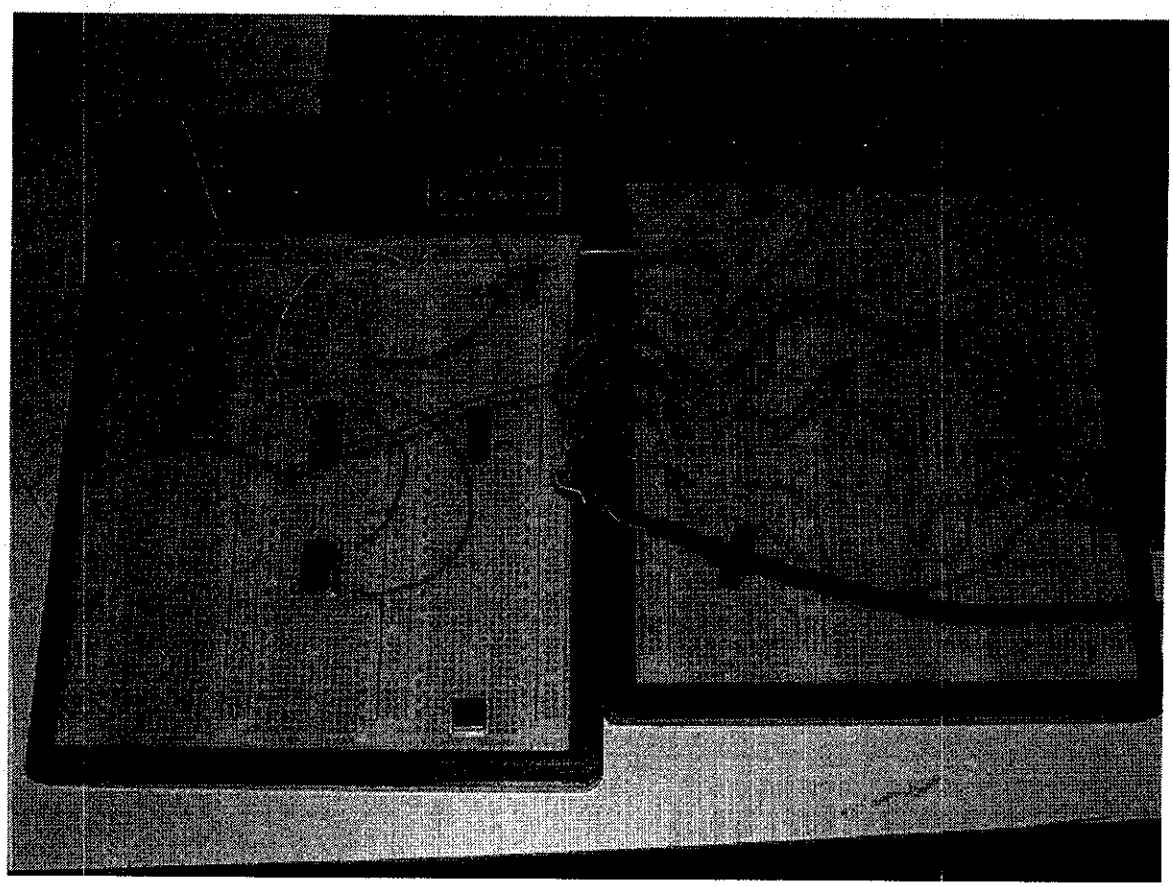


Figure 8-11: PIC CD Changer Module Wiring

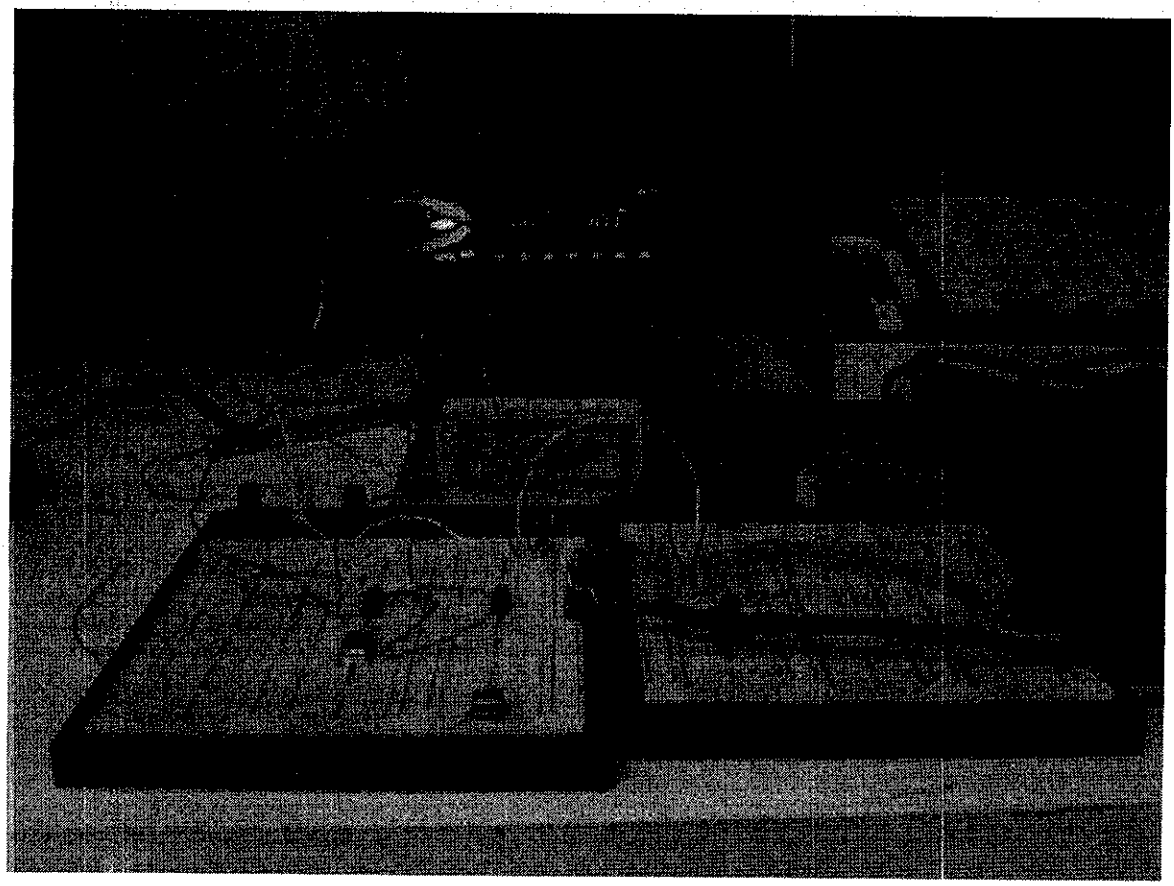


Figure 8-12: Testing the PIC CD Changer Module

8.5 PIC Hardware Accompaniment

8.5.1 SN74HC125 Tri-State Buffer

During development on the protocol detection, we thought we needed a buffer in order to pull the data line to ground and back to high in order to simulate the waveform with the PIC. This interfacing circuitry was later abandoned when it was discovered that its use was not entirely necessary. By using this we were able to simulate the timings needed for a low pulse width according to the timing analyzation from the Tektronix TLA715. When we verified that the buffer acted accordingly we attempted to interface with the head unit using a test program and the head unit succeeded in detecting a CD Changer response. From this we further adapted our code to use the buffer to pull the data pin high and low for the timing for a logical 1 or 0. In addition we added code to detect the M-Bus protocol and send out a response waveform successfully.

FUNCTION TABLE
(each buffer)

INPUTS		OUTPUT
$\overline{OE}$	A	Y
L	H	H
L	L	L
H	X	Z

Table 8-1: Function Table for the SN74HC125 Quadruple Bus Buffer

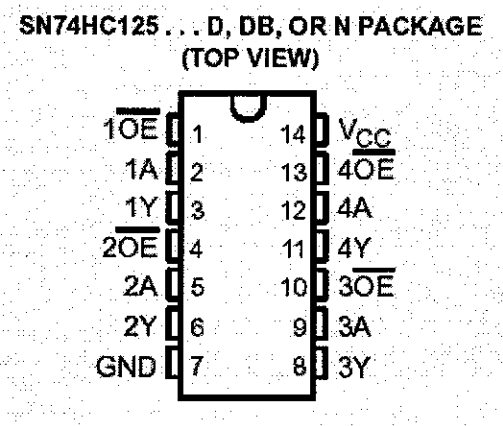


Figure 8-13: Pin Diagram of the SN74HC125 Buffer

8.5.1.1 SN74HC125 Specifications

The operating voltage level of the SN74HC125 buffer is from the range of –0.5V to 7V. We are supplying the buffer with 12V coming from the head unit. Although this is not recommended due to the maximum operating voltage level set at 7V the input and output voltage ratings can be exceeded of the input and output current ratings are taken into consideration. Since the current flow going to the buffer is very low we can essentially supply the 12V needed for our project without repercussions by exceeding the voltage rating. Although these stresses can cause permanent damage to the device and effect reliablilty, the chip was perfectly operational for developmental purposes. Any maximum exposure to the device for prolong periods may directly effect the reliability of the buffer. A buffer or transistor circuit would be used if this was to be implemented for production.

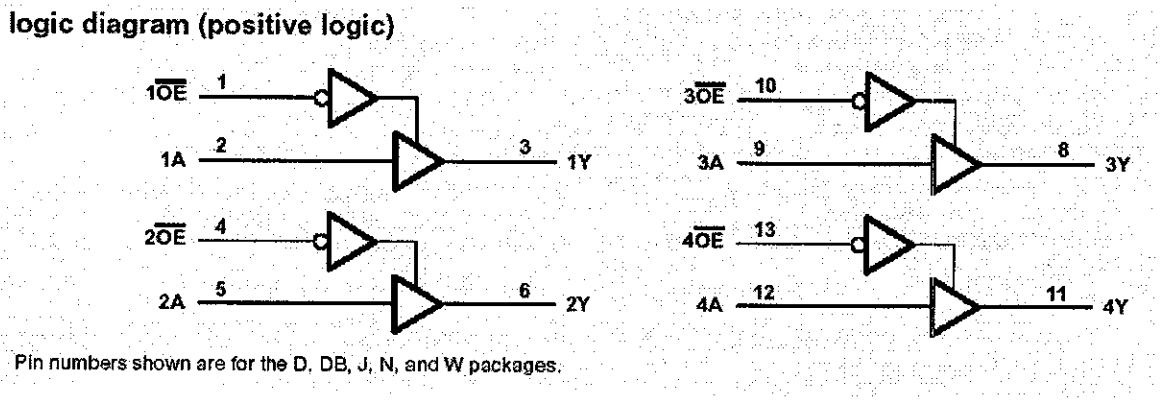


Figure 8-14: Logic Diagram for the SN74HC125

9 *Atmel Hard Drive MP3 Player*

The Atmel Hard Drive MP3 Player allows the user to upload songs to an IDE hard drive and play them by using the PIC CD Changer Emulator Module. In addition the user can simply sending serial characters to the Atmel that will interpret the commands and perform the desired action requested.

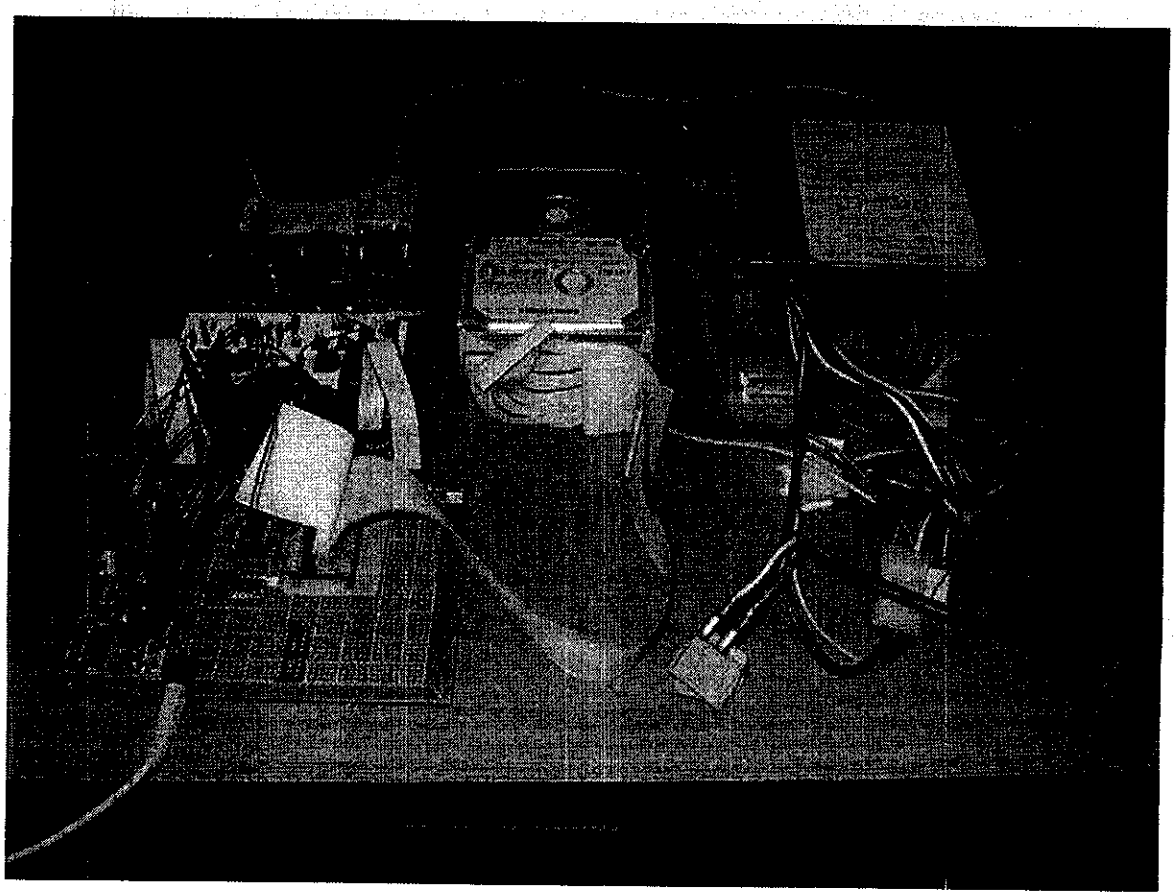


Figure 9-1: Atmel Hard Drive MP3 Player

9.1 Initial Proposed Design

The initial proposed design drafted during Senior Design I is included here to emphasize the changes occurred from the initial proposal and the final product. There were many things implemented, added, and/or deleted that compose the final product from the initial proposal. Along the way of development certain items initially proposed may have not worked or may have not been beneficial for the project.

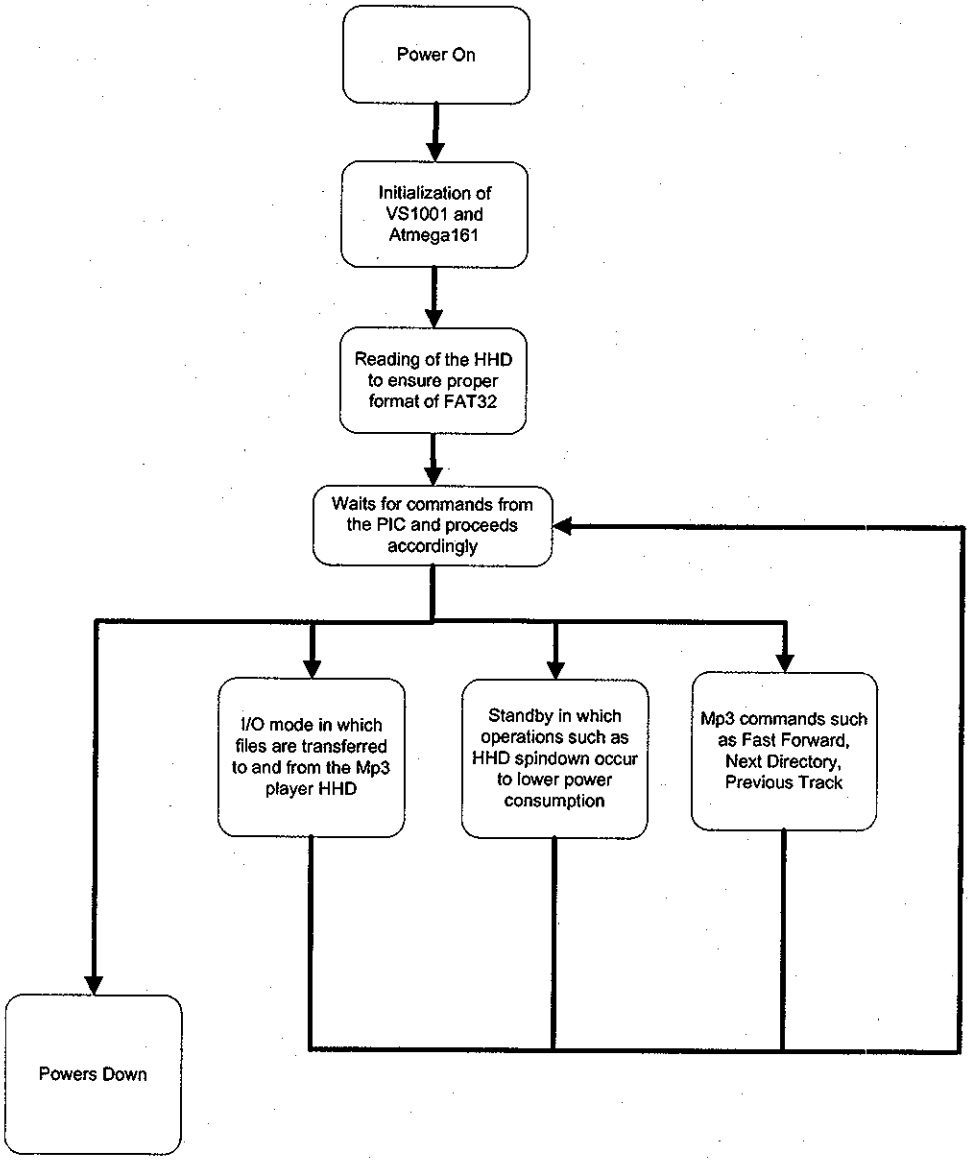


Figure 9-2: Initial Block Diagram

Block Diagram
MP3 CD Changer Emulator

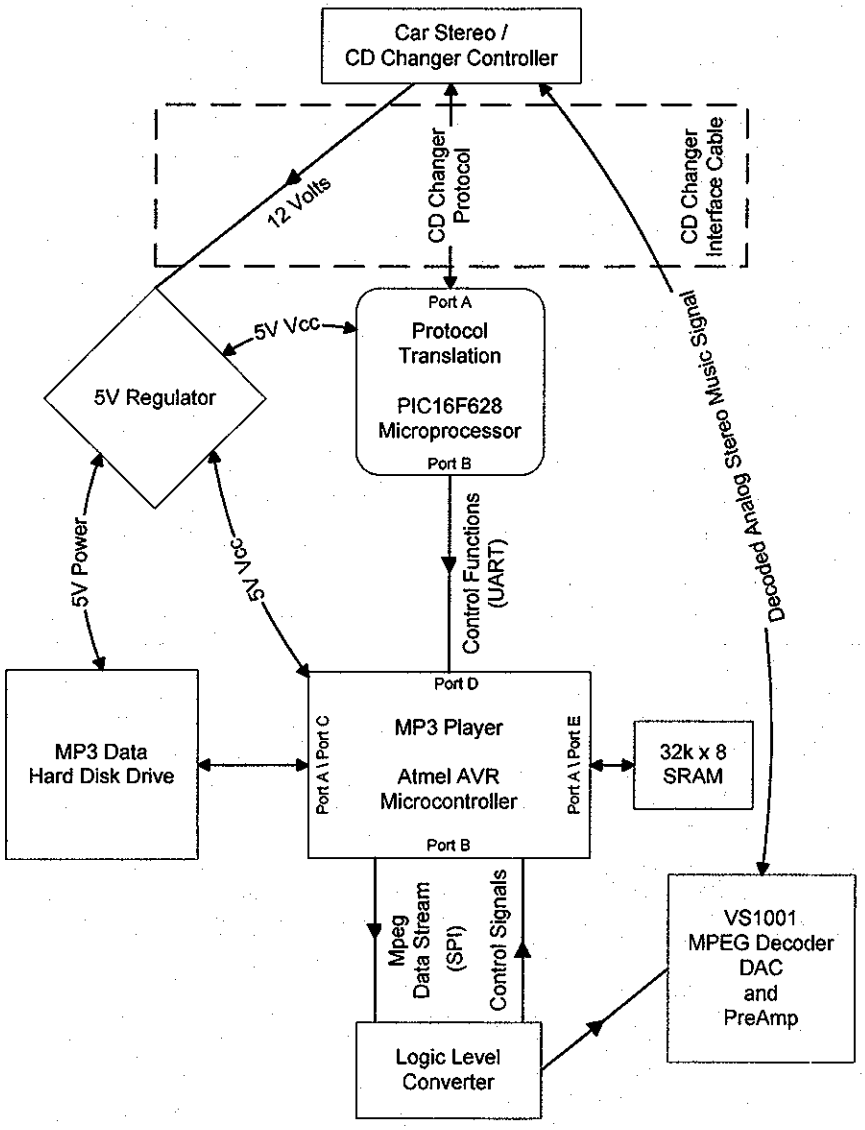


Figure 9-3: Complete System Block Diagram

Command	Description
Next CD	This moves mp3 file selection to the next directory
Previous CD	This moves mp3 file selection to the previous directory
Next Track	Finds next mp3 file in directory
Previous Track	Finds previous mp3 file in directory
Raise Volume	Optional – only used for stand alone player, volume is controlled by the CD head unit
Lower Volume	Optional – only used for stand alone player, volume is controlled by the CD head unit
Play	Plays the selected mp3 file
Pause	Pauses the mp3 file

Table 9-1: Sample Proposed CD Changer Emulator Functions

9.1.1 Regulated Power Supply

As with most electronic devices used in automotive applications the power source originates at the car battery. For CD changers, the operating power arrives through the same interface cable, which carries all other signals. The current through this cable is limited to a low value and the voltage is a fixed value. Low current allows the device only a low amount of power to consume. The voltage supplied is typically 12v due to the car battery or 14.4v due to alternator. The supply voltages needed for individual electrical components is generally 5V or 3.3V.

