Project Number: EE-WZM-1B96

DACS: The Distributed Audio Control System

A Major Qualifying Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Stephen Scott Richardson

Date: 25 July 1997

Approved

Professor William R. Michalson, Advisor

Abstract

The Distributed Audio Control System (DACS) is a hybrid digital and analog system designed to automate common audio tasks such as audio mixing, digital audio cueing, and MIDI device control. The MQP explores the development of a powerful, expandible, and inexpensive system to perform these tasks. The implementation of two custom hardware components, firmware, and software bring the design to fruition, with near production quality.

Acknowledgements

- **Michael Andrews**, for PCB proofing, ideas, and assistance with presentation.
- **J. Nelson Chadderdon**, for LCD backlight inverters and rail of LM1972's.
- **Seann Ives**, for selling me the HP logic analyzer.
- **Professor William Michalson**, for advising the project.
- **Yeasah Pell**, for loan of oscilloscope, laser printer use, random parts, loan of speakers for presentation.
- **Mark and Paula Richardson**, for funding the project.
- My suitemates, Yeasah and Dave, for putting up with the hardware lab in the living room and the testing/software lab in the bedroom.
- All of my friends (Kim, Sarah, Steph, Mike, Yeasah, Todd, and everyone else that I forgot) for their support throughout this project.
- **WPI Lens and Lights**, for the loan of a pair of EAW JF-60 speakers and tripods for the presentation.

Contents

List of Figures

List of Tables

1 Executive Summary

1.1 What Does the System do?

The Distributed Audio Control System (DACS) provides a means for automating common tasks performed in various audio engineering contexts. Automated mixing, compact disc audio cueing, digital audio control, and MIDI device control can be automated using the DACS. This allows complex sequences of events to be controlled with an ease not previously available.

DACS provides these automation capabilities in two distinct ways. Integration with existing off-the-shelf MIDI software provides a means for adding automation to a home-studio environment with relative ease. Additionally, a custom software environment allows development of audio scripts which can be cued manually or synced to a time source. This capability means the DACS is also equally at home in a broadcast audio or theatre audio setting.

1.2 Why Design and Implement this System?

First and foremost, there are no systems on the market currently that offer as many features as DACS for the projected cost. This means that powerful capabilities can be brought within reach of those working with a modest budget.

Secondly, the features of DACS are of use to me, personally. Much of my life revolves around audio work in studio, theatre, and live settings. This project is an outlet for ideas I have had for ways of improving capabilities in these areas.

1.3 Project Goals

When the project began, a general set of goals was established. First, it was desirable to design a practical, marketable product. To do this would require careful balance of quality, affordability, and expandability. Secondly, and most importantly, it was decided that the system would be implemented, producing a product as near to production quality as possible.

These goals were very nearly met. The hardware is functional, with only some performance issues left to conquer. Firmware and software still require additional development time to implement the complete system design, but early tests show that the system has a lot of potential.

1.4 Deliverables

This MQP has produced the following materials:

- **DACS Model 411**, an automated, self-contained line-level mixer in a 3-space rack-mount form factor. The prototype unit supports 32 balanced or unbalanced inputs, eight balanced outputs, and eight unbalanced outputs. The prototype can mix 32x4, 16x8, 8x16, or 4x8x4 channels, and is expandible to 32x16 capability.
- **DACS Model 112**, a self-contained system control board. Traditional audio controls such as faders and buttons are presented, with the addition of liquid crystal and LED displays. The board provides a powerful, intuitive way to use and program the components of the DACS.
- **Linux backend software,** `cdaudio_daemon`, a software component to provide integration of the audio functionality of computer-hosted compact disc drives with DACS. TCP/IP is employed as a means of communication with other DACS elements, furthering the distributed concept.
- **Linux backend software,** `midictrl`, a software component that brings DACS support to off-the-shelf MIDI packages such as *Cakewalk Pro Audio*. The software performs appropriate translations to allow bidirectional interoperation between MIDI devices and the subsystems of DACS.

- **Design Specification,** this document. This document details the design and development of both the completed and uncompleted portions of the project. The design sections for firmware and software are obvious starting points for expansion and improvement of the project.

1.5 Conclusions

For me, DACS is a project of longer-term than an MQP can provide. The MQP has provided a solid foundation upon which the remaining unimplemented firmware and software designs may stand. Even though the project did not meet the initial goals to the letter, it still seems that they were well met. Given the scope of the work completed on the project, and the degree to which it is complete, this makes sense.

DACS is a project that, at least in concept, began for me over two years ago. Considering the potential I see for this system, it is quite likely that development will continue beyond the scope of the MQP. The technology of DACS is viable, affordable, and directly applicable to many applications. Over time, it is hoped that it can be evolved into a marketable product.

The following is a list of known problems or incomplete tasks: (note: not all of these may make sense until this design document has been read through.)

- There is a noise floor approximately 50dB down in the mixer unit. This has not been investigated thoroughly, but it is suspected that it may be due to a grounding issue with the SSM2163 mixer chips.
- Digital "hash" noise can be heard in the outputs of the mixer unit during periods of heavy microcontroller activity. This is likely due to the complete lack of shielding in the unit.
- The bus combining/switching logic in the mixer unit is noisy during switching periods. This is heard as a loud "pop" on the outputs of the unit. It is suspected that this is due to the large switching currents used in the relays, and/or residual DC offsets between buses.
- The mis-designed input trim section requires a re-design and engineering fix. This is a relatively important piece of functionality that is missing in the prototype mixer unit.
- The microcontrollers chosen for the task may not be sufficiently powerful. Early tests are inconclusive, but it is a distinct possibility that a more powerful microcontroller needs to be used.
- The firmware and software are still at a relatively immature stage, from an implementation point of view. For the system to truly be useful, these areas need to be addressed.
- The control board requires another LCD, three buttons, and a data wheel to be mounted. Painting and silkscreened artwork also need to be completed.

This may seem like a large list, but given the sheer number of things that went *correctly*, it is a relatively small list. These items are among the first that will be addressed as the project progresses beyond being an MQP.

2 Introduction

This project represents a complete re-thinking and re-engineering of an independent study project completed between March and May of 1995 by myself and fellow WPI student and long-time friend Michael Andrews. The original project saw the development and implementation of a simple computer-controlled mixer, combined with software for controlling CD-ROM drives and PC sound cards. A simple text-based front end was written to facilitate execution of user-cued events at the push of a button. This system was developed and used in three large theatre productions, serving as the audio control system for all of the sound effects.

While the original system functioned reasonably well for the time frame in which it was developed (approximately *6 weeks*!), it had several shortcomings that needed to be addressed. It was clear in my mind that the product was viable and useful technology for theatre audio applications. Research at the time showed no products with similar capabilities in our price range, and real-world tests showed that the system removed much of the human error inherent in manually-run audio.

A year and a half passed with no major improvements to the original system. When it came time to chose an MQP, it seemed natural to take some concepts from the original system and develop an entirely new system under a more reasonable development schedule. It was clear from the outset of the project that, while the development time frame was more reasonable than the original, the project was still extremely ambitious, especially for a one-person hardware, firmware, and software design and development team.

The first term was spent largely doing overall system design and schematic work. PC board design, chassis fabrication, and PC board stuffing were completed during the second term. Term three largely involved systems testing and integration, firmware and software design and coding, and preparation for project presentations. The final term, an optional one, was used to re-design much of the software, begin some software implementation, and to wrap up loose ends of the project, such as completing the documentation. By no means do I consider the project to be completed. I intend to develop it further, to expand the capabilities and quality of the overall system.

2.1 Design Methodologies

The remainder of this document presents the design and implementation of the hardware, firmware, and software that comprise the DACS. For most design work, a top-down methodology was chosen, except in cases where it made sense to work from the low-level and the high-level simultaneously. Individual hardware and software component specifications were derived from an overall system design, which was formed using a *tabula rasa* approach. Essentially, this involved starting with a clean slate rather than preconceived notions from any previous project work. Only after some preliminary design work was done with this method were learnings from previous work introduced. The resulting synthesis was then checked for practicality. The whole process was iterated until a reasonable design was achieved. This process was carried out at various levels of complexity, right down to the component level in hardware, and code level for firmware and software.

At several points down the development chain, it made sense to re-evaluate goals and specifications set at a higher level. In some instances, it was realized that some high-level goals could not be achieved due to problems at lower levels. In most cases, though, an attempt was made to temper high-level goals and specifications with enough real-world knowledge to avoid any major problems or re-working of the original designs.

At the highest level, the two custom pieces of hardware ended up, in the end, to be very similar to what was originally envisioned. Early renderings, shown in later sections, very closely match the final products. This seems to be a good indication that the high-level design goals were set ambitiously (due to the amount of work it took to get the products to that point), but not unrealistically (due to the fact that it actually *is* at that point).

2.2 Application Analysis

In order to properly form a set of specifications for the components of DACS, a set of potential applications of the system was derived. Four contexts were chosen: a home studio setup, a broadcast application, a theatre application, and a live sound application. Figures 1, 2, 3, and 4 show schematic representations of these possible configurations. In each figure, shaded boxes indicate custom hardware, developed in this MQP. Unshaded boxes indicate off-the-shelf hardware. Custom software runs on each of the computers present in the diagrams. The diagrams have since gone through many iterations since the initial diagrams, hence now reflect actual details of the DACS implementation. Initially, several different diagrams were drawn, out of which these evolved.

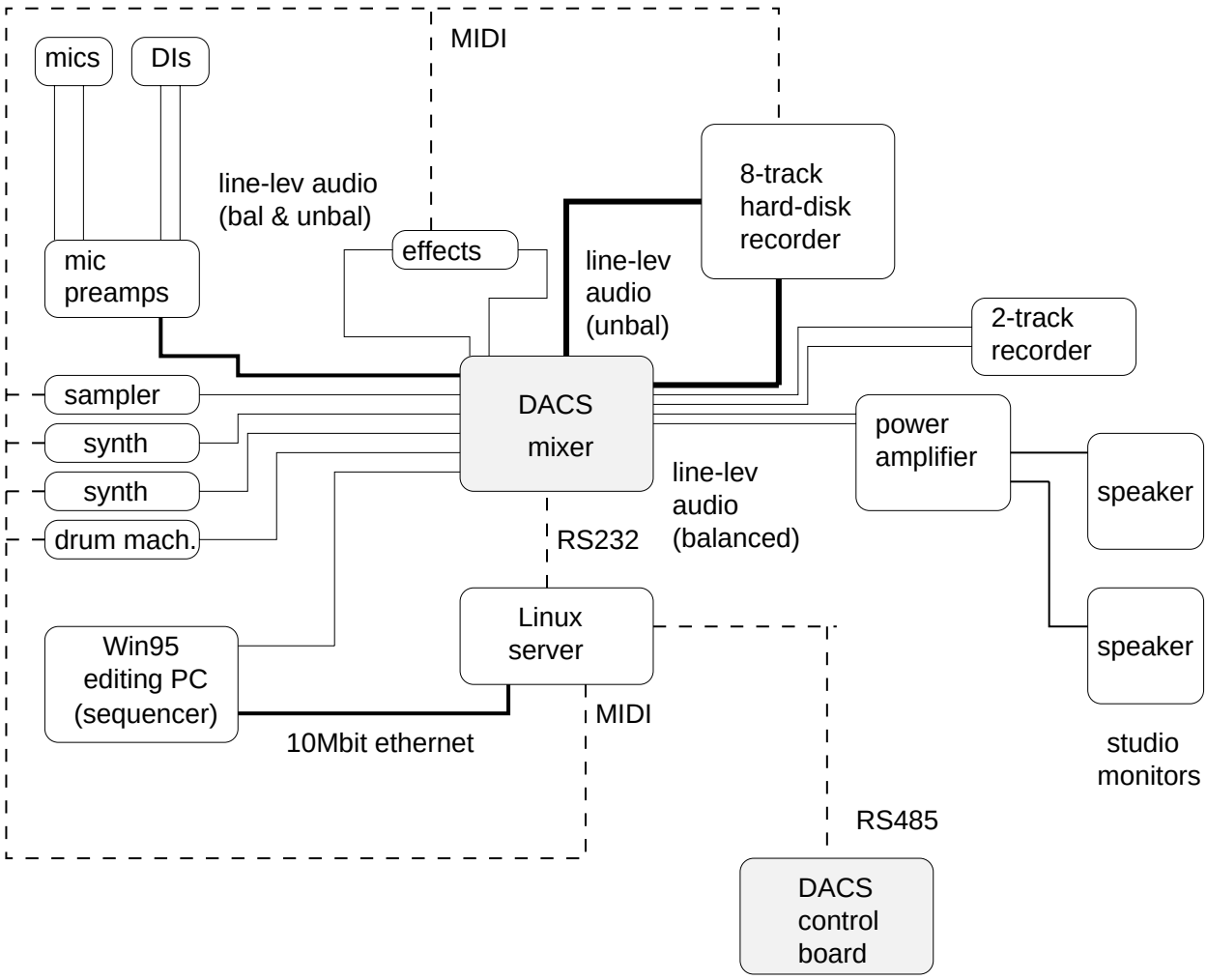


Figure 1: Typical DACS-based home studio configuration.

Through studying potential applications and applying what was learned with the previous system, a rough set of specifications was derived. A partial list is shown below:

- Custom audio hardware should support balanced and unbalanced line-level signals.
- Custom audio hardware should be of semi-professional studio quality, dictating relatively low noise floor and THD specifications. This is an important improvement over the original system, which had noise problems.

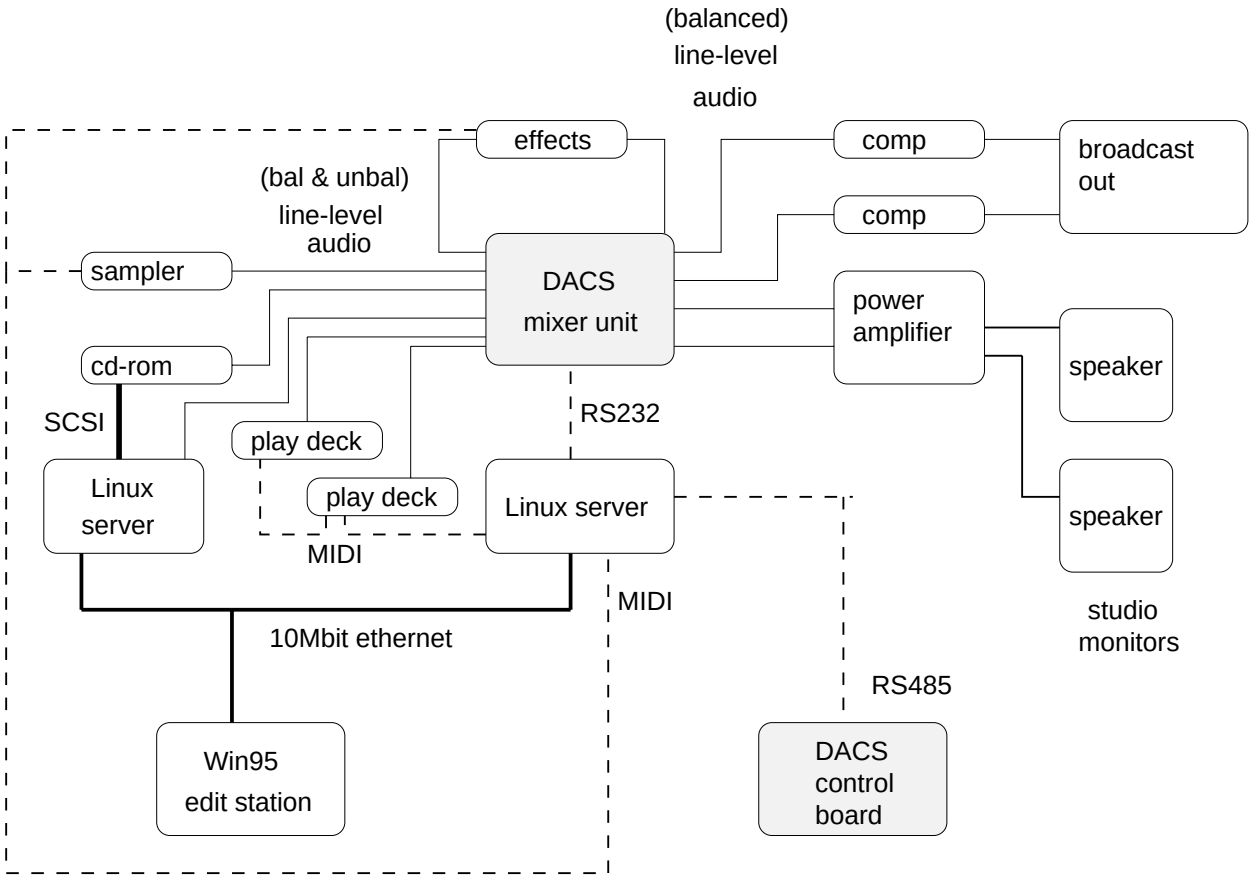


Figure 2: Typical DACS-based broadcast application.

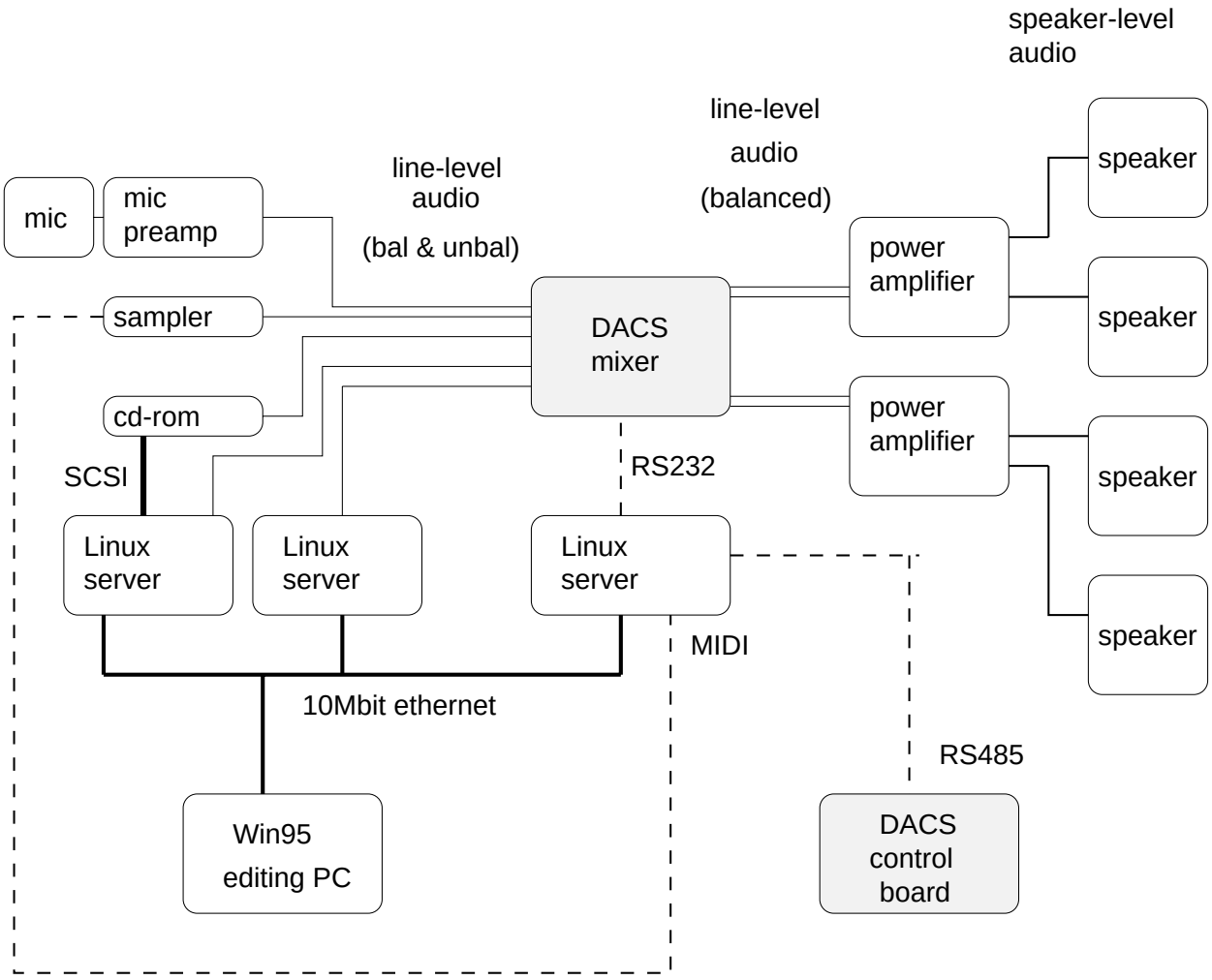


Figure 3: Typical DACS-based theatre audio application.

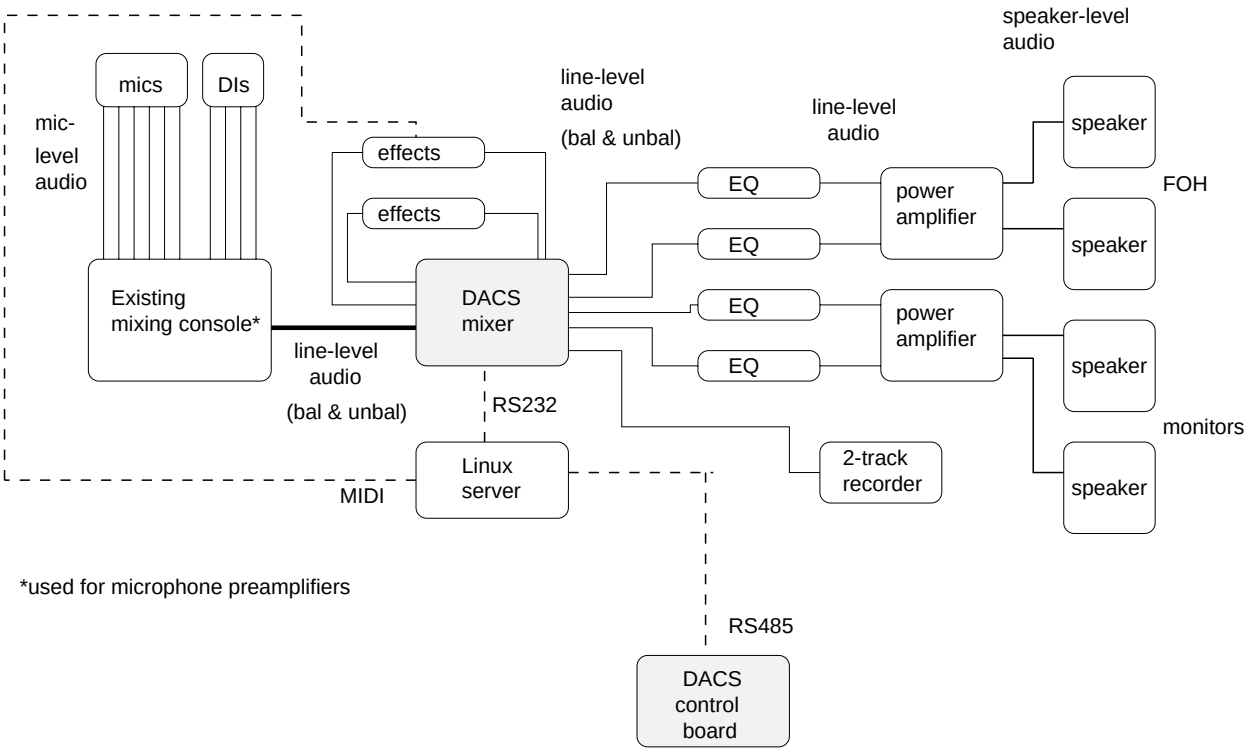


Figure 4: Typical DACS-based live sound application.

- Integration with existing MIDI systems would be beneficial, both in terms of functionality and in terms of reduced time expenditure. This integration would not only include MIDI hardware, but existing MIDI sequencing software.
- If possible, existing audio control standards should be used, to provide compatibility between devices.
- Some applications are set in an environment where syncing to SMPTE time codes would be beneficial for automation purposes.
- Overall, the system needs to be flexible, and dynamically expandible and configurable.
- An intuitive, lucid way of programming and controlling the system must be created. A major shortcoming of the original system was the lack of a decent user interface.
- The use of inexpensive off-the-shelf hardware, such as CD-ROMs for compact disc cueing, provided a significant cost advantage in the first system. This concept should be applied to the DACS.

The two pieces of custom hardware, the DACS mixer and the DACS control board, were settled upon as reasonable devices to design and prototype for the MQP. Other pieces of application-specific hardware were considered, but these two pieces of hardware were applicable to all of the applications considered in the analysis. The DACS mixer is the core of the system, thus it was quite clear that its implementation was necessary. The board was implemented because it is believed that it brings a significant improvement to the usability of the system. The remaining DACS components, firmware and software, were developed to the extent that time permitted.

2.3 System Overview

Figure 5 depicts the logical hierarchy of the DACS. Integration of custom and off-the-shelf hardware is accomplished through the means of custom software components. Network transport layers promote the concept of distributed task servicing. DACS *service providers* appear as services on hosts located on a network, or on a single host (using the internal loopback as a virtual network device). Front-end software accesses individual devices through a series of abstraction layers, such that the actual location and physical characteristics of the underlying hardware providing the service need not be known by that software.

This abstraction paradigm allows generic descriptions of actions to be created which do not depend on the underlying hardware. This would allow, for example, the cues for a touring theatre show to be defined generically, and run on almost any DACS configuration. One can even imagine a CD-ROM with a generic DACS audio script (with all supporting sound effects, etc.) being sold with the script for the play itself.

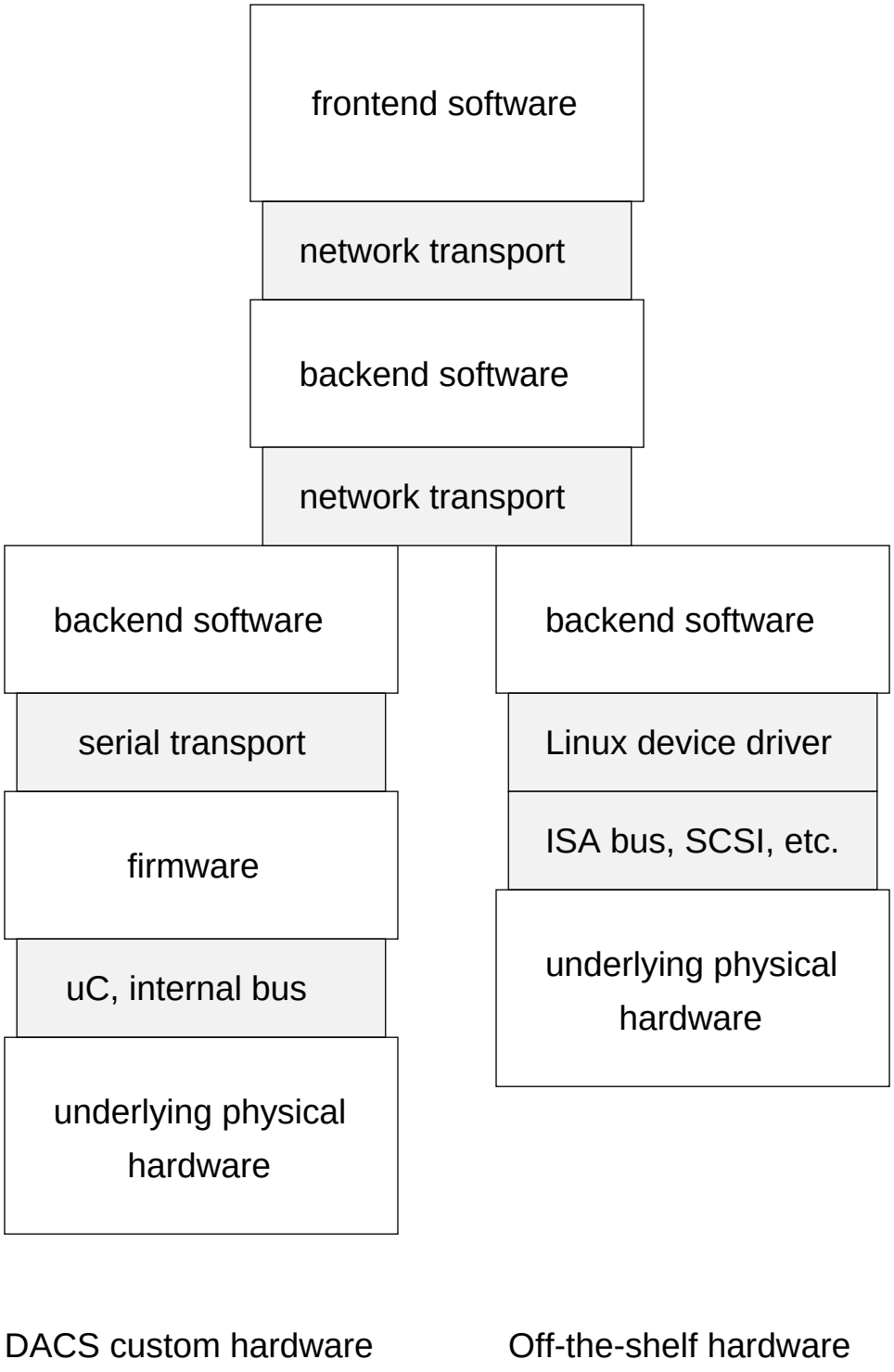


Figure 5: DACS component hierarchy, generalized view.

3 High-Level Hardware Specifications

The high-level hardware specifications are intended to provide an overview of the hardware components developed for the DACS. Decomposition of these specifications is provided in later sections, to the schematic, PC board and component levels.

Some of these specifications have changed over time in a constant attempt to satisfactorily balance the goals set at the system level and the constraints of time, money, and practicality.

3.1 Control Board

The control board is a microcontroller-based remote control unit for the various functions of the DACS. The board shall provide familiar user interfaces for audio mixing, transport, and cue control. In addition, a multi-purpose interface element shall be included, to provide an interface for functions that do not fit into the paradigms mentioned above. Additionally, this interface mechanism shall provide an easily adaptable means for future expansion.

3.1.1 System Description

The user shall be able to control the mixing features of the DACS mixer module via familiar slide potentiometers (faders). The values of these potentiometers shall be digitized with an 8-bit A/D converter, and processed appropriately via the internal microcontroller. Because "flying faders" will not be used on this board due to cost, each channel shall have two LED indicators associated with it to aid in adjusting channel values. The "active" indicator shall light if a particular channel is active in the current mixer/matrix mode. The "grab" indicator, used when editing mix settings, shall light green when the fader has been moved from the current setting, and red when the old value has been "grabbed".

The user shall have control over the channel assignment features of the DACS mixer via a set of pushbutton switches and associated LEDs. The active output channels for a given input channel shall be toggled with these switches. The input channel that is being controlled shall be selectable via two pushbutton switches. An LED display shall provide visual indication of the active channel number.

The user shall be able to control the transports of various physical or virtual audio devices. An interface shall be provided to allow the user to select which device to work with, via a pair of pushbutton switches. Visual indication of the current transport shall be provided via a 16 character alphanumeric LCD. The transport controls shall allow the track index and time index to be adjusted. Track index shall be adjusted via a pair of pushbutton switches. Time index shall be adjusted via a detented rotary encoder wheel. "Play," "Cue," and "Stop" functions shall also be available to the user, via pushbutton switches with LED indicators for visual reference.

The user shall be able to execute a series of pre-programmed cues from the board. These cues may include, but are not limited to, mixer events, transport events, MIDI events, etc. In addition, rudimentary cue stack editing shall be allowed from the board. Cue information shall be presented to the user via a 40x2 LCD. Three buttons shall be used to navigate and execute the cue stack.

The unit shall incorperate an appropriate internal power supply. External connection to this supply shall be made via a standard IEC power connector at the rear of the unit. This power entry unit shall be of the type that filters EMI and RFI interference. It shall be appropriately fused and switchable via a rocker switch.

The board's microcontroller shall communicated with the rest of the DACS via an RS485 4-wire serial interface. This interface shall be presented as a 5-pin male XLR connector, and shall be wired per the DMX512 lighting control specification. This connector was chosen due to its robustness. The adherence to the DMX512 specification merely eliminates any potential equipment malfunction due to accidental interconnection, as DMX512 uses the same connectors,

and will often be present in places this mixer will be used. The serial communication shall take place at 9600 baud.

The unit shall be referred to as **DACS Model 112 - System Control Board**.

3.1.2 Specification Sheet

```
CONNECTORS

 RS485 control ........... :  (1) 5-pin male XLR/Cannon,
                                  panel-mount
 expansion ............... :  (1) 25-pin female D-sub
 power ................... :  (1) IEC, with EMI/RFI filter & fuse


CONTROLS

 channel faders .......... : (16) 2-3/8 inch (60 mm) travel
                                  linear taper slide potentiometers
 multi-function .......... :  (2) tactile pushbutton switches,
                                  PC board mount
                              (1) detented encoder wheel

 output assign ........... : (16) tactile pushbutton switches,
                                  with LED, PC board mount
                              (2) tactile pushbutton switches,
                                  PC board mount
 transport ............... :  (4) tactile pushbutton switches,
                                  PC board mount
                              (3) tactile pushbutton switches,
                                  with LED, PC board mount
                              (1) detented encoder wheel
 cue stack controls ...... :  (3) tactile pushbutton switches,
                                  PC board mount

INDICATORS & DISPLAYS

 channel value grab ...... : (16) T1-3/4 red/green bicolor LEDs
 channel active .......... : (16) T1-3/4 green LEDs
 output channel assign ... : (10) green LEDs
                              (1) dual 7-segment green LED display
 transport function ...... :  (3) green LEDs
 RS485 link data ......... :  (1) T1-3/4 green LED
 transport track & time .. :  (3) dual 7-segment green LED displays,
 multi-function display .. :  (1) backlit LCD,
                                  40x8 alphanumeric characters,
                                  plus 240x64 graphics capability
 transport control display :  (1) backlit LCD,
                                  16x1 alphanumeric characters
 cue-stack control display :  (1) backlit LCD,
                                  40x2 alphanumeric characters

ENCLOSURE
```

```
dimensions .............. :  19 inches (482mm) wide (EIA rack)
                            14 inches (356mm) tall (EIA 8 RU)
                             4 inches (102mm) deep max.
top and rear panel ...... :  steel (acrylic on prototype)
bottom, top and side .... :  steel (aluminum on prototype)
color ................... :  satin black
external component layout :  as per layout diagrams, below.
panel artwork ........... :  single-color silkscreen, white,
                            laid out as below.


CONSTRUCTION

   All components shall be PC-board mount style, except
where not possible. Liquid crystal displays shall be covered
with transparent acrylic or equivalent material.  Faders and rotary
encoders shall be mounted to the case for additional durability.


MARKETING

target retail price ..... :  $899 for base unit


additional modules ...... :  16-fader expansion module
                            16-channel meter bridge

                            modules connect to expansion port
```

3.1.3 Panel Layout

The panel layout for the control board is shown in figure 6.

3.1.4 Concept Rendering

Figure 7 shows the original concept rendering of the control board.

3.2 Mixer Unit

The mixer unit is a microcontroller-based module that shall provide the capability to mix line-level audio signals in an automated fashion. The unit shall be able to mix any of its active inputs to any of its active outputs at any level, within the attenuation range of the mixer. The signal architecture of the mixer/matrix shall be user-reconfigurable, allowing many input channels to be mixed down to few outputs, few inputs to be mixed to many outputs, or an equal number of inputs to be mixed to an equal number of outputs.

3.2.1 System Description

In stock configuration, the unit shall provide thirty-two line-level audio inputs capable of receiving balanced or unbalanced audio signals. Eight balanced and eight unbalanced line-level audio outputs shall also be present on the rear panel of the unit. These outputs may be converted to all balanced or all unbalanced merely by changing modules. Depending on the mode and hardware configuration of the mixer, all inputs and outputs may not be available at a given time.

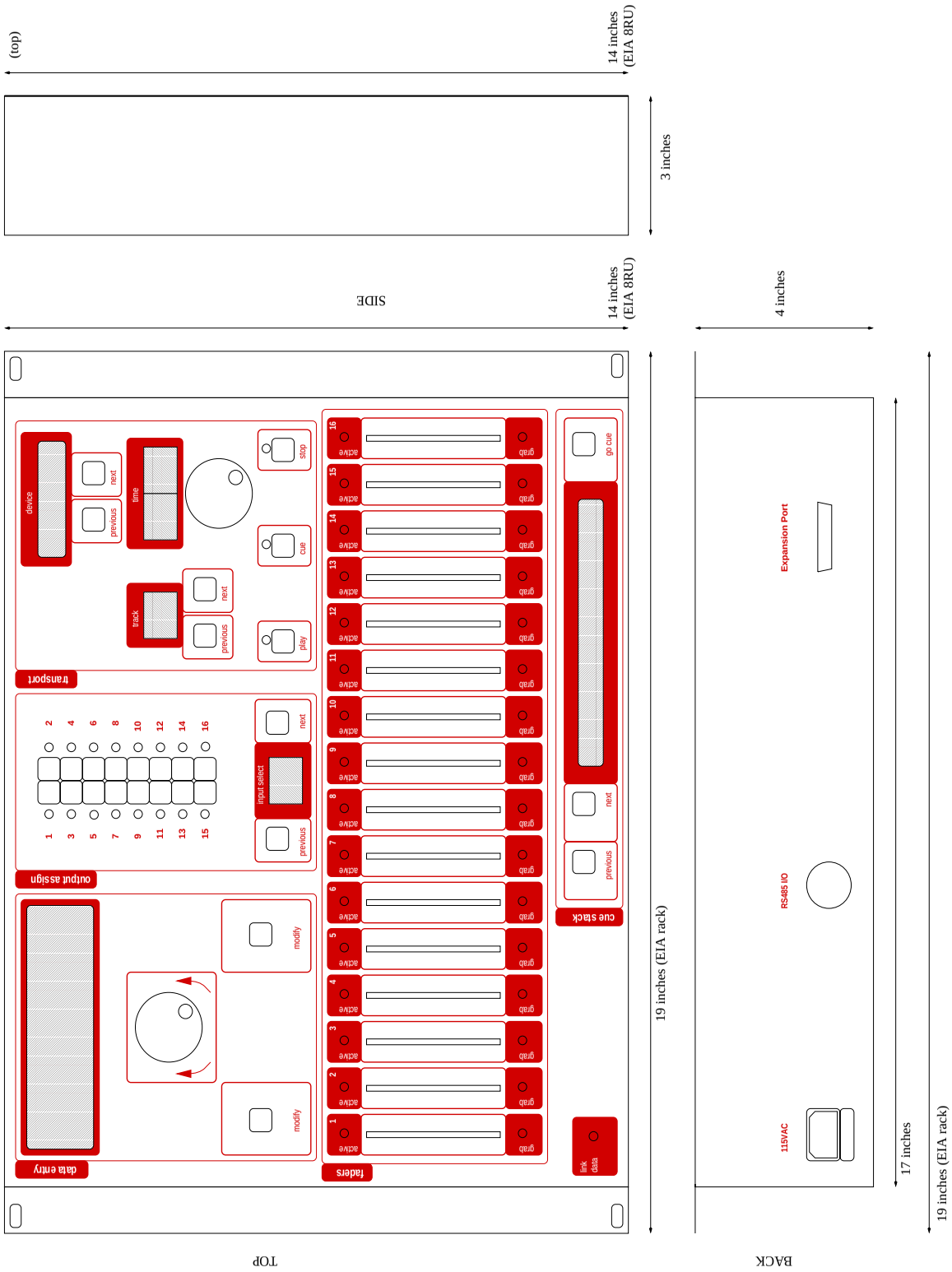


Figure 6: Control board, panel layout and silkscreen artwork.

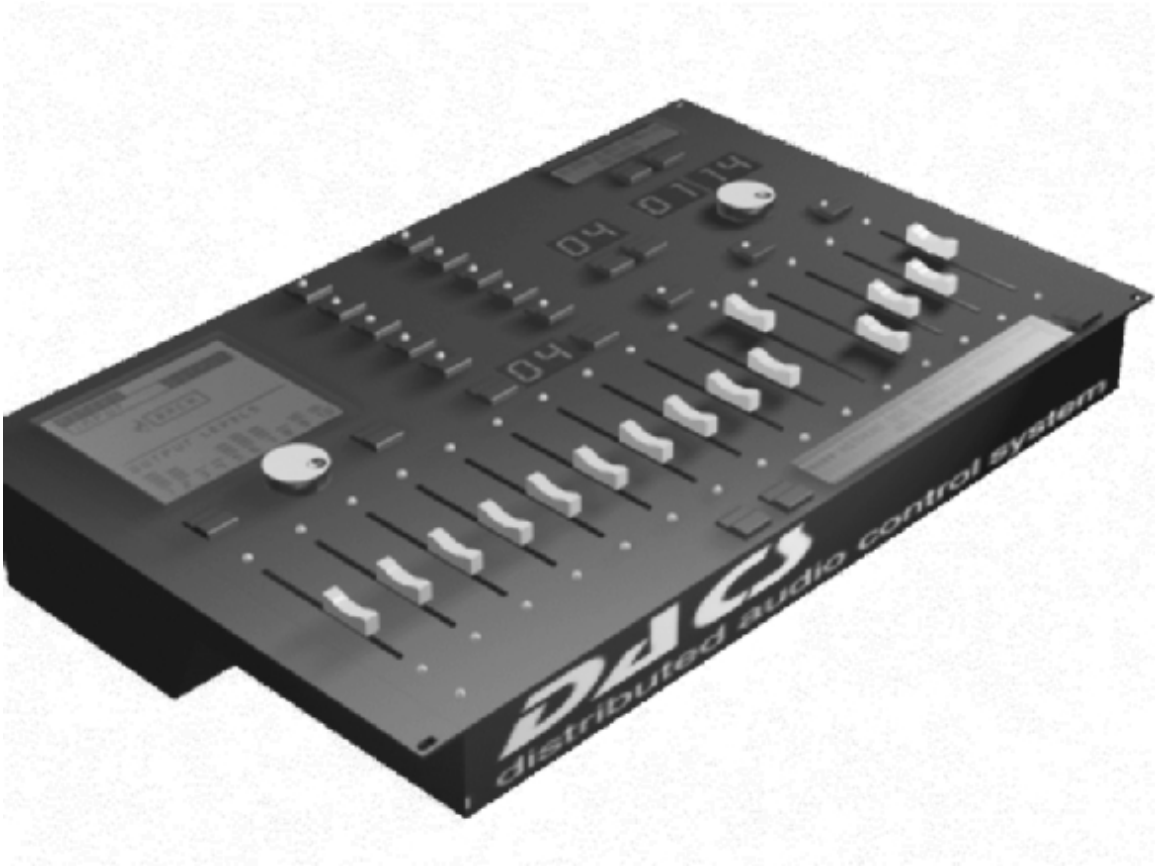


Figure 7: Control board concept rendering.

The digitally-controlled mixer circuitry shall behave in a logarithmic fashion, to match the response of the human ear. The mixer shall provide from 0dB to 63dB of attenuation capability, in 1dB steps. Additionally, near-infinite attenuation ("mute") shall be provided for each input channel. This dictates that the digital controller for the audio mixer shall have at least 6 bits of resolution.

Digitally controlled switches shall be used to reconfigure the audio paths of the mixer. These switches shall be of the "clickless" type, such that signals may be re-routed without any undesirable noise entering the system.

The unit shall incorperate an 8-bit A/D converter and appropriate analog multiplexer and driver circuitry to allow monitoring of all input and output signals, after the input buffers and before output drivers. This setup shall form the basis for a digital VU meter. The VU meter shall be capable of reporting signals from -10dBu to +18dBu. The converter shall function linearly, with any necessary logarithmic conversion taking place in software. This VU data can be displayed on the mixer unit itself via its LCD and/or transmitted over the microprocessor's serial uplink, to be displayed on a host PC or the DACS control board.

Mixer modes and other relevant information shall be displayed to the user via an LCD panel on the front of the unit. A single LED shall indicate serial link activity. Another LED shall indicate that the unit is powered.

The unit shall be constructed in a modular fashion, such that future upgrades are possible by simply adding or changing cards. The choice of output channel boards shall be left to the user, allowing a mixer with all balanced, all unbalanced, or half balanced and half unbalanced outputs to be constructed. In addition to the upgrade capability, the PC boards shall be designed such that a larger model mixer unit can be built without re-engineering the electronics. Future upgrade options may include, but are not limited to: equalization modules, microphone preamplifier modules, DSP effects engines, and digital audio interfaces (for AES/EBU and S/PDIF-compliant digital audio devices).

The unit shall incorperate linear internal power supplies. Switching power supplies are unacceptable in this unit due to the large quantity of electrical noise they create. Separate power buses for digital and analog circuitry shall be provided, to reduce digital noise in the analog portions of the mixer. These supplies shall be sufficiently capable of powering add-on modules. External connection to these supplies shall be made via a standard IEC power connector at the rear of the unit. This power entry unit shall be of the type that filters EMI and RFI interference. It shall be appropriately fused and switchable via a front-panel rocker switch.

The mixer unit's microcontroller shall communicate with the rest of the DACS via an RS232 3-wire serial interface. This interface shall be presented as a 9-pin female D-subminiature connector, and shall be wired per the RS232 9-pin Data Communications Equipment (DCE) spec. This allows direct connection to a host PC with a straight-through cable. The serial communication shall take place at 19200 baud.

The unit shall be referred to as **DACS Model 411 - Modular Automated Audio Mixer**.

3.2.2 Specification Sheet

```
AUDIO INPUT AND OUTPUT

 Input maximum level ........ : +24dBu
 Input impedance ............ : 10K ohms or greater
 Output maximum level ....... : +24dBu
 Output impedance ........... : 100 ohms or less


FREQUENCY RESPONSE

 20Hz to 30KHz or better, +0dB, 3dB down
```

any single input to any single output

NOISE

80dB S/N ratio or better, +0dBu reference input signal,
any single balanced input to any single balanced output

TOTAL HARMONIC DISTORTION

0.05% THD or better,
any single balanced input to any single balanced output

DIGITALLY-CONTROLLED AUDIO MIXER

```
configurations ..... : 8x16, 16x8, 32x4, 4 discrete 8x4 (stock)
                       16x16, 32x8, 32x16 with expansion options
attenuation ........ : 0dB to 63dB
resolution ......... : 1dB (64 steps)
```

DIGITALLY-CONTROLLED INPUT TRIM

```
gain ............... : at least 12dB
resolution ......... : 1dB (64 steps)
```

INTERNAL ANALOG-TO-DIGITAL CONVERTER (FOR DIGITAL VU METER)

```
adc resolution ..... : 8 bits
adc acquisition range ..... : -10dBu to +18dBu +/- 1dB
adc acquisition points ..... : all audio inputs (post-buffers)
                               all audio outputs (pre-drivers)
```

CONNECTORS

AUDIO INPUT MODULES (4 MODULES TOTAL)

```
bal/unbal input module ... :  (8) tip-ring-sleeve 1/4" phone jacks
```

AUDIO OUTPUT MODULES (2 MODULES TOTAL)

```
unbalanced output module . :  (8) tip-sleeve 1/4" phone jacks
balanced output module ... :  (8) panel-mount 3-pin male XLR/Cannon
```

PERMANENT

```
 RS232/422 control ........ :  (1) female 9-pin D-sub, wired DCE
 power ..................... :  (1) IEC, with EMI/RFI filter & fuse


CONTROLS

 power ..................... :  (1) panel-mount rocker switch


INDICATORS & DISPLAYS

 power ..................... :  (1) T1-3/4 green LED
 RS232/422 link data ...... :  (1) T1-3/4 bicolor yel/grn LED
 multi-function display ... :  (1) backlit LCD,
                                   40x2 alphanumeric characters


ENCLOSURE

 dimensions ................ : 19 inches wide (EIA rack)
                              5.25 inches tall (EIA 3 RU)
                              14 inches deep max
 front/rear panel material  : aluminum
 other panel material ..... : steel, top and bottom vented
 module plate material .... : aluminum
 case color ................ : satin black
 module plate color ....... : satin black
 external component layout  : as in diagrams below
 panel artwork ............. : single color silkscreen, white,
                              as in diagrams below


CONSTRUCTION

 Liquid crystal display shall be covered by transparent acrylic
 or equivalent material.  PC board mount connectors shall be
 used for all back panel connections.  These connectors
 shall also be mounted to the panel for additional strength.


MARKETING

 target retail price ... : $2000 for the base unit, including:
                              32 balanced/unbalanced line inputs
                               8 unbalanced line outputs
                               8 balanced line outputs
                               4 quad mixer modules
                                 (8x16, 16x8, 32x4, 4x8x4 mixer modes)



 additional modules .... :  8 output balanced board (replace one existing)
```

```
8 output precision balanced board (replace one)
8 output unbalanced board (replace one)
quad mixer module to expand system
   4 additional brings system to 16x16 and 32x8
   12 additional brings system to 32x16
8 input equalizer board, 3-band
8 input equalizer board, 4-band (2-parametric)
DSP effects module

modules are installed internally and via
  back-panel brackets.
```

3.2.3 Panel Layout

The panel layout for the mixer unit is shown in figure 8.

3.2.4 Concept Rendering

Figure 9 shows the original concept rendering of the mixer unit.

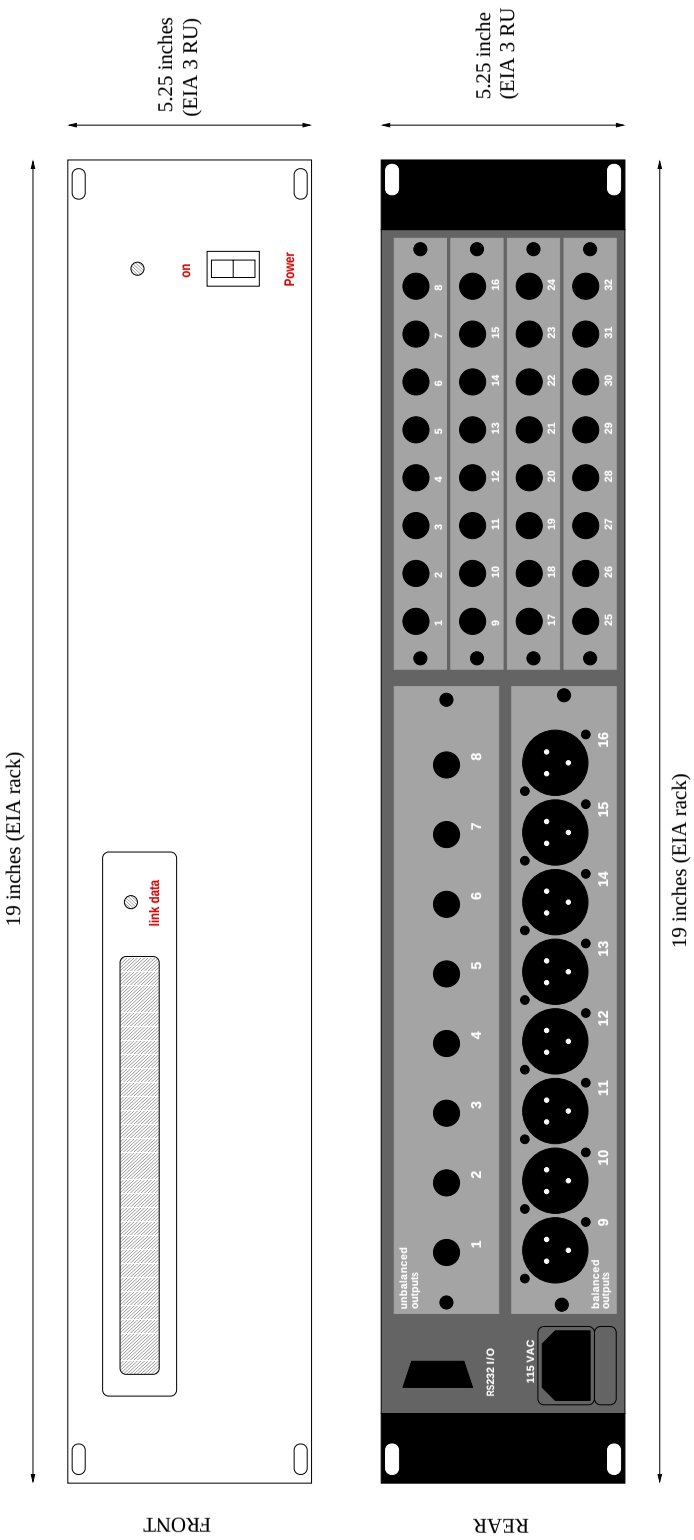


Figure 8: Mixer unit panel layout and silkscreen artwork.

Figure 9: Mixer unit concept rendering.

4 Module-Level Hardware Specifications

4.1 Control Board

4.1.1 System Description and Diagram

The control board is a simple piece of hardware, as it is mainly a variety of input and output devices connected to a microcontroller. Many types of output devices are used, ranging from simple LEDs to graphics-capable liquid crystal displays. Three main types of input devices are used: buttons, datawheels and faders.

Figure 10 shows the overall system diagram. This view depicts the system in terms of each of the input and output elements.

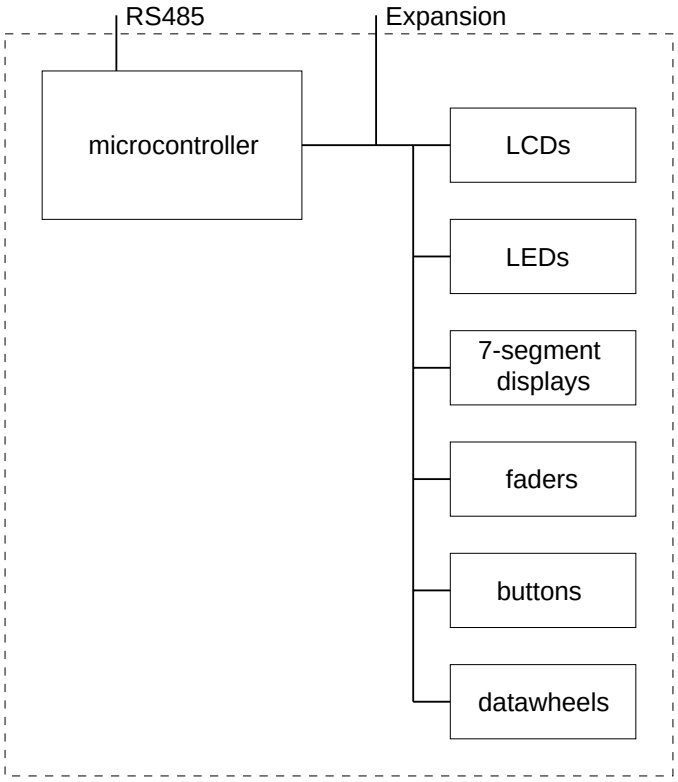


Figure 10: Control board, overall system diagram.

4.1.2 Module Overview

Figure 11 shows the various components of the control board, at the module level. The control board is designed in a modular fashion, such that larger models with dozens of faders, transport controls, etc. can be built. In addition, an expansion port provides the option to connect external add-ons, such as external fader boards and meter bridges.