Initially we intended to design a switch mode power supply. This design is presented shortly, however, implementation of this design would have required many additional components. The

9.1.2 µA7800 Series Voltage Regulator

In order to accommodate the operating voltage (Vcc) levels of the PIC16F628 of 5V, a voltage regulator was needed prevent damage to the PIC. This is acheved with the typical 7805 part. This voltage regulator is able to take the 12V that is coming from the head unit to the 5V needed for PIC operation. A set of signal smoothing capacitors could also be used but after analysis, we found they were not essential.

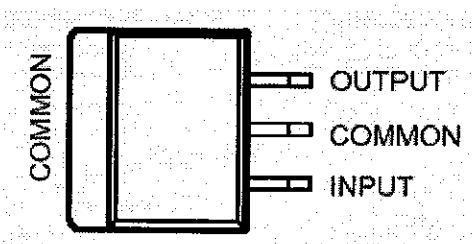


Figure 9-4: Pin Diagram of the µA7800 Series Positive-Voltage Regulator

9.1.2.1 µA7800 Series Voltage Regulator Specifications

This voltage regulator can deliver up to 1.5A of output current, more than enough to power our circuit. For a current this high, a heat sink would be required. The internal current-limiting features of this particular regulator prevent them from overloading. With this safety measure of the regulator rest assured that none of our circuitry will overload the voltage regulator. The recommended input voltage operating conditions of this voltage regulator ranges from 7V to 25V. Typically the output of this voltage regulator ranges from 4.8V to 5.2V.

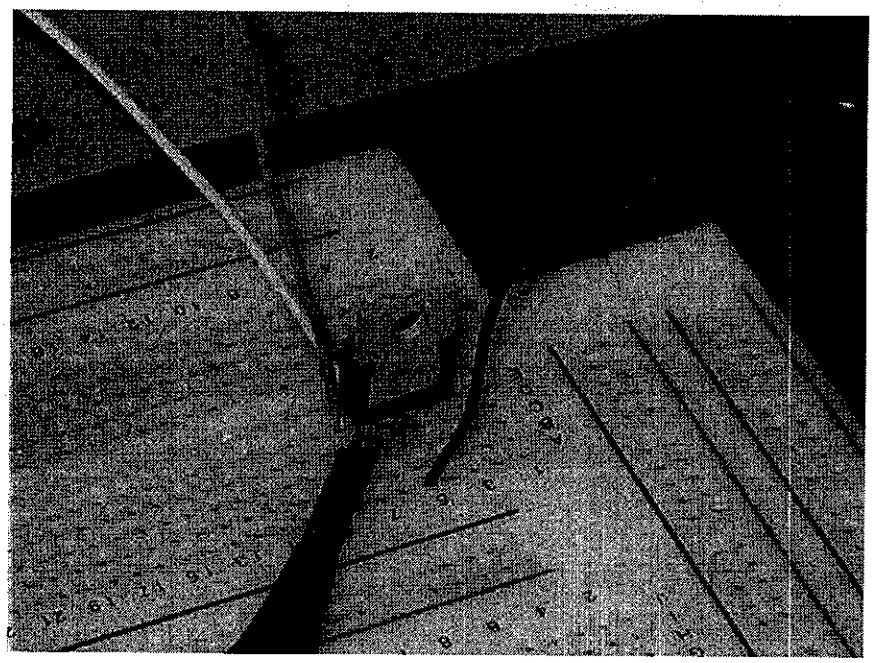


Figure 9-4: Picture of the µA7800 Series Positive-Voltage Regulator

9.1.2.2 Step-Down DC-DC Controller

These voltages are achieved through a step-down DC-DC controller. Maxim's MAX1626 controls the switch mode power supply. This chip offers an extremely high efficiency rating for our device. The efficiency of greater than 90% aids in low power consumption required by our application. Maxim rates the supply voltage (V+) for the MAX1626 between +3v (-0.3v lower limit) and +17v. Our operating voltage falls within these limits. The details on the MAX1626 usage are found in the datasheet that gives all details of design for the regulated power supply.

9.1.2.3 MAX1626 Detailed Description

The MAX1626 are step-down DC-DC controllers designed primarily for use in portable computers and battery-powered devices. Using an external MOSFET and current-sense resistor allows design flexibility and the improved efficiencies associated with high-performance P-channel MOSFETs. A unique, current-limited, pulse-frequency-modulated (PFM) control scheme gives these devices excellent efficiency over load ranges up to three decades, while drawing around 90µA under no load. This wide dynamic range optimizes the MAX1626 for battery-powered applications, where load currents can vary considerably as individual circuit blocks are turned on and off to conserve energy. Operation to a 100% duty cycle allows the lowest possible dropout voltage, extending battery life. High switching frequencies and a simple circuit topology minimize PC board area and component costs. Figure 1 shows a typical operating circuit for the MAX1626.

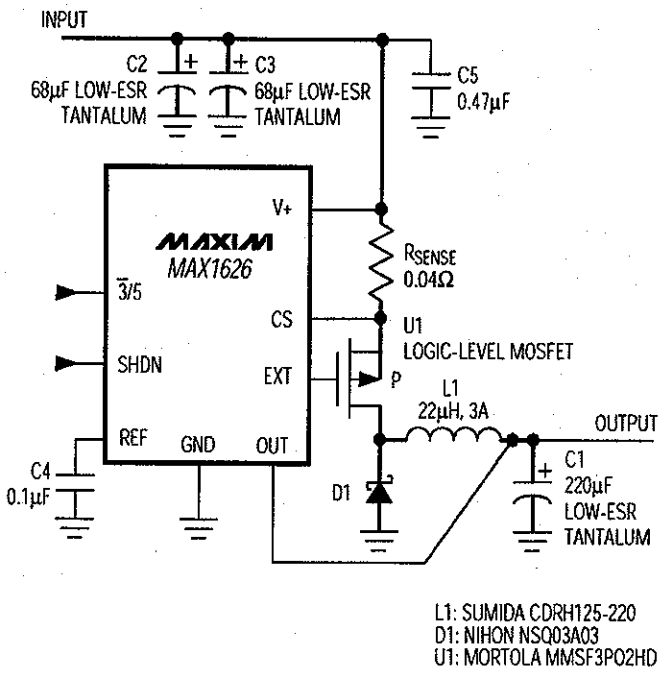


Figure 9-4: MAX1626 Typical Operating Circuit

9.1.2.4 Setting the Output Voltage

The MAX1626's output voltage can be selected to 3.3V or 5V by using the 3/5 pin. The output voltage can be set to and exact value, besides 3V or 5V, by using two resistors in a voltage divider set-up. The specifics of this are described in the MAX1626 datasheet and will be omitted here since our application requires 5V output. In order to achieve this 5V output we will connect the 3/5 inputs to Vcc. If 3.3V were desired we would connect the input to GND.

9.1.2.5 External Switching Transistor

The MAX1626 drive P-channel enhancement mode MOSFETs. The EXT output swings from GND to the voltage at V+. To ensure the MOSFET is fully on, use logic-level or low-threshold MOSFETs when the input voltage is less than 8V. The datasheet lists recommended suppliers of switching transistors. The four important parameters for selecting a P-channel MOSFET are drain-to-source breakdown voltage, current rating, total gate charge (Qg), and $R_{DS(ON)}$. The drain-to-source breakdown voltage rating should be at least a few volts higher than V+. Choose a MOSFET with a maximum continuous drain current rating higher than the peak current limit, which is also described in the datasheet.

9.1.2.6 Why Use a Switching Regulator?

For battery management, the only other choice is a linear regulator. Linear regulators only step down, and efficiency is equivalent to the output voltage divided by the input voltage. On the other hand, switching regulators operate by passing energy in discrete packets over a low-resistance switch, so they can step up, step down, and invert. In addition, they offer higher efficiency than linear regulators. Using a transformer as the energy-storage element also allows the output voltage to be electrically isolated from the input voltage.

The one disadvantage of the switching regulator is noise. Any time you move charge indiscrete packets, you create noise or ripple. But the noise can often be minimized using specific control techniques and through careful component selection.

9.1.3 Atmel AVR CPU

In order to drive all of the main functions of the MP3 CD Changer Emulator we will be using one of Atmel's AVR 8-bit RISC microprocessors. This central part of the device will command all functions of MP3 play, memory access, execution of commands relayed from the PIC module and more. The Atmel AVR chips support self-programming and in system programming as well. This makes programming the chip's Flash memory easier and makes it possible to program through USB or serial with no need to remove the chip.

9.1.3.1 Atmel AT90S8515

The AT90S8515 is part of the 8-bit AVR family from Atmel. This chip is very common . It is so common, radioshack.com sells them. The chip became available to us for free as part of the STK500 developers board. This was ideal because documentation was available for the chip. Although, Atmel is very well documented.

The AVR uses low-power CMOS architecture making it ideal for our application. Since it is CMOS, ESD (Electrostatic Discharge) precautions must be taken into consideration. This will be taken into consideration in both chip handling and the device's casing. The ATMega161 provides the following features: 6K bytes of In-System or Self-programmable Flash, 512 bytes EEPROM, 1K byte of SRAM, 35 general-purpose I/O lines, 32 general purpose working registers, Real-time Counter, 3 flexible timer/counters with compare modes, internal and external interrupts, 2 programmable serial UARTs, programmable Watchdog Timer with internal oscillator, an SPI serial port and 3 software-selectable power saving modes. All of these features influenced our decision for using this microprocessor. The internal Flash and EEPROM will be used to store all program information eliminating the need for using external PROM chips. This will simplify the design's size and power consumption.

9.1.3.2 Atmel ATMega161 Pin Descriptions

Each pin on the ATmega161 corresponds to particular task. As usual there is a supply voltage (Vcc) and Ground pin. The other basic pins include the RESET, XTAL1 and XTAL2. The RESET input is connected to Vcc through a current limiting resistor under normal operation. A hardware reset is generated when the RESET pin is sent low via a push button switch between the RESET pin and ground. The two external crystal pins, XTAL1 and XTAL2, are connected to a 7.3728 MHz crystal in the usual manner. This includes the crystal and two equal value capacitors. The crystal also determines what clock the source code is compiled for. The remainder of these pins corresponds to bits of the chips four ports. Each of the four ports is a bi-directional I/O port with internal pull-up resistors and have tri-state outputs. The basic functions of the ports are introduced in the datasheet summary shown in the appendix. The ports also have special functions that will be used by our device. Table 9-2 shows the specific functions of each pin.

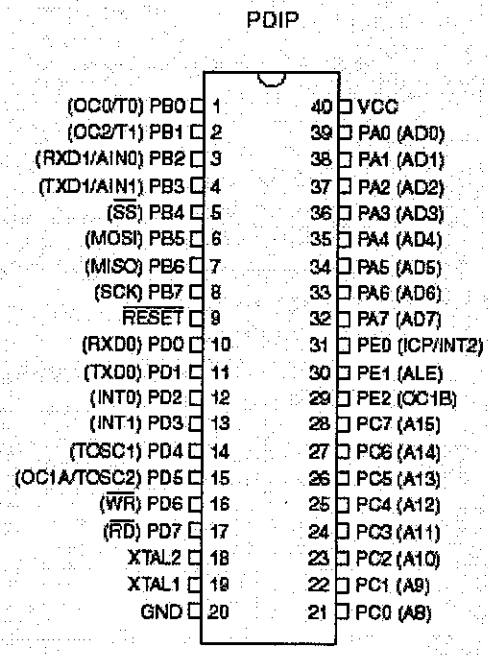


Figure 9-5: Pin Diagram for the Atmege161

Port A:
Port A serves as a Multiplexed Address/Data port when using external memory interface.
Port B:
Bit 7 – SCK, Master clock output for use with slave clock input for a SPI channel. Always functions as Input.
Bit 6 – MISO, Master Data Input. Connects to a slave data output pin for a SPI channel.
Bit 5 – MOSI, Master Data Output. Connects to a slave data input pin for a SPI channel.
Bit 4 – SS, Slave Select for SPI operation.
Bit 3 – TXD1/AN1 TXD1, Transmit Data. Data output for USART AN1, Analog Comparator Negative Input
Bit 2 – RXD1/AN0 RXD1, Receive Data, Data input for USART AN0, Analog Comparator Positive Input
Bit 1 – OC2/T1 T1, Timer/Counter1 OC2, Output Compare Match Output
Bit 0 – OC0/T0 T0, Timer/Counter0 OC0, Output Compare Match Output
Port C:
Port C serves as and address high output when using external memory
Port D:
Bit 7 – RD, External Memory Read Strobe
Bit 6 – WR, External Memory Write Strobe
Bit 5 – OC1, Output Compare
Bit 5 and 4 – TOSC1/TOSC2
Bit 3 – INT1, External Interrupt
Bit 2 – INT0, External Interrupt
Bit 1 – TXD0, USART TX
Bit 0 – RXD0, USART RX
Port E:
Bit 2 – OC1B, Output Compare
Bit 1 – ALE
Bit 0 – ICP/INT2 ICP, Input Capture Pin INT2, External Interrupt

Table 9-2: Atmega161 Pin Descriptions

9.1.3.3 Using Special Port Functions

As seen in Table 9-2, each port has a special function. Since the VS1001 uses SPI to communicate, we will use the Special Functions of Port B to interface with it. The data sheet for the VS1001 will be referred to for proper pin connections. Since Port A and C are used for addressing external memory, they will be used together to access the 16 bit memory structure of the ATA hard disk drive. Port C will be used to address the high order byte as specified in Table 9-2.

9.1.3.4 Atmel AVR Instruction Set

Available on the Atmel web site is a complete instruction set reference. Although this document is greater than 150 pages and omitted from this paper, it has been printed in its entirety for use while programming the Atmel AVR microchip. This reference will be used heavily during program development.

9.1.4 Universal Serial Bus

Initially we wanted to be able to upload the songs with a USB interface. Although this would have been nice and easy for the end-user we ended up not implementing this option.

In order to improve ease of use, the MP3 CD changer emulator will implement a Universal Serial Bus (USB) interface as an optional feature. This would allow the device to be connected to a personal computer for the high-speed transfer of MP3 music files. The device will be connected to the PC through the USB port and seen as a regular hard disk drive. This should permit an easy drag and drop update of music files. If needed we will program a stand alone application for the PC using programming libraries and Dynamic Link Libraries (DLLs) provided by the USB chip manufacturer. This eliminates the need to program complicated USB drivers. Also, it would be possible for the CPU to be reprogrammed via the USB port using a boot loader algorithm. This would permit easy device updates without removing the device from its case. If a stand-alone PC application is fabricated, the firmware programming will be updateable using the application and USB connection.

RD# (*input*): When pulled low, RD# takes the 8 data lines from a high impedance state to the current byte in the FIFO's receive buffer. Taking RD# high returns the data pins to a high impedance state and prepares the next byte (if available) in the FIFO to be read.
WR (*input*): When taken from a high state to a low state, WR# reads the 8 data lines and writes the byte into the FIFO's transmit buffer. Data written to the transmit buffer is immediately sent to the host PC and placed in the RS-232 buffer opened by the application program.
TXE# (*output*): When high, the FIFO's 384-byte transmit buffer is full or busy storing the last byte written. Do not attempt to write data to the transmit buffer when TXE# is high.
RXF# (*output*): When low, at least 1 byte is present in the FIFO's 128-byte receive buffer and is ready to be read with RD#. RXF# goes high when the receive buffer is empty.

9.1.4.1 Personal Computer Configuration

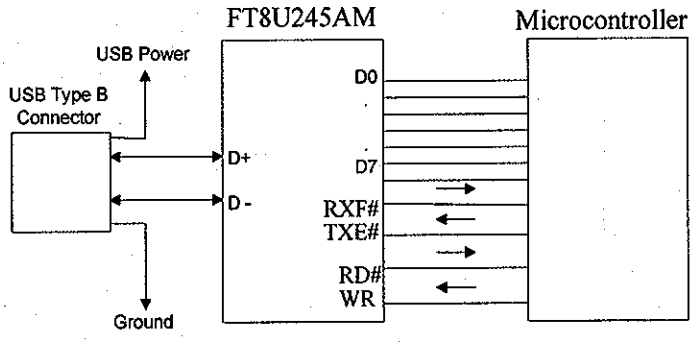
By using FTDI's virtual COM Port drivers, the peripheral looks like a standard COM Port to the application software. The Commands to set the baud rate are ignored because the device always transfers data at its fastest rate regardless of the application's baud rate setting. All drivers from FTDI are royalty free and can be used without cost. There are also DLL libraries available for use. We are not positive if this will allow us to see the hard drive on our device as a simple removable hard disk drive. This will be revealed to us with more experimentation with our prototype.

9.1.4.2 Internal USB Descriptor

The FT8U245AM is designed to interface directly with an EEPROM (specifically the 93C46) for storing configurable parameters that include the PID (Product ID), VID (Vendor ID), device description and manufacturer name. With the FT8U245AM you can use your own VID and PID or just use FTDI's. The VID and other configurable parameters can be written to the EEPROM using the program 232PROG.EXE which can be downloaded from FTDI's website. This program will also generate a serial number and write it to the EEPROM granted registration. If we planned to commercialize this project as a product and use a USB port, we would be required to register our own PID and VID with the USB-IF. This registration is included with the $2,500 membership fee, or you can become a non-member USB-IF Logo Licensee for $1,500. An example source code for defining the device descriptor is given in the appendix of the FT8U245AM datasheet.

9.1.4.3 FTDI USB Chip

A recent article in Nuts and Volts Magazine (May 2001 Issue) discussed the ease of using a USB chip from FTDI Chip. This chip, the FT8U245AM, provides an easy cost-effective method of transferring data to and from our device and a host PC at a rapid rate. The chip rates data transfer rates up to 8 Million bits (1 Megabyte) per second. This would definitely speed up the transfer of MP3 files. It uses a FIFO (First In First Out) design making it easy to interface to any CPU either by mapping the USB Chip into the Memory/IO map of the CPU, using DMA or controlling the device via IO ports. The FT8U245AM chip offers us a simple USB solution for our project. With this chip there is no need to worry about the underlying USB protocols and no time wasted debugging device drivers and firmware.