Unfortunately, there is very little module duplication possible in this piece of hardware. The obvious options were to make one large module, or use the approach presented here. This design was chosen with an eye towards expandability as well as manufacturing cost. Manufacturing one large PC board costs a significant amount of money, whereas some money is saved by the duplication of the largest board, the fader module. It can be argued that the cost of the components required to connect all of the boards together is significant, but in general it seems a wise decision from an engineering standpoint to make smaller modules that can be combined into

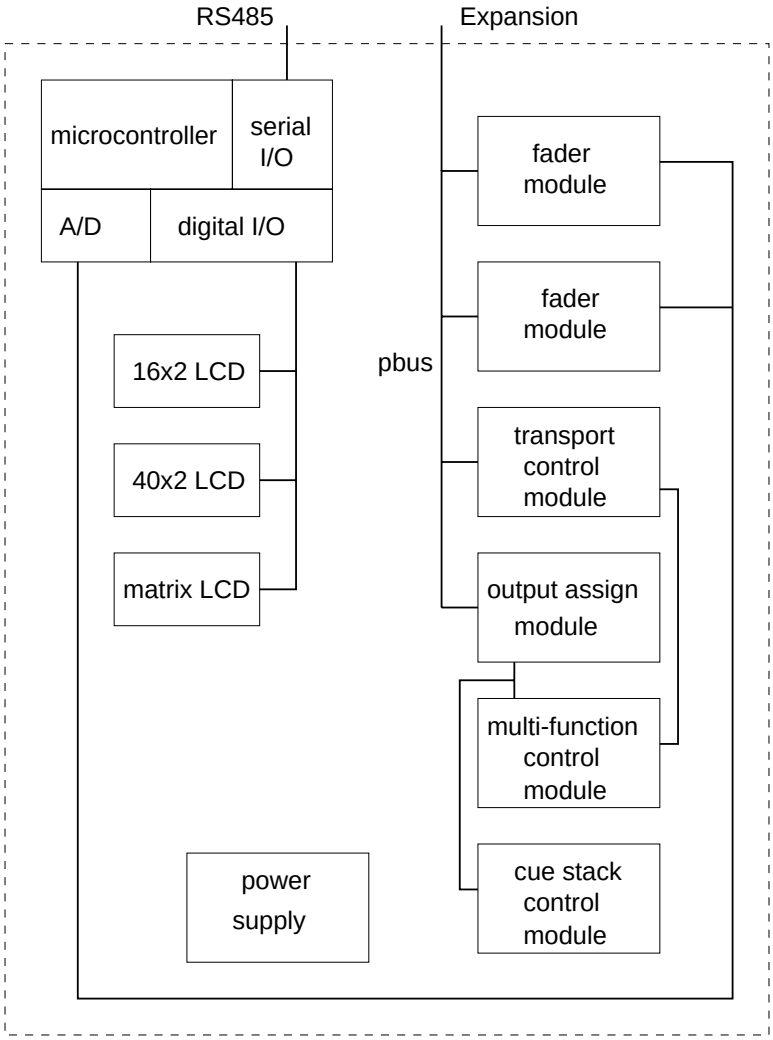


Figure 11: Control board, module-level system diagram.

many different forms, not to mention debugged in a more sane fashion. This technique is often employed in lighting control boards, where different sized boards are built from small, modular building blocks.

4.1.3 Fader Module

Each fader module shall contain eight linear taper potentiometers, sixteen LEDs, and the necessary Pbus interface circuitry. Figure 12 shows the functional diagram of a fader module. The prototype of the control board shall contain two of these fader modules, providing a total of sixteen faders.

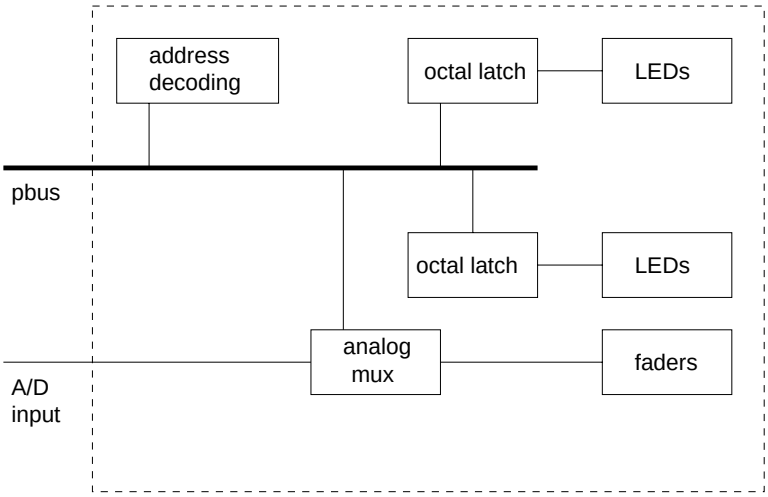


Figure 12: Control board, fader module diagram.

4.1.4 Output Assignment Module

The output assignment module shall contain eighteen momentary buttons, two digits of 7-segment LED displays, sixteen LEDs, and the necessary Pbus interface circuitry. Additionally, it shall provide a means of interfacing the additional buttons of the cue and universal interface sections of the control board. Figure 13 shows the functional diagram of the output assign board.

4.1.5 Transport Control Module

The transport control module shall contain 7 buttons, a datawheel, 3 LEDs, six digits of 7-segment displays, and necessary interface circuitry on a PC board. Additionally, a connection for the datawheel of the universal interface section shall be provided. Figure 14 shows the functional diagram of the transport control board.

4.1.6 Microcontroller Module

The microcontroller module used for in the control board shall be of sufficient computing capability to handle polling of all of the input hardware (buttons, faders, etc.), driving of the three LCD modules, and serial communication at 19.2Kbps.

For the prototype, a pre-made Motorola 8MHz MC68HC11 microcontroller board will act as the embedded processor for the control board. This board contains an RS232 port, an 8-input A/D converter, an I/O port, address decoding logic, and contrast adjustment for liquid crystal displays. Figure 15 shows this microcontroller board.

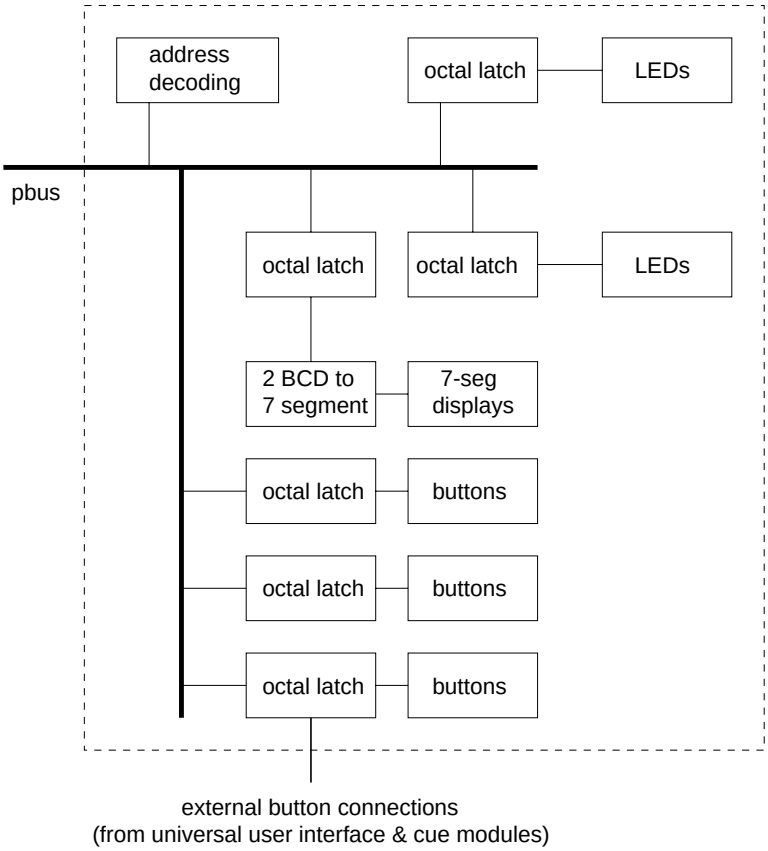


Figure 13: Control board, output assignment module diagram.

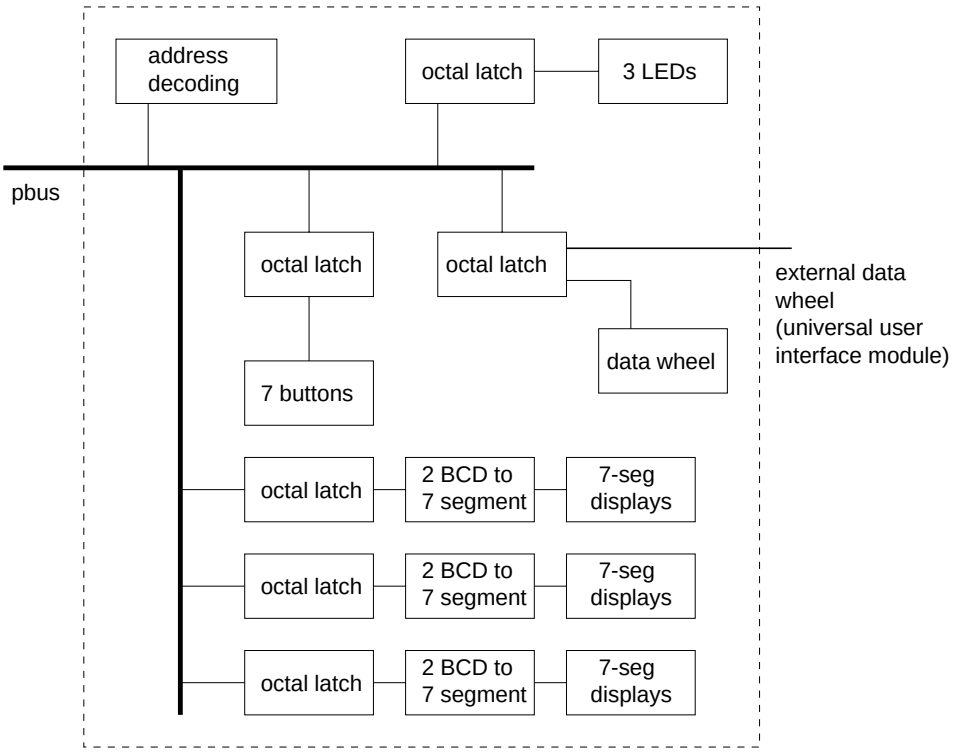


Figure 14: Control board, transport module diagram.

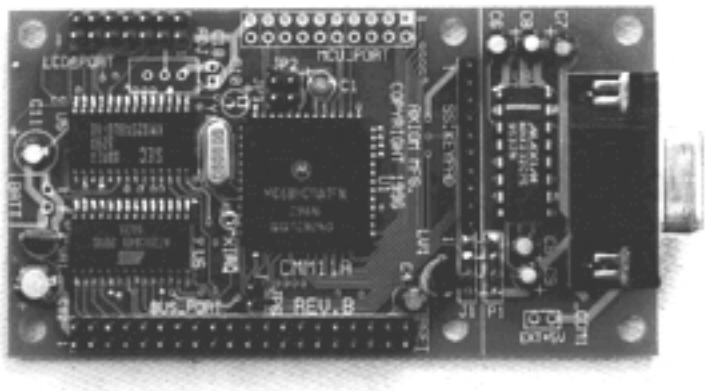


Figure 15: Axiom CMM-11A8 single-board computer, used in the mixer unit.

In a marketed model, a custom microcontroller board would be developed that includes only the necessary features. However, due to the time frame of this project, it makes sense to use a pre-manufactured microcontroller board. While it would certainly have been possible to develop a microcontroller board, the time spent debugging the inevitable problems would have significantly taken away from firmware and software development.

4.2 Mixer Unit

4.2.1 Overall System Description and Diagram

The mixer unit is a hybrid analog-digital device. The system is broken up into several logical blocks. Modules dedicated to handling audio inputs and outputs work in conjunction with matrix-mixer modules to provide the necessary functionality. A digital VU meter board provides audio level monitoring capability to the system. Additionally, a microcontroller is used to control the functions of the unit, with an LCD to display status information.

Figure 16 shows the overall mixer unit diagram. This view shows the system in terms of its functional blocks.

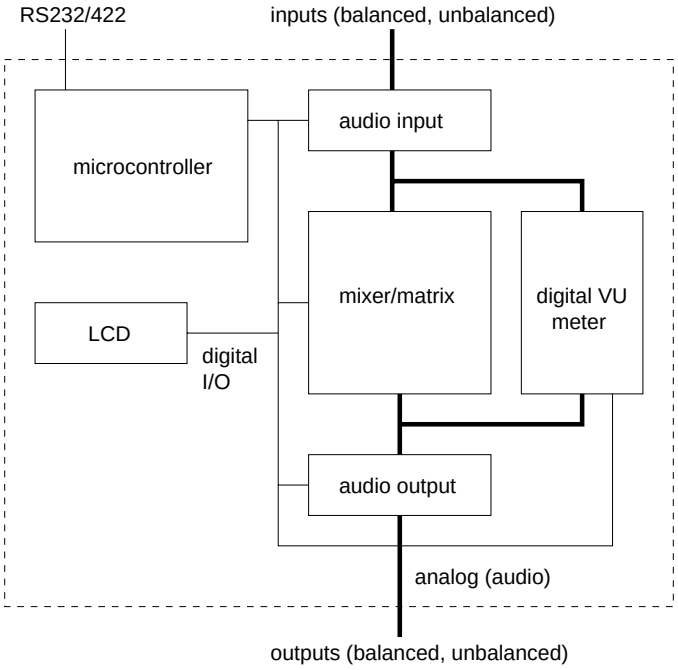


Figure 16: Mixer unit, overall system diagram.

4.2.2 Module Overview

Figures 17 and 18 show the system architecture of the mixer unit at the module level. The system is laid out in a modular fashion, such that it can be expanded and upgraded from an initial configuration.

Audio input and output boards mounted to panel-mounted aluminum brackets, allow reasonably simple upgrades. Audio mix PC boards can be added to the system as well, however this upgrade process is slightly more involved, as a different wiring harness is needed to connect to the boards.

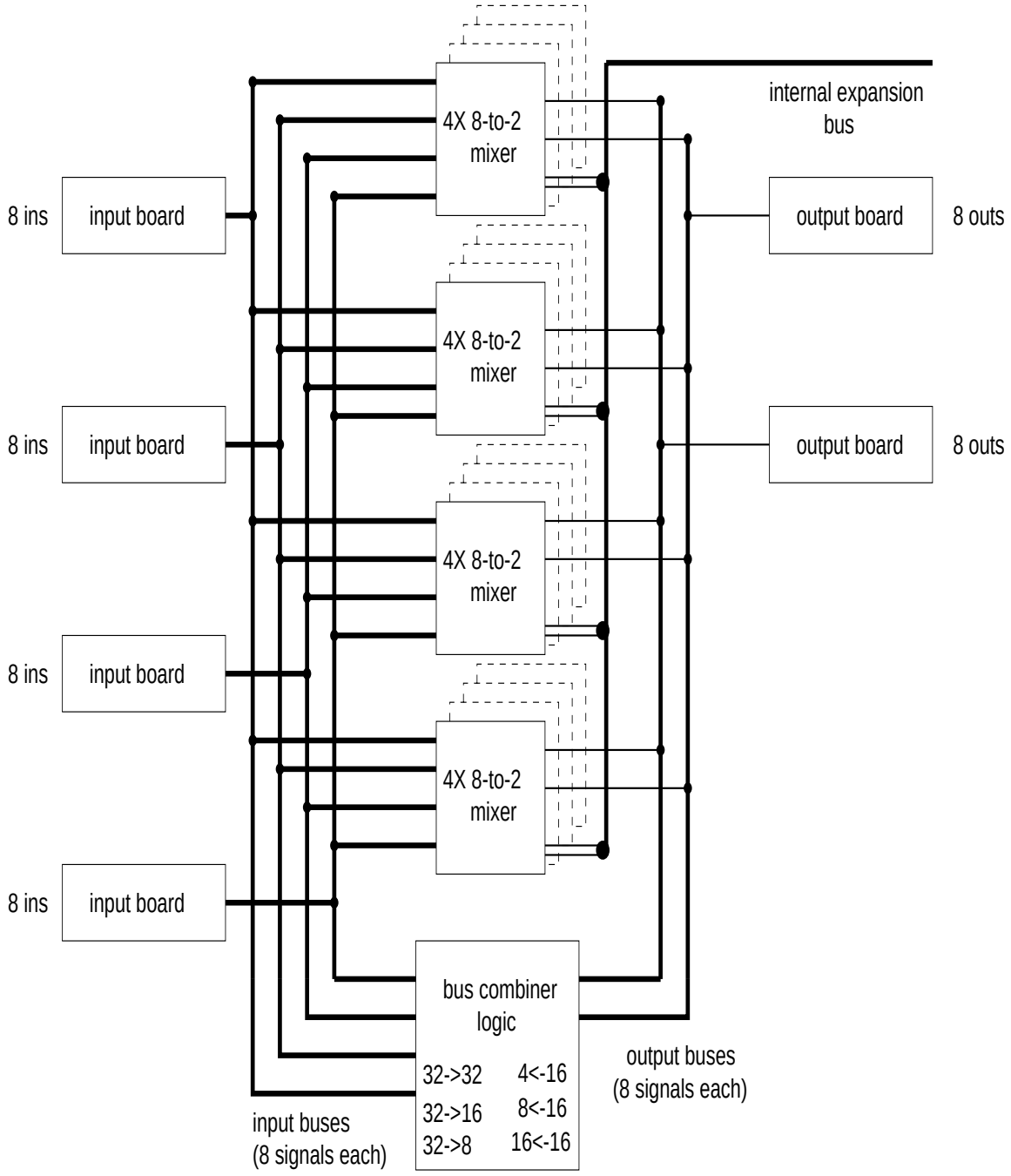


Figure 17: Mixer unit, module-level system diagram (analog).

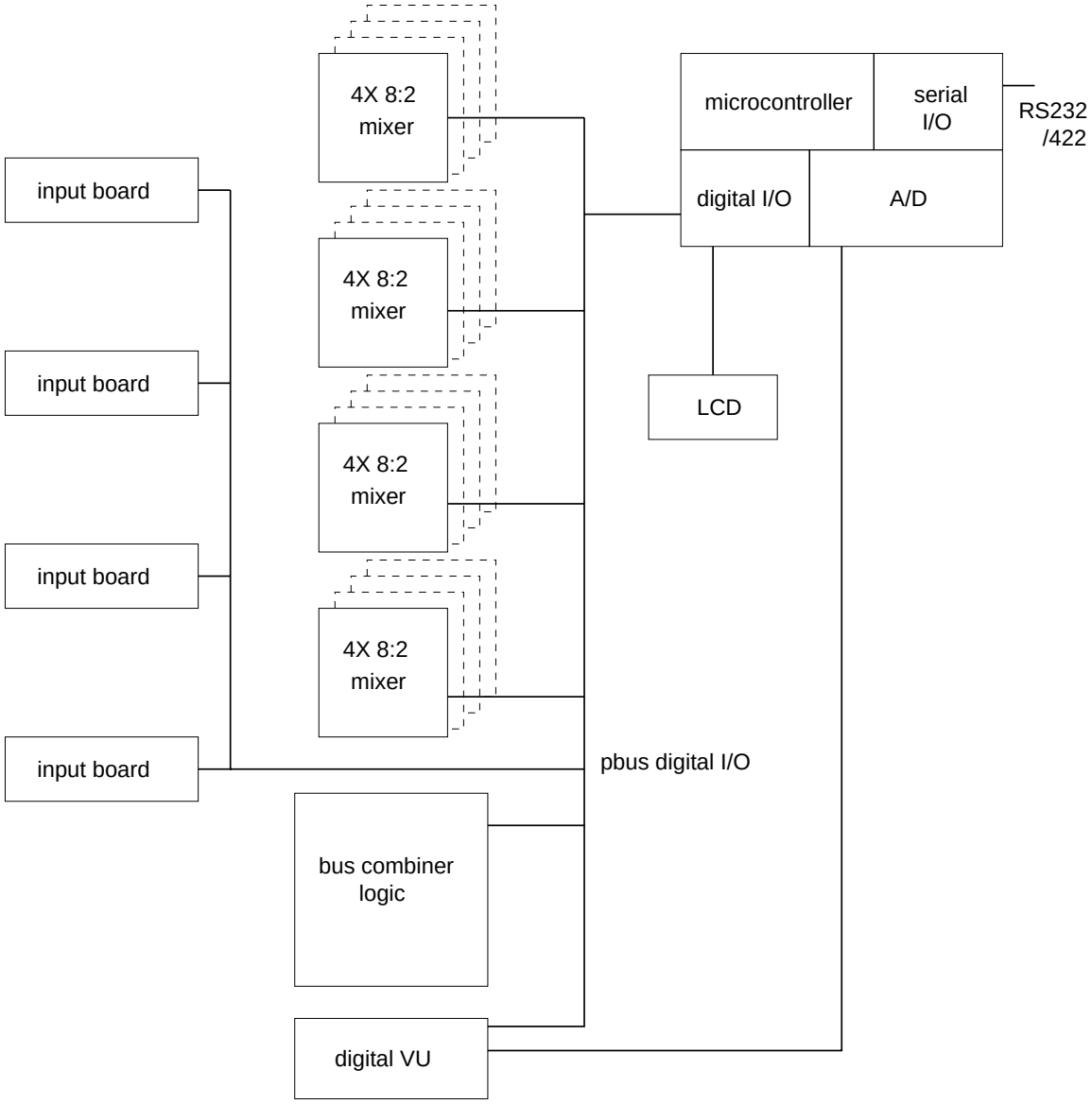


Figure 18: Mixer unit, module-level system diagram (digital).

This particular scheme of modularity was chosen for two main reasons. First, it extends the ability to expand and upgrade the system. Secondly, there are substantial manufacturing cost reductions involved due to the duplication of several of the modules.

4.2.3 Audio Input Module

As per high-level specifications, each audio input can be either balanced or unbalanced. Each input module consists of eight tip-ring-sleeve 1/4 inch jacks, appropriate circuitry for processing the input signals, and appropriate Pbus interface circuitry. As per specifications, a digitally-controlled gain stage shall be employed, to provide a trim adjust for each input.

Figure 19 shows a block diagram of an audio input module. A standard system has four of these modules.

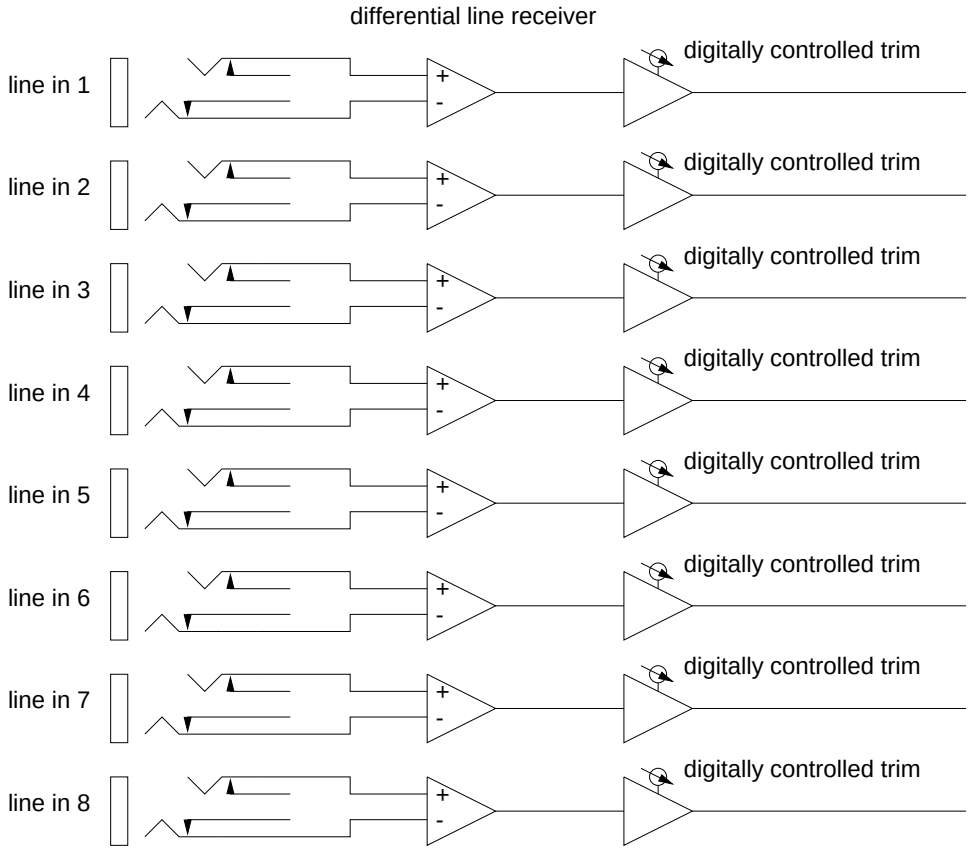


Figure 19: Mixer unit, audio input module diagram.

4.2.4 Audio Mix Modules

The audio mix modules are the core of the mixer unit. Each module shall contain four eight-to-two attenuating mixer chips. These mixers shall digitally controlled, and shall provide at least 63dB of attenuation in 1dB steps, as per the high-level specification.

At this stage, it made sense to attempt to locate such a part, and indeed, one was found. The SSM2163 from **Analog Devices** is such a hybrid mixing device. It has respectable specifications, and is about $12 in quanitites between 10 and 100.

One output from each mixer chip is fed to a position on each output bus. With the system in stock configuration with four mix boards, one chip drives each output bus line. In larger

configurations, up to four mixer chips (from different mixer modules) may be attached to any one output bus line. The remaining output from each mixer chip shall be available for future upgrades such as inputs to DSP modules, etc.

Figure 20 depicts a mix module. A stock system has only four of these boards, but up to sixteen may be used in a system.

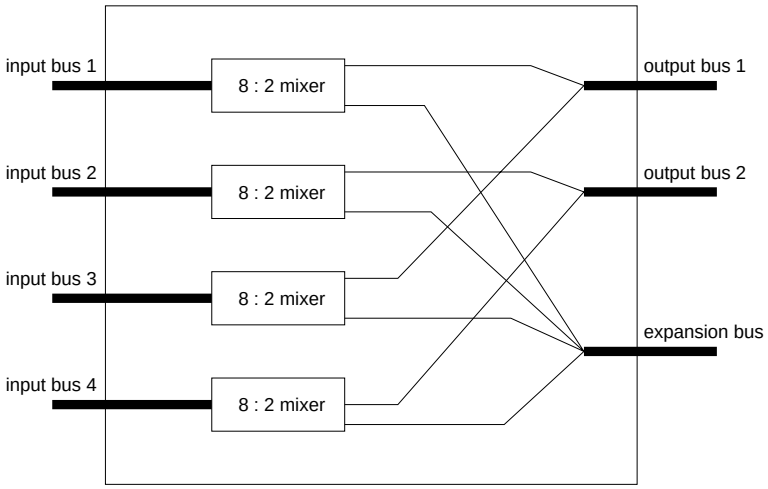


Figure 20: Mixer unit, audio mix module diagram. Systems shall have four, eight or sixteen of these modules.

4.2.5 Bus Combiner Module

The bus combiner module provides the capability to dynamically reconfigure the signal architecture of the unit. Since the input and output sections are divided up in to buses of 8 audio signals, these buses can be combined or separated, changing the architecture of the mixer. This is only useful in systems with less than sixteen mix modules, as there are not enough mixer chips to fully cover a 32x16 mix configuration. In a four or eight mix module system, bus combination is necessary, to reduce the number of inputs to gain a large amount of outputs, or vice-versa.

Figure 21 shows the input bus combiner diagram, while figure 22 shows the output bus combiner diagram. These are shown separately only for clarity; they shall be combined in the same module on the prototype.

4.2.6 Digital VU Module

The digital VU module multiplexes and processes the audio signals from all of the input and output buses. The output from this board shall be suitable for input to an analog-to-digital converter operating in the 0-5V range.

Figure 23 shows the functional diagram of the digital VU module.

4.2.7 Audio Output Module

As per high-level specifications, balanced and unbalanced audio output formats are available on the mixer unit.

Balanced output modules consist of eight XLR/Cannon style connectors and necessary driver circuitry. Figure 24 shows the respective functional diagram.

Unbalanced output modules consist of eight tip-ring-sleeve 1/4 inch jacks, a PC board, and a metal mounting bracket. Figure 25 shows the respective functional diagram.

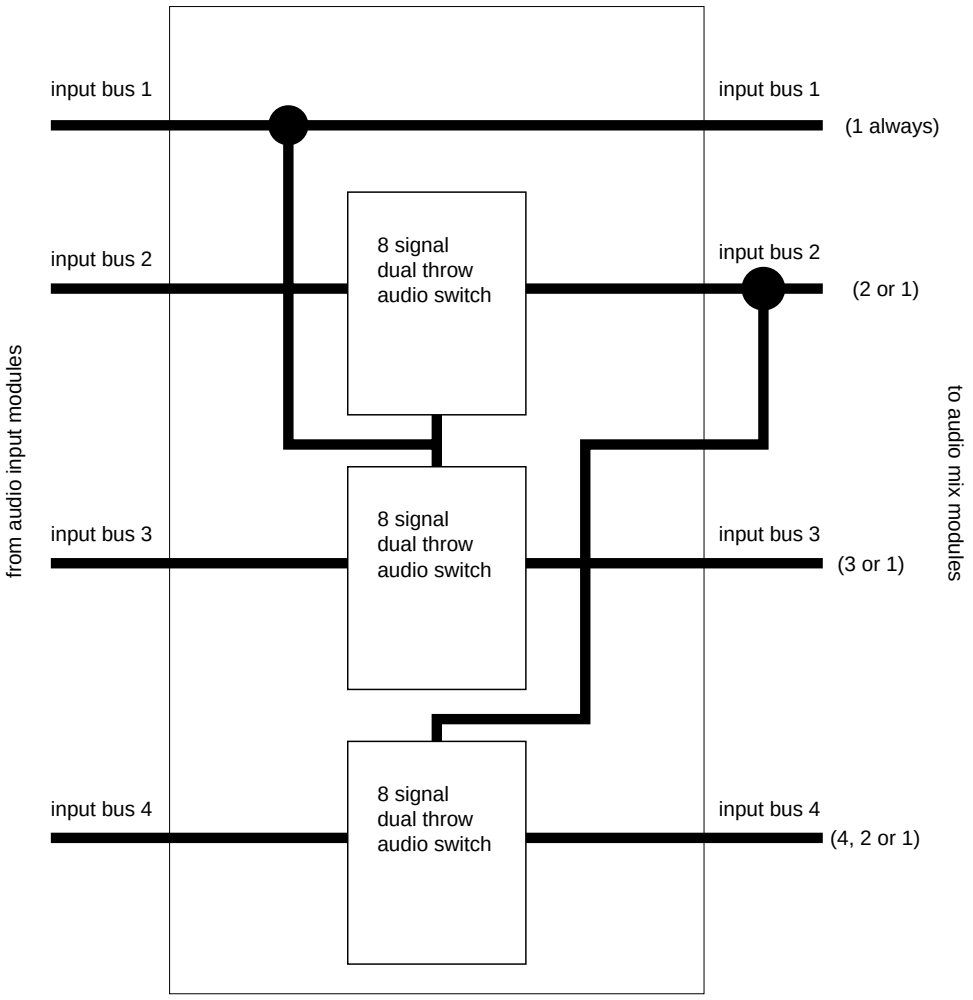


Figure 21: Mixer unit, bus combiner module diagram (input buses).

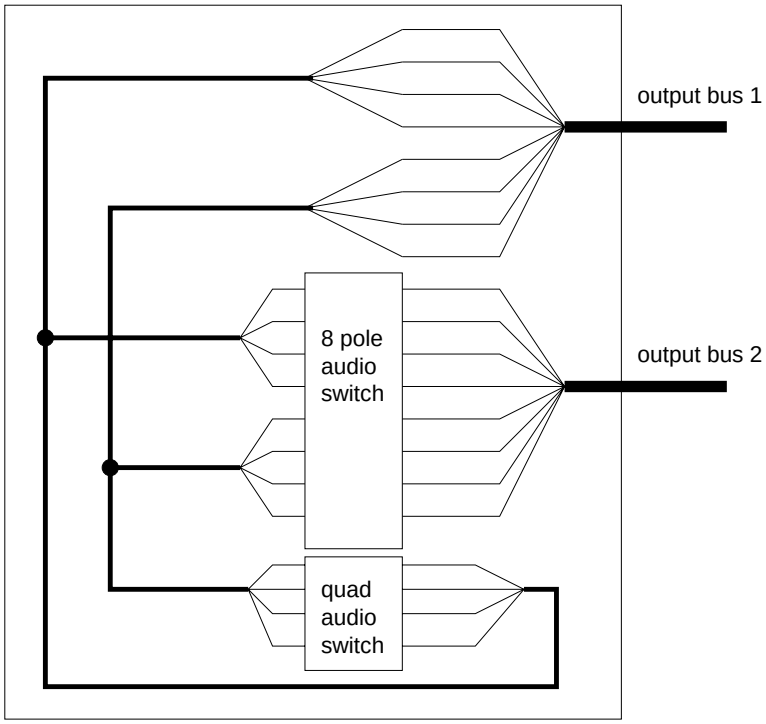


Figure 22: Mixer unit, bus combiner module diagram (output buses).

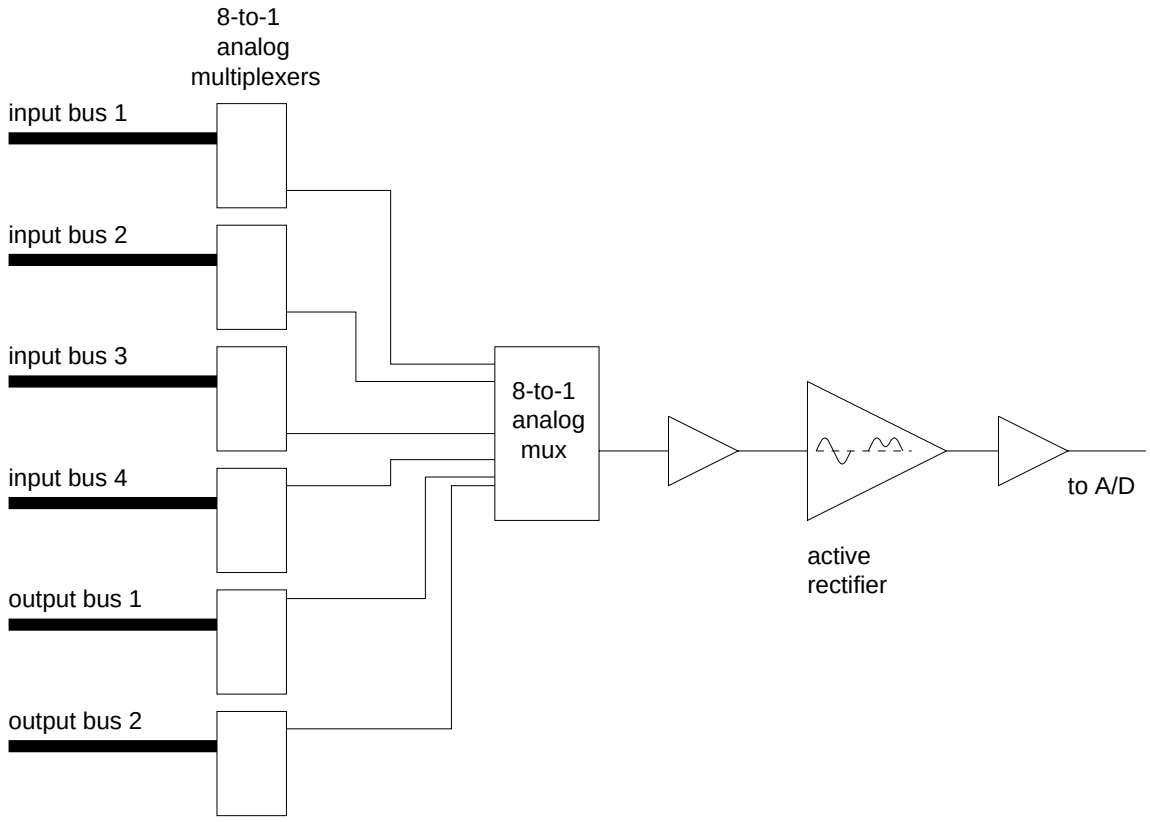


Figure 23: Mixer unit, digital VU module diagram.

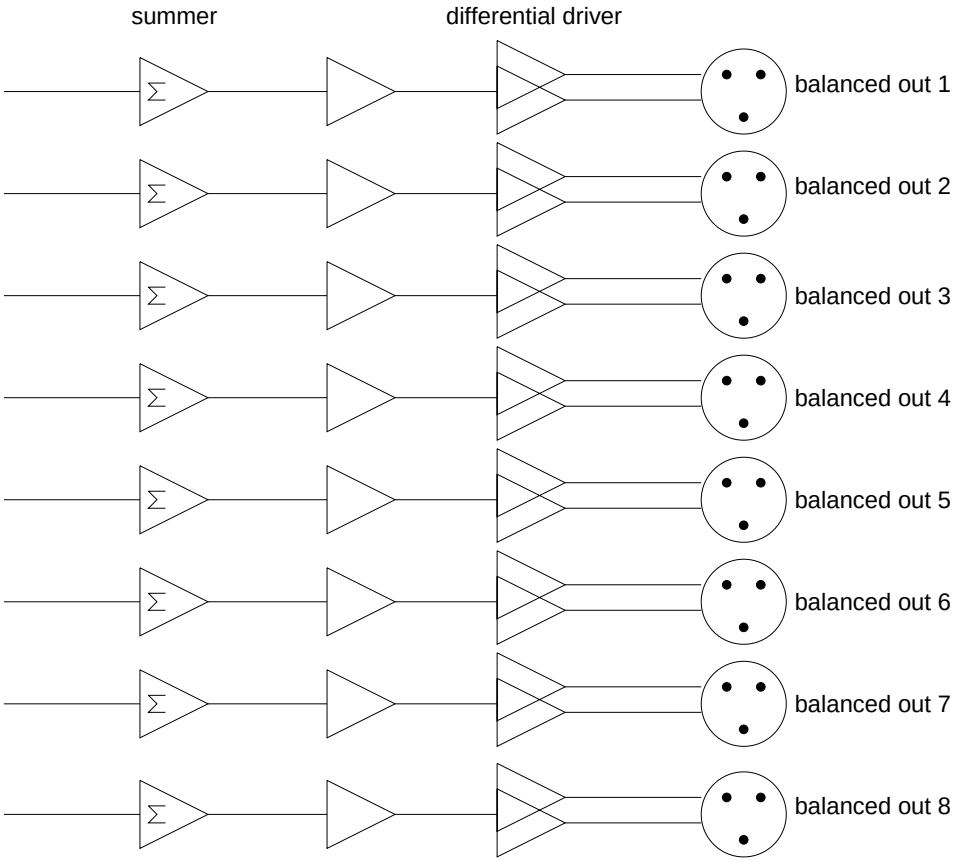


Figure 24: Mixer unit, balanced audio output module diagram.

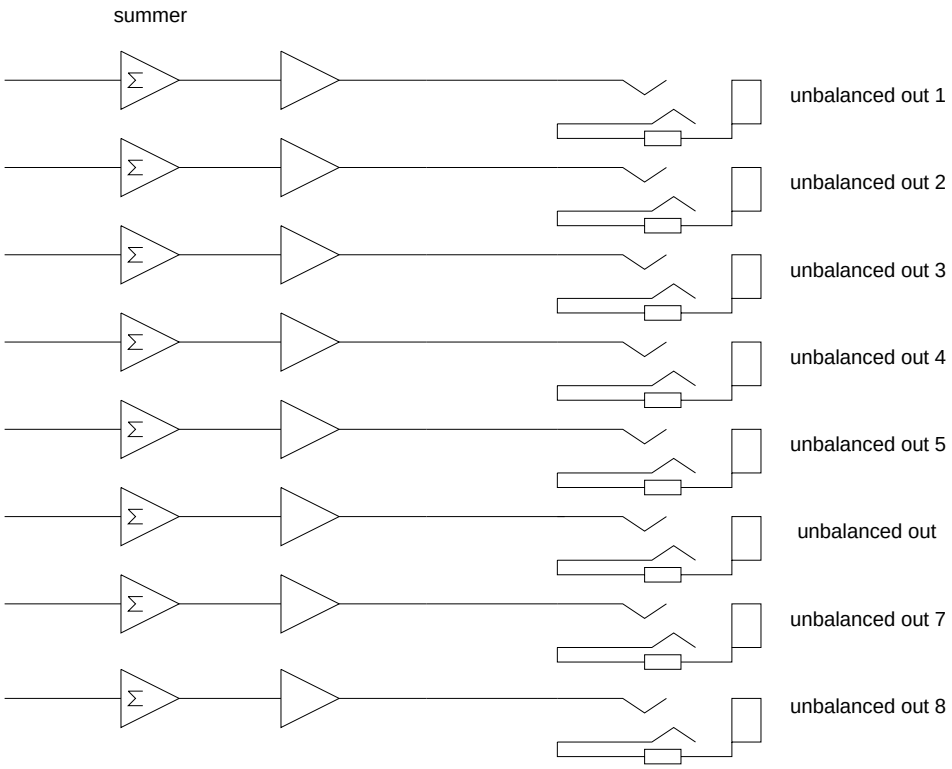


Figure 25: Mixer unit, unbalanced audio output module diagram.

4.2.8 Microcontroller Module

For the prototype, a pre-made Motorola 8MHz MC68HC11 microcontroller board will act as the embedded controller for the mixer unit. This board contains an RS232 port, an RS485/RS422 port, an 8-input A/D converter, and an 8255-based peripherial I/O controller. In addition, it includes an I/O port, address decoding logic, and contrast adjustment for liquid crystal displays. Figure 26 shows this microcontroller board.

In a marketed model, a custom microcontroller board would be developed that includes only the necessary features. However, due to the time frame of this project, it makes sense to use a pre-manufactured microcontroller board. While it would certainly have been possible to develop a microcontroller board, the time spent debugging the inevitable problems would have significantly taken away from firmware and software development.

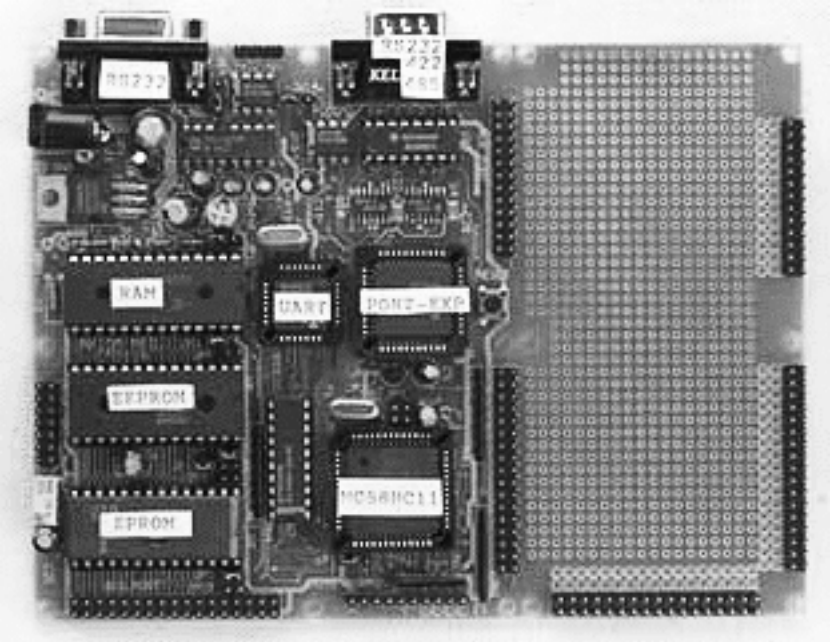


Figure 26: Axiom CMD-11A8 single-board computer, used in the mixer unit.

5 Inter-Module Specifications

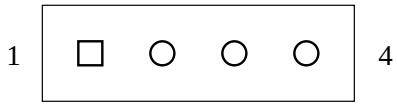
The mid-level hardware specifications are intended to bridge the gap between the high-level designs presented in the last section and the next section. As such, the main intent is mostly to provide exposure to the elements that make up each component, as well as some overall component details.

5.1 Control Board

5.1.1 Power

The control board is primarily a digital device, and as such does not require any negative DC voltages. The primary supply voltage is +5V, thus greatly simplifying both the power supply and power distribution methodology.

Figure 27 shows the pinouts for the connector used to distribute power to each of the PC boards inside of the control board.



pin	voltage
1	+5V
2	GND
3	GND
4	+5V

Figure 27: 4 circuit 0.156in spaced connector, used for power distribution in the control board.

There shall be 10μF of power supply bypass capacitance for each supply line used on each PC board. Additional 0.1μF bypass capacitors shall be placed near each IC, as appropriate.

5.2 Mixer Unit

5.2.1 Gain Structure Diagram

Taking audio specifications in to account, overall system gain structure diagrams have been constructed. Figure 28 depicts the input gain structure for a single channel. Included in this section is gain reduction for a balanced input, adjustable trim control, and an overall gain reduction before passing to the next stage.

Figure 29 depicts the gain structure for the mix stage of the mixer unit. This stage only attenuates the input signal, up to 63dB, as per the high-level specification.

Figure 30 shows the gain structure for the output stage. Note a constant input section gain, and a constant gain on the balanced outputs.

5.2.2 Power

Since the mixer unit is a hybrid digital/analog device, several power supplies are necessary. It is desirable to keep digital and analog supplies separate, as digital circuits tend to inject a lot of noise on to their power supply rails.

A ± 7V power supply is necessary, since the mixer chips chosen can not operate outside of the range of ± 7.5V. The specifications of the system dictate that large signals from external devices (up to +24dBu) may be used with the system. This dictates that a ± 14V power supply

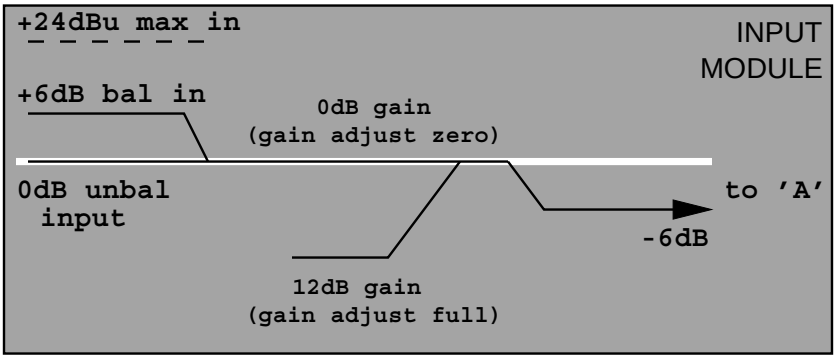


Figure 28: Mixer unit gain structure diagram, input section.

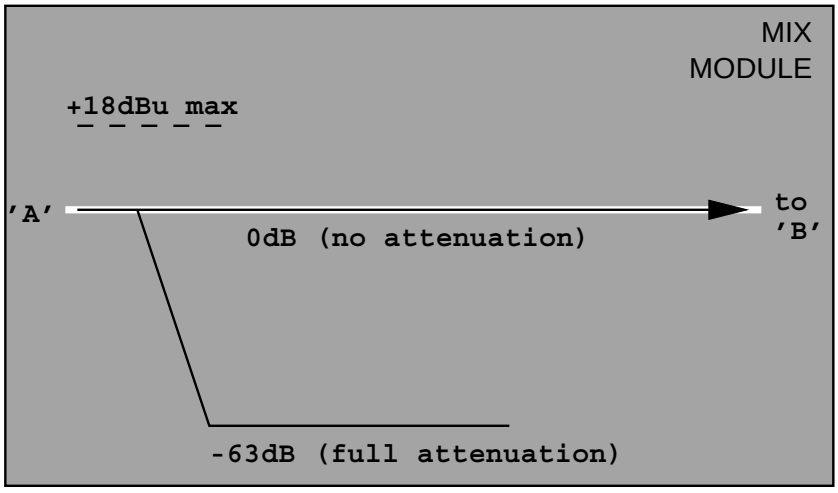


Figure 29: Mixer unit gain structure diagram, mixer section.

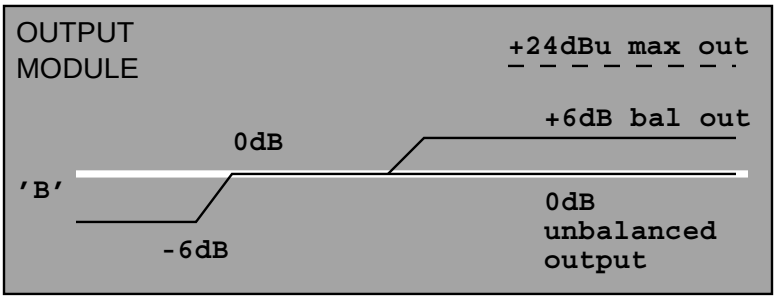
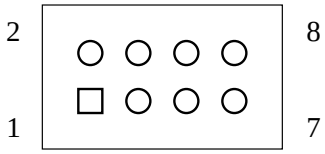


Figure 30: Mixer unit gain structure diagram, output section.

be used for the rails of the input and output boards. Adding a +5V power supply for the digital elements of the system rounds out the power requirements of the system.

Each board receives its return grounds from the power connector, *not* from any of the other connectors present on the board. All ground signals are connected at the power supply. This star grounding technique shall be employed to eliminate ground loop problems.

Figure 31 shows the pinouts of the power connector for boards in the mixer unit.



pin	voltage
1	+5V (digital)
2	GND (digital)
3	-14V (analog)
4	-7V (analog)
5	+14V (analog)
6	+7V (analog)
7	N.C.
8	GND (analog)

Figure 31: 8 circuit **Molex** *Mini-Fit Jr.* connector, used for power distribution in the mixer unit.

5.2.3 Audio Signal Distribution

Several audio signals need to be distributed among many modules in the mixer unit. To accomplish this, a bus architecture is used. Each bus carries eight audio signals on standard ribbon cable. Connection to each PC board is made via standard IDC sockets, and board-mounted headers.

The signals on each ribbon cable bus are arranged such that they are separated by a signal ground. This helps to prevent crosstalk between channels carried on the same bus. The ground signals are *only* to be connected at the bus combiner module, eliminating ground loop problems.

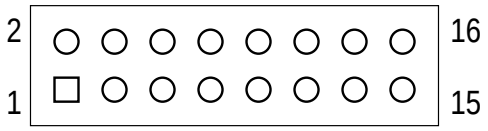
Figure 32 shows the pinouts for the audio input buses, of which there are four. Figure 33 shows the pinouts for the two audio output buses. The signal levels on these buses shall be no higher than +18dBu (about 6.15V peak).

5.3 Digital Control Bus

The modular design of the DACS subsystems begs for a flexible means of connecting a variety of PC boards together such that the internal microcontroller (μC) may communicate with them. Thus, a "pseudo-bus" (dubbed the Pbus) is defined, to be generated by the μC . The Pbus is a master-slave type of bus, where the master is the microcontroller board, and the slaves are the various devices connected to the bus. The signal levels on the bus are standard TTL levels.

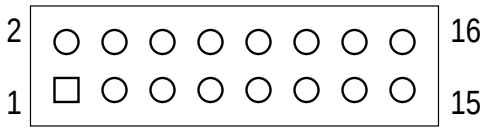
To remove any confusion, it should be noted that the Pbus is *not* the same as the μC bus, but is rather built on top of the underlying μC bus structure. This abstraction layer allows many different types of controllers to act as Pbus masters, thus allowing a virtually limitless choice of microcontrollers and microprocessors.

Additionally, simple PC-hosted hardware can be constructed to allow easy Pbus development, testing and debugging. A simple, but extraordinarily useful PC-hosted bus-snooper will likely be constructed in order to aid later firmware debugging tasks.



pin	function (bus A) (bus B) (bus C) (bus D)
1	GND†
2	audio in (1) (9) (17) (25)
3	GND†
4	audio in (2) (10) (18) (26)
5	GND†
6	audio in (3) (11) (19) (27)
7	GND†
8	audio in (4) (12) (20) (28)
9	GND†
10	audio in (5) (13) (21) (29)
11	GND†
12	audio in (6) (14) (22) (30)
13	GND†
14	audio in (7) (15) (23) (31)
15	GND†
16	audio in (8) (16) (24) (32)

Figure 32: Audio input bus pinouts. The 32 audio inputs are carried on four 16-pin headers. † GND is *only* connected at the bus combiner board.



pin	function (bus A) (bus B)
1	GND†
2	audio out (1) (9)
3	GND†
4	audio out (2) (10)
5	GND†
6	audio out (3) (11)
7	GND†
8	audio out (4) (12)
9	GND†
10	audio out (5) (13)
11	GND†
12	audio out (6) (14)
13	GND†
14	audio out (7) (15)
15	GND†
16	audio out (8) (16)

Figure 33: Audio output bus pinouts. The 16 audio outputs are carried on two 16-pin headers. † GND is *only* connected at the bus combiner board.

5.3.1 Physical Specifications

The Pbus is physically built around commonly available 20-pin DIP headers, ribbon cable and insulation displacement connectors (IDC). The pinouts are depicted in figure 34. Note the presence of a single ground pin. Slaves on the Pbus shall not use this as a ground; it is present to simplify connection of an external bus snooper.

Typical μC connection to the Pbus shall be made via a few byte-wide I/O ports, a peripheral I/O controller such as the venerable Intel 8255, or through a piece of programmable logic.

With 7 bits of address space and a byte-wide data bus, there exist 128 input and 128 output ports in this bus. I/O maps are given for the control board and mixer unit in a later section.

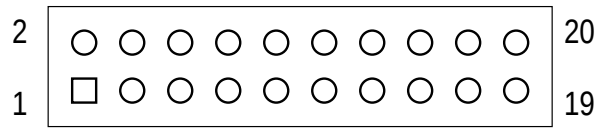
5.3.2 Timing and Levels

The Pbus is a relatively low-speed digital I/O bus. Timing is not exceptionally critical, as the speed of the bus is in the sub-1MHz range. Nevertheless, a visual reference to the bus timings is useful to fully understand the Pbus . Figure 35 depicts the timing for the Pbus master reading data from a slave device. Figure 36 shows the timing for the Pbus master writing data to a slave.

Note that the control signals of the bus are level-sensitive, not edge-triggered. During the time when a particular input or output window is active, there is a transparent window between the master and the active slave(s), in the direction dictated by the direction control line.

5.3.3 Control Board I/O Map

Figure 37 shows the Pbus I/O map used in the control board.



pin	function	direction (from μC)
1	data 0	I/O
2	bus $\overline{latch}$ (to latch data to/from devices)	out
3	data 1	I/O
4	GND (for Pbus snooper)	N/A
5	data 2	I/O
6	$\overline{clock}$ (for serial devices)	out
7	data 3	I/O
8	addx 0	out
9	data 4	I/O
10	addx 1	out
11	data 5	I/O
12	addx 2	out
13	data 6	I/O
14	addx 3	out
15	data 7	I/O
16	addx 4	out
17	$\overline{sense}$ (for device auto-sensing)	in
18	addx 5	out
19	bus $\overline{read}/write$	out
20	addx 6	out

Figure 34: DACS Pbus , physical description. A standard 0.100 inch DIP header of 20 pins shall be used to connect each PC board to the bus. The bus itself shall be carried on 20 conductor ribbon cable, with each PC board connection made via a 20 conductor IDC. All signals are standard TTL levels.