9.1.5 STA013 MP3 Decoder Chip

The STA013 MP3 Decoder Chip was our first choice for MP3 decoding until we discovered a one-chip solution that included a Digital to Analog Converter (DAC). Although the STA013 suited our needs it would have needed a separate DAC to function. With the VS1001 we opted for the single-chip solution. For more information about the STA013 please refer to the section labeled STA013 MP3 Decoder.

9.2 Hard Disk Drive Connectivity

We use a Hard Disk Drive as our MP3 data source. We could use either a typical 3.5 inch drive, but we desire to use the smaller 2.5 inch laptop hard disk drive. This drive will not only be a better choice for shock resistance within the automobile but will also consume less power. The laptop hard drive will operate at off a 5 volt supply therefore will not require an additional power supply. The laptop drive will also give us the benefit of using a single ribbon cable for both data and power. This helps keep the size of the device low. Unfortuanately, the laptop hard drive option failed as discussed later.

9.2.1 ATA Specification

All specific details of the ATA hard disk connection can be found online within the ATA/ATAPI Specification. This is a working document that will be updated or revised as needed. Since the specification is over 300 pages in length, it will not be included in its entirety. The ATA connector pin-out will be discussed briefly.

9.2.2 Connecting to IDE Drive

A 44-pin 2mm IDE connector will interface directly with a 2.5 inch lap-top hard disk drive. To access the 16-bit IDE interface, Port A and Port C are used for accessing the 16-bit IDE data. Using the pin-out for the connector from within the datasheet (shown in Figure) Port A and Port C will be connected to the low and high order bytes of the hard drive respectively.

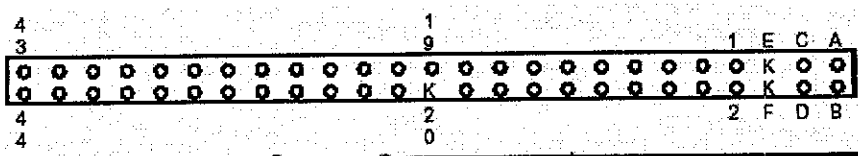
9.2.3 Powering the Hard Drive

One huge benefit of using a 2.5 inch laptop hard disk drive is that the drive is powered through the same connector and cable. These power connections are shown through the following pin-out diagram. The laptop drive has 44 pins while the conventional IDE drive has 40 pins and a 4 wire power connector. The laptop drive eliminates this power connector which simplifies the design of the connection. Using connector pins 41 thru 44 the power is connected for the laptop drives. Since the laptop hard drive was unsuccessful, we will power the hard drive using a split (12V/5V) power supply for demo purposes.

9.2.4 Laptop 2.5" Hard Drive

Initially the decision to use a laptop hard drive was based on the portability and the protection included with that type of device. We ended up using a regular 3.5" IDE hard drive since our available laptop hard drives failed.

This drive will not only be a better choice for shock resistance within the automobile but will also consume less power. The laptop hard drive will operate at off a 5-volt supply therefore will not require an additional power supply. The laptop drive will also give us the benefit of using a single ribbon cable for both data and power. This helps keep the size of the device low.



Signal name	Connector contact	Conductor		Connector contact	Signal name
Vendor specific (see note)	A			B	Vendor specific (see note)
Vendor specific (see note)	C			D	Vendor specific (see note)
(keypin) (see note)	E			F	(keypin) (see note)
RESET-	1	1	2	2	Ground
DD7	3	3	4	4	DD8
DD6	5	5	6	6	DD9
DD5	7	7	8	8	DD10
DD4	9	9	10	10	DD11
DD3	11	11	12	12	DD12
DD2	13	13	14	14	DD13
DD1	15	15	16	16	DD14
DD0	17	17	18	18	DD15
Ground	19	19	20	20	(keypin)
DMARQ	21	21	22	22	Ground
DIOW-:STOP	23	23	24	24	Ground
DIOR-:HDMARDY-:HSTROBE	25	25	26	26	Ground
IORDY:DDMARDY-:DSTROBE	27	27	28	28	CSEL
DMACK-	29	29	30	30	Ground
INTRQ	31	31	32	32	reserved
DA1	33	33	34	34	PDIAG-
DA0	35	35	36	36	DA2
CS0-	37	37	38	38	CS1-
DASP-	39	39	40	40	Ground
+5 V (logic) (see note)	41	41	42	42	+5 V (Motor) (see note)
Ground (return) (see note)	43	43	44	44	Reserved - no connection (see note)
NOTE – Pins that are additional to those of the 40-pin cable.					

Table 9-3: Laptop HDD Pin Diagram

9.3 Final Product Design

We would have liked to have a completely milled PCB available for demonstration. This would show the portablilty and self containment of the system. No additional power supply would be needed since the power supplied through the m-bus cable would be utilized. Unfortunately we were only able to show the wire wrap version and an additional power supply was needed since our laptop hard drives failed. The final product design differs a bit from the initial proposed design. Below is the completed mp3 player ready to interface with the PIC module.

Figure 9-6: Final Product Design

9.3.1 Hardware

This section will describe the many different hardware components used to develop the MP3 Hard Drive player.

9.3.1.1 MAXIM RS-232 Transceivers – MAX202

The Maxim transceivers are designed for RS-232 communications interfaces where ± 12V supplies are not available. The maximum rating for the MAX202 ranges from -0.3V to +6V. The MAX202 consists of three sections charge-pump voltage converters, drivers (transmitters), and receivers. The typical operating circuit is shown below. This is the exact schematic provided in the Atmel STK500 AVR developers board for development purposes. For the final device, this would be implemented the same way. For the demonstration, we will use the circuit built onto the STK500.

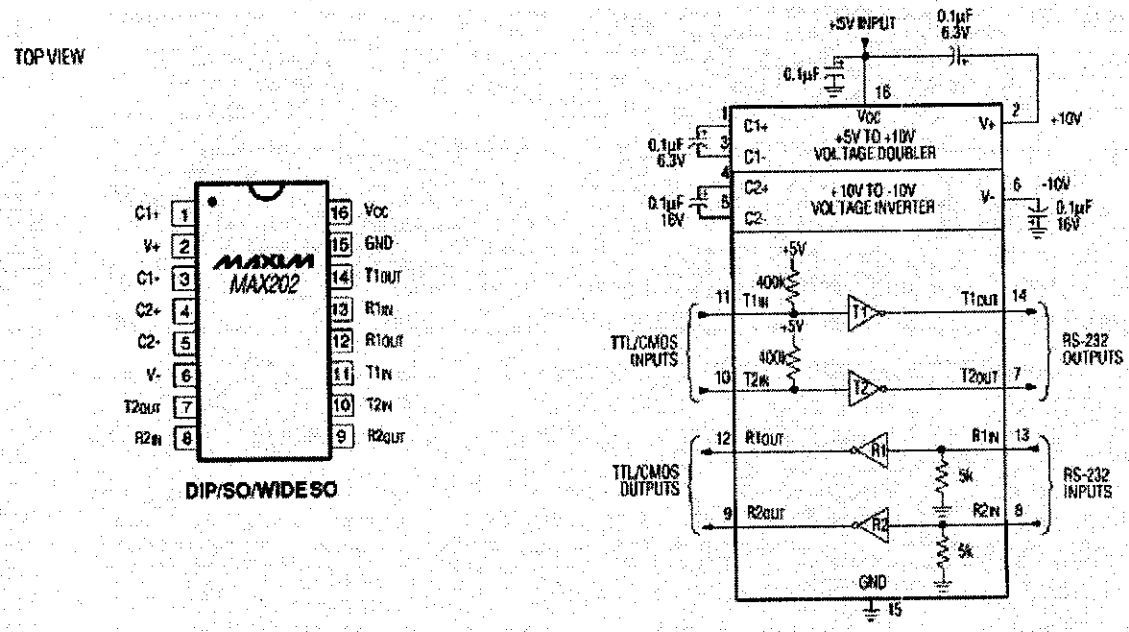


Figure 9-7: Pin Diagram and connection schematic for Maxim MAX202

PIN	CONNECTION	
1	Received Line Signal Detector, sometimes called Carrier Detect (DCD)	Handshake from DCE
2	Receive Data (RD)	Data from DCE
3	Transmit Data (TD)	Data from DTE
4	Data Terminal Ready	Handshake from DTE
5	Signal Ground	Reference point for signals
6	Data Set Ready (DSR)	Handshake from DCE
7	Request to Send (RTS)	Handshake from DTE
8	Clear to Send (CTS)	Handshake from DCE
9	Ring Indicator	Handshake from DCE

Table 9-4: DB9 RS-232 Pin Connections for the MAX202

9.3.1.2 Atmel AVR STK500

The STK500 is a starter kit for the Atmel's AVR Flash microcontrollers. We used this kit as a tool in rapid prototyping. This board interfaced directly with the hard drive interfacing circuitry and the mp3 decoder board. It is apparent in the figure below that we were able to interface to the on board port headers using ribbon cables and jumper wires.

The STK500 kit includes the following:

- AVR Studio Software Interface
- RS-232 for programming and configuration
- Parallel and serial high-voltage programming of AVR devices
- I/O Ports accessible through pin header connections
- RS-232 port for general use
- Flexible clocking, voltage, and reset system
- Supports In-System programming of all AVR microcontrollers
- Can program external target systems through sockets

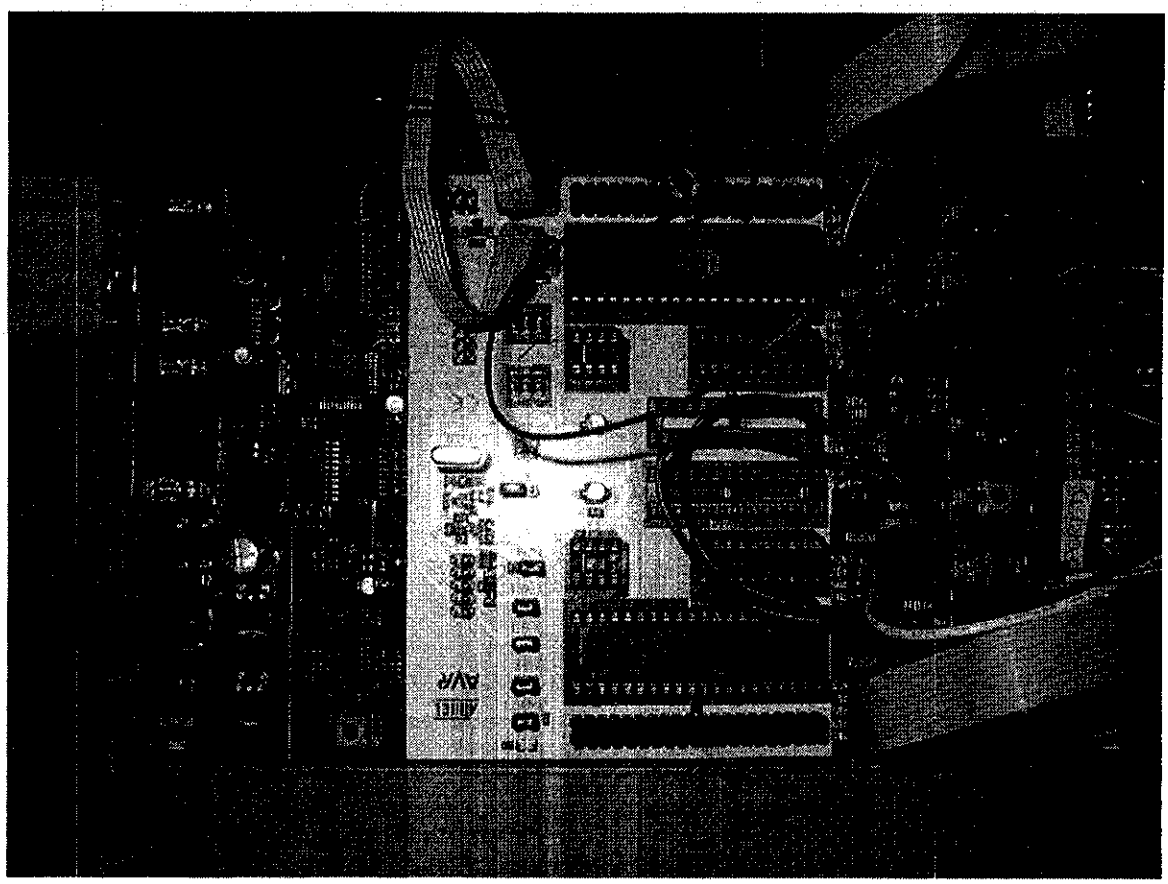


Figure 9-8: Picture of the STK500 Atmel AVR Developmental Board

9.3.1.3 IDE 3.5" Hard Drive

When trying to develop the MP3 player we initially started using a laptop 2.5" hard drive. Trying to interface with the device the laptop hard drive failed due to malfunctioning drives. In trying to eliminate the cause of the problem we tried another laptop hard drive and that failed as well. We then decided to use a regular desktop IDE Hard Drive for the product and it has not failed and works well with our project. We used a total of 5 hard drives during the project since the previous 4 drives failed during use, and misuse.

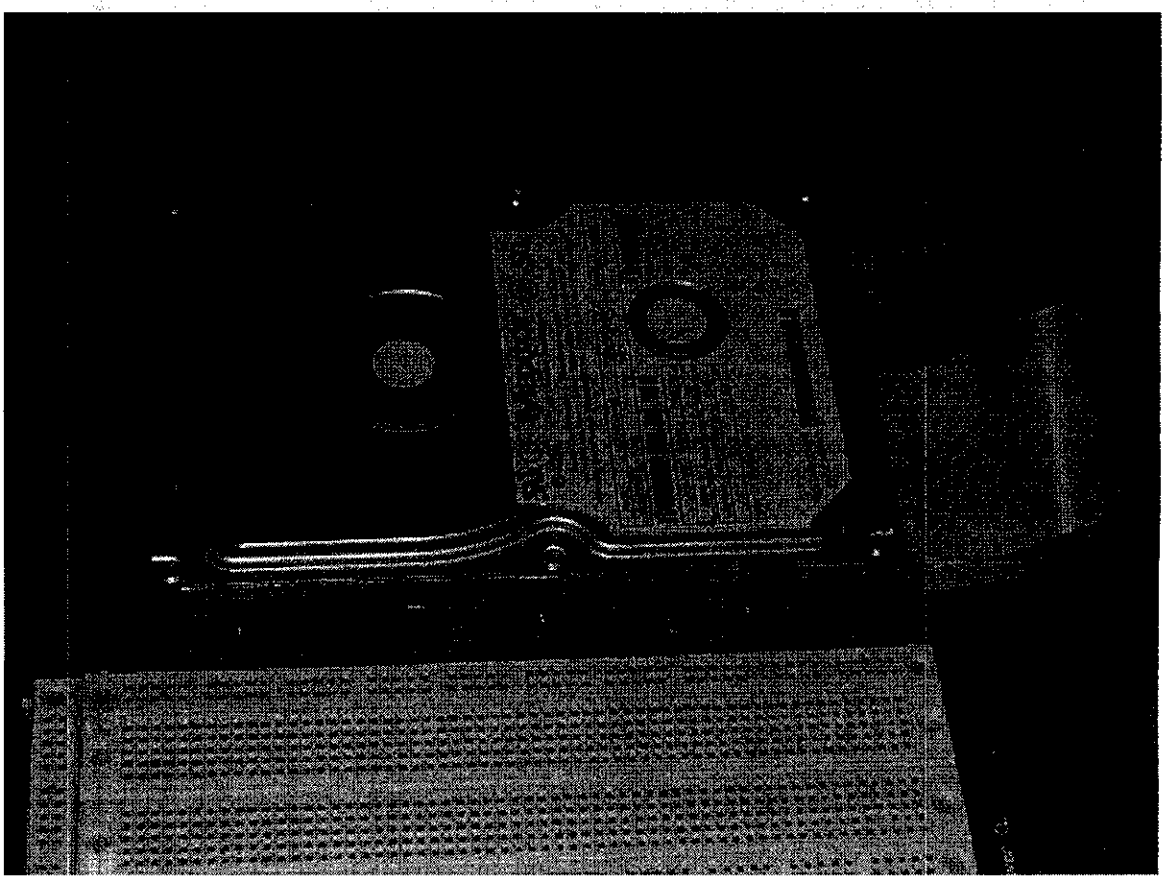


Figure 9-9: IDE 3.5" Hard Drive used for the Atmel MP3 player

9.3.1.4 Split 12V/5V Computer Power Supply

Since we suffered the loss of 4 hard drives, we decided to use a 200W power supply for easier implementation for demonstration and the signal remained a constant signal. This supply would not be used in actual implementation at all. Our production model would convert the 12V signal coming from the headunit to power the complete system, this would require no additional supply. A laptop hard drive would illuminate the need for a dual mode supply

Figure 9-10: Picture of the MP3 Hard Drive Player and 200W Power Supply

9.3.1.5 Wire Wrapping

Most of the developmental board for the MP3 player is wire wrapped. The final product is wire wrapped as well. A special tool called a "wire-wrap" tool has two holes. The wire and a quarter-inch of insulated wire are placed in a hole hear the edge of the tool. The hole in the center of the tool is placed over the post. The tool is rapidly twisted. The result is that 1.5 to 2 turns of insulated wire are wrapped around the post, and atop that, 7 to 9 turns of bare wire are wrapped around the post. The post has room for three such connections, although usually only two are needed. This permits manual wire wrapping to be used for repairs.

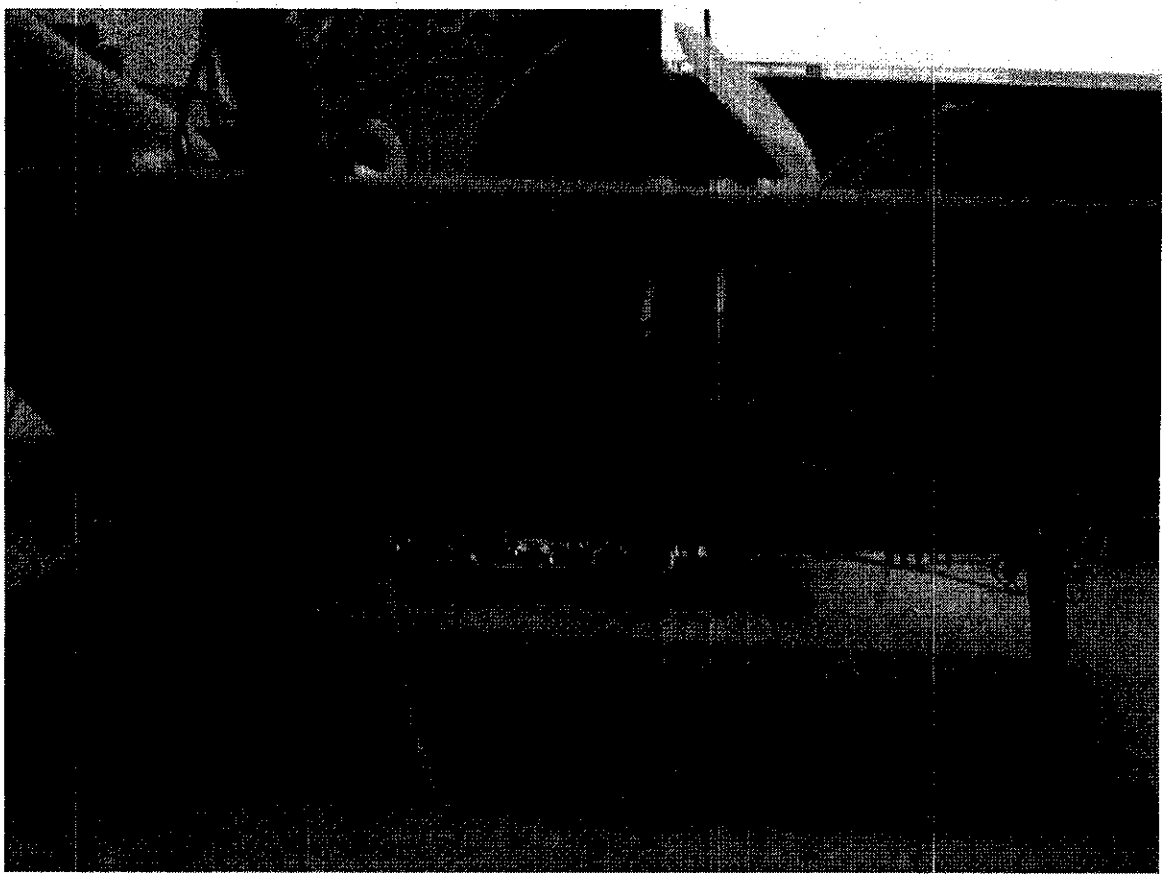


Figure 9-11: Picture of Wire Wrapping

9.3.1.6 PCB Board Milling

We did mill some PCB boards for our project. The boards mainly consisted of multiple custom designed wire wrap adaptors. We also attempted to mill a complete board with many difficulties. Since there are no plate thru holes, all via holes and thru hole parts must be soldered on both sides making a connection thru the board. During this procedure, many pads and traces lifted. Time restraints made it impossible to perfect a usable board before demonstration. The board design was completed using Cadsoft Eagle. The final board design is shown on the following pages. Daniel milled all of the boards for the project. Below are some pictures of one of the boards that he milled.