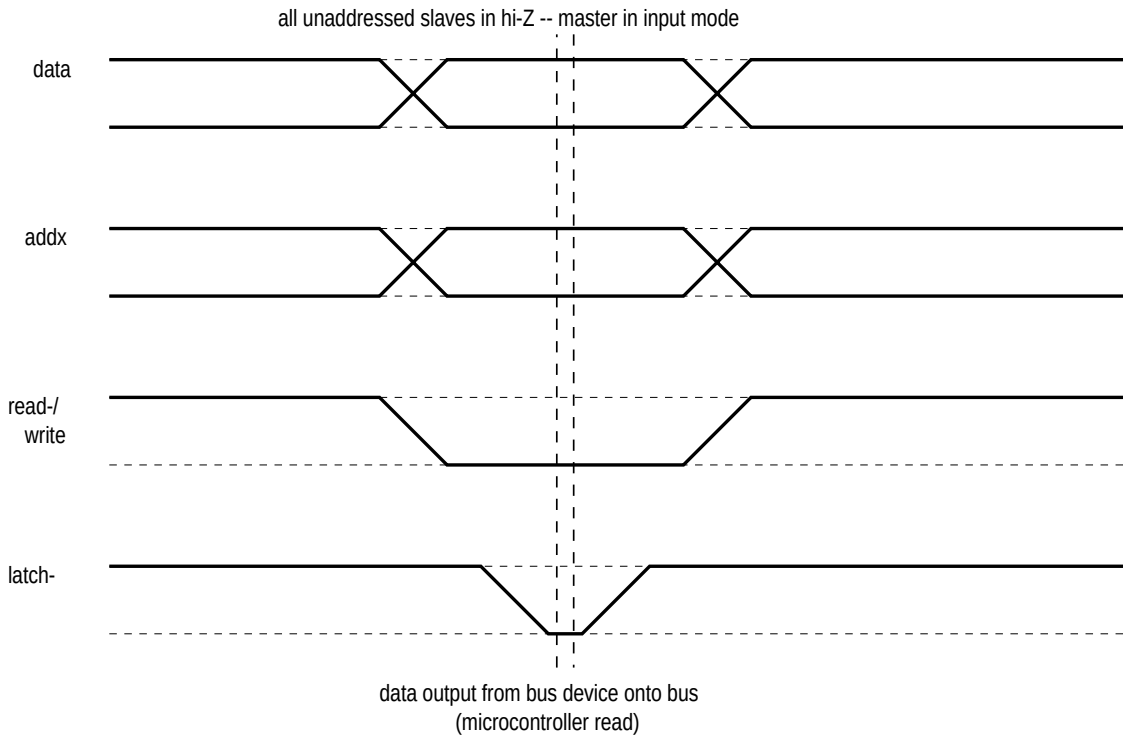


Figure 35: Pbus timing diagram for input (Pbus master reading slave).

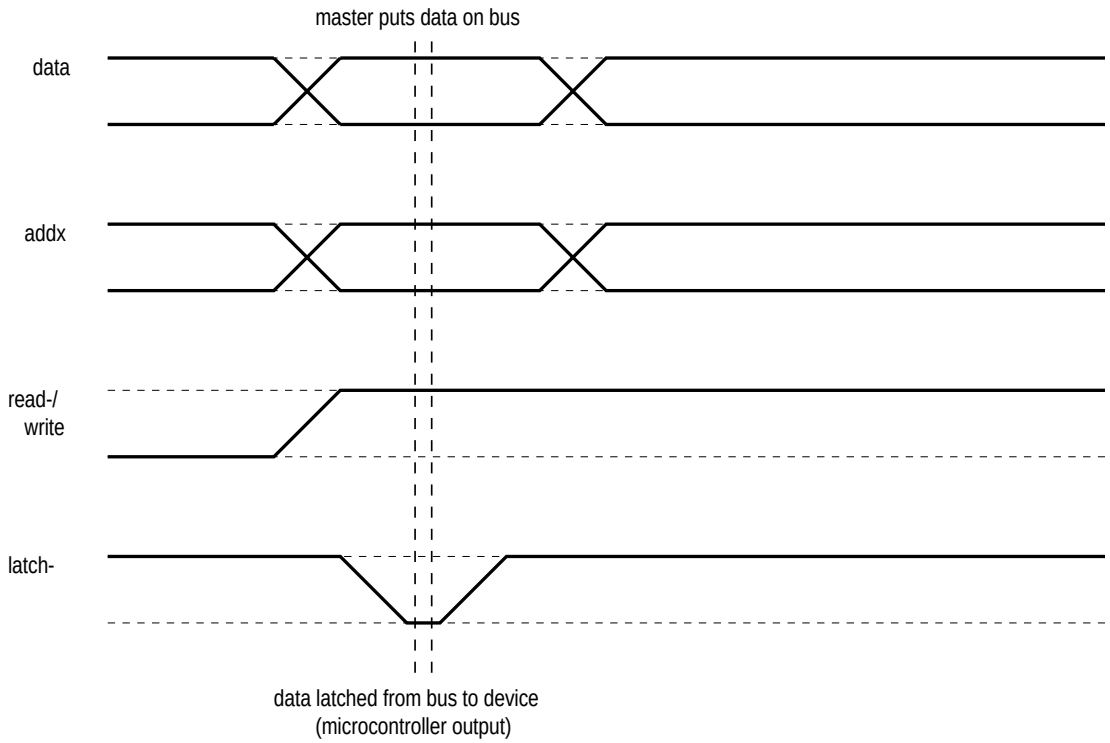


Figure 36: Pbus timing diagram for output (Pbus master writing to slave).

address range	direction (from μC)	function
0x00 - 0x0F	out	fader analog mux address
0x10 - 0x2F	out	LEDs (individual and 7-segment)
0x30 - 0x4F	in	switches (buttons, etc.)
0x50 - 0x57	in	quadrature decoders (datawheels)
0x58 - 0x7F	—	reserved for expansion

Figure 37: DACS control board prototype Pbus I/O map.

address range	direction (from μC)	function
0x00 - 0x0F	out	input board trim control
0x10 - 0x2F	out	mixer module control
0x30 - 0x37	out	bus combiner control
0x38 - 0x3F	out	digital VU control
0x40 - 0x7F	—	reserved for expansion

Figure 38: Mixer unit Pbus I/O map.

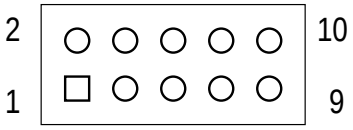
5.3.4 Mixer Unit I/O Map

Figure 38 shows the Pbus I/O map for the mixer unit.

5.3.5 A/D Converter Connection

Several of the modules in both the control board and the mixer unit require connection to the A/D converter present on the microcontroller modules. A generalized scheme for connecting devices to these A/D inputs is specified, such that a single ribbon cable with several inexpensive IDC connectors can be connected between all devices requiring A/D access. Each of these modules shall be equipped with a 10-pin 0.100 inch DIP header, with pinouts as shown in figure 39.

A map of A/D port use is given for each piece of hardware in a later section.



pin	function
1	A/D converter input 1 (0-5VDC)
2	A/D converter input 2 (0-5VDC)
3	A/D converter input 3 (0-5VDC)
4	A/D converter input 4 (0-5VDC)
5	A/D converter input 5 (0-5VDC)
6	A/D converter input 6 (0-5VDC)
7	A/D converter input 7 (0-5VDC)
8	A/D converter input 8 (0-5VDC)
9	reserved
10	reserved

Figure 39: Analog-to-digital converter connector pinouts.

5.3.6 Control Board A/D Map

The analog input map for the prototype DACS board is shown in figure 40. Note that very few of the inputs are used.

A/D input	function
1	leftmost 8 fader inputs (muxed)
2	rightmost 8 fader inputs (muxed)
3-8	reserved for expansion

Figure 40: DACS board prototype A/D connector assignments.

6 Hardware Schematics

6.1 Control Board

6.1.1 Fader Module

The fader module uses a standard block of Pbus interface logic. A 16V8 GAL (gate array logic) chip shall provide address decoding functionality. Figure 41 shows the pin assignments of this piece of programmable logic. While other means of address decoding are certainly possible, a GAL was used to reduce chip count, and provide enhanced functionality. The fact that GALs are completely programmable allows any address to be decoded. Thus, different GALs with different addressing may be used for duplicate boards in a system. The VHDL code used to generate the GALs is included in the appendices, on page 220.

To reduce the number of A/D ports required, the eight slide potentiometers are connected to a 4051 8-to-1 analog multiplexer. The schematic shown in figure 42 shows how these potentiometers are connected. A single enable line generated by the GAL enables the address lines of the 4051.

Sixteen individually addressable status LEDs are driven with a pair of 74LS373 octal latches. The LEDs are arranged such that the '373 is sinking the current when the LED is on, thus the status of the LEDs will appear inverted. Note that other schemes could have been used to drive the LEDs, but the latched method reduces CPU load considerably, as there is no refresh cycle to chew up CPU time, as in a multiplexed design. A pair of enable lines, generated by the address decode GAL, activate each latch. Figure 43 shows one of two of the LED driver circuits needed for the fader module.

6.1.2 Output Assign Module

The output assign modules uses a 16V8 GAL for address decoding and enable signal generation. Figure 44 shows the pin assignments for the GAL used in the output assign module. The VHDL code used to generate the GALs is included in the appendices, on page 222.

The status of 18 pushbutton switches need to be sensed by the microcontroller. Many methods exist for connecting switches to a microcontroller bus. To keep parts count and CPU use down, simple latches are used to latch the status of eight switches at a time. Firmware debouncing routines shall take the place of hardware debouncing techniques. The lack of a multiplexing scheme means less CPU time is wasted in reading in the values of the switches, while the firmware debouncing routines mean fewer components are used on the board. Figure 45 shows the circuit used to achieve these goals. Three of these circuits are present, providing a total of 24 pushbutton inputs. Six of these inputs are used for pushbuttons in the universal user interface and cue stack sections of the control board.

The output assign module contains several LED indicators that need to be controlled by the microcontroller. A simple current-sinking scheme involving 74LS373 octal latches is used, identical to that of the fader module. Figure 46 shows the circuit.

The output assign module contains a pair of 7-segment LED displays, controlled by the microcontroller. While it certainly would have been possible to drive the displays in a similar fashion to the bare LEDs present on the same board, this was decided against due to the extra Pbus ports needed. Instead, BCD-to-7-segment decoders were used, with their inputs driven from a 74LS373 latch. Figure 47 shows the circuit used to drive the 7-segment displays. Note the use of the ripple zero-blanking feature of the 74LS247 BCD-to-7-segment chips. This feature will blank both displays when the a double-zero is displayed, and blank the leftmost display when its value is zero.

6.1.3 Transport Control Module

As in other DACS modules, a 16V8 GAL is used for Pbus address decoding and enable signal generation. Figure 48 shows the pin layouts for this GAL. The VHDL code used to generate the

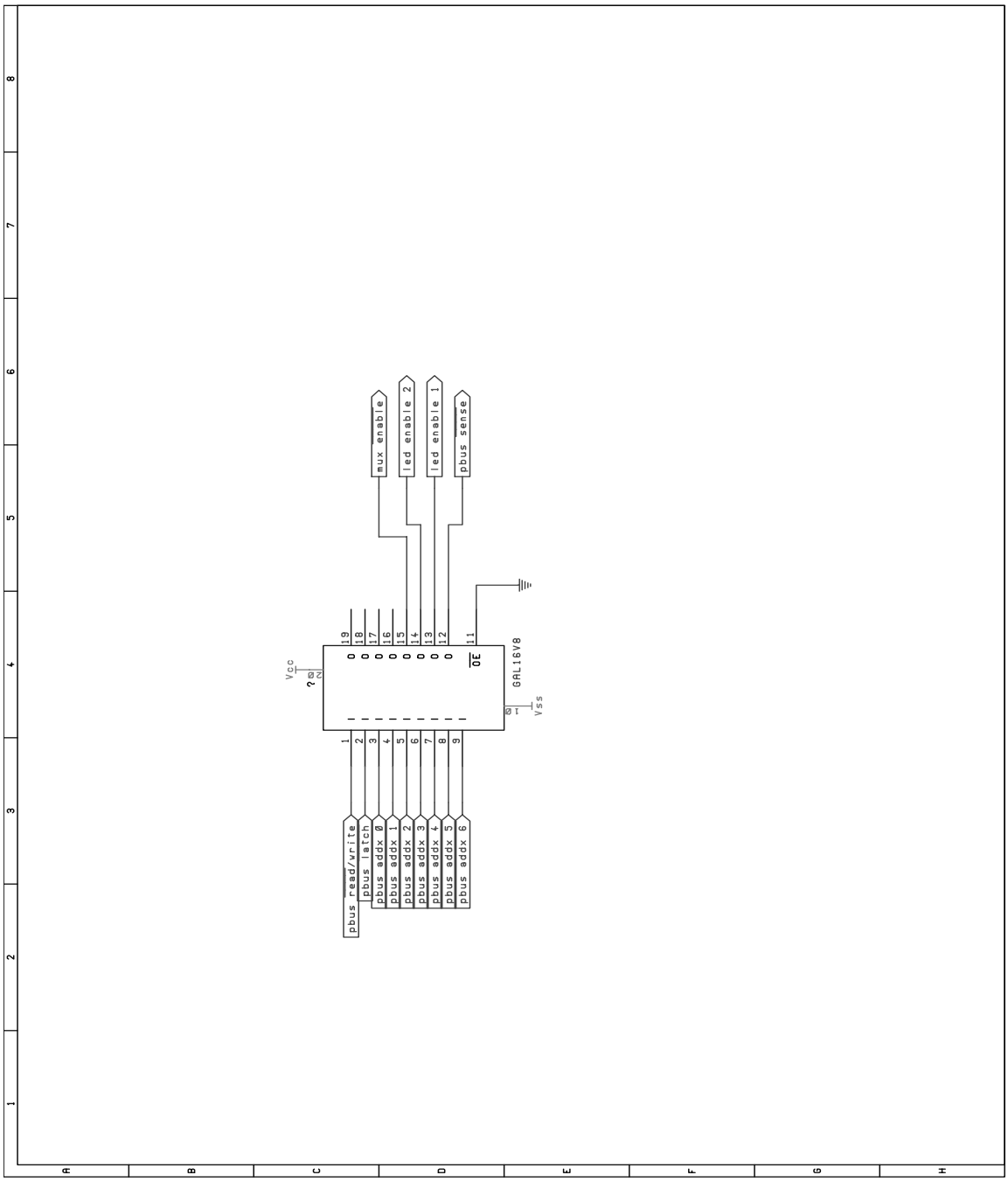


Figure 41: Fader module, address decoding GAL pin layout.

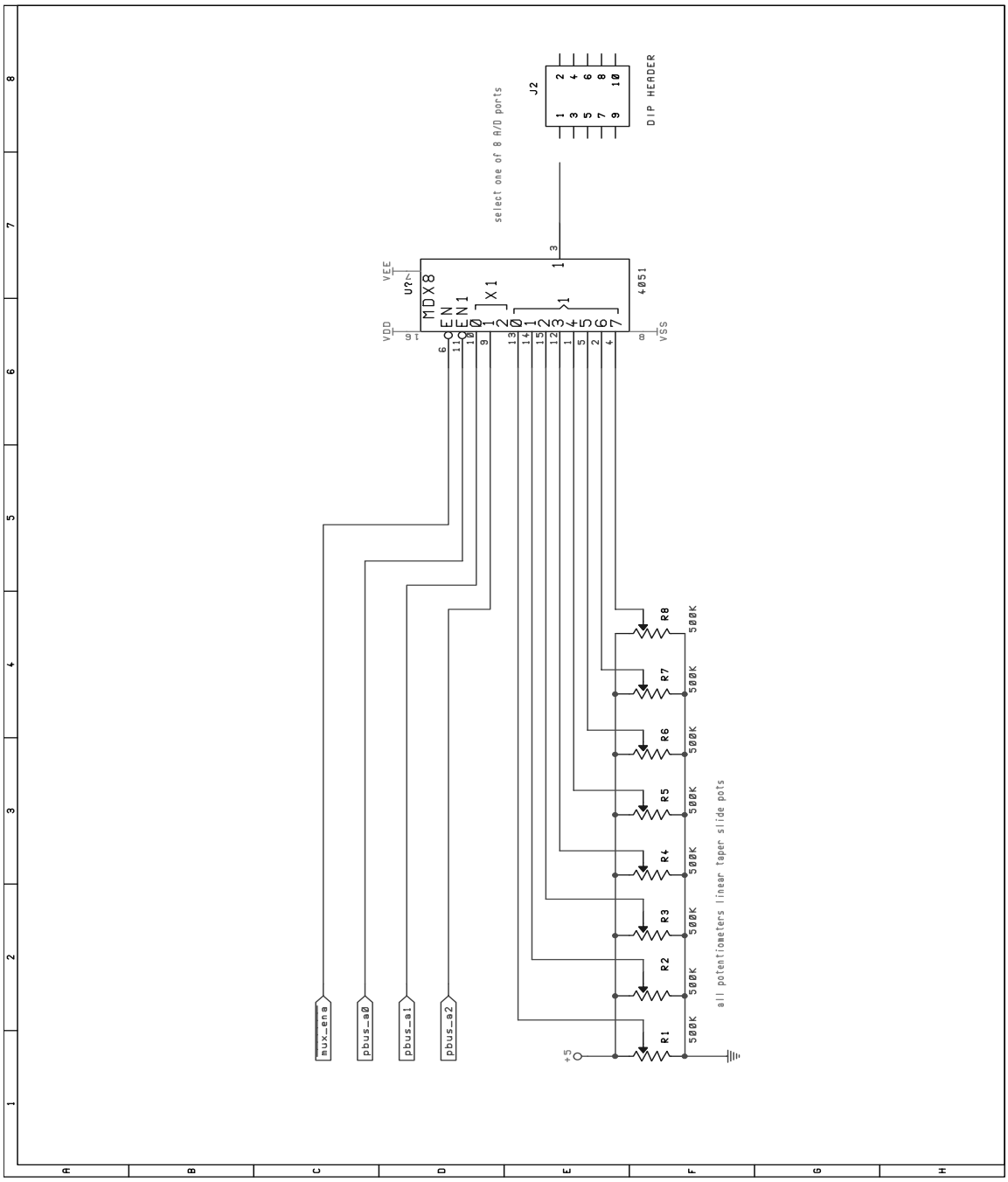


Figure 42: Fader module, analog multiplexer circuit for potentiometers.

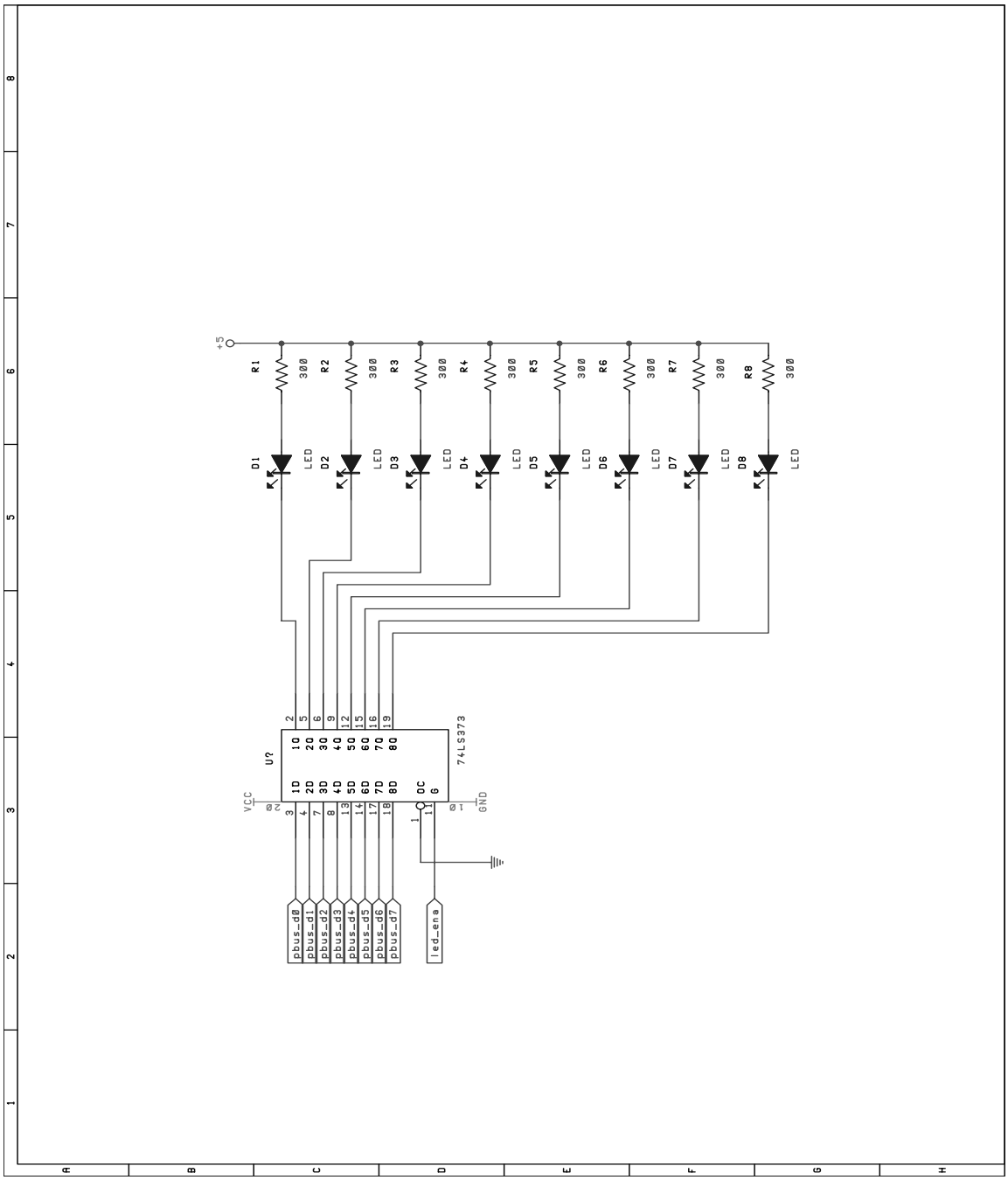


Figure 43: Fader module, LED driver schematic. The module needs two of these circuits, for a total of 16 LEDs.

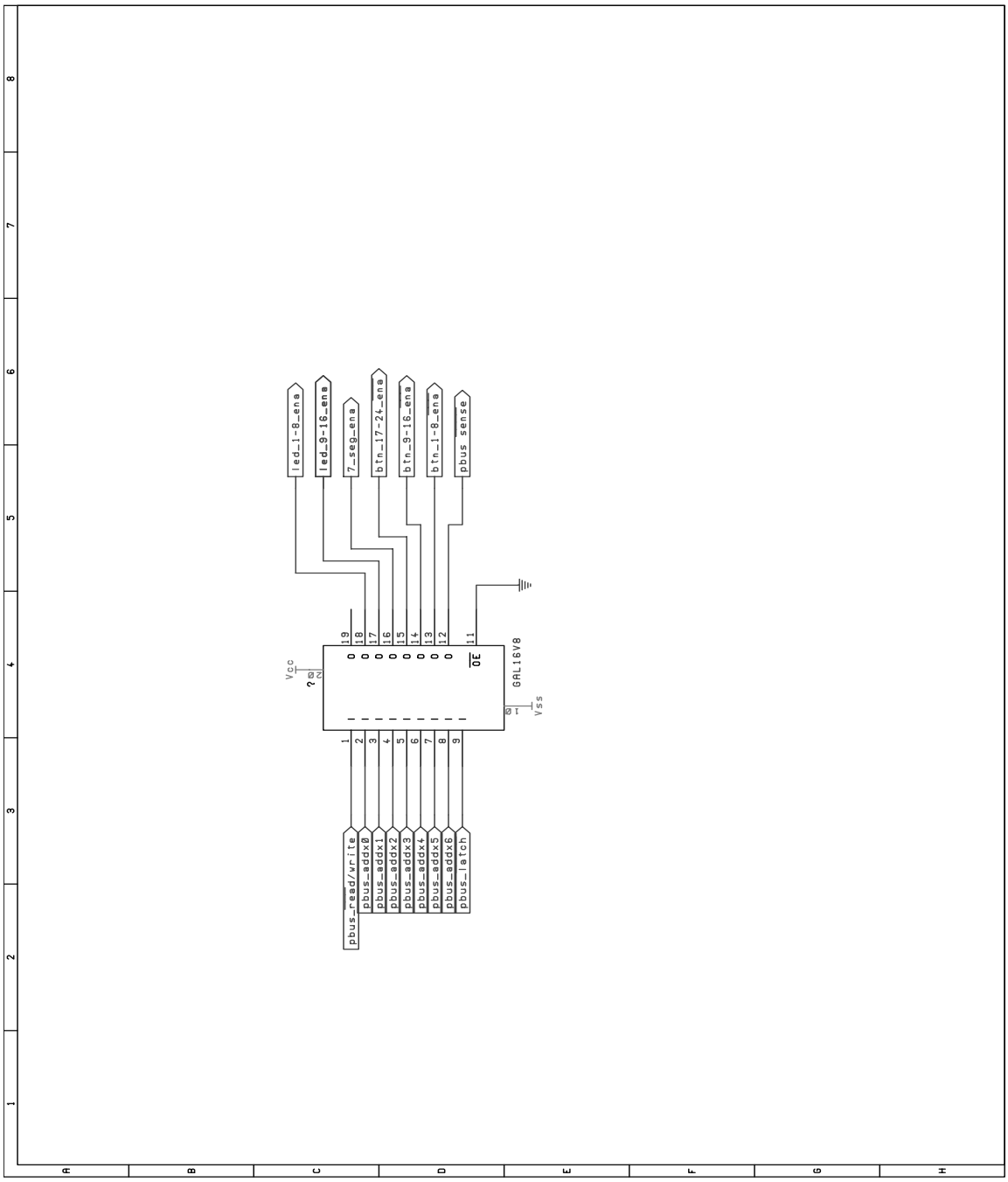


Figure 44: Output assign module, address decoding GAL.

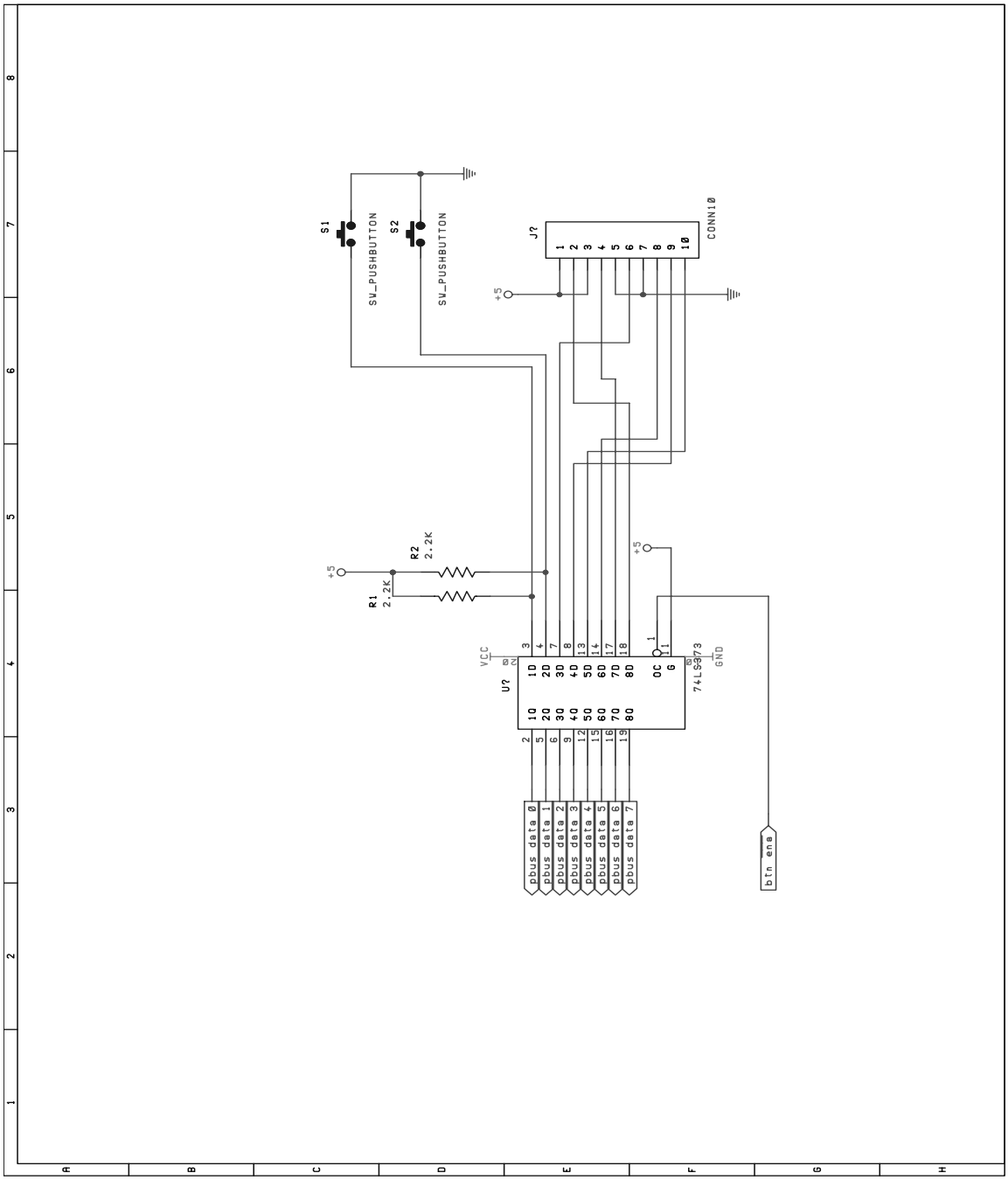


Figure 45: Output assign module, momentary pushbutton decoding schematic. The output assign module uses three of these circuits, for a total decoding capability of 18 on-board and 6 off-board buttons.