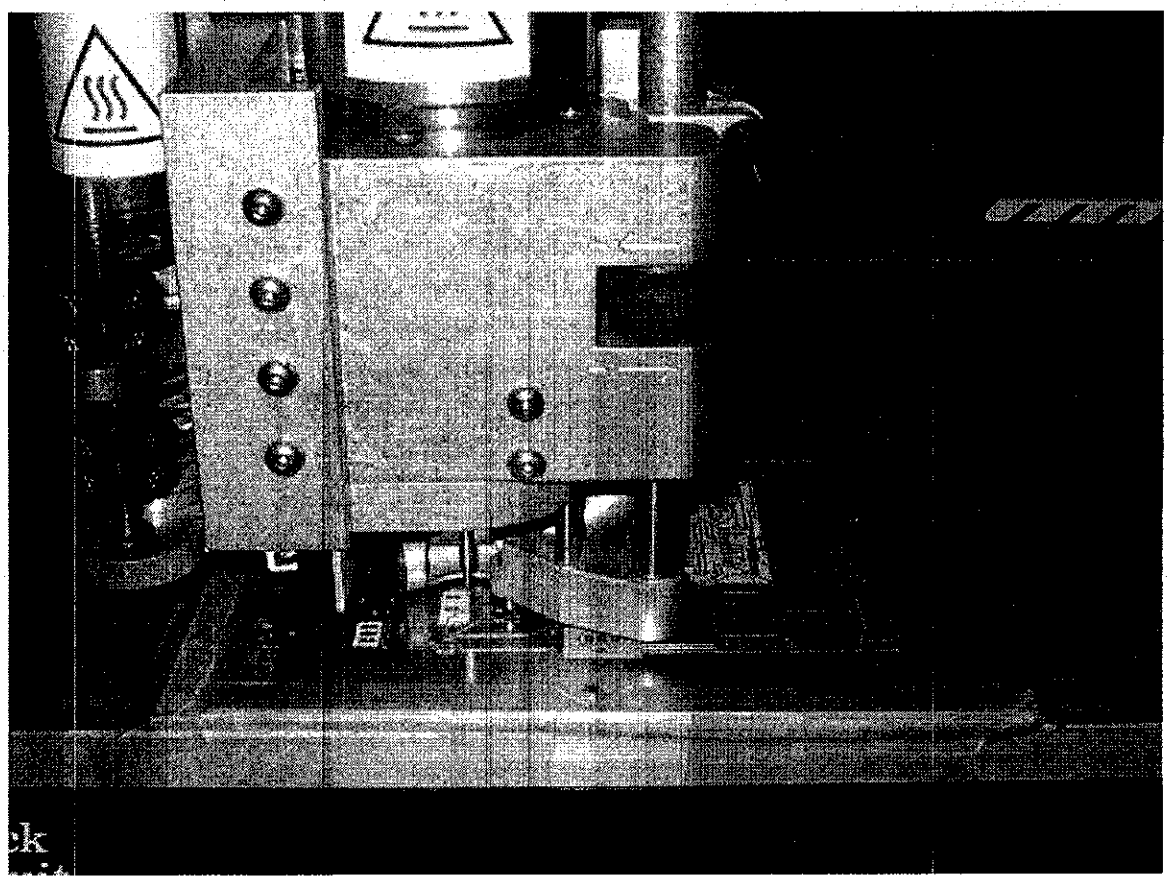


Figure 9-12: Milling one of our PCB Boards for the project

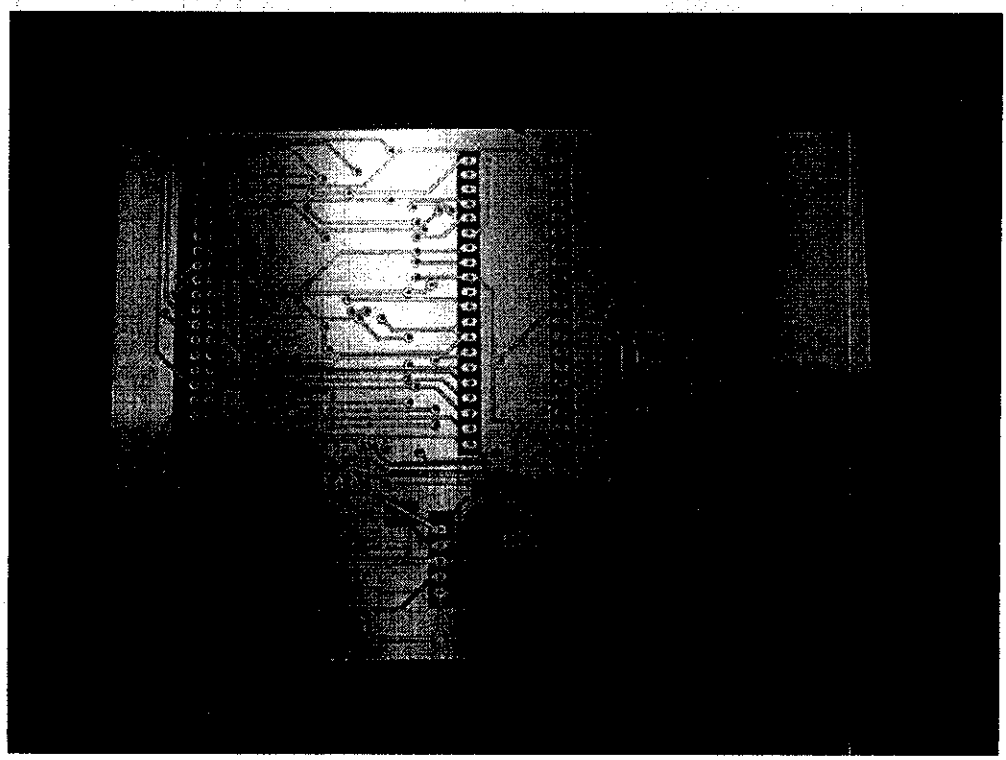
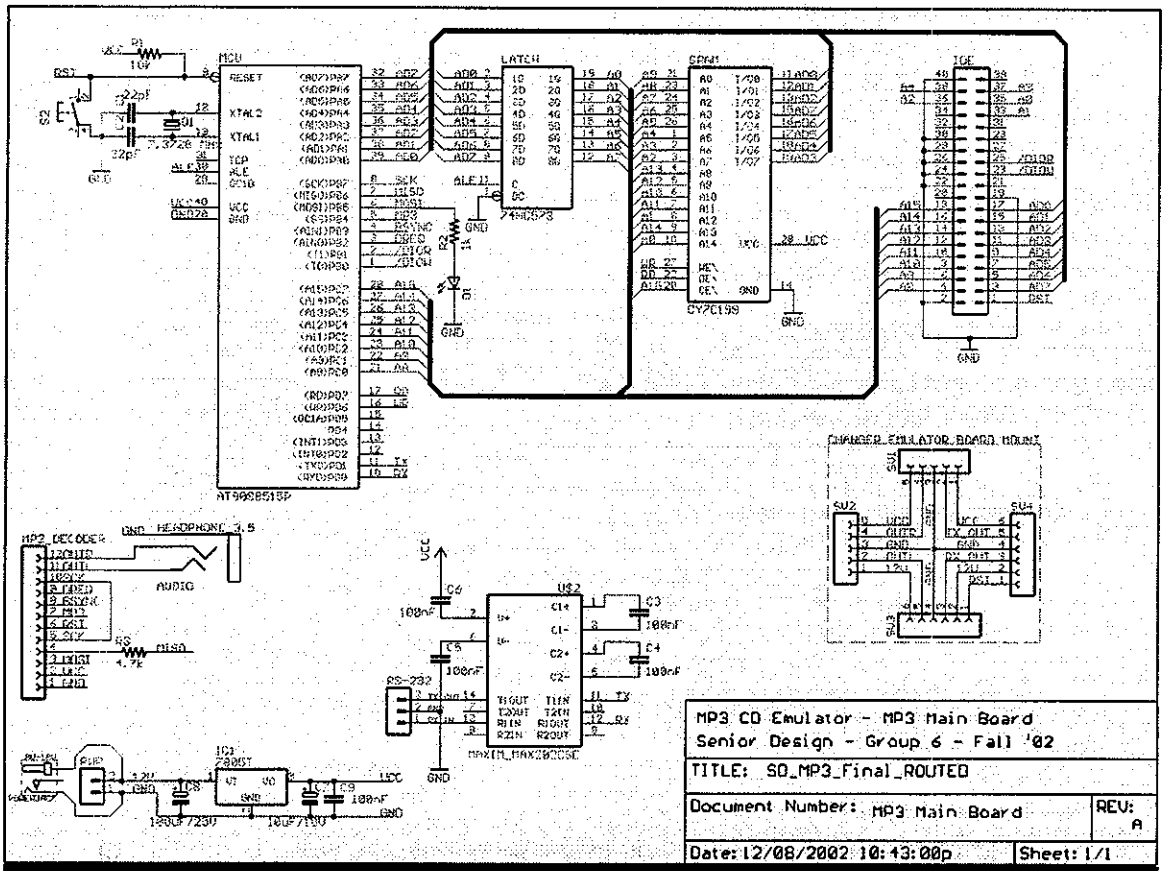
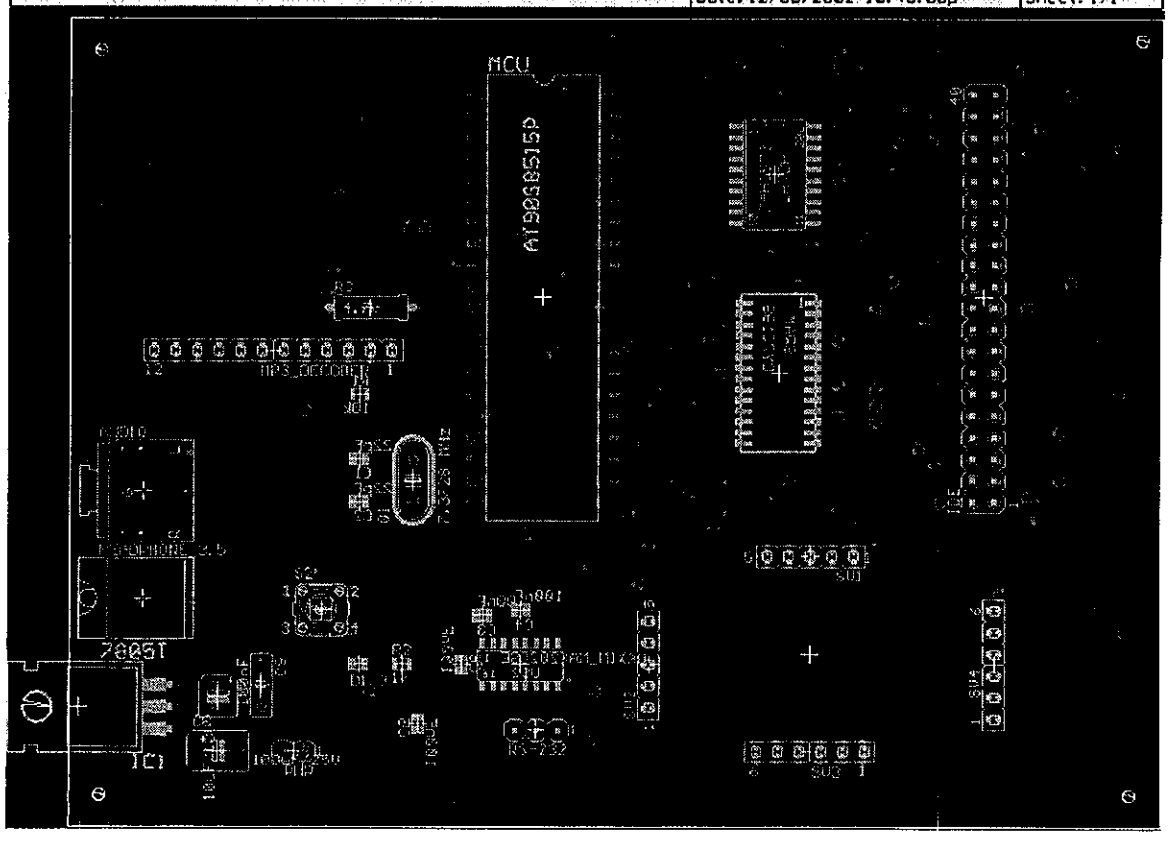


Figure 9-13: The PCB Board after the milling process

Figure 9-14: The Nice UCF Pegasus Logo on our PCB Board


MCU
LATCH
CDINT
IDE
CHANGER EMULATOR BOARD MOUNT
MP2_DECODER
HEADPHONE_3.5
AUDIO
RS-232
MAXIM_MAX202CSE
7805T
MP3 CD Emulator - MP3 Main Board
Senior Design - Group 6 - Fall '02
TITLE: SD_MP3_Final_ROUTED
Document Number: MP3 Main Board
REV: A
Date: 12/08/2002 10:43:00p
Sheet: 1/1


MCU
AT90S8515P
MP3_DECODER
HEADPHONE_3.5
7805T
RS-232

9.3.1.7 Connecting an IDE Hard Drive to a Microcontroller

From a controller point of view an IDE interface could be described as a set of I/O ports. The IDE interface has a 8/16 bits I/O bus, two /CS lines, a /WR and /RD line, three address bits and one interrupt. In this description I assume the most traditional IDE interface.

When implementing a 16-bits I/O port all you need is a bi-directional I/O port and some control bits to generate the /RD, /WR, and the other signals required.

The following pictures show the printed circuit board milled for the project. The purpose of this board was to allow us to wire wrap a laptop hard drive header. The pins on top were later used to stack a conventional PC hard drive to the top.

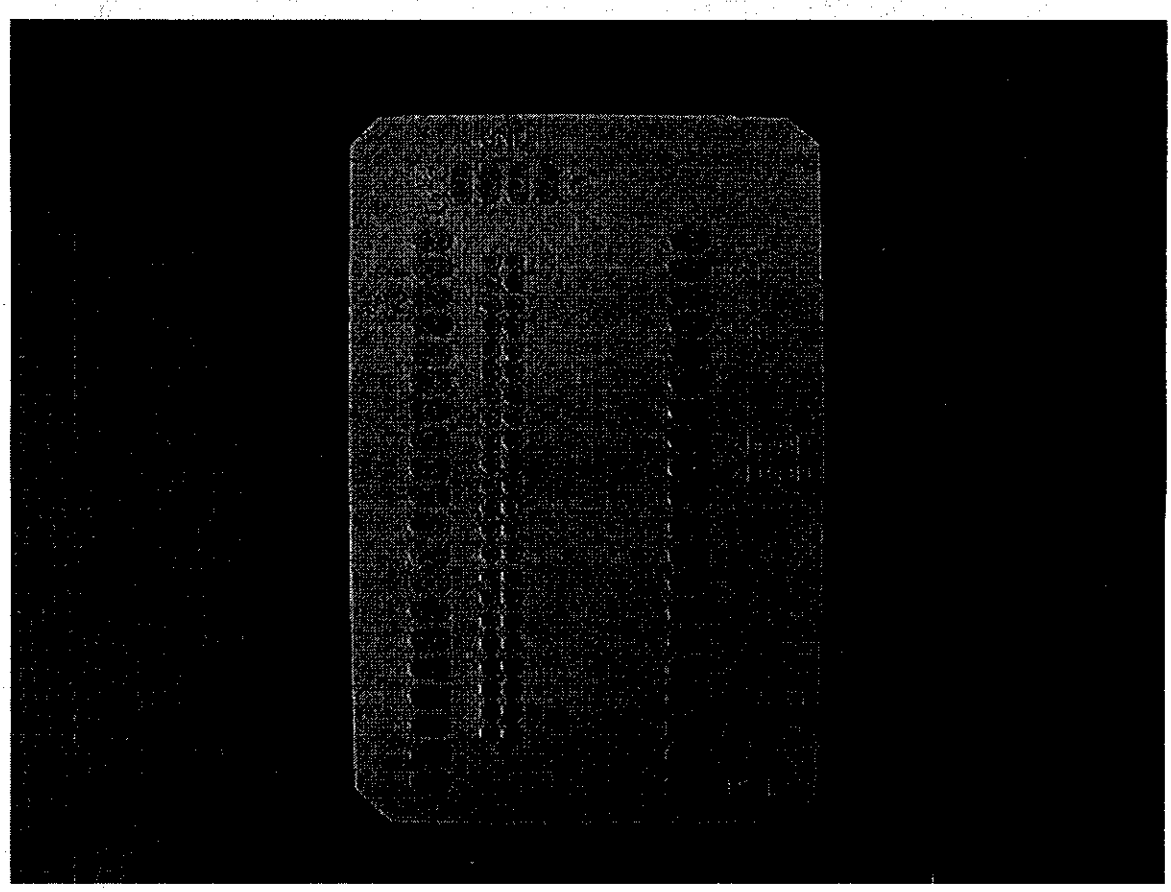


Figure 9-15: IDE Connector Interface (Front)

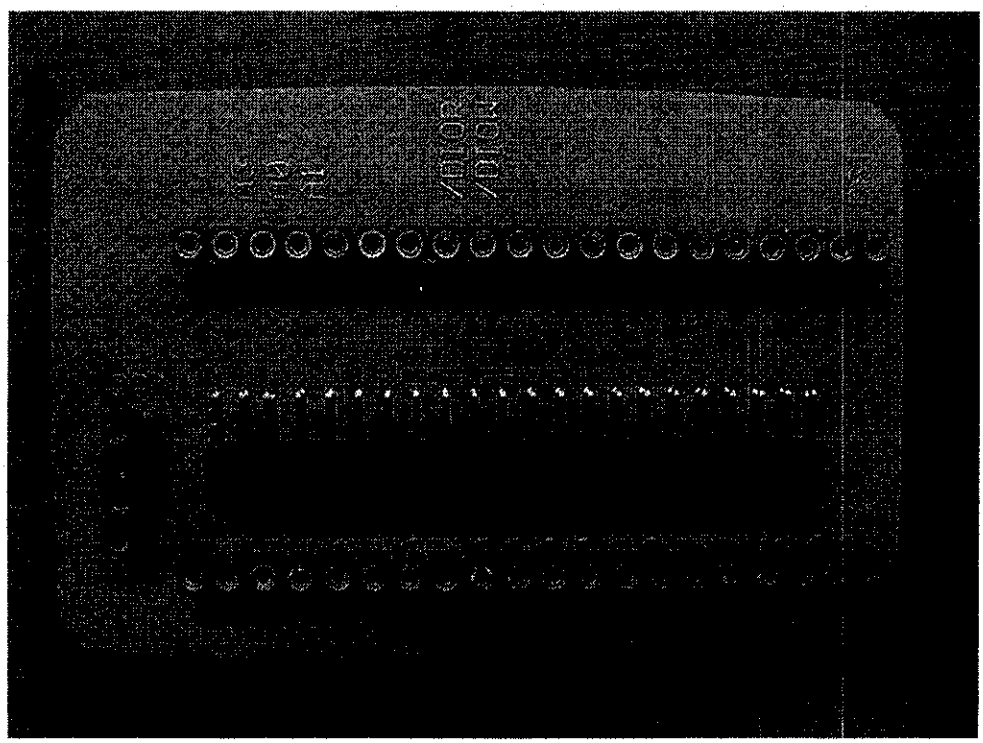


Figure 9-16: Pins Connected onto the IDE Interface (Front)