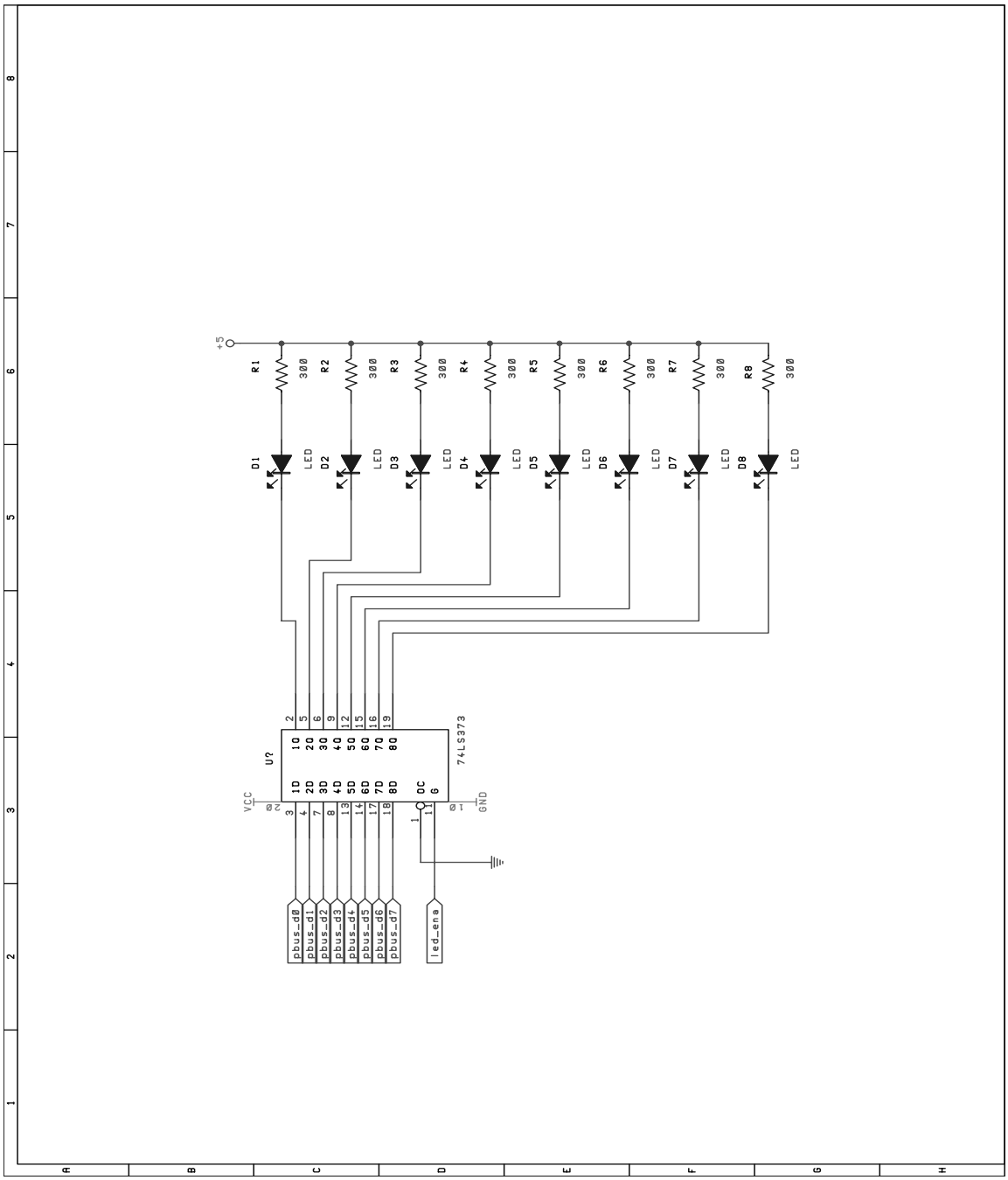


Figure 46: Output assign module, LED driver schematic. This module uses two such circuits, for a total of 16 LEDs.

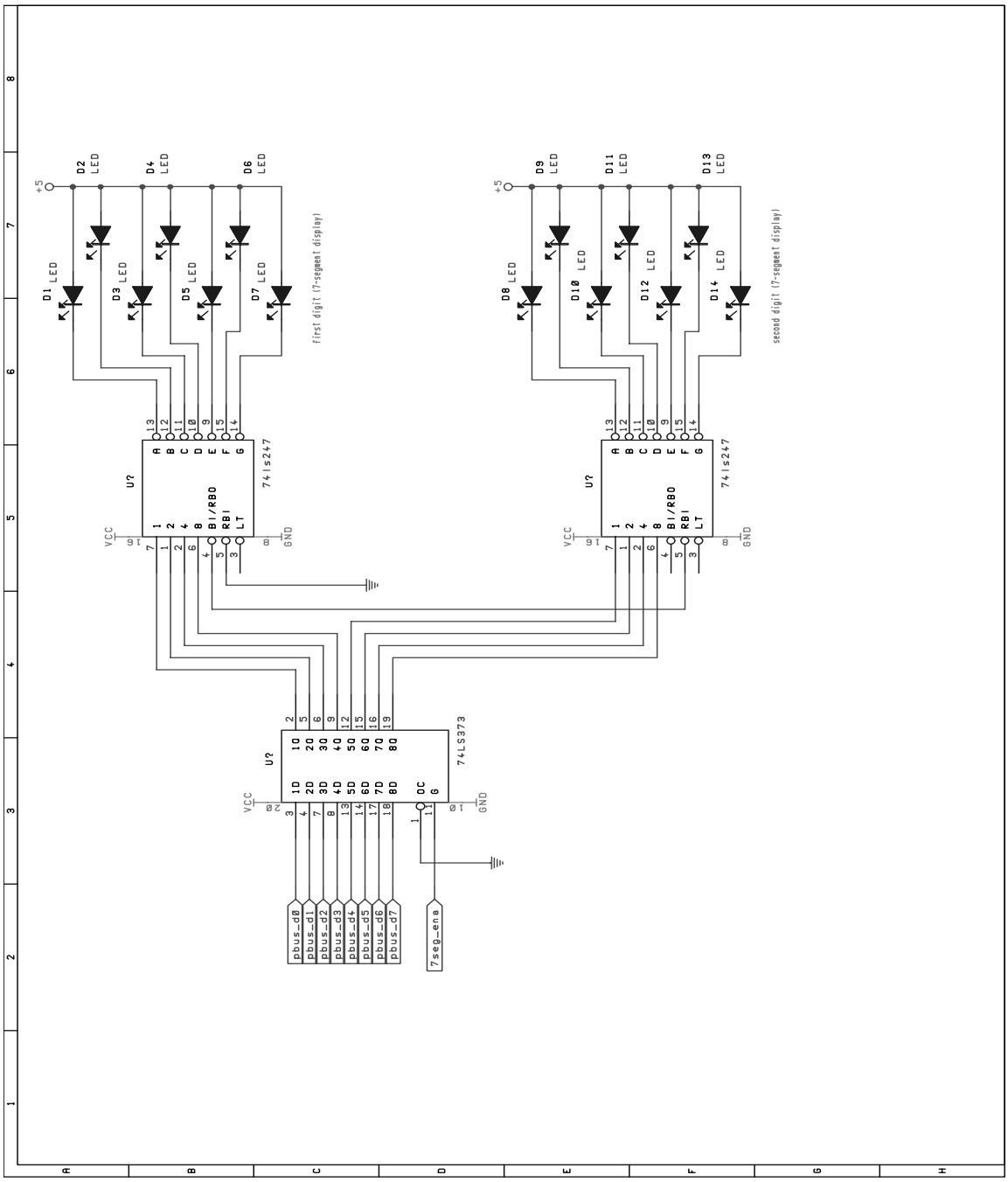


Figure 47: Output assign module, 7-segment display decoder and driver schematic.

GALs is included in the appendices, on page 225.

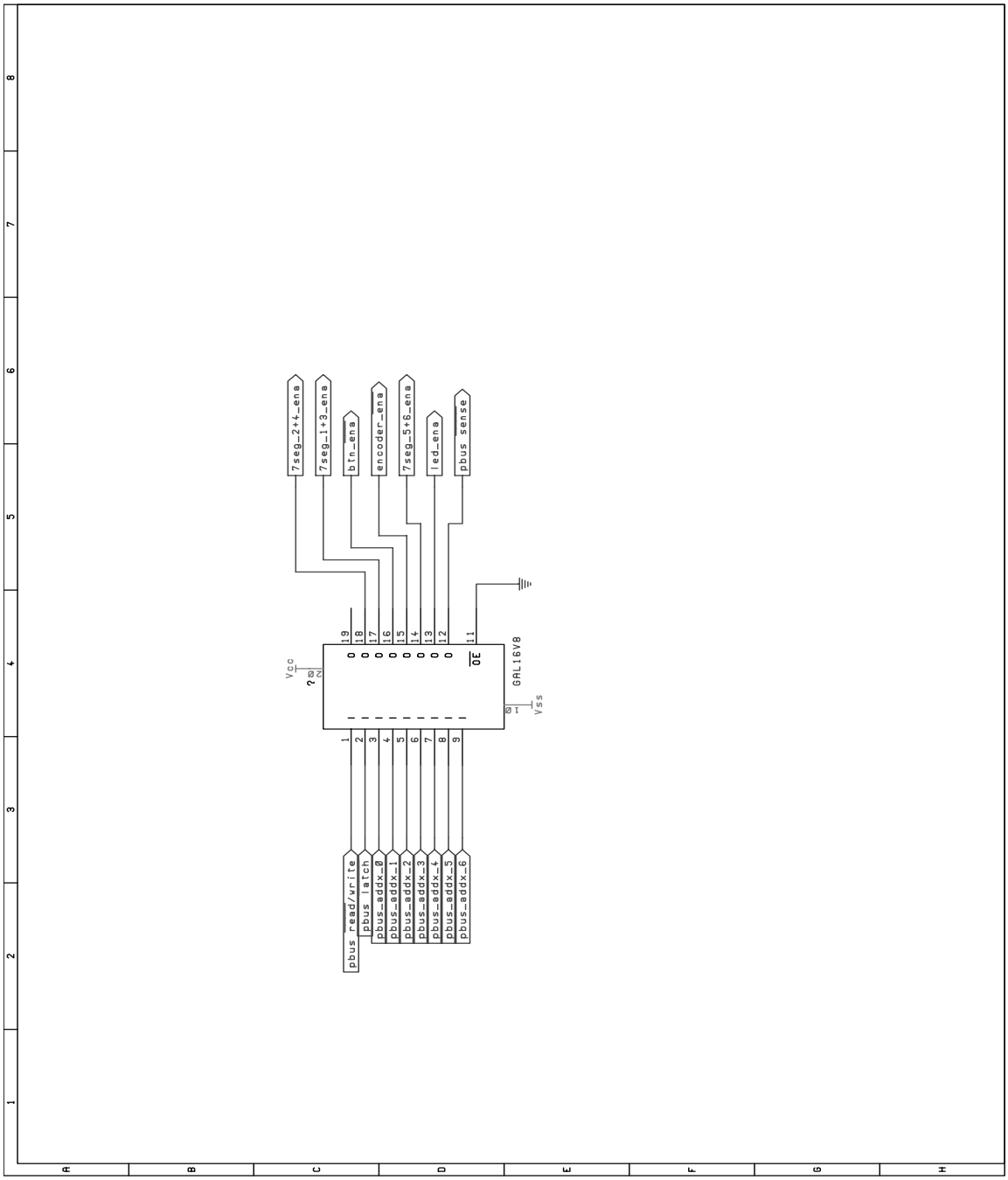


Figure 48: Transport control module, address decoding GAL schematic.

The transport control module contains 7 pushbuttons, the status of which need to be detected by the microcontroller. The identical latching and firmware debouncing scheme is used in this module as is used in the output assign module. Figure 49 shows this circuit.

A rotary shaft encoder is used as a data entry wheel in the transport control module. This encoder outputs two bits, which represent the position of the shaft. Hardware quadrature decoders can be used to give direction and clock signals. However, in the interest of conserving

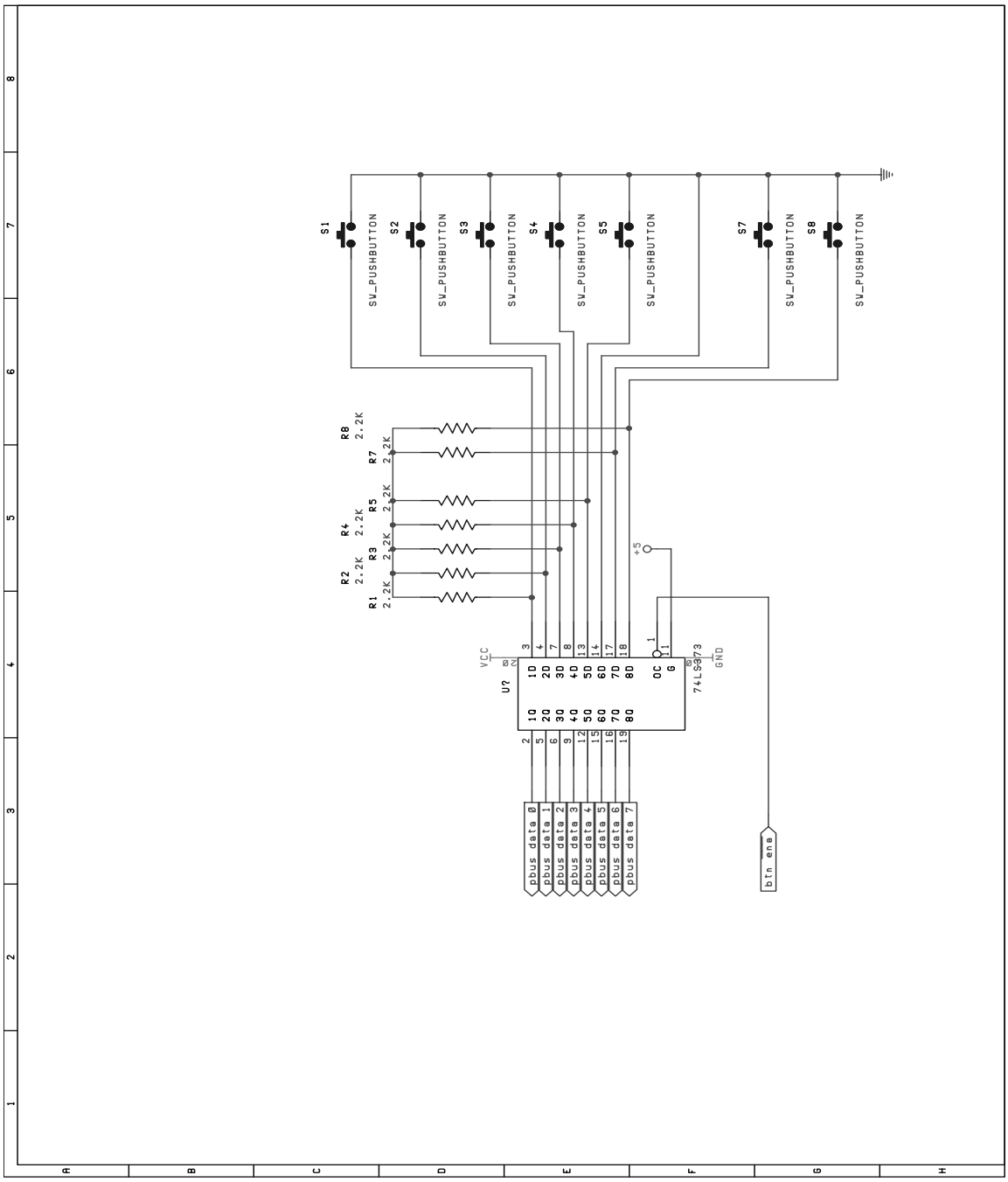


Figure 49: Transport control module, momentary pushbutton decoder schematic.

board space, this decoding will take place in firmware. A simple latching scheme is used to input the data from the encoder. Figure 50 shows the circuit for the encoder. Note that the rotary encoder present in the universal user interface section of the control board also connects to this circuit.

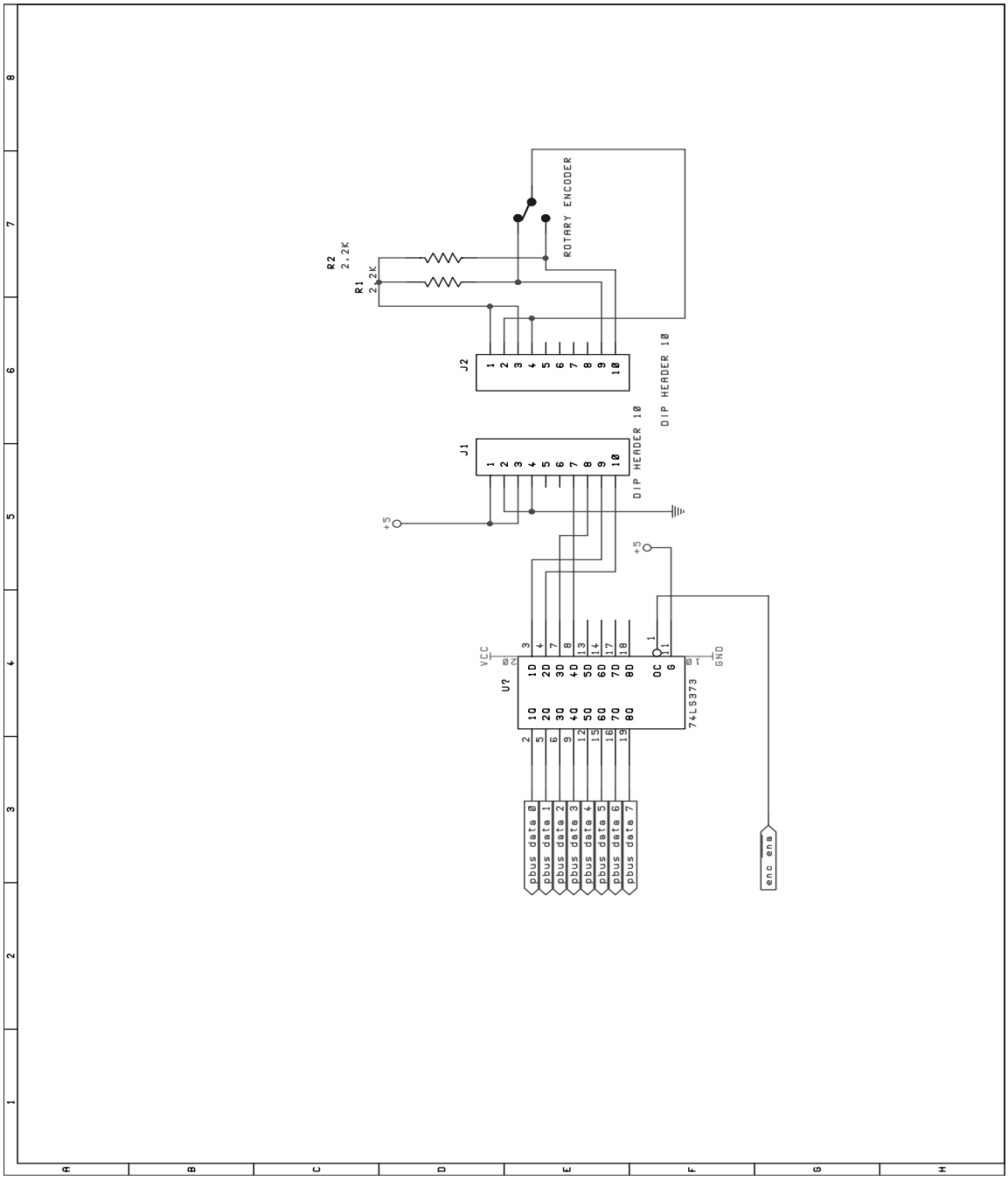


Figure 50: Transport control module, rotary encoder interface schematic. This also acts as the interface for the encoder on the universal interface module.

Three LEDs are present on the transport control module, to indicate various statuses. A simple 74LS373 current-sinking setup is used to drive these LEDs, as shown in figure 51.

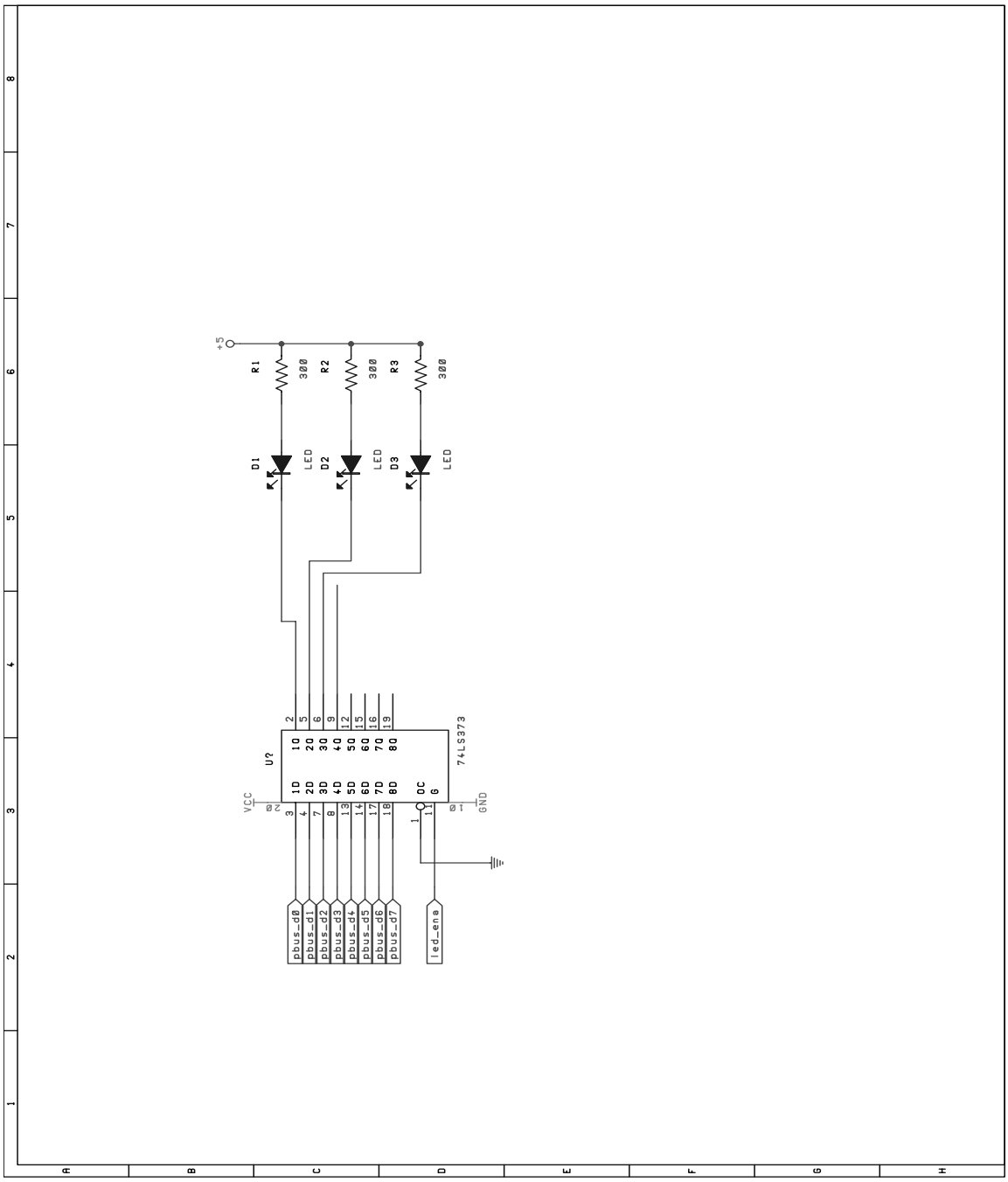


Figure 51: Transport control module, LED driver schematic.

A pair of 7-segment displays are present in the transport control module to display track information. These displays are connected in a very similar fashion to those in the output assign module. Figure 52 shows the schematic.

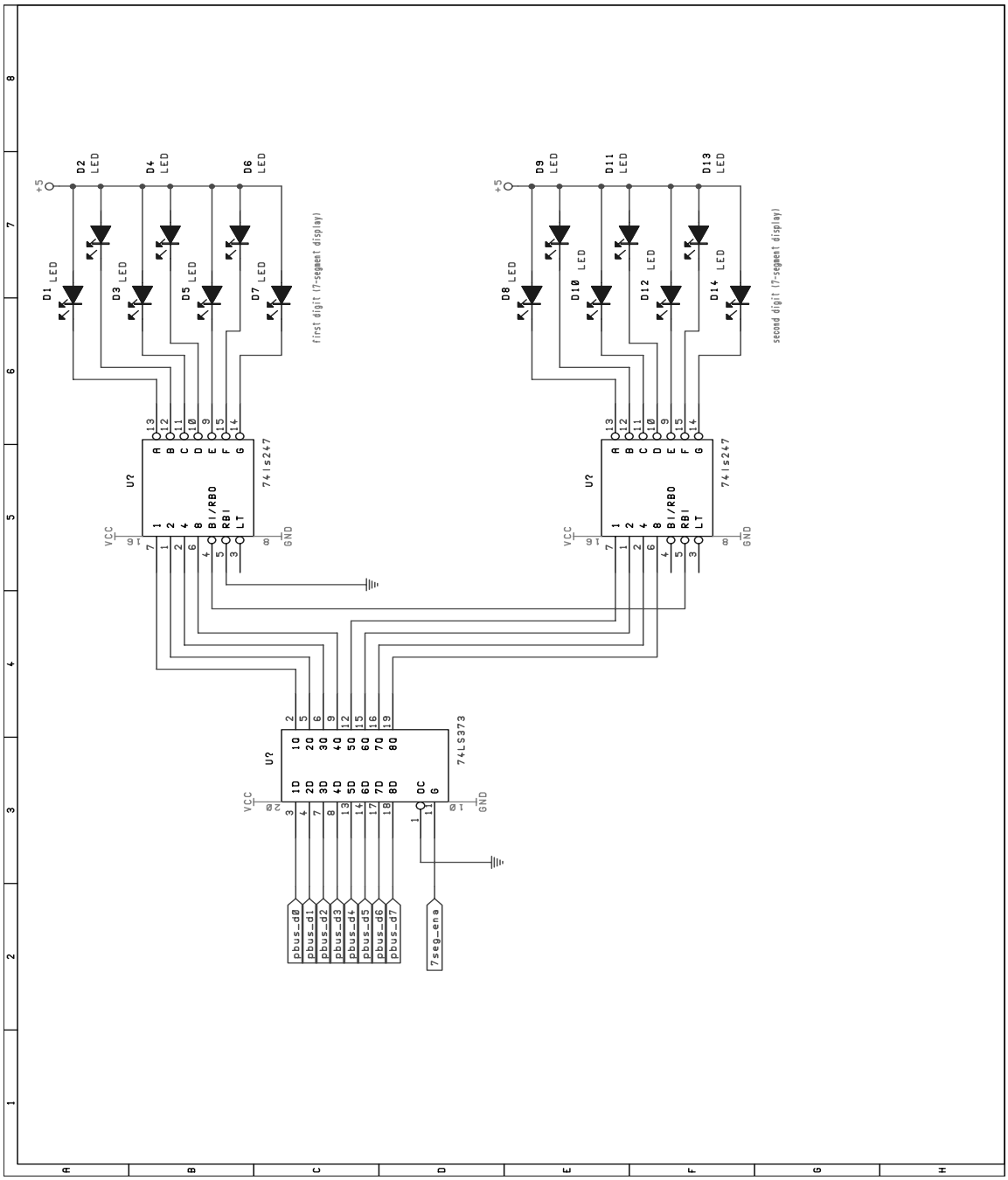


Figure 52: Transport control module, track select 7-segment display decoder and driver schematic.