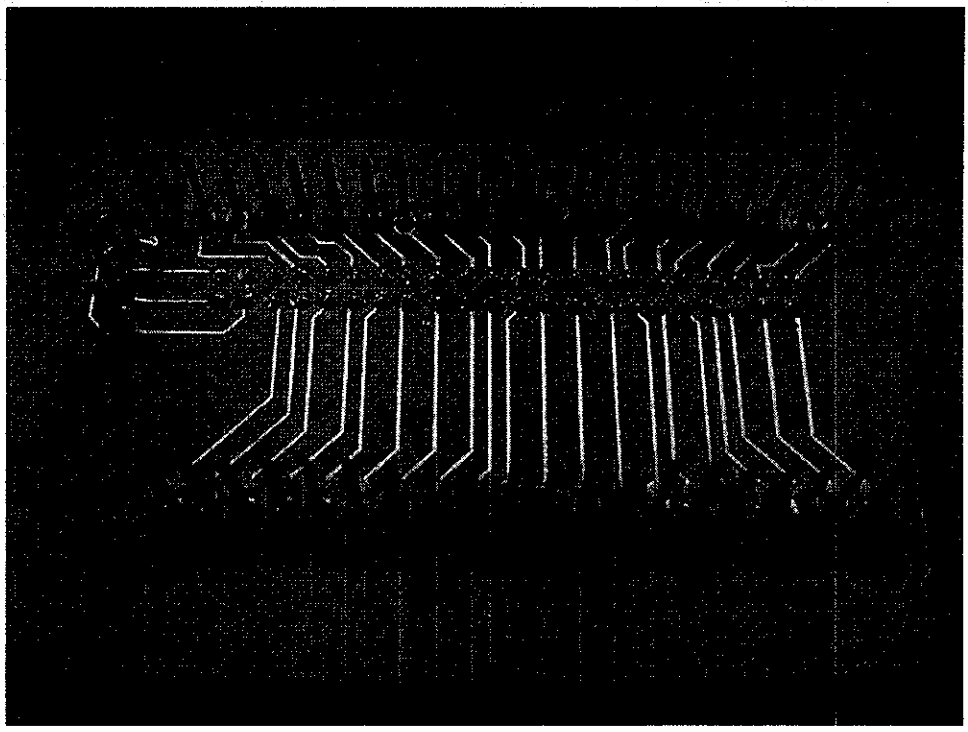


Figure 9-17: Pins Connected onto the IDE Interface (Back)

ATA Hard Drive connector pin descriptions

Pin Number	Name	Function
1	/RESET	Allow signal level on this pin will reset all connected devices
2, 19, 22, 24, 26, 30, 40	GND	Ground, interconnect them all and tie to controller's ground signal
3,5,7,9,11,13,15,17	D7...D0	Low data bus, 3=D7 ... 17=D0. This part of the bus is used for the command and parameter transfer. It is also used for the low byte in 16-bits data transfers.
4,6,8,10, 12,14,16,18	D8...D15	High data bus, 4=D8 ... 18=D15. This part of the bus is used only for the 16-bits data transfer.
20	-	This pin is usually missing. It is used to prevent mis-connecting the IDE cable.
21, 27	/IOREADY	To slow down a controller when it is going too fast for the bus
23	/WR	Write strobe of the bus.
25	/RD	Read strobe of the bus.
28	ALE	Some relic from the XT time.
31	IRQ	Interrupt output from the IDE devices.
32	IO16	Used in an AT interface to enable the upper data bus drivers.
34	/PDIAG	Master/slave interface on the IDE bus itself.
35	A0	Addresses of the IDE bus. With these you can select which register of the IDE devices you want to communicate.
33	A1	
36	A2	
37	/CS0	The two /CS signals of the IDE bus. Used in combination with the A0 ... A2 to select the register on the IDE device to communicate with.
39	/ACT	A low level on this pin indicates that the IDE device is busy.

Table 9-5: IDE Pin Detailed Description

The signals:
A0, A1, A2, /CS0, /CS1, /WR, /RD and /RESET are always outputs from the controller to the IDE bus.

The signals:
IRQ and /ACT are always outputs from the IDE bus to the controller (IRQ can be tri-stated by the IDE device when two devices are connected to the IDE bus.) /ACT can drive a LED (with resistor of course).

The signals:
D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15 are bi-directional. They are output from the controller to the IDE bus when writing, output from the IDE device to the controller when reading information.

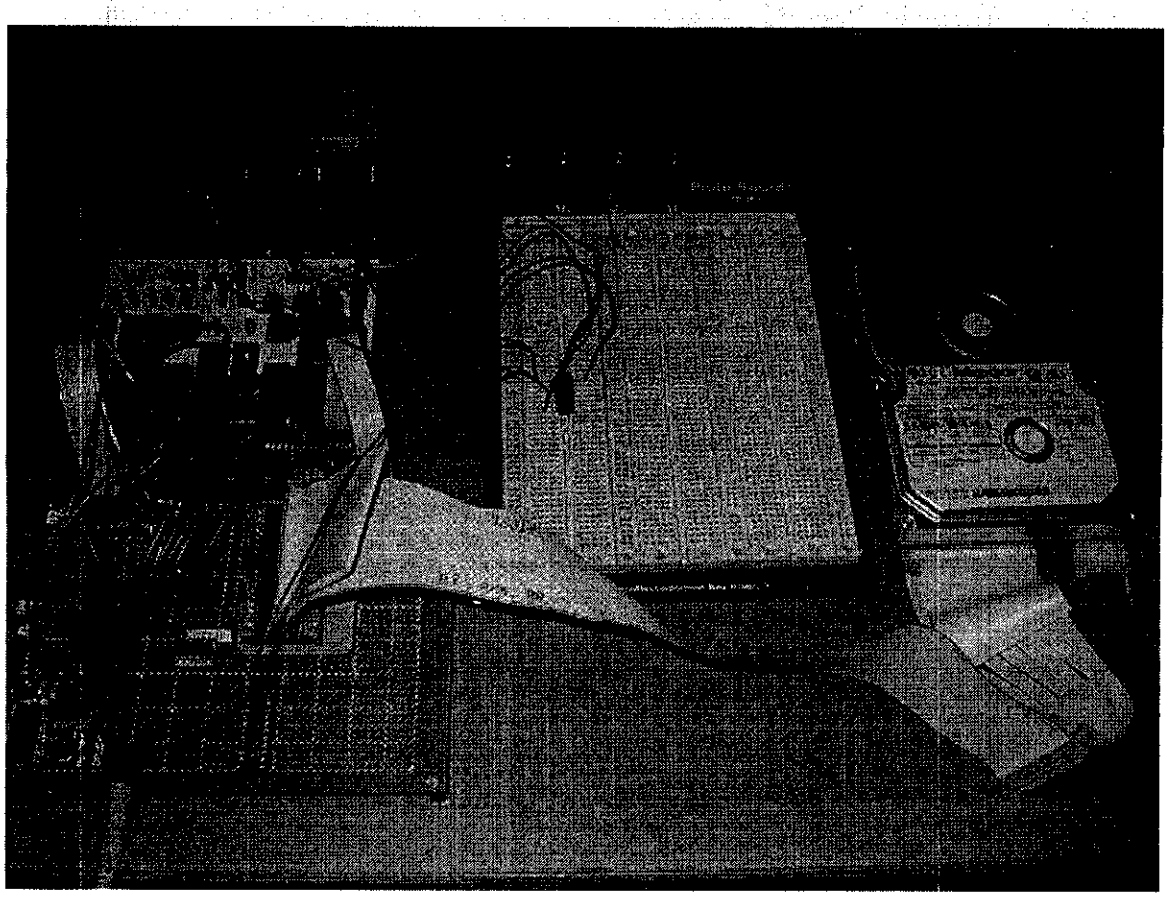


Figure 9-18: IDE Cable going from the Developmental Board to the Hard Drive

9.3.1.7.1 Status Register

Both the primary and secondary status register use the same bit coding. The register is a read register.

- bit 0: error bit. If this bit is set then an error has occurred while executing the latest command. The error status itself is to be found in the error register.
- bit 1: index pulse. Each revolution of the disk this bit is pulsed to '1' once.
- bit 2: ECC bit. if this bit is set then an ECC correction on the data was executed.
- bit 3: DRQ bit. If this bit is set then the disk either wants data (disk write) or has data for you (disk read).
- bit 4: SKC bit. Indicates that a seek has been executed with success.
- bit 5: WFT bit. Indicates a write error has happened.
- bit 6: RDY bit. Indicates that the disk has finished its power-up. Wait for this bit to be active before doing anything (except reset) with the disk.
- bit 7: BSY bit. This bit is set when the disk is doing something for you. You have to wait for this bit to clear before you can start giving orders to the disk.

9.3.1.7.2 Interrupt and Reset Register

- bit 1: IRQ enable. If this bit is '0' the disk will give and IRQ when it has finished executing a command. When it is '1' the disk will not generate interrupts.
- bit 2: RESET bit. If you pulse this bit to '1' the disk will execute a software reset. The bit is normally '0'.

9.3.1.7.3 Active Status Register

This is a read register.

- bit 0: master active. If this bit is set then the master IDE device is active.
- bit 1: slave active. If this bit is set then the slave IDE device is active.
- bits 5..2: complement of the currently active disk head.
- bit 6: write bit. This bit is set when the device is writing.
- bit 7: in a PC environment this bit indicates if a floppy is present in the floppy drive.

9.3.1.7.4 Error Register

The error register indicates what went wrong when a command execution results in an error. The fact that an error has occurred is indicated in the status register, the explanation is given in the error register. This is a read register.

- bit 0: AMNF bit. Indicates that an address mark was not found.
- bit 1: TK0NF bit. When this bit is set the drive was not able to find track 0 of the device.
- bit 2: ABRT bit. This bit is set when you have given an indecent command to the disk. Mostly wrong parameters (wrong head number etc..) cause the disk to respond with error flag in the status bit set and the ABRT bit set.
- bit 3: MCR bit. Indicated that a media change was requested.
- bit 4: IDNF bit. Means that a sector ID was not found.
- bit 5: MC bit. Indicates that the media has been changed.
- bit 6: UNC bit. Indicates that an uncorrectable data error happened. Some read or write errors could cause this.
- bit 7: Reserved

9.3.1.8 FAT Data Structure

FAT stands for File Allocation Table. What this data structure does is define a singly linked list of clusters of a file. At this point a directory or file container is merely a special file with an attribute to show that it is a directory. The thing that makes a directory unique is that the data or contents of the file are a series of 32 byte FAT directory entries. Essentially a directory is no different from a file.

9.3.2 Software

Software used to aid the development of the MP3 Hard Drive Player. Without these developmental tools the project and the realization of the MP3 Hard Drive would be impossible.

9.3.2.1 FDISK and FORMAT

Using an IDE Hard Drive we used these Microsoft utilities in order to ready our hard drive for user upload of MP3 music. The FDISK and FORMAT commands can be found on any Windows Startup Disk and requires these files in order to define the cluster size and the way to handle information specifically for our project. FORMAT drive: /z:n where drive is the drive you wish to format and n is the number of sectors per cluster multiplied by 512. FDISK was used to create hard drive partition a single fat32 partition. We used the maximum partition size.

Example:

- format c: /z:1 creates a 512 Bytes cluster
- format c: /z:2 creates a 1024 Bytes (1 KB) cluster
- format c: /z:4 creates a 2kb cluster.

For our case the syntax for the format command was: format c: /z:8 since that is the cluster size required by our firmware implemented.

9.3.2.2 IsoPro

IsoPro 2.0 is a Windows 9x/NT/ME/2000 software program that imports electronic CAD files and prepares them for use by Quick Circuit machines. IsoPro features a completely new Windows user interface designed to make the circuit board able to be milled. The program produces the Isolation of the traces contained on the printed circuit board and drives the CAM.

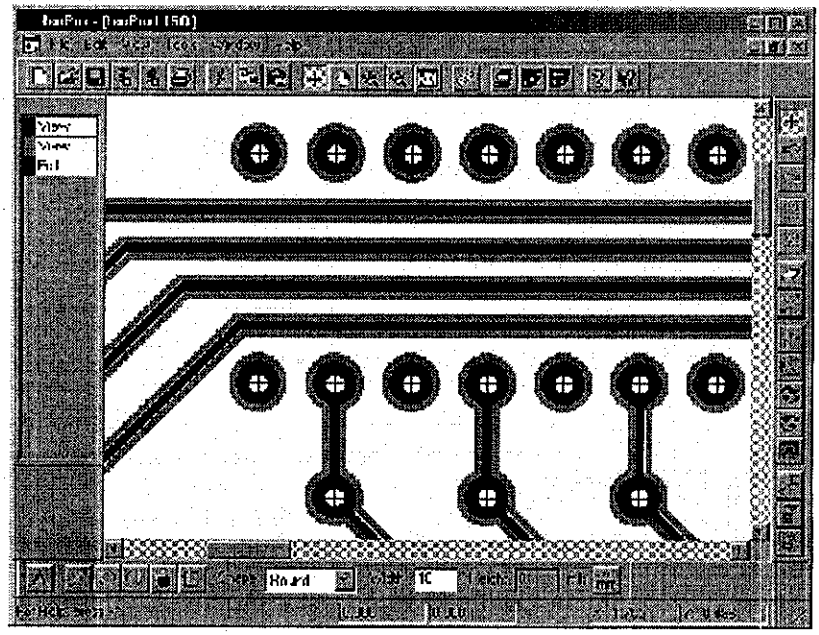


Figure 9-19: Screenshot of IsoPro Software

9.3.2.3 EAGLE Layout Editor

Cadsoft EAGLE is another PCB board layout editor that allows a user to design the PCB board and mill it according to the design. All boards were created using Cadsoft Eagle since the light version is free and it is a very powerful too with many part libraries available. Integrated in eagle is a CAM post processor. This allows us to export the files to extended gerber format for milling in ISOPro. In addition to extended gerber, the CAM post processor is able to export Excellon format for automatic placement of drill holes. The drill holes have in the past been drawn by hand. The functionality of the program took a good deal of research and experimentation.

Generating Excellon Drill Data
This CAM job can be used to generate data for drilling machines in Ecxellon format.

Run drillcfg.ulp in the Layout Editor window to generate a rack file first, then start the CAM Processor and load this job.

The board house needs two files:
*.drl - rack file
*.drd - drill data

Generating Extended Gerber Format
This CAM job consists of five sections that generate data for a two layer board.

You will get five gerber files that contain data for:
component side *.cmp
solder side *.sol
silkscreen component side *.plc
solder stop component side *.stc
solder stop solder sid *.sts

The last three files are not used for milling in our lab. Also, for the solder side layout, mirror is checked by default. It is desireable to turn this off.

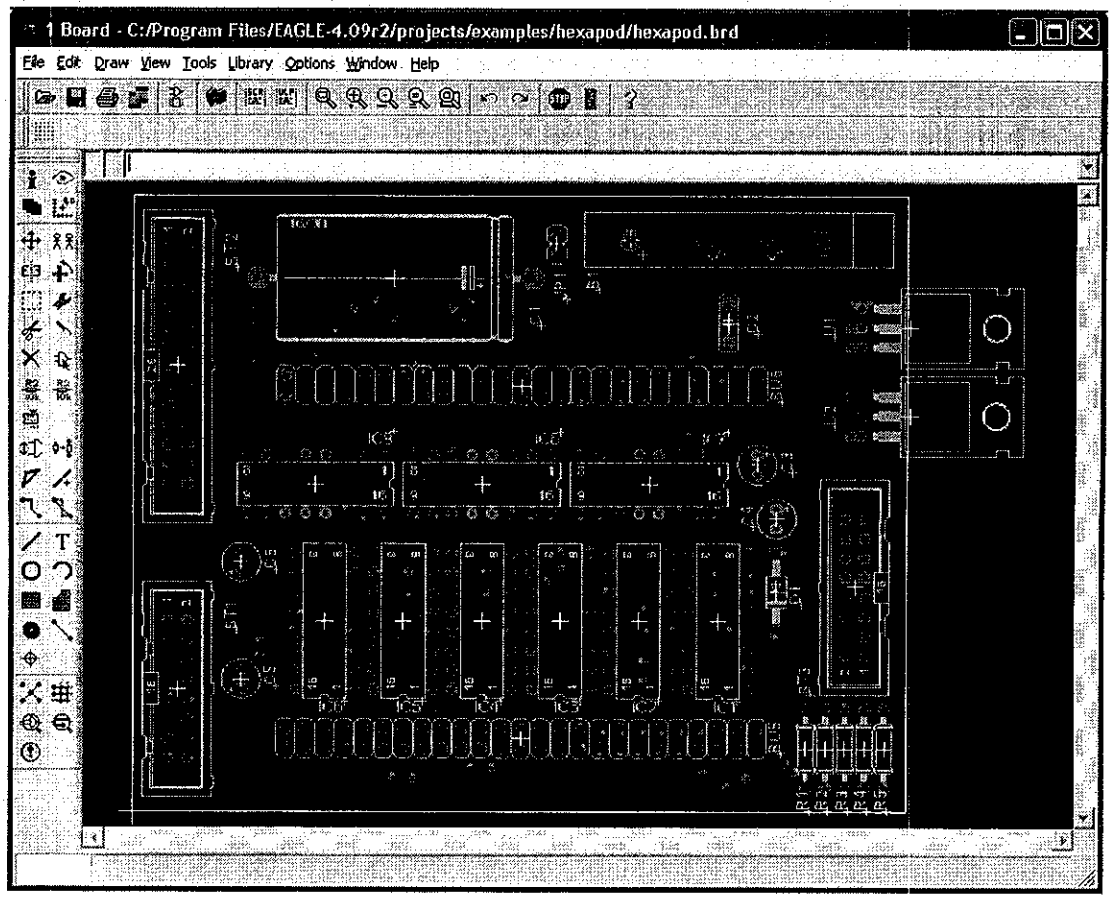


Figure 9-20: Screenshot of the EAGLE Layout Editor Software

9.3.2.4 Terminal Program by Bray++

This program was also very useful in debugging and testing the MP3 player code to see if the hardware was working properly. For more information please refer to the section labeled Terminal Program by Bray++.

9.3.2.5 AVR-GCC

The AVR-GCC C compiler (and assembler) that is made available through the GNU project under the GNU general public license. Using this open-source compiler was necessary to compile the C programs for the AVR microcontroller. This used with AVR Studio as a Third party compiler. This does not aid in the ease of programming.

9.3.2.6 AVR Studio

AVR Studio is a complete development suite, and contains an editor and a simulator that we used to write our code, and then see how it will run on an AVR device. This is the free IDE for AVR programming provided by Atmel.

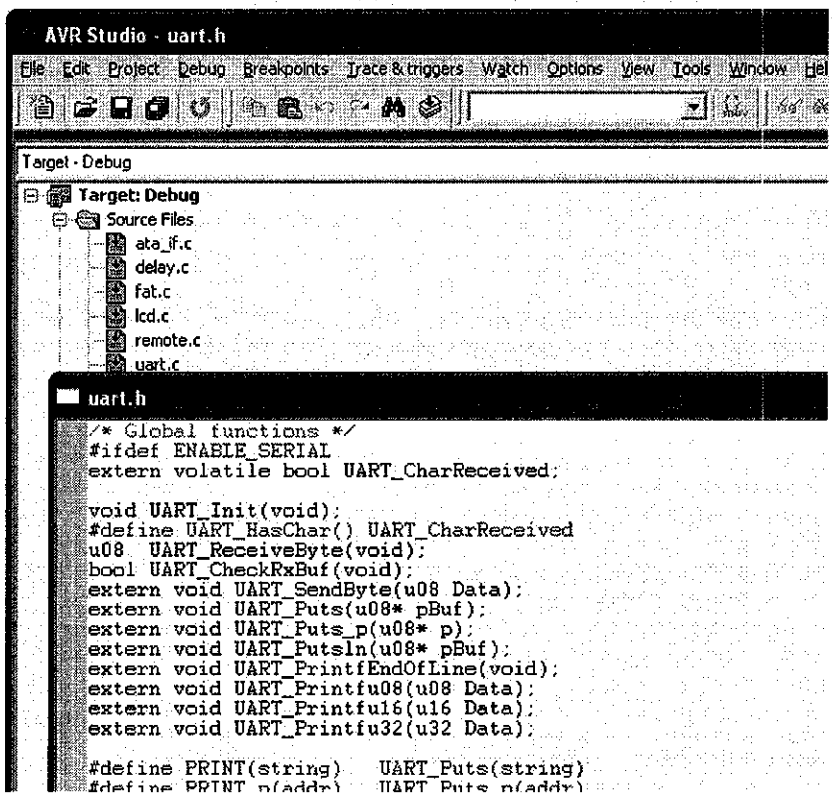


Figure 9-21: Screenshot of AVR Studio

9.3.2.7 STK5000 with AVR Studio

STK500 programming is integrated with AVR Studio. The Flash, EEPROM, and all fuse options can be programmed individually or with sequential programming option.