Another four 7-segment displays are used in this module to display a time value. These displays are connected in a similar fashion to the track indicator, with the exception that the ripple-blanking and zero-blanking features are not used. Figure 53 shows the schematic used. Two of these circuits are present in the module.

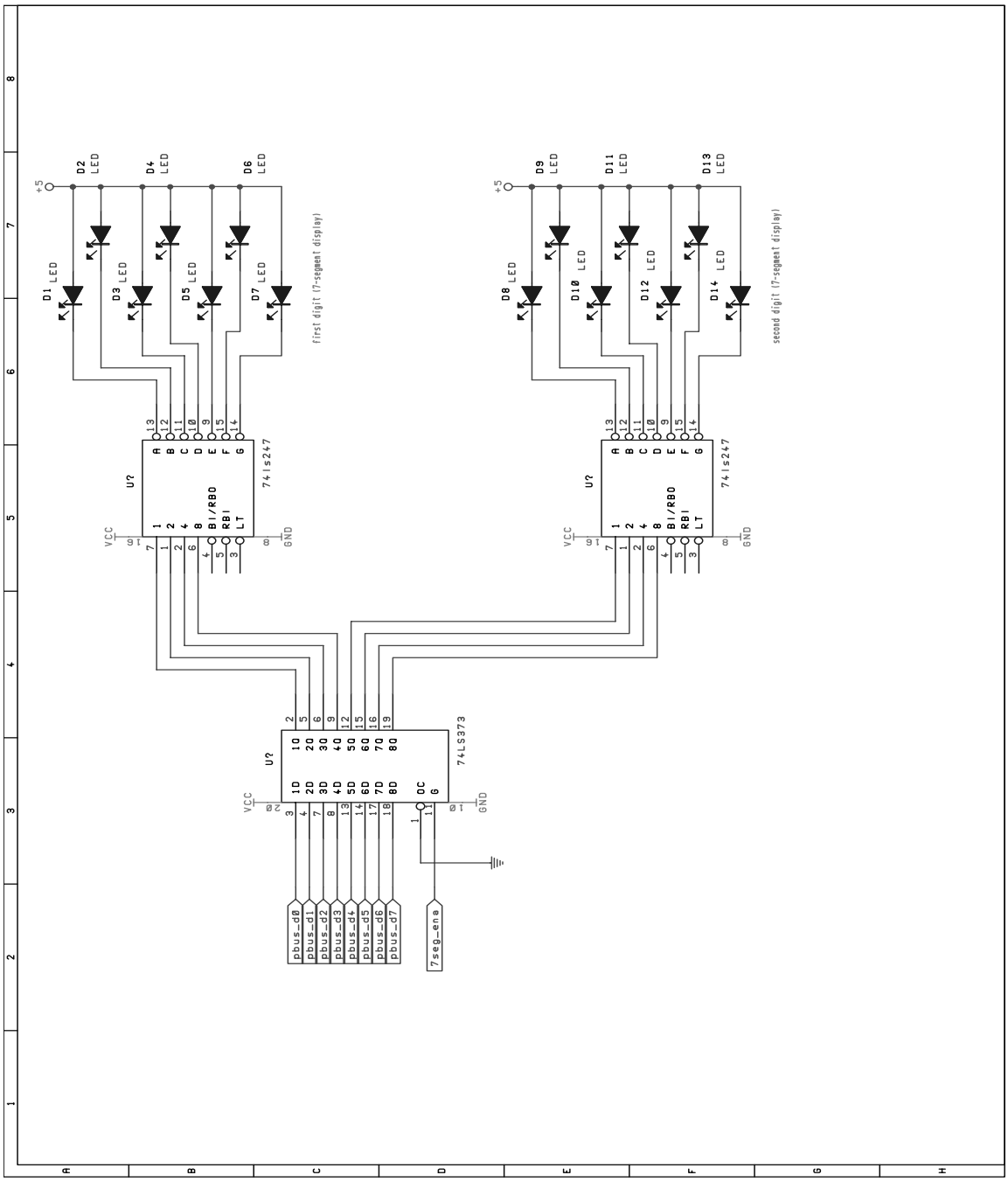


Figure 53: Transport control module, time indicator 7-segment display decoder and driver schematic.

6.1.4 Universal User Interface and Cue Stack Submodules

As previously mentioned, the rotary encoder present in the universal user interface section connects to the transport control section. Figure 54 shows this connection and the associated circuit.

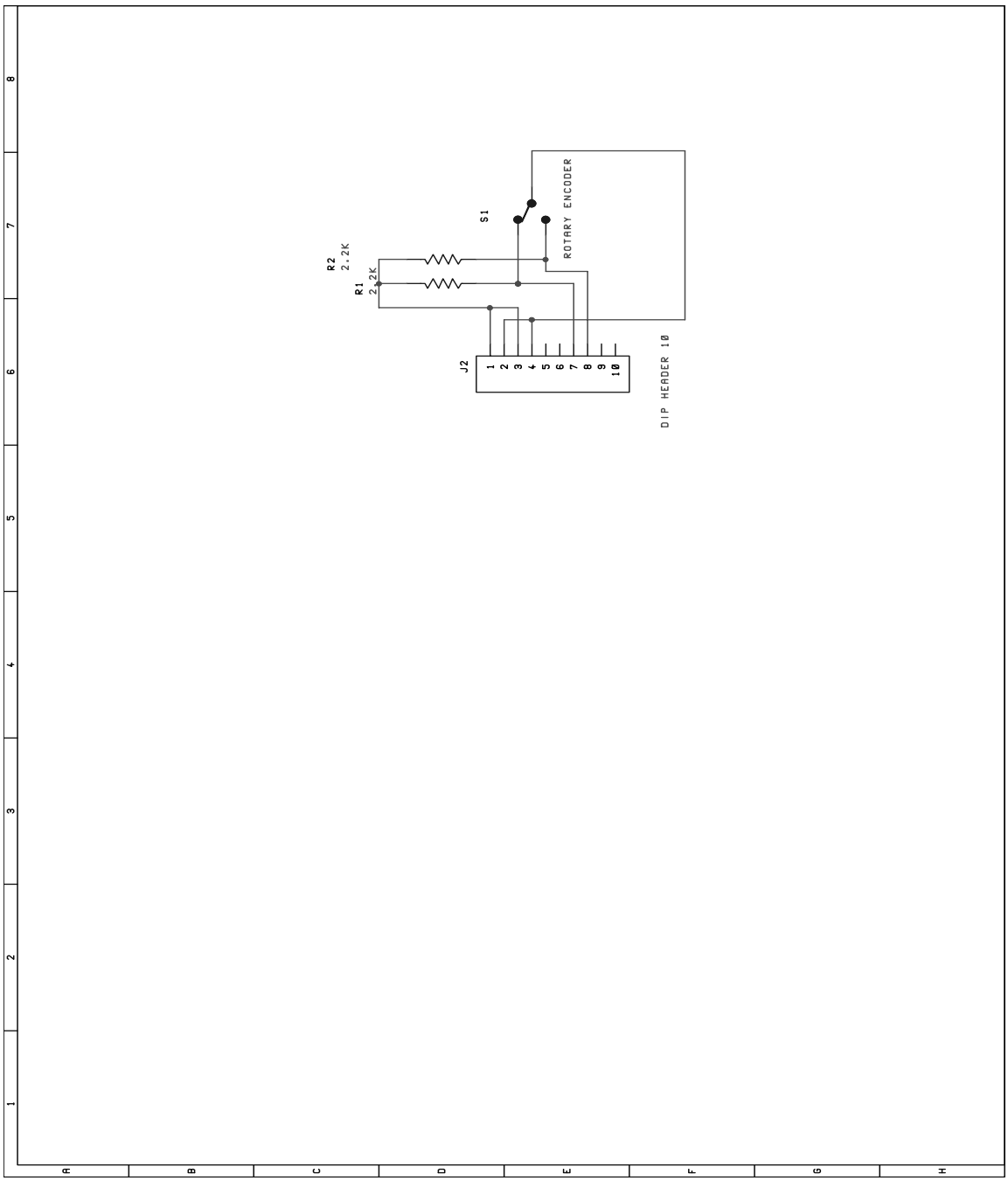


Figure 54: Universal user interface submodule, rotary encoder schematic. This circuit connects to the transport control encoder interface.

The universal user interface module also contains a pair of pushbutton switches. These switches interface to the output assign board, as shown in figure 55.

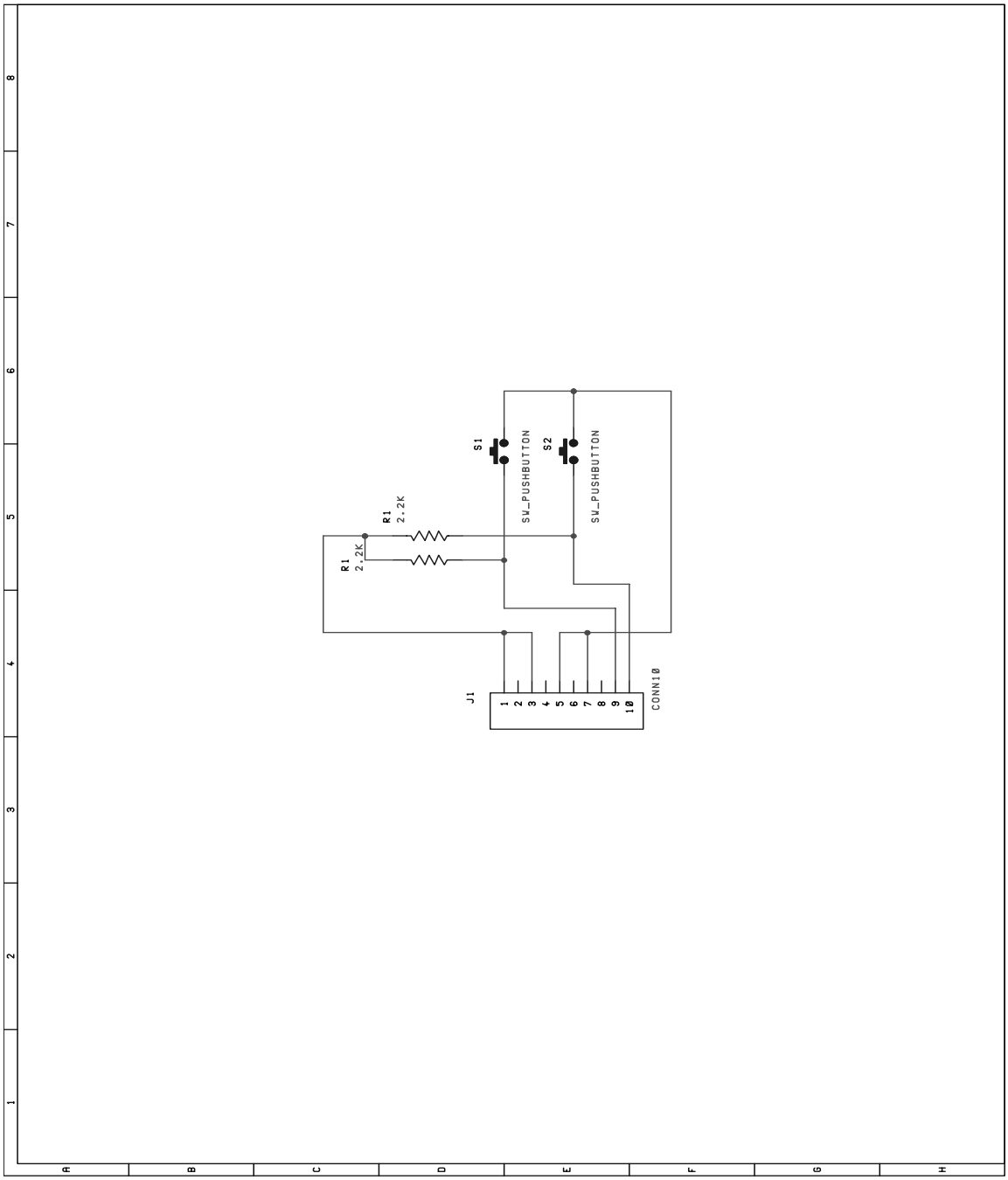


Figure 55: Universal user interface submodule, momentary pushbutton interface schematic. This circuit connects to the output assign button interface.

The cue stack control section contains three pushbutton switches that interface to the output assign module. Figure 56 shows this connection.

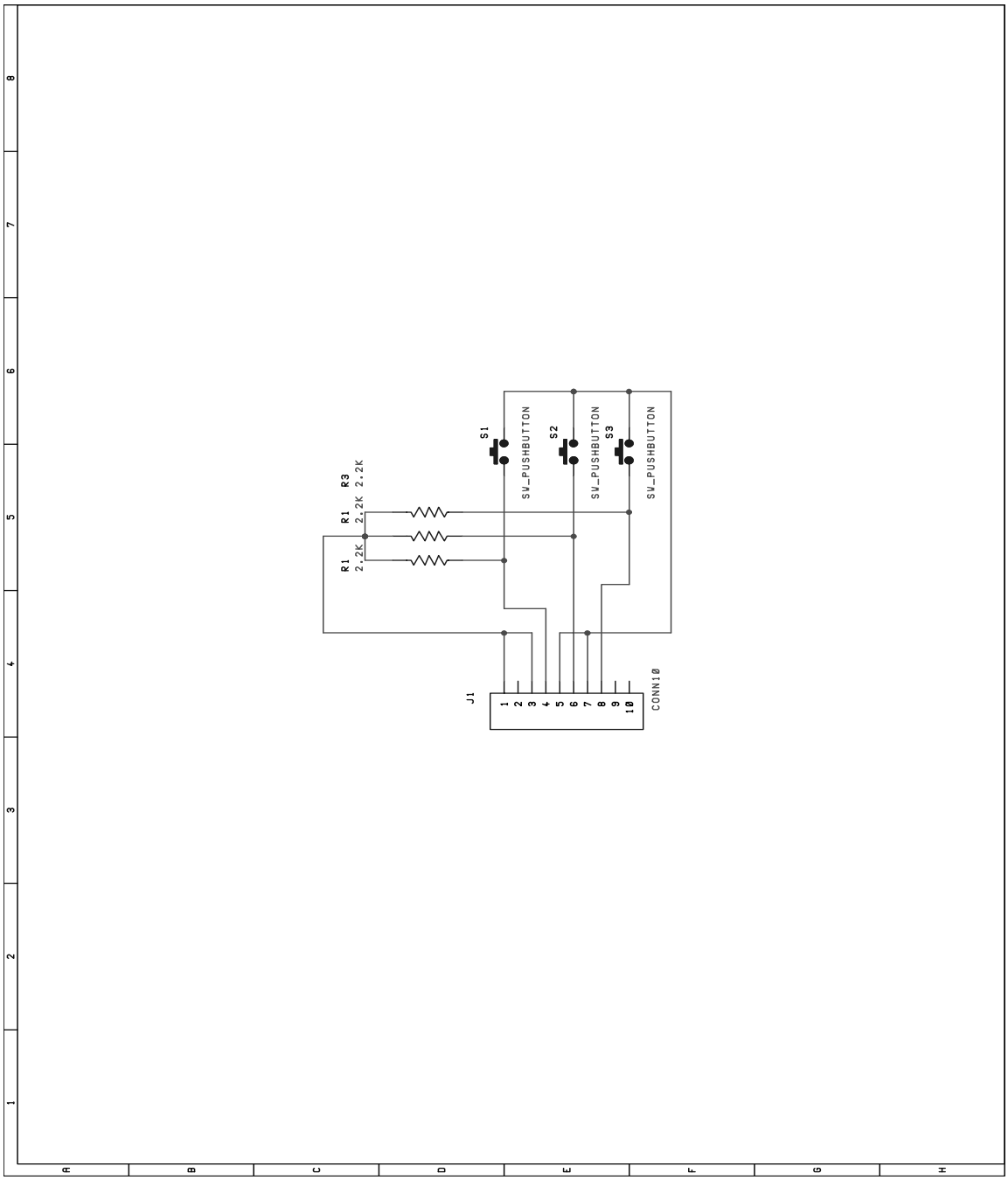


Figure 56: Cue stack submodule, momentary pushbutton interface schematic. This circuit connects to the output assign button interface.

6.1.5 Microcontroller Module (pbus and LCD interfaces)

(to be converted from paper documentation from Axiom.. also need schematic capture of 8255-based Pbus controller, which is on paper only)

6.1.6 Power Supply

An off-the-shelf switch-mode power supply is used, thus no schematic is present. The supply provides +5V, and ±12V.

6.2 Mixer Unit

6.2.1 Audio Input Module

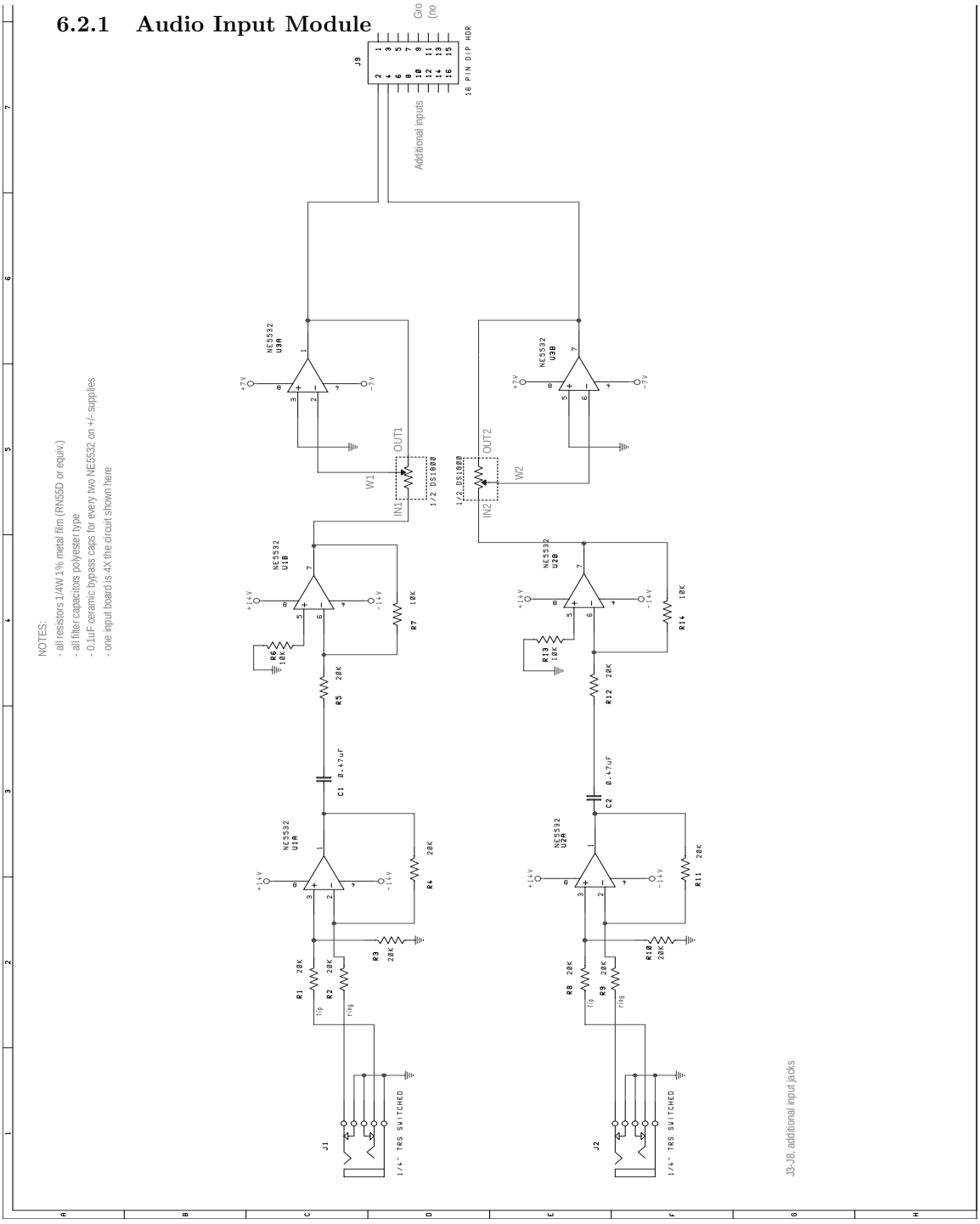


Figure 57: Audio input module, line receiver, buffer, and gain adjust schematic (2/8 of module).

This is the original design for the audio input module. Through clever use of a normalling jack, either balanced or unbalanced signals may be input. Capacitive coupling is used to remove any residual DC offset from the input.

A trim stage, based on the Dallas Semiconductor DS1800 digital potentiometer, provides audio trim capabilities. Unfortunately, when this circuit was designed (and later fabricated), it was not known that the DS1800 is not a bipolar device, thus it is inappropriate in this situation. For the prototype, this functionality was sacrificed, but a stock of National Semiconductor LM1972 bipolar digital volume controls was obtained. At some point in the future, the circuit will be redesigned using this part.

6.2.2 Audio Mix Module

The mix module, the central component in the mixer unit, is actually fairly straightforward. Each module contains four of the **Analog Devices** SSM2163 8-to-2 attenuating mixer chips. The eight audio inputs on each of these chips are connected to the audio input buses, in a one-to-one fashion. The left output of each of the chips passes through a resistor and on to one of the output buses. The resistor is present here because of the reconfigurable mix architecture. The outputs of several chips are combined (either with the bus combiner/switcher module, or by the presence of several mix modules in the unit). The combination is accomplished through the use of a standard op-amp summing circuit, present on the input stage of the audio output module.

Figure 58 shows 1/4 of the audio connections for an audio mix module. An entire module is simply this same circuit repeated four times, with each output connected to a separate output bus point.

As in other DACS modules, a standard 16V8 GAL is used for address decoding. Figure 59 shows the pin assignments for the GAL. The VHDL code used to generate the GALs is included in the appendices, on page 230.

Such that all mixer chips on a module could be simultaneously, yet individually programmed, their data and enable inputs are all connected to a 74LS373. All clock signals are tied together, and to the Pbus master clock signal through a pair of 74LS04 inverters (acting as buffers in this case). This means that on any given write cycle, four mixer chips may be programmed at once, but if the need arises, a single mixer may be programmed without disturbing the other chips. Figure 60 shows the digital schematic of the mixer module.

6.2.3 Bus Combiner and Switcher Module

Relays were chosen over an electronic switching scheme for reasons of cost and simplicity. Many types of audio-quality electronic switches exist, but most are rather expensive. Additionally, many have limitations such as operating voltage ranges, latch-up problems, etc. Relays are inexpensive, and do not have these limitations. However, relays consume a fair amount of current, and depending on the type, they may not switch silently. These limitations seem to outweigh the potential problems involved with electronic switches, however.

To switch the source of the second input bus between the second group of inputs and the first group of inputs, four DPDT relays are used. The normally-closed portion of these relays connects the second bus to the second group of inputs by default. Figure 61 shows the schematic for the audio portion of this circuit.

In a similar fashion, the source of the third input bus is switchable between input group one and input group three. Four DPDT relays are used for this switching, with the normally-closed portion defaulting the switcher to group three. Figure 62 shows the schematic for the audio portion of this circuit.

Lastly, the source of the fourth input bus is switchable between input group four, and the input *bus* two. This allows the fourth input bus to be sourced from the fourth, second, or first input group. Figure 63 shows the schematic of the audio portion of the circuit.

In addition to switching the input buses, the output buses can be combined, down to just four outputs. This is done in only two steps. Combining the first output bus with the second output bus reduces the number of outputs to from sixteen to eight. Further combining the top four with the bottom four lines reduces the number of outputs to four.

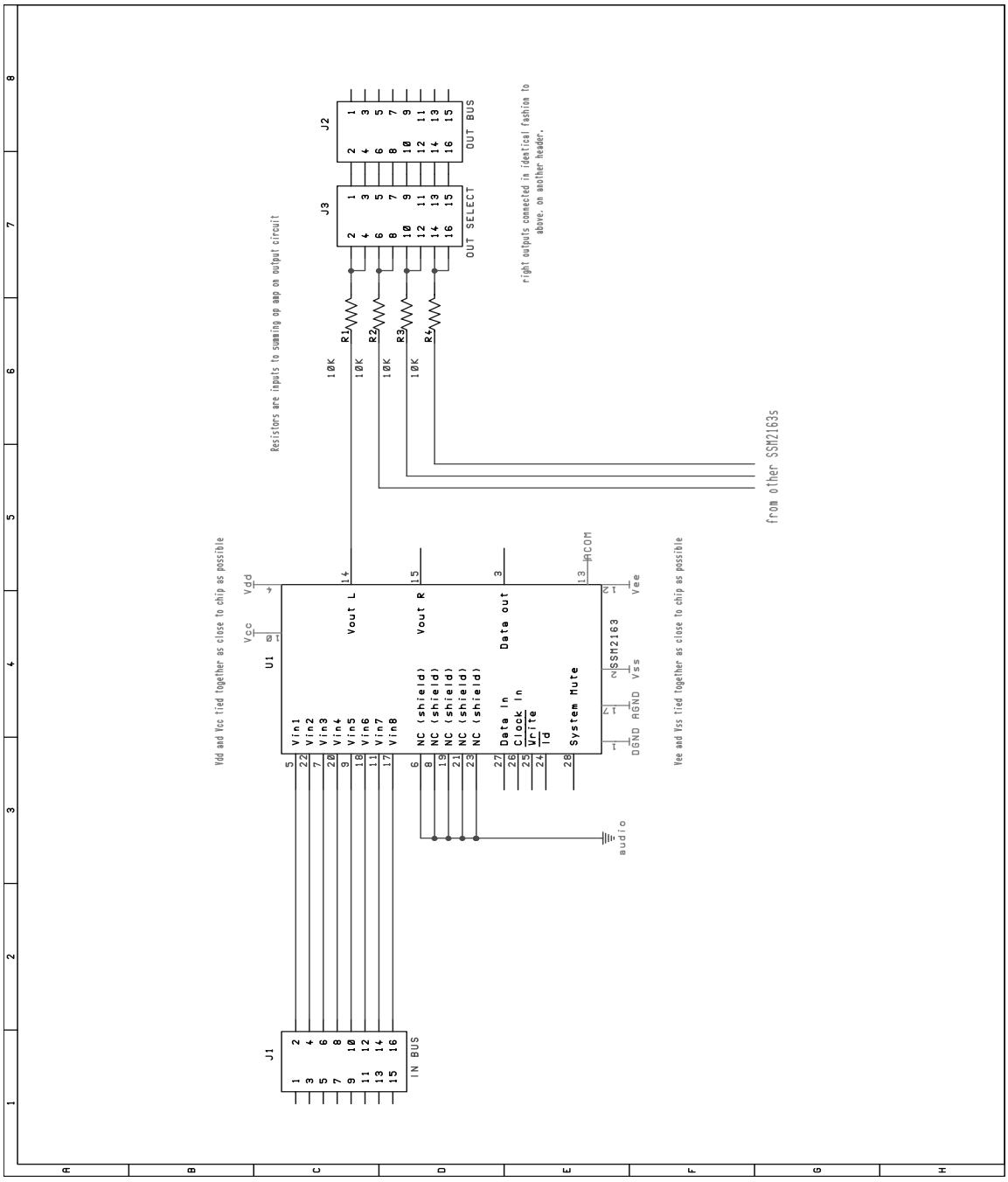


Figure 58: Audio mix module, analog circuit schematic (1/4 of module).