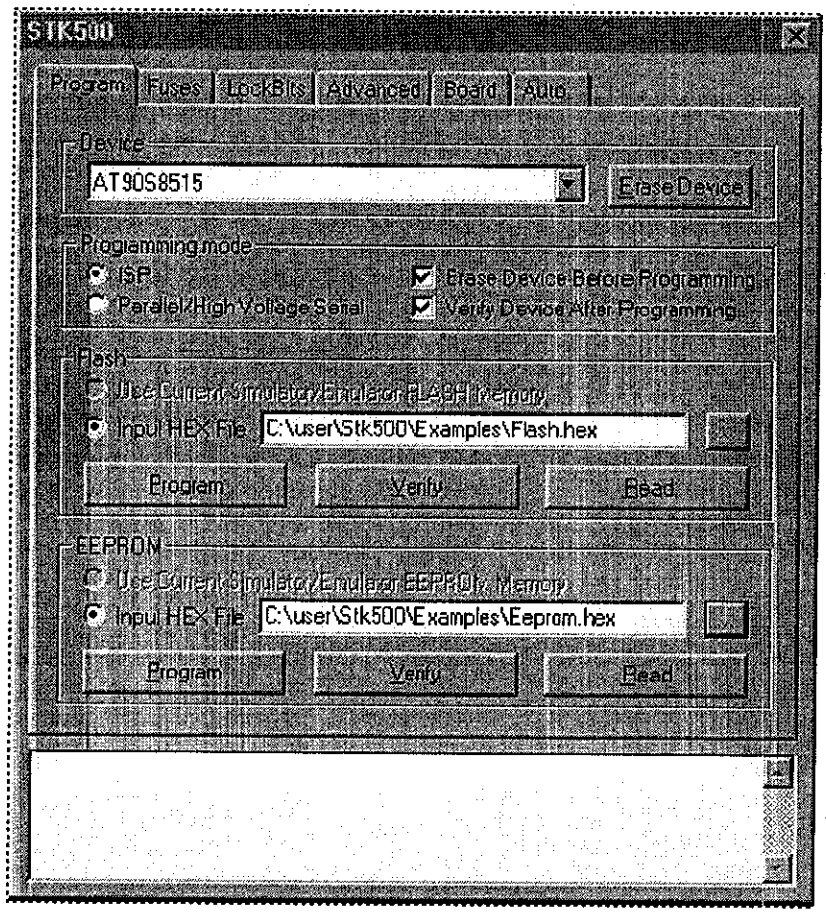


Figure 9-22: Programming the STK500 inside the AVR Studio Software

9.3.3 Troubleshooting

9.3.3.1 Why doesn't my MP3 song play?

At certain times the MP3 player may not recognize a song due to certain bit-rate settings or some unconventional method of MP3 encoding. Please decode the MP3 into a wave format and re-encode the MP3 file with a reliable codec.

9.3.3.2 Why do I hear a loud noise from the MP3 Player?

Certain files may be encoded and pops and static may result. If it is the hard drive player then turn off the power, push the reset button to discharge the circuit and re-power the MP3 player.

9.3.3.3 Why doesn't the Alpine Head Unit recognize button commands?

Make sure that the PIC module is set in the socket as instructed. Power down the circuit and check the cable connections. Make sure that the button is not stuck. If a delay occurs then before pressing the button wait approximately 3 seconds to send out the next command.

9.3.3.4 What should I do when my MP3 player screeches?

Either the MP3 file being played is corrupted or the Atmel MP3 Hard Drive Player has locked up. Please turn off power to the unit, discharge the circuit by pressing the reset button, and re-power the circuit.

9.3.3.5 How come the MP3 won't change tracks?

Make sure that the serial connection going from the PIC CD Changer Emulator Module to the Atmel MP3 Hard Drive Player is connected. Reseat the cables if necessary.

10 Appendix

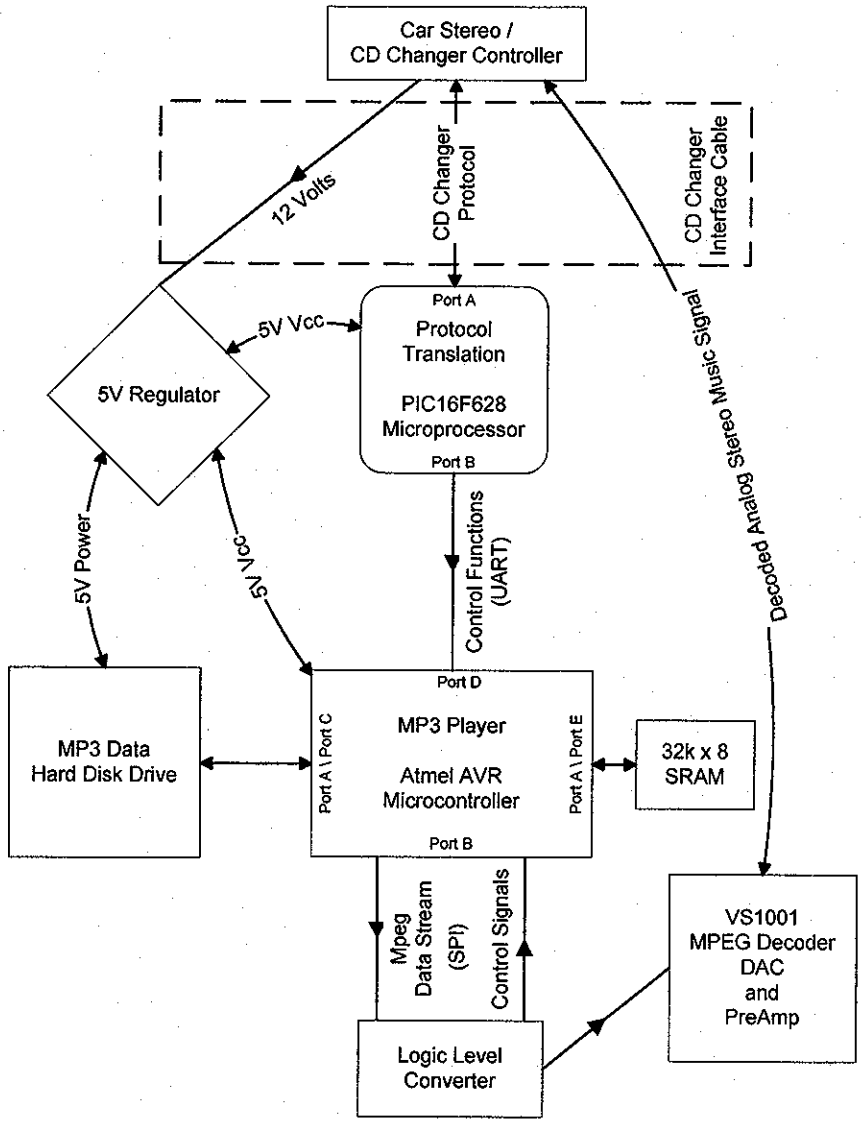
10.1 Schematic and Circuit Diagrams

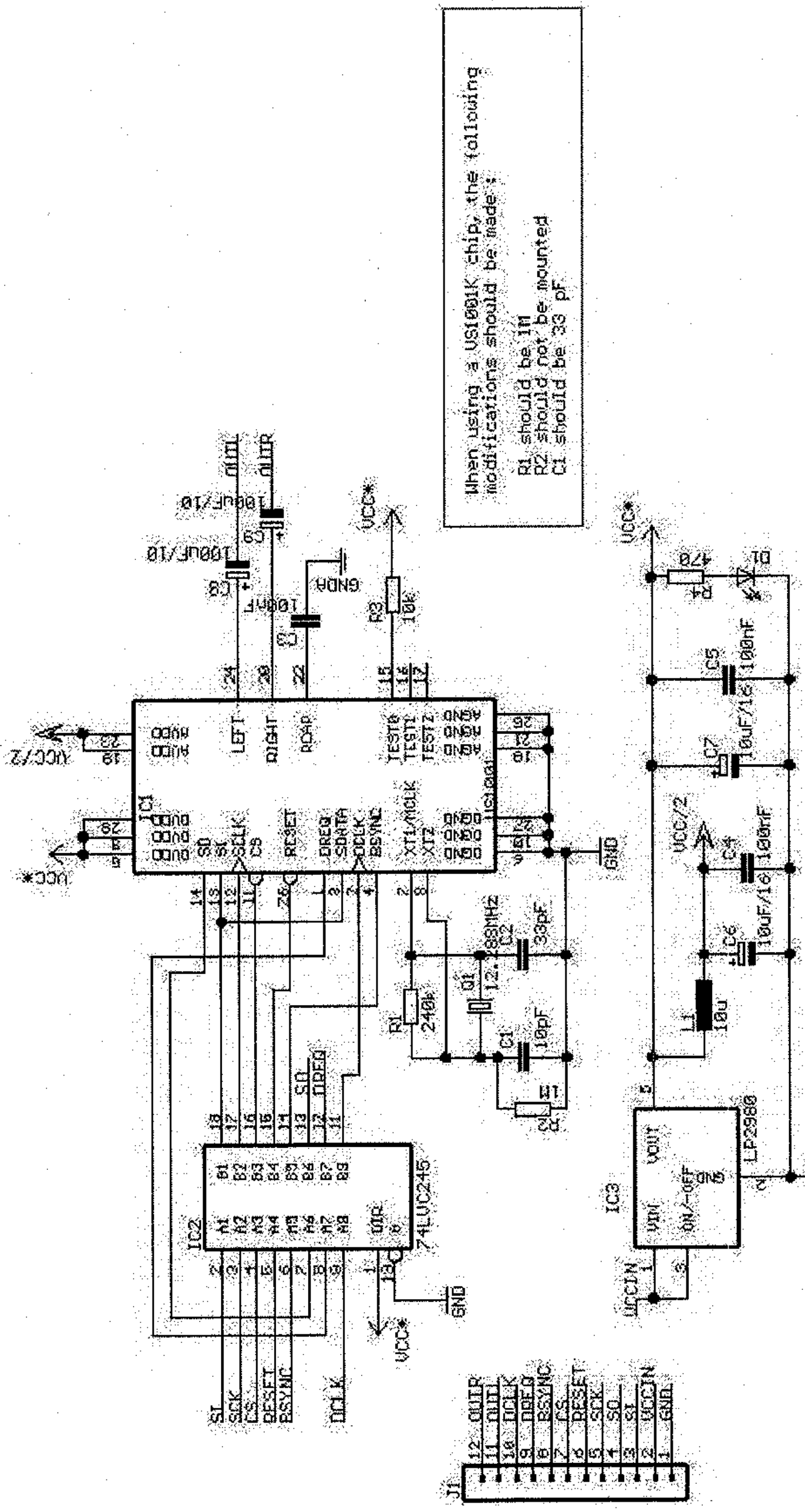
10.2 M-Bus PIC Code

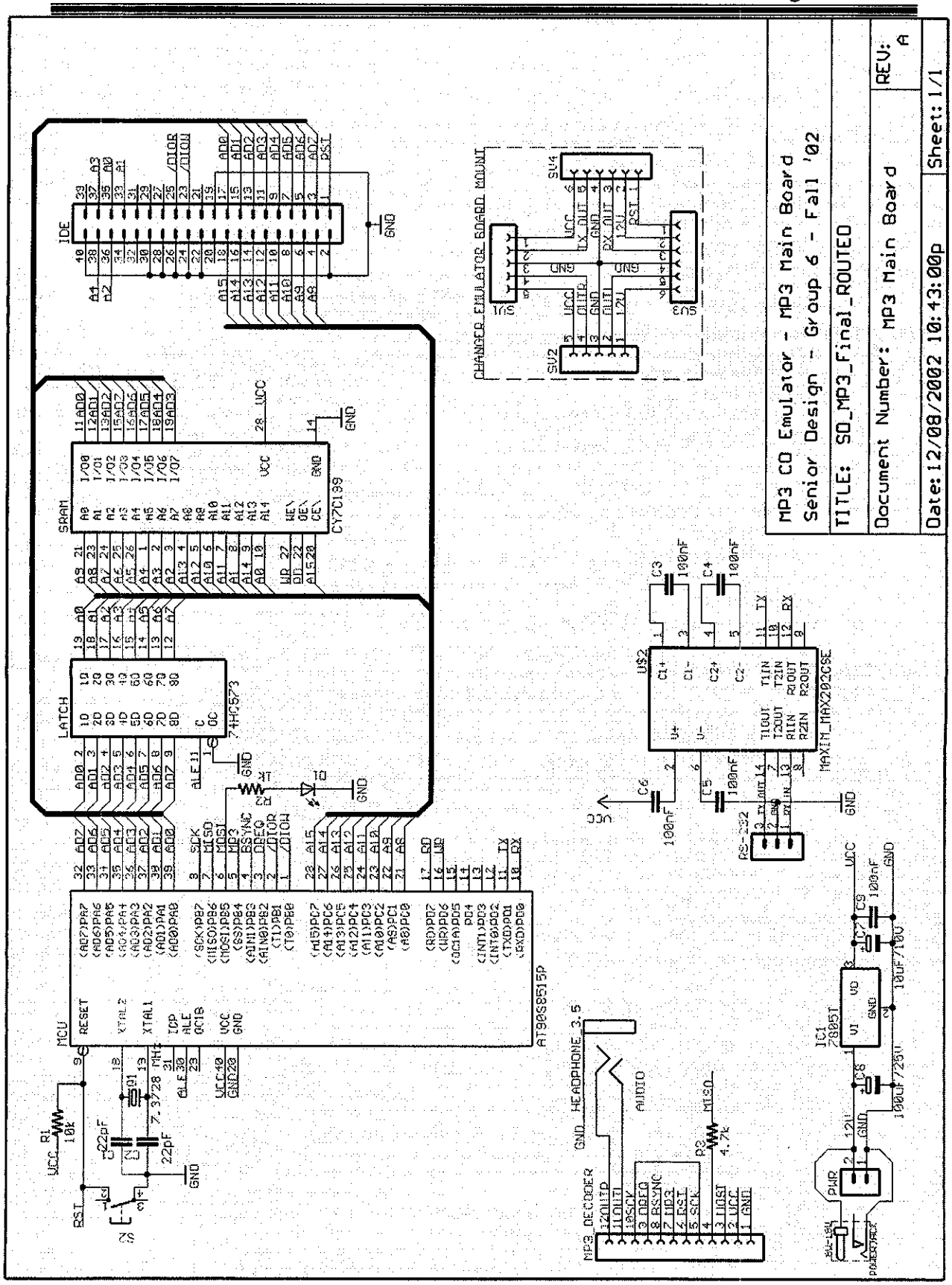
10.3 Log of M-Bus Packets

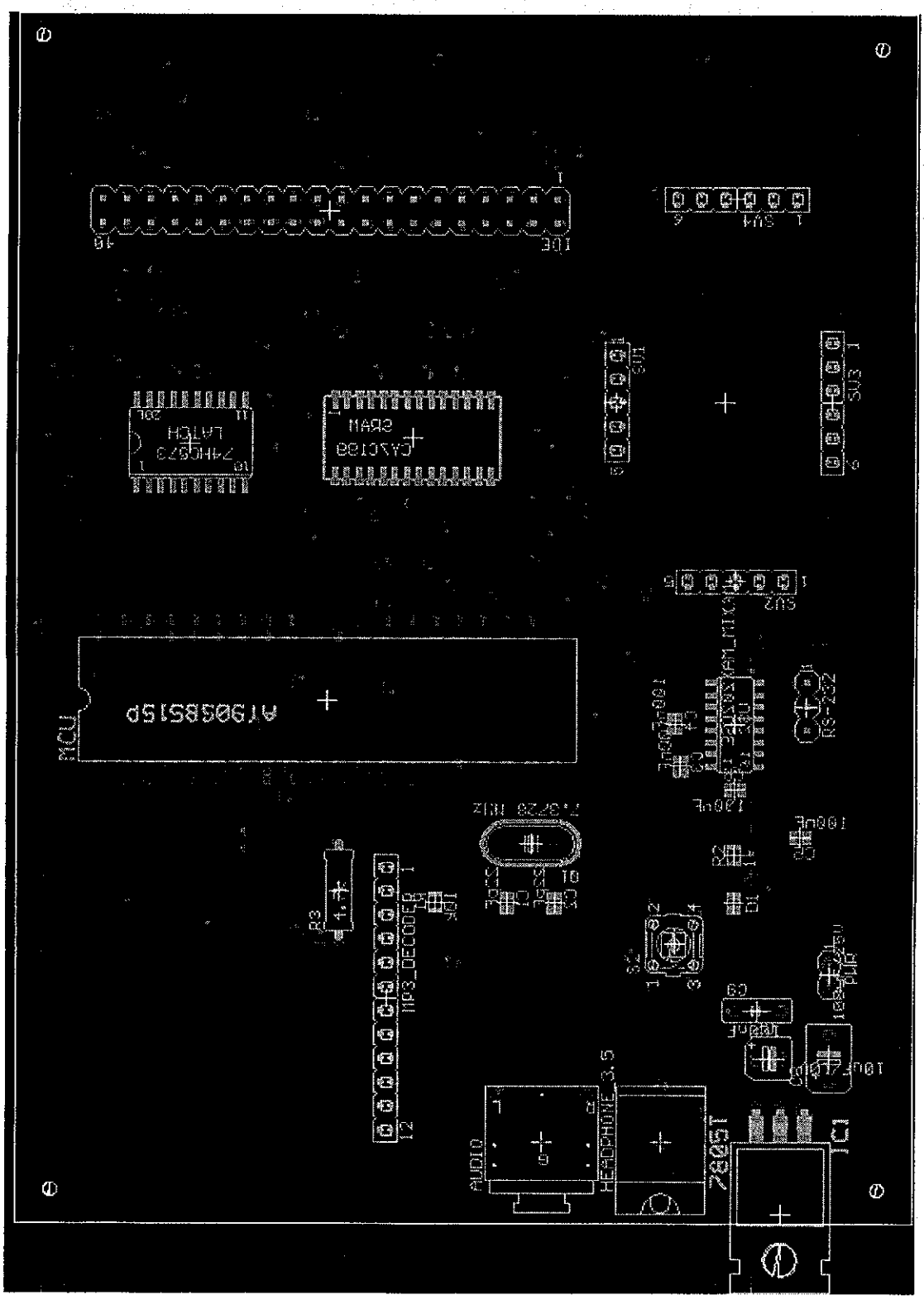
Schematic and Circuit Diagrams

Block Diagram
MP3 CD Changer Emulator




When using a US1001K chip, the following modifications should be made:
R1 should be 1M
R2 should not be mounted
C1 should be 33 pF


MP3 CD Emulator - MP3 Main Board
Senior Design - Group 6 - Fall '02
TITLE: SD_MP3_Final_ROUTED
Document Number: MP3 Main Board
REV: A
Date: 12/08/2002 10:43:00p
Sheet: 1/1
IDE
SRAM
CY7C199
74HC573
LATCH
MCU
AT90S8515P
MAXIM_MAX202CSE
RS-232
MP3 DECODER
HEADPHONE 3.5
AUDIO
CHANGER EMULATOR BOARD MOUNT
IC1
78L05
PWR


MCU
AT90S8515P
SRAM
CX52088
74HC573
LATCH
SW1
SW2
SW3
MP3_DECODER
AUDIO
HEADPHONE
7.8351

M-Bus PIC Code

```
'-------------------------------------------------------------------------
'-- CD Changer Emulator
'-------------------------------------------------------------------------
'   Date:
'           Fall Semester 2002
'
'   Company:
'           University of Central Florida
'
'   Group:
'           Daniel Gallagher
'           Ethan Steffens
'           Tony Trinh
'
'   Description:
'           This Pic Basic/assembly code allows for communication
'           between the head unit and the Atmel.  Using the PIC16F628
'           to interpret the protocol this code allows the necessary
'           functions to output head unit commands to the Atmel.
'
'   Comments:
'           When debugging with Serout2 you the pause statements at the
'           beginning of each GoSub packet routine needs to be changed.
'           Make them shorter.
'
'-------------------------------------------------------------------------
'-- Revision History
'-------------------------------------------------------------------------
'   1.0  - Initial Coding
'          Wrote the Wait664 for testing
'   1.1  - Added the TRISA and TRISB settings for the PORT direction
'          Included interrupt flag behavior
'   1.1a - Added the Track Command Hex Codes
'   1.1b - Added Interrupt saves and Buffer locations
'   1.1c - Added code for capture of data bits
'          Added code to SendByte to Head Unit and SendSerial to Atmel
'          IdleStatus Subroutine
'   1.2  - Added to capture 8 bytes
'          Checking first byte for action
'          Added InitStatus1 and InitStatus2
'   1.3  - Added additional control lines
'          Recoded BitLoop and BitNibLoop to accommodate timing issues
'          Subroutines CntrlPinSet0 and CntrlPinSet1
'          Modified wait times
'   1.4  - Modified the CP1 and CP2 Pins
'          Added waitcount and the proper timing for the protocol
'          Added First Byte Checking for IdleStatus
'          Deleted some commented code, (See v1.3)
'   1.5  - Added waitLoop and waitcountLoop for msec subroutine
'          Modified code to reflect the new subroutines
'          Added wait times for the InitStatus1 and InitStatus2
'   1.6  - Ported code over to PicBasic Pro
'          Code rewritten, further PIC assembly code (See v1.5)
'          Wrote CntrlPinSet0,1 using asm directives
'   1.6a - Pse1, Pse2, Play1, Play2, Nxt, Prev subroutines coded
```

```
'    1.6b - Edited GetPollingData to properly get the bits needed
'    1.7  - Added the Serial in alias
'           Added the SerOut2 code to communicate with the AtMega161L
'           Removed previous defined constants (See 1.6b)
'    1.8  - Asm Interrupt Handler for Pulsin head unit commands
'           Changed the DataPin to PORTB.3 Changed the pulseWidth
routine
'           Changed so pulseWidths are stored in a pulseArray
'    1.9  - Back to Polling for Data
'           Detection of the Head Unit Polling command working
'               Puts only one packet in buttonBytes
'      2.0    - Optimized loops
'               ControlPin configured to directly interface with DataPin
'               Old transfer packets taken out
'               Idle, Init1, Init2 recoded
'      2.1  - Major code reorganization
'               Waiting loop put in
'               Many if's taken out
'      2.2  - Different clock cycle, 20mhz now
'      2.3  - Got it working again.
'      2.4  - Adding change source function
'             - Changed the the track number... didn't work, going back
'      3.0    - Adding Next Track, Previous Track,
'      3.1  - Still adding the functions.. filename wrkmb3_4
'      4.0  - Got the functions working, switching the display for all
functions
'      4.1  - Implemented NextDisc(folder), PreviousDisc(folder)
'      5.0  - Implemented more functions, HDD spin down
'-------------------------------------------------------------------------

      INCLUDE "MODEDEFS.BAS"

'--------------------------------------
'-- Connections
'--------------------------------------
'    PIC RB3 <- From Head Unit MBus Pin 3
'
'    PIC RA1 = Control Pin  (Protocol)
'
'    PIC RB2 -> Atmel
'
'--------------------------------------

DataPin           VAR         PORTB.3          ' This is the Data Pin
for receiving and sending the protocol
ControlPin2       VAR         PORTA.2          ' Control Pin 2
SerialOut         VAR         PORTB.2          ' Used to issue
commands to the Atmel
'SerialIn         VAR         PORTB.1          ' For future
implentation

'-------------
'-- Defines --
'-------------
DEFINE                  PULSIN_     MAX 300
DEFINE                  OSC         20
```

```
'---------------
'-- Variables --
'---------------
bitBuffer          VAR          BYTE              ' To rotate the bits
into captureBuffer
tempBuffer         VAR          BYTE              ' Temporary byte Buffer
to store into captureBuffer
captureBuffer      VAR          BYTE[3]           ' Store all the
captured bits here
buttonByte1        VAR          BYTE              ' First CaptureBuffer
Byte
buttonByte2        VAR          BYTE              ' Second CaptureBuffer
Byte
buttonByte3        VAR          BYTE              ' Third CaptureBuffer
Byte
pulseWidth         VAR          WORD              ' Variable to store
pulseWidth
loopValue1         VAR          BYTE
loopValue2         VAR          BYTE
sendBuffer         VAR          BYTE
Meminitchange      VAR          BIT
MemoryP                VAR          BIT           'Used to keep track of
which pause
MemoryuP           VAR    BIT                     'Used to keep track of
play and unpause
Memory1or2         VAR          BIT               'Used to keep track of
which unpause
pulseArray         VAR    WORD[24]
temp               VAR    BYTE
MemTrack           VAR          BYTE

'------------------
'-- Main Program --
'------------------
TRISA.2 = 1

Memory1or2 =0
Meminitchange = 0
MemoryP = 0
MemoryuP = 0
MemTrack = $00


Idleloop:

      bitBuffer = $00
      tempBuffer = $00
      captureBuffer[0] = $00
      captureBuffer[1] = $00
      captureBuffer[2] = $00

      GoSub GetPollingData     ' Wait for a packet
      GoSub Restpac            ' Get a packet
      GoSub ScanButtons        ' Scan to see if we recognize any buttons
```

```
      GoTo Idleloop

      End

'-------------------------------------------
'-- GetPollingData
'-------------------------------------------
'   This funtion is used to capture the bits
'   1 is a pin pulled low for roughly 1.21 ms
'   0 is a pin pulled Low For roughly 709 us
'-------------------------------------------
GetPollingData:

Waiting:
                                    'Polls until it gets something
      PulsIn DataPin, 0, pulseArray[0]
      IF (pulseArray[0] != 0) Then Capture            'if pulsin
doesn't time out then it polls

      GoTo Waiting

Capture:
      For loopValue1 = 0 to 22
      'changing from 23 to 22 cuz i +1
            PulsIn DataPin, 0, pulseArray[loopValue1 + 1]
      'changed to +1
      Next loopValue1

      Return


'-------------------------------------------
'-- Restpac
'-------------------------------------------
'   This funtion is used to get the rest of
'   the packet.
'-------------------------------------------
Restpac:

      temp=0

      For loopValue2 = 0 to 2

            For loopValue1 = 0 to 7

                  IF (pulseArray[temp] = 0) Then
                        captureBuffer[loopValue2] = tempBuffer
                        GoTo Ret
                  EndIF

                  IF (pulseArray[temp] < 500) Then
                        bitBuffer = 0
                        GoSub FillByte
                        GoTo nxtrest
                  EndIF
```

```
                    IF (pulseArray[temp] > 500) Then
                          bitBuffer = 1
                          GoSub FillByte
                    EndIF

nxtrest:
                    temp = temp + 1

              Next loopValue1

              captureBuffer[loopValue2] = tempBuffer
              tempBuffer = 0
        Next loopValue2

Ret:
        Return

'-----------------------------------------
'-- FillByte
'-----------------------------------------
'   This funtion is used to get fill the
'   tempBuffer
'-----------------------------------------
FillByte:
        Asm
              rrf _bitBuffer,1
              rlf _tempBuffer,1
        EndAsm

Return

'-------------------------------------------------------------------------
'-- Status & Button Push Packets
'-------------------------------------------------------------------------
'   IdleStatus
'   HeadUnit - 18A 11.7ms
'   CD Changer - 982
'-------------------------------------------------------------------------
'   InitStatus1
'   Head Unit - 111819 13.71ms
'   CD Changer - 9F005147 8.67ms 9C10115772572 9.51ms 9F005165 8.39ms
9F005165 8.67ms 9F005165 8.39ms 9C10115772572 9.79ms 9A00000000004
9.79ms 9D000000005 8.39ms 9B9103000019 8.67ms 9910301022000013
'-------------------------------------------------------------------------
'   InitStatus2
'   Head Unit - 111406 15.39ms
'   CD Changer - 991030102200009B 751.89ms 9F005165 8.79ms
99203010220009A 751.89ms 9F005165 8.79ms 99203010220009A
'-------------------------------------------------------------------------
'   Pse1 (Pause)
'   Head Unit - 111024 22.8906ms
'     CD Changer - 9940101035700015 8.62835ms 9F005268
'-------------------------------------------------------------------------
```

```
'-------------------------------------------------------------------
'     Pse2 (Pause)
'     Head Unit - 111024 15.4851ms
'     CD Changer - 9930101035700021
'-------------------------------------------------------------------
'     Play1
'     Head Unit - 111011 18.5952ms
'     CD Changer - 9F001E67 8.62539ms 9940301014300012
'-------------------------------------------------------------------
'     Play2
'     Head Unit - 111011 13.0561ms
'     CD Changer - 9940301014300012
'-------------------------------------------------------------------
'     Nxt
'     Head Unit - 113003102 5.0123ms
'     CD Changer - 9940201001100017 8.62924ms 9F007068 9.83995ms
9B920300001C 9.83837ms 9950301000000017 8.62783ms 9F007068 9.84038ms
9920301000000018 9.84176ms 9920301000000018
'-------------------------------------------------------------------
'     Prev
'     Head Unit - 113002101 19.2171ms
'     CD Changer - 9F00F060 9.84107ms 9B960200001F 8.63064ms
9950201000000018
'-------------------------------------------------------------------
ScanButtons:
            buttonByte1 = captureBuffer[0]
            buttonByte2 = captureBuffer[1]            ' Storing the
captureBuffer into buttonByte variables
            buttonByte3 = captureBuffer[2]

            'IDLESTATUS
            IF (buttonByte1 == $18) AND (buttonByte2 == $0A) Then
                  GoSub IdleStatus
                  TRISA.2 = 1
                  GoTo RetoMain
            EndIF

            'INITSTATUS1
            IF (buttonByte1 == $11) AND (buttonByte2 ==     $18) Then
                  GoSub InitStatus1
                  TRISA.2 = 1
                  SerOut2 SerialOut, 16416, ["G"]
                  GoTo RetoMain
            EndIF

            'INITSTATUS2
            IF (buttonByte1 == $11) AND (buttonByte2 == $14) Then
                  GoSub InitStatus2
                  TRISA.2 = 1
                  SerOut2 SerialOut, 16416, ["G"]
                  GoTo RetoMain
            EndIF

            'Next Track and Previous Track
            IF (buttonByte1 == $11) AND (buttonByte2 == $30) Then
```

```
                    'PREVIOUS TRACK
                    IF (buttonByte3 <= MemTrack) OR (buttonByte3 == $15)
OR (buttonByte3 == $14) Then
                          GoSub PreviousTrack
                          TRISA.2 = 1
                          SerOut2 SerialOut, 16416, ["p"]
                          MemTrack = $02
                          GoTo RetoMain
                    EndIF

                    'PREVIOUS TRACK
                    IF (buttonByte3 <= MemTrack) OR (buttonByte3 == $98)
OR (buttonByte3 == $99) Then
                          GoSub PreviousTrack
                          TRISA.2 = 1
                          SerOut2 SerialOut, 16416, ["p"]
                          MemTrack = $02
                          GoTo RetoMain
                    EndIF

                    'NEXT TRACK
                    IF (buttonByte3 > MemTrack) Then
                          GoSub NextTrack
                          TRISA.2 = 1
                          SerOut2 SerialOut, 16416, ["n"]
                          MemTrack = $02
                          GoTo RetoMain
                    EndIF

              EndIF


              'If statement for Next Disc and Previous Disc
              IF (buttonByte1 == $11) AND ((buttonByte2 &%11110000) ==
$30) Then

                    ' PREVIOUS DISC
                    IF ((buttonByte2 &%00001111) == 6) OR ((buttonByte2
&%00001111) == 5) OR ((buttonByte2 &%00001111) == 1) Then
                          GoSub PreviousDisc
                          TRISA.2 = 1
                          SerOut2 SerialOut, 16416, ["D"]
                          GoTo RetoMain
                    EndIF

                    'NEXT DISC
                    IF ((buttonByte2 &%00001111) == 2) OR ((buttonByte2
&%00001111) == 3) OR ((buttonByte2 &%00001111) == 4) Then
                          GoSub PreviousDisc
                          TRISA.2 = 1
                          SerOut2 SerialOut, 16416, ["d"]
                          GoTo RetoMain
                    EndIF

              EndIF
```

```
'If statement for initchange, pause and unpause
IF (buttonByte1 == $11) AND (buttonByte2 == $10) Then

      'INITCHANGE1and2
      IF (buttonByte3 == $11) AND (MemoryuP == $00) Then

            IF (Meminitchange == 0) Then
                  Meminitchange = 1
                  GoTo RetoMain
            EndIF

            IF (Meminitchange == 1) Then
                  GoSub InitChange1
                  TRISA.2 = 1
                  SerOut2 SerialOut, 16416, ["g"]
                  Meminitchange = 0
                  GoTo RetoMain
            EndIF

      EndIF

      'PAUSE
      IF (buttonByte3 == $24) Then

            IF (MemoryP == $00) Then
                  GoSub Pausing1
                  TRISA.2 = 1
                  GoTo RetoMain
            EndIF
            IF (MemoryP == $01) Then
                  GoSub Pausing2
                  TRISA.2 = 1
                  SerOut2 SerialOut,16416, ["G"]
                  GoTo RetoMain
            EndIF

      EndIF

      'UNPAUSE
      IF (buttonByte3 == $11) AND (MemoryuP == $01) Then

            IF (Memory1or2 == $00) Then
                  GoSub UnPause1
                  TRISA.2 = 1
                  GoTo RetoMain
            EndIF
            IF (Memory1or2 == $01) Then
                  GoSub UnPause2
                  TRISA.2 = 1
                  SerOut2 SerialOut,16416, ["g"]
                  GoTo RetoMain
            EndIF

      EndIF

EndIF
```

```
RetoMain:
      Return

'-----------------------
'-------Packets---------
'-----------------------

PreviousDisc:
            Pause 50
            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $6F
            GoSub SendByte
            sendBuffer = $6A
            GoSub SendByte

            Pause 9

            sendBuffer = $9B
            GoSub SendByte
            sendBuffer = $91
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $1B
            GoSub SendByte

            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $20
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $13
            GoSub SendByte

            Pause 50

            sendBuffer = $9F
            GoSub SendByte
```

```
sendBuffer = $00
GoSub SendByte
sendBuffer = $6F
GoSub SendByte
sendBuffer = $6A
GoSub SendByte

Pause 9

sendBuffer = $9B
GoSub SendByte
sendBuffer = $D1
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $1F
GoSub SendByte

Pause 50

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $45
GoSub SendByte
sendBuffer = $53
GoSub SendByte

Pause 9

sendBuffer = $9B
GoSub SendByte
sendBuffer = $B1
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $19
GoSub SendByte

Pause 9

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $29
GoSub SendByte
```

```
sendBuffer = $59
GoSub SendByte

Pause 9

sendBuffer = $9B
GoSub SendByte
sendBuffer = $C1
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $10
GoSub SendByte

Pause 50

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $29
GoSub SendByte
sendBuffer = $6C
GoSub SendByte

Pause 9

sendBuffer = $9B
GoSub SendByte
sendBuffer = $81
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $1C
GoSub SendByte

Pause 50

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $51
GoSub SendByte
sendBuffer = $65
GoSub SendByte

Pause 9
```

```
            sendBuffer = $9B
            GoSub SendByte
            sendBuffer = $91
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $1C
            GoSub SendByte

            Pause 9

            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $51
            GoSub SendByte
            sendBuffer = $65
            GoSub SendByte

            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $12
            GoSub SendByte

            GoSub PreviousTrack                 'Added this code



'           sendBuffer = $
'           GoSub SendByte
'           sendBuffer = $
'           GoSub SendByte
'           sendBuffer = $
'           GoSub SendByte
'           sendBuffer = $
'           GoSub SendByte
'           sendBuffer = $
```

```
'           GoSub SendByte
'           sendBuffer = $

'           GoSub SendByte

            Return


PreviousTrack:
            Pause 50
            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $52
            GoSub SendByte
            sendBuffer = $68
            GoSub SendByte

            Pause 9

            sendBuffer = $9B
            GoSub SendByte
            sendBuffer = $91
            GoSub SendByte
            sendBuffer = $02
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $1A
            GoSub SendByte

            Pause 9

            GoSub SendByte
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $50
            GoSub SendByte
            sendBuffer = $20
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $18
            GoSub SendByte

            Return
```

```
NextTrack:
            Pause 50

            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $52
            GoSub SendByte
            sendBuffer = $68
            GoSub SendByte

            Pause 9

            sendBuffer = $9B
            GoSub SendByte
            sendBuffer = $91
            GoSub SendByte
            sendBuffer = $02
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $1A
            GoSub SendByte

            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $50
            GoSub SendByte
            sendBuffer = $20
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $18
            GoSub SendByte

            Return


Pausing1:
            Pause 50
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $40
            GoSub SendByte
```

```
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $10

            GoSub SendByte
            sendBuffer = $35
            GoSub SendByte
            sendBuffer = $70
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $15
            GoSub SendByte

            Pause 9

            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $52
            GoSub SendByte
            sendBuffer = $68
            GoSub SendByte

            MemoryP = 1                    'set to 1 so next time it
goes to pausing 2 routine

            Return

Pausing2:


            Pause 50
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $30
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $35
            GoSub SendByte
            sendBuffer = $70
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $21
            GoSub SendByte

            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $30
```

```
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $10

            GoSub SendByte
            sendBuffer = $35
            GoSub SendByte
            sendBuffer = $70
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $21
            GoSub SendByte

            MemoryuP = 1
            MemoryP = 0                 'set to zero so next time program
goes to pausing 1 routine

            Return

UnPause1:
            Pause 50
            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $52
            GoSub SendByte
            sendBuffer = $68
            GoSub SendByte

            Memory1or2 = 1              'set so next time program goes to
unpause2 routine

            Return

Unpause2:
            Pause 50
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $40
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $35
            GoSub SendByte
            sendBuffer = $70
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $15
            GoSub SendByte
```

```
            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $50

            GoSub SendByte
            sendBuffer = $20
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $18
            GoSub SendByte


            Memory1or2 = 0          'set to 0 so next time the program
will go to unpause 1
            MemoryuP = 0            'set to 0 so next time the program
will go to inital change

            Return


InitChange1:
            Pause 50
            sendBuffer = $99
            GoSub SendByte
            sendbuffer = $20
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $13
            GoSub SendByte

            Pause 9

            sendBuffer = $99
            GoSub SendByte
            sendbuffer = $20
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
```

```
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte

sendBuffer = $00
GoSub SendByte
sendBuffer = $13
GoSub SendByte

Pause 9

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $6F
GoSub SendByte
sendBuffer = $6A
GoSub SendByte

Pause 9

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $6F
GoSub SendByte
sendBuffer = $6A
GoSub SendByte

Pause 9

sendBuffer = $9B
GoSub SendByte
sendBuffer = $92
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $19
GoSub SendByte

Pause 9

sendBuffer = $99
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $00
```

```
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte

            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $12
            GoSub SendByte

            Pause 9

            sendBuffer = $99          ' adding in seeing if it works for
display
            GoSub SendByte
            sendBuffer = $50
            GoSub SendByte
            sendBuffer = $20
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $18
            GoSub SendByte

            Return

IdleStatus:
            Pause 50
            sendBuffer = $98
            GoSub SendByte
            sendBuffer = $2F
            GoSub SendNibble

            Return

InitStatus1:
            Pause 50
            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $51
            GoSub SendByte
            sendBuffer = $47
            GoSub SendByte

            PauseUs 8670
```

```
sendBuffer = $9C
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $11
GoSub SendByte
sendBuffer = $57

GoSub SendByte
sendBuffer = $72
GoSub SendByte
sendBuffer = $57
GoSub SendByte
sendBuffer = $2F
GoSub SendNibble

PauseUs 9510

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $51
GoSub SendByte
sendBuffer = $65
GoSub SendByte

PauseUs 8390

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $51
GoSub SendByte
sendBuffer = $65
GoSub SendByte

PauseUs 8670

sendBuffer = $9F
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $51
GoSub SendByte
sendBuffer = $65
GoSub SendByte

PauseUs 8390

sendBuffer = $9C
GoSub SendByte
sendBuffer = $10
GoSub SendByte
sendBuffer = $11
```

```
GoSub SendByte
sendBuffer = $57
GoSub SendByte
sendBuffer = $72
GoSub SendByte
sendBuffer = $57
GoSub SendByte
sendBuffer = $2F

GoSub SendNibble

PauseUs 9790

sendBuffer = $9A
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $4F
GoSub SendNibble

PauseUs 9790

sendBuffer = $9D
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $5F
GoSub SendNibble

PauseUs 8390

sendBuffer = $9B
GoSub SendByte
sendBuffer = $91
GoSub SendByte
sendBuffer = $03
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $00
GoSub SendByte
sendBuffer = $19
```

```
            GoSub SendByte

            PauseUs 8670

            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $10 'changed from a $10
            GoSub SendByte
            sendBuffer = $30 'changed from a $30

            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $22 'changed from a $22
            GoSub SendByte
            sendBuffer = $00 'changed from a $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $13
            GoSub SendByte

            Return

InitStatus2:
            Pause 50
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $10 'changed from a $10
            GoSub SendByte
            sendBuffer = $30 'changed from a $30
            GoSub SendByte
            sendBuffer = $10
            GoSub SendByte
            sendBuffer = $22 'changed from a $22
            GoSub SendByte
            sendBuffer = $00 'changed from a $00
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $9B
            GoSub SendByte

            Pause 752

            sendBuffer = $9F
            GoSub SendByte
            sendBuffer = $00
            GoSub SendByte
            sendBuffer = $51
            GoSub SendByte
            sendBuffer = $65
            GoSub SendByte

            PauseUs 8790
```

```
            sendBuffer = $99
            GoSub SendByte
            sendBuffer = $20 'changed from a $20
            GoSub SendByte
            sendBuffer = $30 'changed from a $30
            GoSub SendByte
            sendBuffer = $14
            GoSub SendByte
            sendBuffer = $22 'changed from a $22
            GoSub SendByte

            sendBuffer = $00 'changed from a $00
            GoSub SendByte
            sendBuffer = $09
            GoSub SendByte
            sendBuffer = $AF
            GoSub SendNibble

            Return

'-----------------------------------------------------
'-- SendByte
'-----------------------------------------------------
'   sends a byte to head unit.
'   load byte to sendBuffer before calling subroutine
'-----------------------------------------------------
SendByte:
                        For loopValue1 = 0 to 7
                              Asm
                                    rlf         _sendBuffer, 1
                                    btfsc STATUS, C
                                    Call  _CntrlPinSet1

                                    btfss STATUS, C
                                    Call  _CntrlPinSet0
                              EndAsm
                        Next loopValue1

                        Return

'-----------------------------------------------------
'-- SendNibble
'-----------------------------------------------------
'   sends a byte to head unit.
'   load byte to sendBuffer before calling subroutine
'-----------------------------------------------------
SendNibble:
                        For loopValue1 = 0 to 3
                              Asm
                                    rlf         _sendBuffer, 1
                                    btfsc STATUS, C
                                    Call  _CntrlPinSet1

                                    btfss STATUS, C
                                    Call  _CntrlPinSet0
                              EndAsm
```

```
                        Next loopValue1

                        Return

'-----------------------------------------------
'-- CntrlPinSet0
'-----------------------------------------------
'   Sets the RA1 & RA2 Control Pins for Logical 0
'-----------------------------------------------
CntrlPinSet0:
'                       High  ControlPin1
                        Low         ControlPin2

                        PauseUs     693

'                       Low         ControlPin1
                        High  ControlPin2

                        PauseUs 2440
                        'TRISA.2 =1
                        Return

'-----------------------------------------------
'-- CntrlPinSet1
'-----------------------------------------------
'   Sets the RA1 & RA2 Control Pins for Logical 1
'-----------------------------------------------
CntrlPinSet1:
'                       High  ControlPin1
                        Low         ControlPin2

                        PauseUs     1938

'                       Low         ControlPin1
                        High  ControlPin2

                        PauseUs 1196
                        'TRISA.2 =1
                        Return
```

Log of M-Bus Packets

** 1 is a pin pulled low for roughly 1.21 ms **
** 0 is a pin pulled low for roughly 709 us **

---

analyzation of idle (not on changer)

first complete sequence:: 0001 1000 1010 1001 1000 0010

second complete sequence:: 0001 1000 1010 1001 1000 0010

analyzation of idle on changer

first sequence:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 1001 0000 0000 0000 0001 1110

second sequence:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0010 0000 0000 0000 0000 0001 1000

third sequence:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0010 0001 0000 0000 0000 0001 0111

fourth sequence:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0010 0010 0000 0000 0000 0001 0110

analyzation of disc change

238.58ms after:: changeto2 0001 0001 0011 0010 0000 0000 0001 0000 0001 (15.43ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.62ms) 1001 1011 1001 0010 0000 0000 0001 0000 0000 0000 0000 0001 1010 (9.99ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011 (15.44ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.62ms) 1001 1011 1101 0010 0000 0000 0001 0000 0000 0000 0001 1110

781.12 ms after:: 1001 1111 0000 0000 0010 1010 0101 1100 (9.99ms) 1001 1011 1011 0010 0000 0000 0001 0000 0000 0000 0001 1100 (9.76) 1001 1111 0000 0000 0100 0101 0101 0011 (8.62ms) 1001 1011 1100 0010 0000 0000 0001 0000 0000 0000 0001 1101

446.07ms after:: 1001 1111 0000 0000 0100 0101 0110 0010 (9.76ms) 1001 1011 1000 0010 0000 0000 0001 0000 0000 0000 0001 1001

116 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 (9.76ms) 1001 1011 1001 0010 0000 0000 0000 0000 0000 0000 0001 1001 (8.62ms) 1001 1111 0000 0000 0110 1111 0110 1010 (9.99ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2 secs after:: 0001 1000 1010 (19.30ms) 1001 1000 0010

211.80 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 (9.76ms) 1001 1100 0010 0000 0001 0001 0001 0110 0010 0100 0110 0111 1000 (9.76ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.63ms) 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0001 0101

132.34 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 (9.86ms) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0001 0110

764.44 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

second sequence after change:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

anaylzation of forwardtrack.tla

first sequence (track 1):: 1001 1001 0100 0000 0001 0000 0001 0000 0001 0001 0101 0000 0000 0000 0001 0001

second sequence (track 1):: 1001 1001 0100 0000 0001 0000 0001 0000 0001 0001 0110 0000 0000 0000 0001 0100

fastforward command:: 0001 0001 0011 0000 0000 0010 0001 0000 0001 1001 1111 0000 0000 0101 0010 0110 1000 1001 1011 1001 0001 0000 0010 0000 0000 0000 0000 0001 1010 1001 1001 0101 0000 0010 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 1000

first sequence (track 2):: 1001 1001 0100 0000 0010 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 1000

second sequence (track2):: 1001 1001 0100 0000 0010 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 0101

analyzation of pause.tla

sequence before pause:: 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 0101 0000 0000 0000 0001 0111

sequence before pause:: 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 0110 0000 0000 0000 0001 0110

pause:: 0001 0001 0001 0000 0010 0100 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 0111 0000 0000 0000 0001 0101 1001 1111 0000 0000 0101 0010 0110 1000 0001 0001 0001 0000 0010 0100 1001 1001 0011 0000 0001 0000 0001 0000 0011 0101 0111 0000 0000 0000 0010 0001 1001 1001 0011 0000 0001 0000 0001 0000 0011 0101 0111 0000 0000 0000 0010 0001

2 secs after pause:: 0001 1000 1010 1001 1000 0010

2 secs after that:: 0001 1000 1010 1001 1000 0010

1.7 secs afer that:: 0001 0001 0001 0000 0001 0001 1001 1111 0000 0000 0101 0010 0110 1000 0001 0001 0001 0000 0001 0001 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 0111 0000 0000 0000 0001 0101 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 0111 0000 0000 0000 0001 0101

1.28 secs after:: 1001 1001 0100 0000 0001 0000 0001 0000 0011 0101 1000 0000 0000 0000 0001 1100

changeto5.tla

source change:: 0001 0001 0001 0000 0001 0001 0001 0001 0001 0000 0001 0001 1001 1001 0010 0000 0001 0000 0001 0000 0000 0001 0000 0000 0000 0000 0001 0011 1001 1001 0010 0000 0001 0000 0001 0000 0000 0001 0000 0000 0000 0000 0001 0011 1001 1111 0000 0000 1101 0001 0110 1101 1001 1111 0000 0000 1101 0001 0110 1101 1001 1100 0101 0000 0001 0001 0001 0100 0101 0101 0011 0111 0010 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0101 1001 1011 1001 0101 0000 0001 0000 0000 0000 0000 0001 1111 1001 1001 0001 0000 0001 0000 0000 0001 0000 0000 0000 0000 0001 0010

1.25 secs after:: 1001 1111 0000 0000 1101 0001 0110 1101 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 0000 0000 0000 0000 0001 0101

680 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 0001 0000 0000 0000 0001 0110

800 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 0010 0000 0000 0000 0001 0111

800 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 0011 0000 0000 0000 0001 1000

Disk4missing.tla

inital change:: 0001 0001 0001 0000 0001 0001 0001 0001 0001 0000 0001 0001 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011 1001 1111 0000 0000 0110 1111 0110 1010 1001 1111 0000 0000 0110 1111 0110 1010 1001 1011 1001 0010 0000 0000 0000 0000 0000 0000 0001 1001 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2 secs after:: 0001 1000 1010 1001 1000 0010

181 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 1001 1100 0010 0000 0001 0001 0001 0110 0010 0100 0110 0111 1000 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0101 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0101

105.8 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0110

790 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

796 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

803 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0011 0000 0000 0000 0001 0111

800 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0100 0000 0000 0000 0001 0010

799 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0101 0000 0000 0000 0001 0001

4.5 ms after::(changeto1) 0001 0001 0011 0001 0000 0000 0001 0000 0100 (15.8ms) 1001 1111 0000 0000 0110 1111 0110 1010 (9.53ms) 1001 1011 1001 0001 0000 0000 0001 0000 0000 0000 0001 1011 (8.17ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011

230.4 ms after:: 1001 1111 0000 0000 0110 1111 0110 1010 (9.53ms) 1001 1011 1101 0001 0000 0000 0001 0000 0000 0000 0001 1111

772 ms after:: 1001 1111 0000 0000 0100 0101 0101 0011 (8.39ms) 1001 1011 1011 0001 0000 0000 0001 0000 0000 0000 0001 1001 (8.64ms) 1001 1111 0000 0000 0010 1001 0101 1001 (8.64ms) 1001 1011 1100 0001 0000 0000 0001 0000 0000 0000 0001 0000

448 ms after:: 1001 1111 0000 0000 0010 1001 0110 1100 (9.97ms) 1001 1011 1000 0001 0000 0000 0001 0000 0000 0000 0001 1100

140.8 ms after:: 1001 1111 0000 0000 0101 0001 0110 0101 (8.40ms) 1001 1011 1001 0001 0000 0000 0000 0000 0000 0000 0001 1100 (10.0ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.40ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2 secs after:: 0001 1000 1010 (22.8ms) 1001 1000 0010

670 ms after:: 1001 1111 0000 0000 0101 0001 0110 0101 (8.4ms) 1001 1100 0001 0000 0001 0001 0101 0111 0111 0010 0101 0111 0010 (9.8ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.4ms) 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0001 0101

399 ms after:: 1001 1111 0000 0000 0101 0001 0110 0101 (8.6ms) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0110

822.6 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

812.9 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

.....

last packet with #:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0110 0000 0000 0000 0001 0100

584.7 ms after::(changeto3) 0001 0001 0011 0011 0000 0000 0001 0000 0010 (15.9ms)

1001 1111 0000 0000 0101 0001 0110 0101 (8.64ms) 1001 1011 1001 0011 0000 0000 0001 0000 0000 0000 0000 0001 1001 (8.42ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011

172 ms after:: 1001 1111 0000 0000 0101 0001 0110 0101 (8.42ms) 1001 1011 1101 0011 0000 0000 0001 0000 0000 0000 0001 1101

760 ms after:: 1001 1111 0000 0000 0010 1010 0101 1100 (9.78ms) 1001 1011 1011 0011 0000 0000 0001 0000 0000 0000 0001 1011 (17.1ms) 1001 1111 0000 0000 0110 0100 0101 0010 (9.78ms) 1001 1011 1100 0011 0000 0000 0001 0000 0000 0000 0001 1110

530.3 ms after:: 1001 1111 0000 0000 0110 0100 0110 0011 (8.87ms) 1001 1011 1000 0011 0000 0000 0001 0000 0000 0000 0001 1010

116.7 ms after:: 1001 1111 0000 0000 1000 1110 0110 0111 (7.96ms) 1001 1011 1001 0011 0000 0000 0000 0000 0000 0000 0001 1010 (9.55ms) 1001 1111 0000 0000 1000 1110 0110 0111 (8.42ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2.02 secs after:: 0001 1000 1010 (16.6ms) 1001 1000 0010

288.6 ms after:: 1001 1111 0000 0000 1000 1110 0110 0111 (8.64ms) 1001 1100 0011 0000 0001 0001 0001 0111 0000 0011 0100 0111 0001 (8.64ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.64ms) 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0001 0101

502.5 ms after:: 1001 1111 0000 0000 1000 1110 0110 0111 (8.4ms) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0010 110

789.8 ms after::(plys song) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

797.3 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

.....

last packet with #:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0001 0000 0000 0000 0000 0001 0101

4.7 ms after:: (changeto4) 0001 0001 0011 0100 0000 0000 0001 0000 0111 (24.07ms) 1001 1111 0000 0000 1000 1110 0110 0111 (8.71ms) 1001 1011 1001 0100 0000 0000 0001 0000 0000 0000 0000 0001 0000 (9.8ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011

228 ms after:: 1001 1111 0000 0000 1000 1110 0110 0111 (8.4ms) 1001 1011 1101 0100 0000 0000 0001 0000 0000 0000 0001 1100

763.11 ms after:: 1001 1111 0000 0000 0110 0100 0101 0010 (9.65ms) 1001 1011 1011 0100 0000 0000 0001 0000 0000 0000 0001 1110 (9.46ms) 1001 1111 0000 0000 1000 0010 0101 1010 (9.84ms) 1001101111000100000000000001000000000000000011011

436.4 ms after:: 1001 1111 0000 0000 1000 0010 0110 1011 (8.71ms) 1001 1011 1000 0100 0000 0000 0001 0000 0000 0000 0001 1111

115.5 ms after:: 1001 1111 0000 0000 1010 1100 0110 0111 (8.71ms) 1001 1011 1001 0100 0000 0000 0000 0000 0000 0000 0001 1111 (8.71ms) 1001 1111 0000 0000 1010 1100 0110 0111 (8.52ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

963.5 ms after:: 1001 1111 0000 0000 1010 1100 0110 0111 (8.86ms) 1001 1011 1001 0100 0000 0000 0001 0000 0000 0000 0001 0000 (9.57ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011

106.3 ms after:: 1001 1111 0000 0000 1010 1100 0110 0111 (8.47ms) 1001 1011 1101 0100 0000 0000 0101 0000 0000 0000 0001 0000

740.65 ms after:: 1001 1111 0000 0000 1000 0001 0101 1011 (8.70ms) 1001 1011 1011 0101 0000 0000 0001 0000 0000 0000 0001 1101 (8.70ms) 1001 1111 0000 0000 1010 0000 0101 1010 (9.39ms) 1001 1011 1100 0101 0000 0000 0001 0000 0000 0000 0001 1100

454.37 ms after:: 1001 1111 0000 0000 1010 0000 0110 1011 (8.93ms) 1001 1011 1000 0101 0000 0000 0001 0000 0000 0000 0001 0000

125.3 ms after:: 1001 1111 0000 0000 1101 0001 0110 1101 (8.02ms) 1001 1011 1001 0101 0000 0000 0000 0000 0000 0000 0001 0000 (9.61ms) 1001 1111 0000 0000 1101 0001 0110 1101 (8.47ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2 secs after:: 0001 1000 1010 (22ms) 1001 1000 0010

403.8 ms after:: 1001 1111 0000 0000 1101 0001 0110 1101 (8.47ms) 1001 1100 0101 0000 0001 0001 0001 0100 0101 0101 0011 0111 0010 (9.84ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.24ms) 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0001 0101

354.8 ms after:: 1001 1111 0000 0000 1101 0001 0110 1101 (8.47ms) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0110

807.29 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

809.12 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

.....

last packet with #:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0111 0000 0000 0000 0001 0011

374.7 ms after::(changet06) 0001 0001 0011 0110 0000 0000 0001 0000 0101 (18.7ms) 1001 1111 0000 0000 1101 0001 0110 1101 (8.76ms) 1001 1011 1001 0110 0000 0000 0001 0000 0000 0000 0000 0001 1110 (9.98ms) 1001 1001 0010 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0011

182.6 ms after:: 1001 1111 0000 0000 1101 0001 0110 1101 (8.52ms) 1001 1011 1101 0110 0000 0000 0001 0000 0000 0000 0001 1010

785.48 ms after:: 1001 1111 0000 0000 1001 1111 0101 0110 (9.98ms) 1001 1011 1011 0110 0000 0000 0001 0000 0000 0000 0001 0000 (9.74ms) 1001 1111 0000 0000

1011 1101 0101 0110 (9.98ms) 1001 1011 1100 0110 0000 0000 0001 0000 0000 0000 0001 1001

445.82 ms after:: 1001 1111 0000 0000 1011 1101 0110 0111 (8.61ms) 1001 1011 1000 0110 0000 0000 0001 0000 0000 0000 0001 1101

138.69 ms after:: 1001 1111 0000 0000 1111 0000 0110 0000 (8.57ms) 1001 1011 1001 0110 0000 0000 0000 0000 0000 0000 0001 1101 (8.76ms) 1001 1111 0000 0000 1111 0000 0110 0000 (9.79ms) 1001 1001 0001 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0010

2 secs after:: 0001 1000 1010 (20.93ms) 1001 1000 0010

376.44 ms after:: 1001 1111 0000 0000 1111 0000 0110 0000 (9.79ms) 1001 1100 0110 0000 0001 0001 0100 0101 0011 0101 1001 0111 1011 (8.68ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.24ms) 1001 1001 0101 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0000 0001 0101

104.18 ms after:: 1001 1111 0000 0000 1111 0000 0110 0000 (9.57ms) 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001 0110

805.19 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0001 0000 0000 0000 0001 0101

789.6 ms after:: 1001 1001 0100 0000 0001 0000 0001 0000 0000 0000 0010 0000 0000 0000 0001 1000

hookingup.tla

first sequence:: 0001 1000 1010 (11.7ms) 1001 1000 0010

10.91 ms after:: 0001 0001 0001 1000 0001 1001 (13.71ms) 1001 1111 0000 0000 0101 0001 0100 0111 (8.67ms) 1001 1100 0001 0000 0001 0001 0101 0111 0111 0010 0101 0111 0010 (9.51ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.39ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.67ms) 1001 1111 0000 0000 0101 0001 0110 0101 (8.39ms) 1001 1100 0001 0000 0001 0001 0101 0111 0111 0010 0101 0111 0010 (9.79ms) 1001 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0100 (9.79ms) 1001 1101 0000 0000 0000 0000 0000 0000 0000 0000 0101 (8.39ms) 1001 1011 1001 0001 0000 0011 0000 0000 0000 0000 0000 0001 1001 (8.67ms) 1001 1001 0001 0000 0011 0000 0001 0000 0010 0010 0000 0000 0000 0000 0001 0011

240 ms after:: 0001 0001 0001 0100 0000 0110 (15.39ms) 1001 1001 0001 0000 0011 0000 0001 0000 0010 0010 0000 0000 0000 0000 1001 1011

751.89 ms after:: 1001 1111 0000 0000 0101 0001 0110 0101 (8.79ms) 1001 1001 0010 0000 0011 0000 0001 0000 0010 0010 0000 0000 0000 1001 1010

2 secs after:: 0001 1000 1010 (12.33ms) 1001 1000 0010