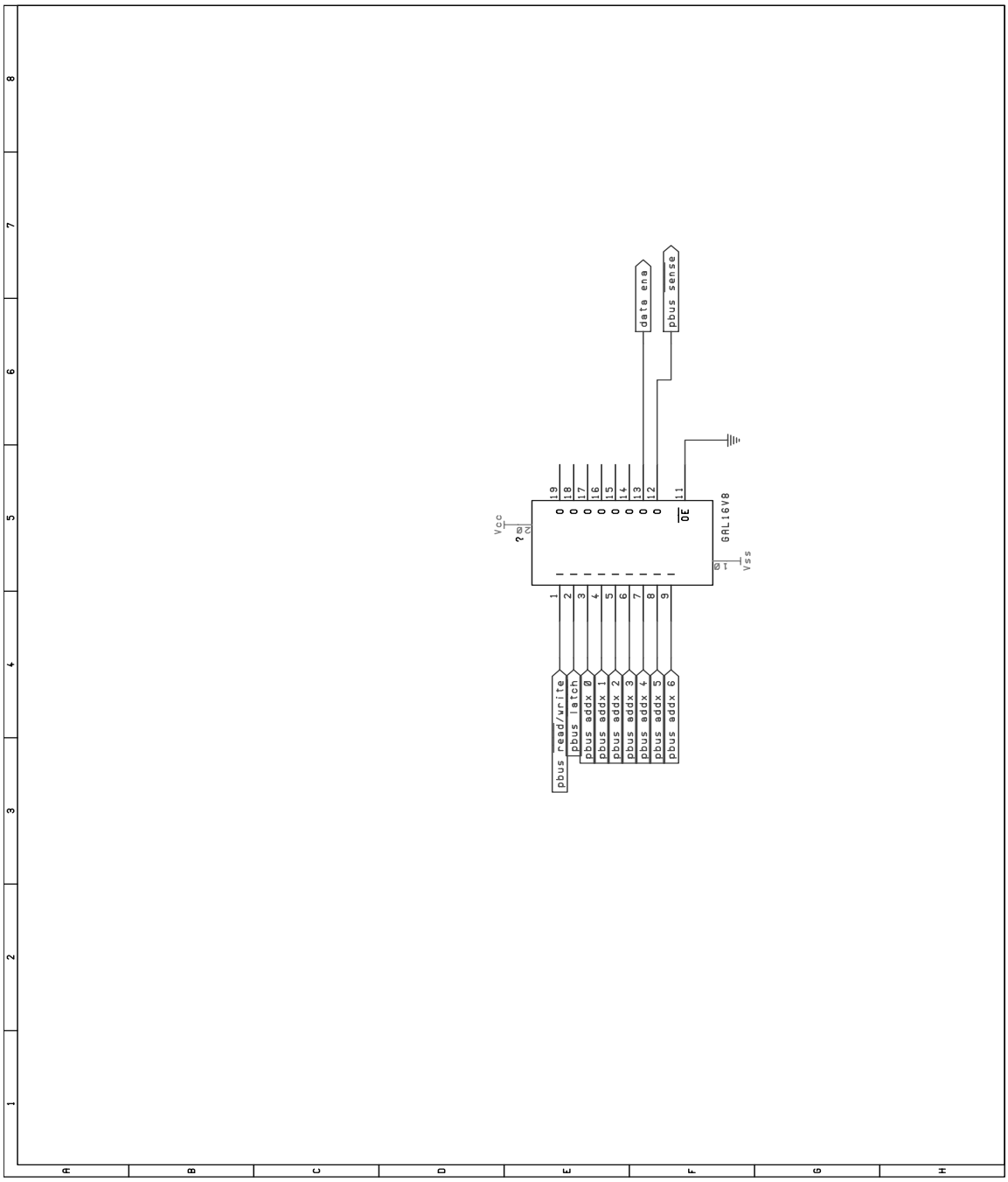


Figure 59: Audio mix module, address decode GAL.

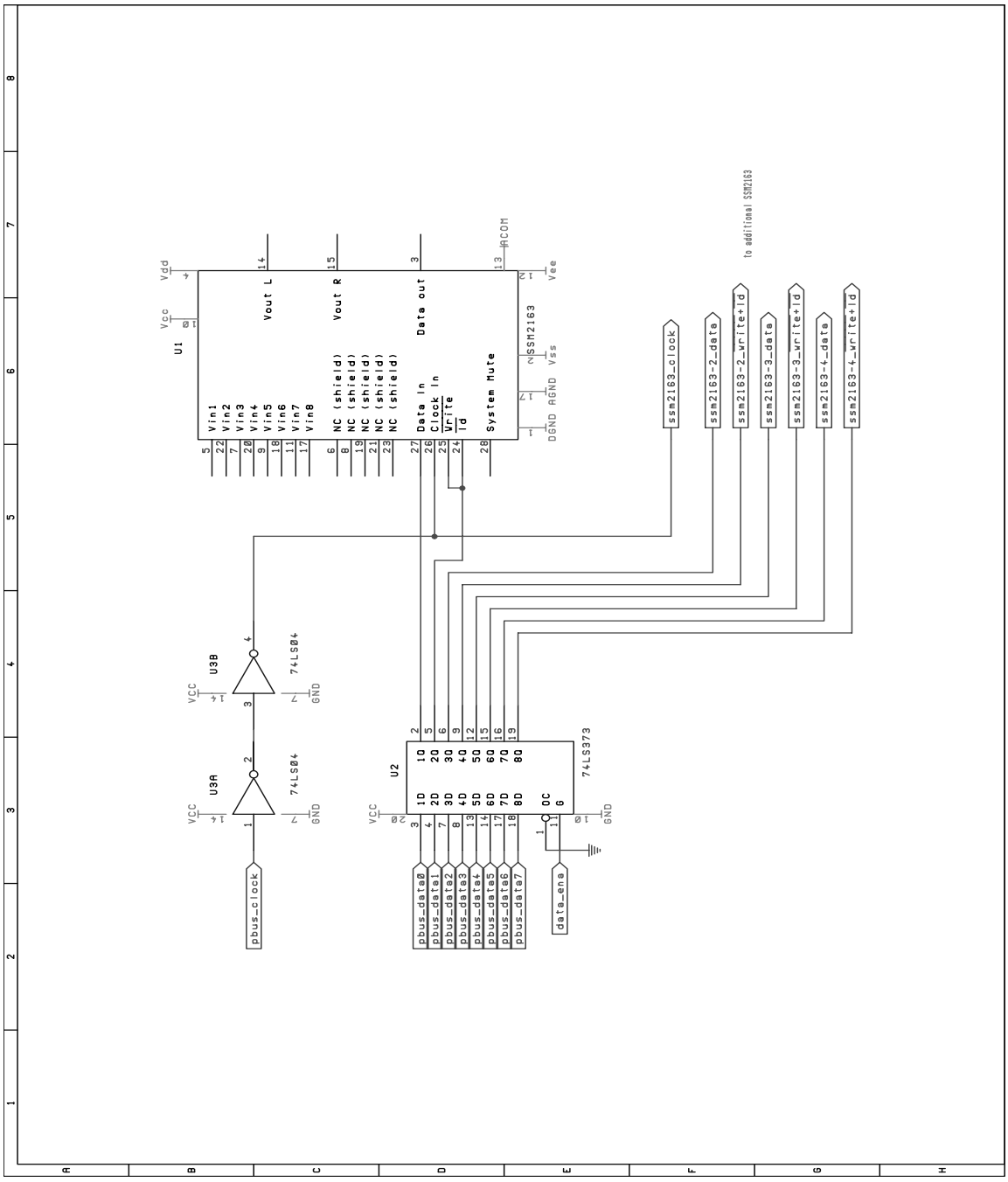


Figure 60: Audio mix module, digital circuit schematic (1/4 of module).

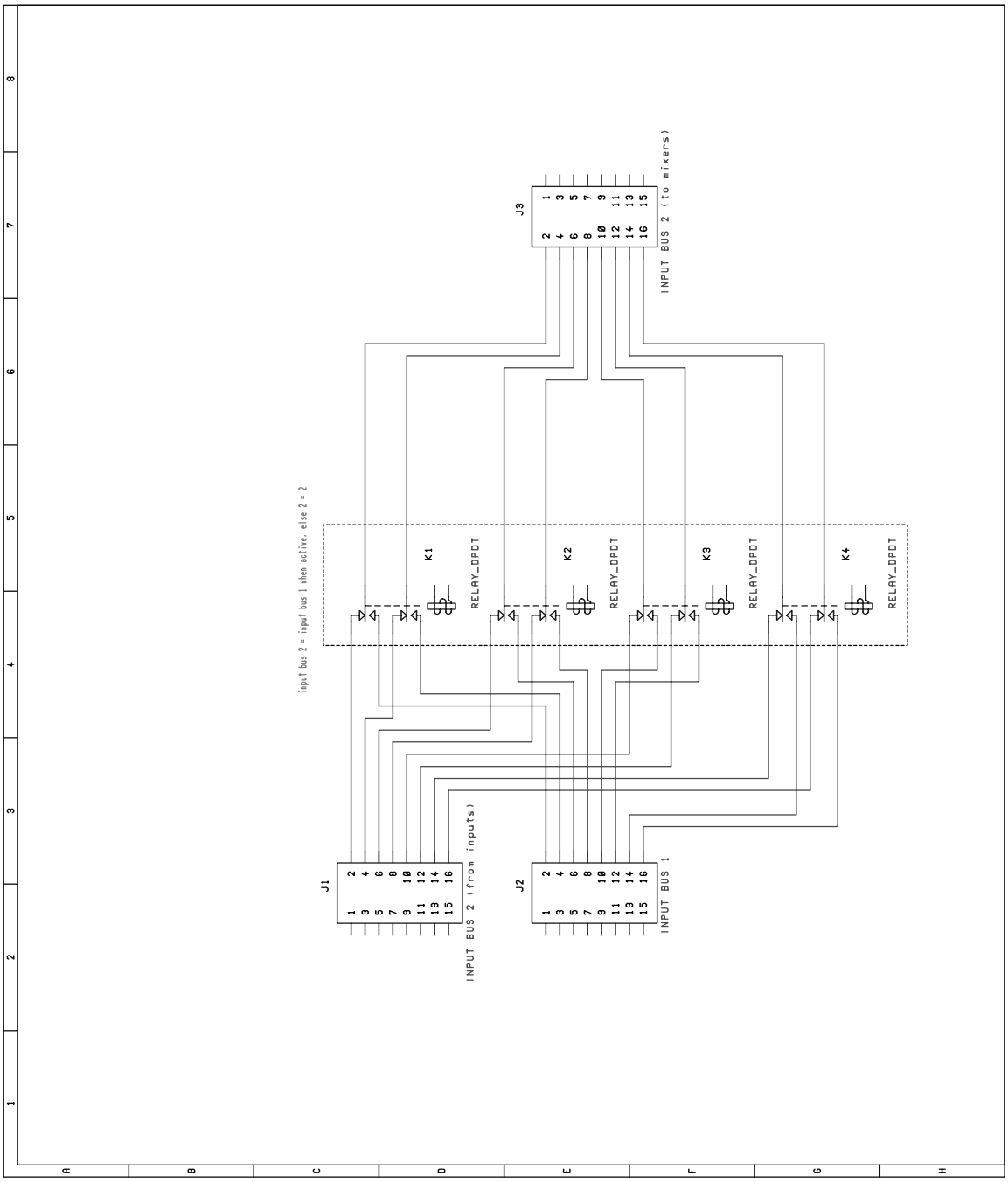


Figure 61: Bus switcher module, audio bus switching scheme for input bus 2.

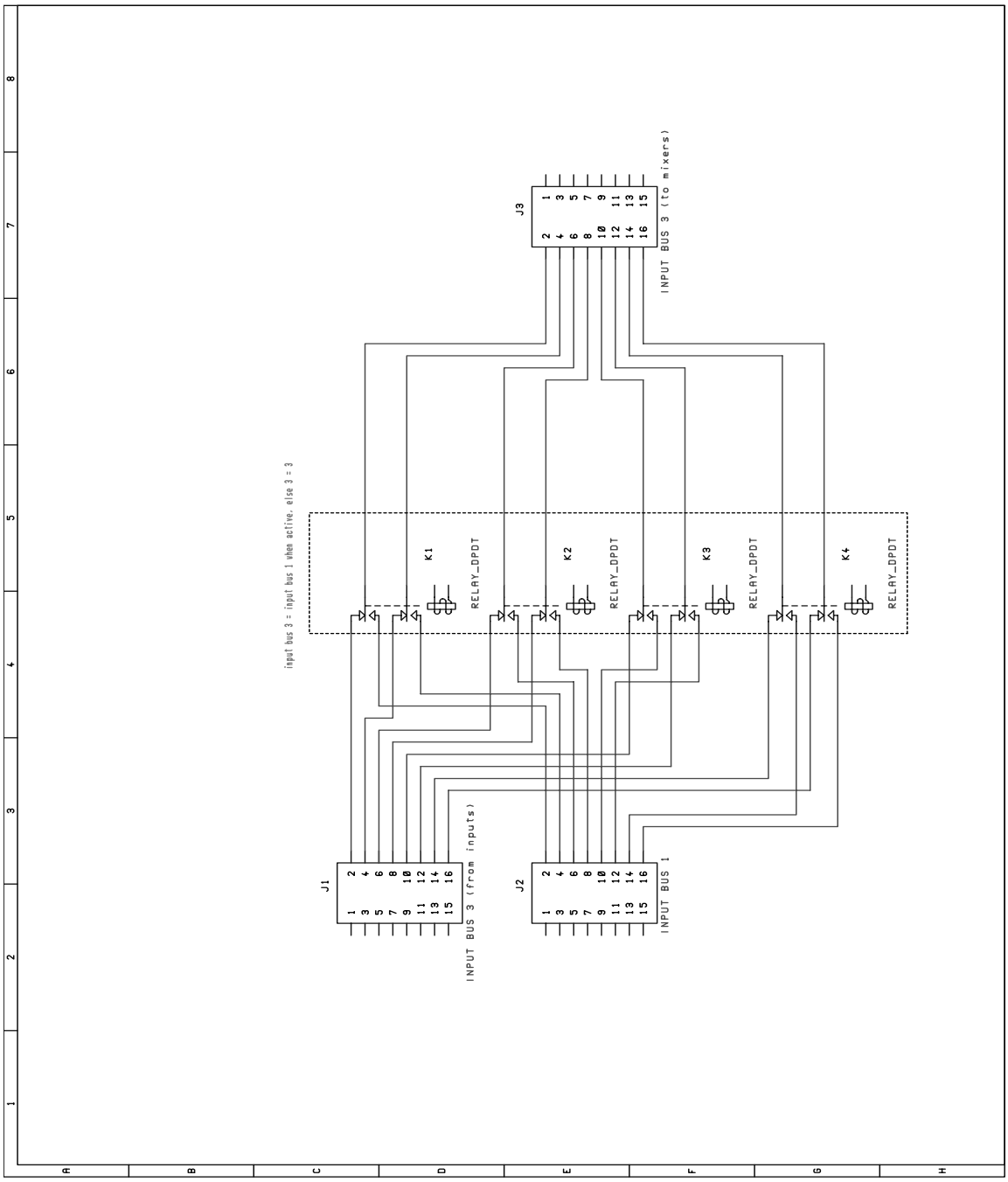


Figure 62: Bus switcher module, audio bus switching scheme for input bus 3.

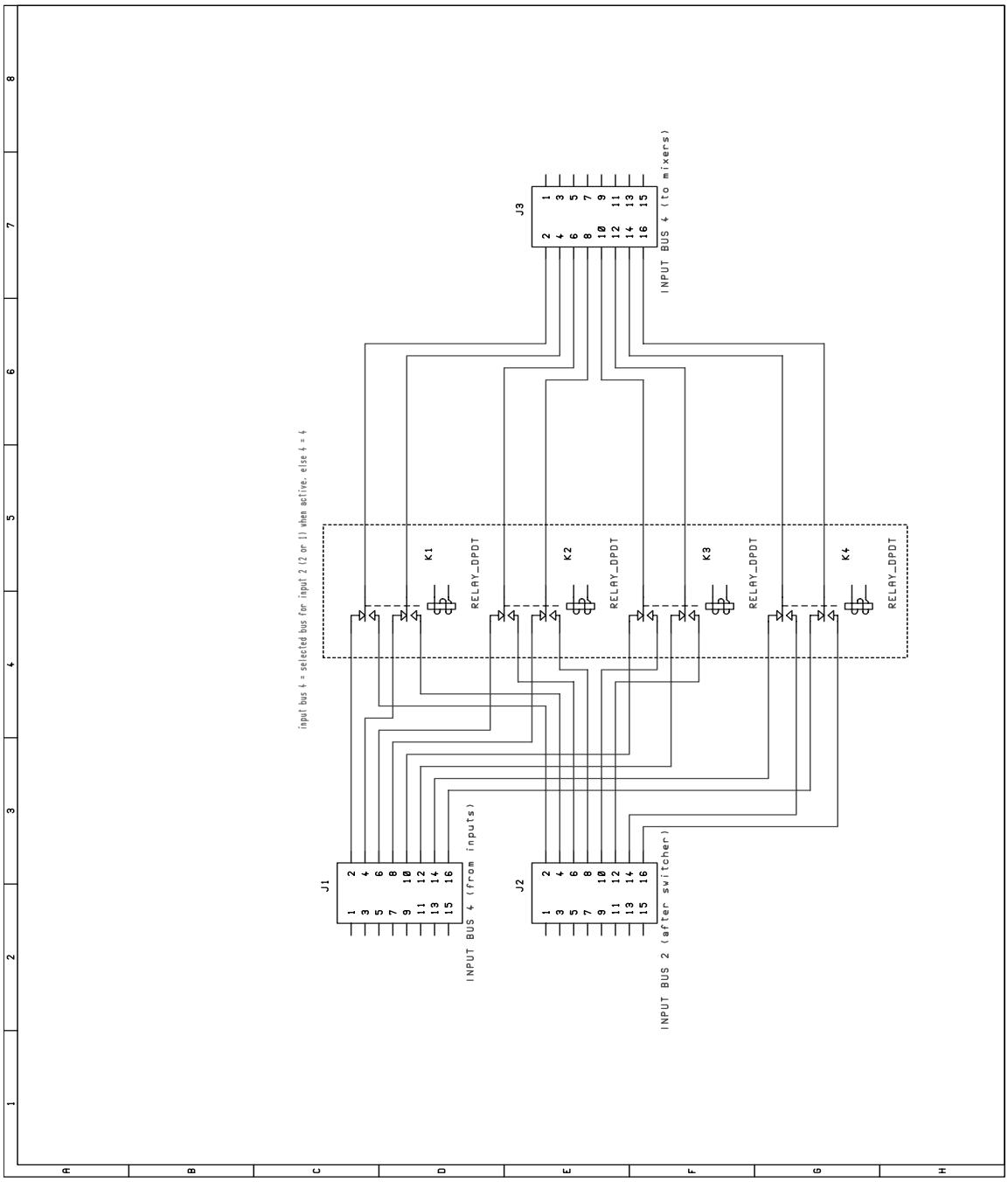


Figure 63: Bus switcher module, audio bus switching scheme for input bus 4.

As in the other portions of the bus switcher/combiner module, DPDT relays are used for switching. Four relays are used to combine the first and second output buses. The normally-closed sections of the relays aren't used, while the normally-open sections are used to connect the respective lines of each bus together. To connect the first four with the last four, a pair of DPDT relays are used in a similar fashion. Figure 64 shows the schematic for this circuit.

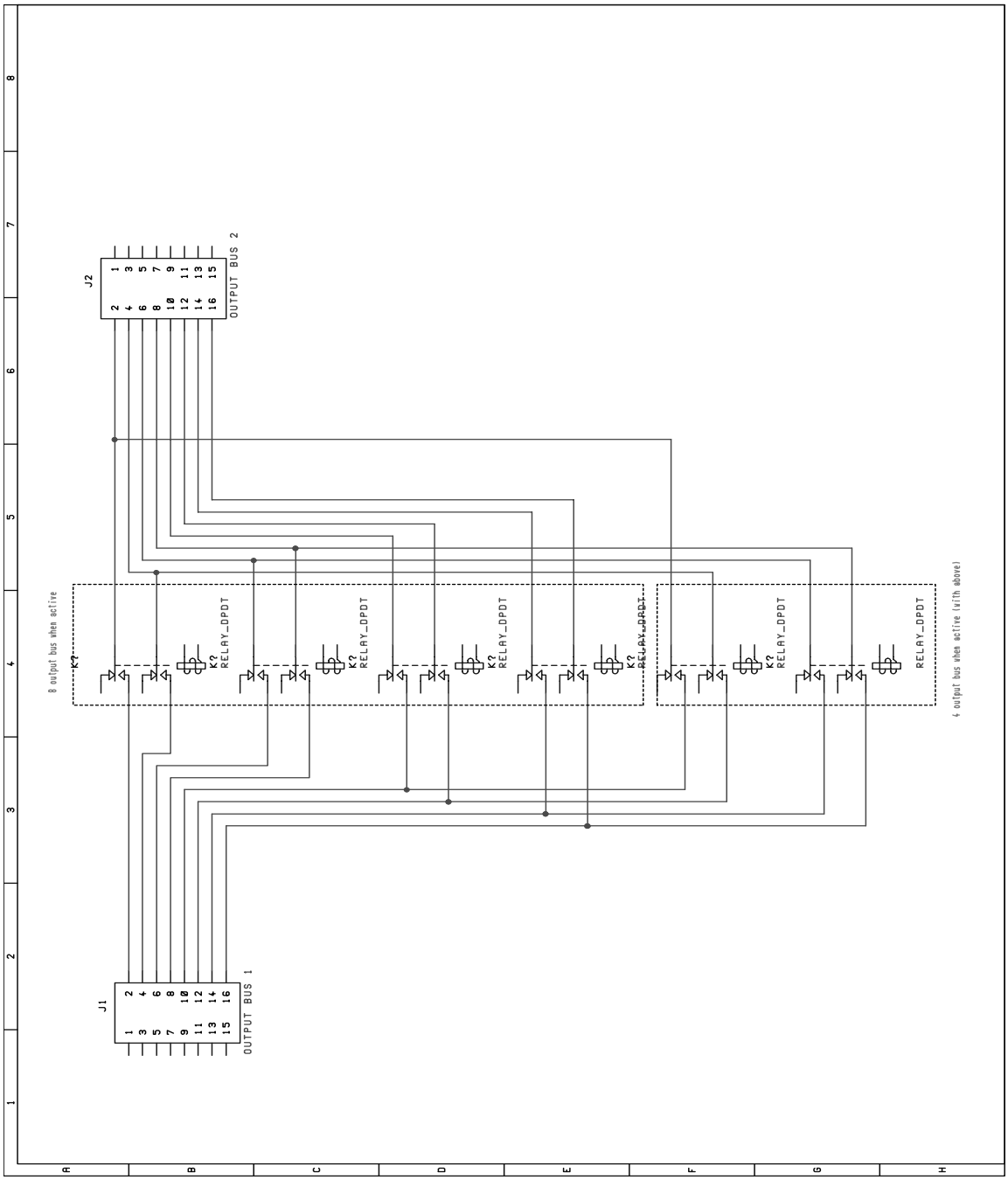


Figure 64: Bus combiner module, output bus combination schematic.

As with other Pbus modules, a standard 16V8 GAL is used for address decoding. Figure 65 shows the pin assignments for the GAL. The VHDL code used to generate the GALs is included

in the appendices, on page 231.

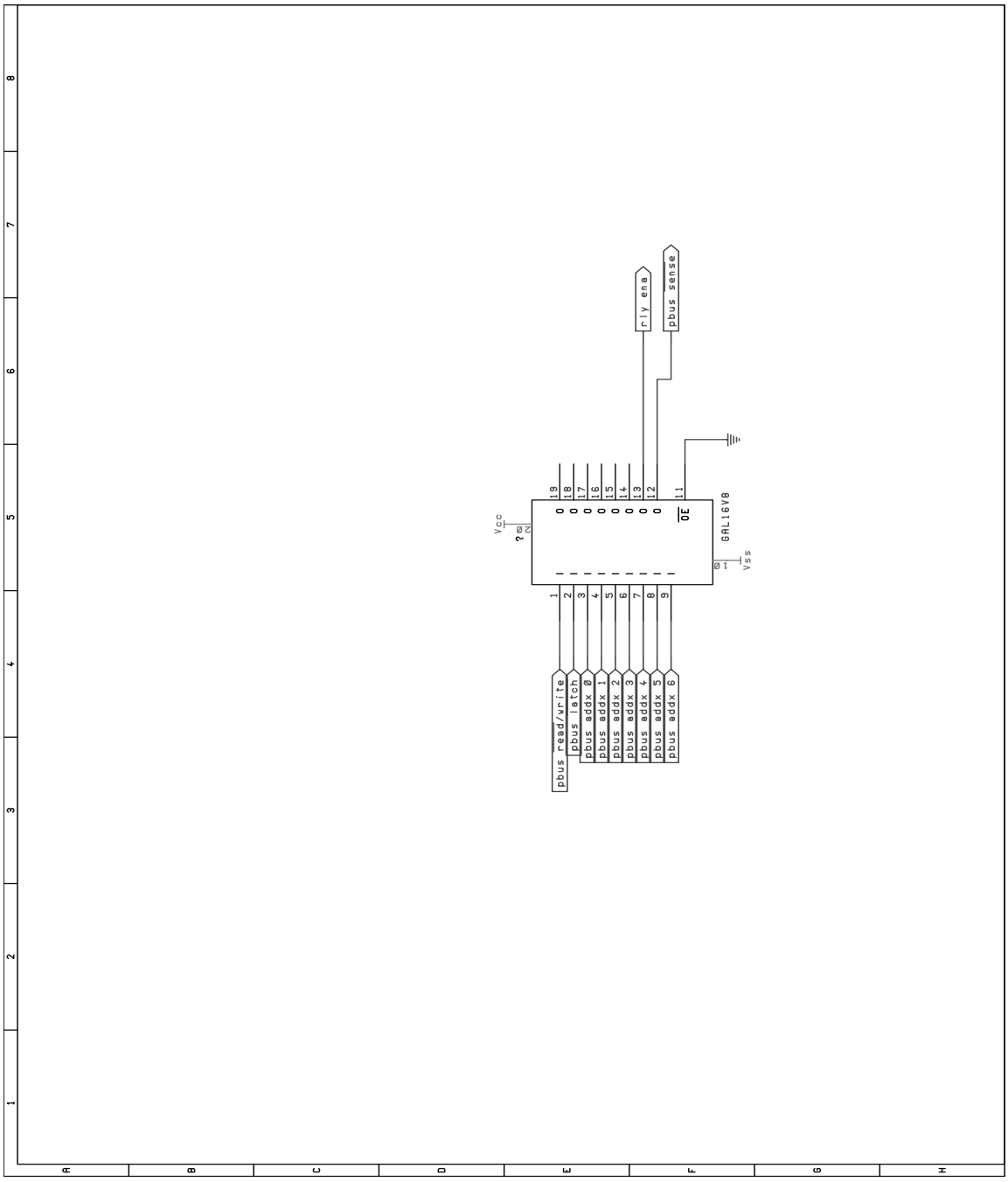


Figure 65: Bus switcher/combiner module, address decoding GAL.

To drive the relays, a series of NPN power transistors are used. The transistors are driven by a 74LS373 latch. Diodes are present across all relay coils, to prevent transistor damage due to the inductive kick of the relays. Figure 66 shows the schematic for this circuit.

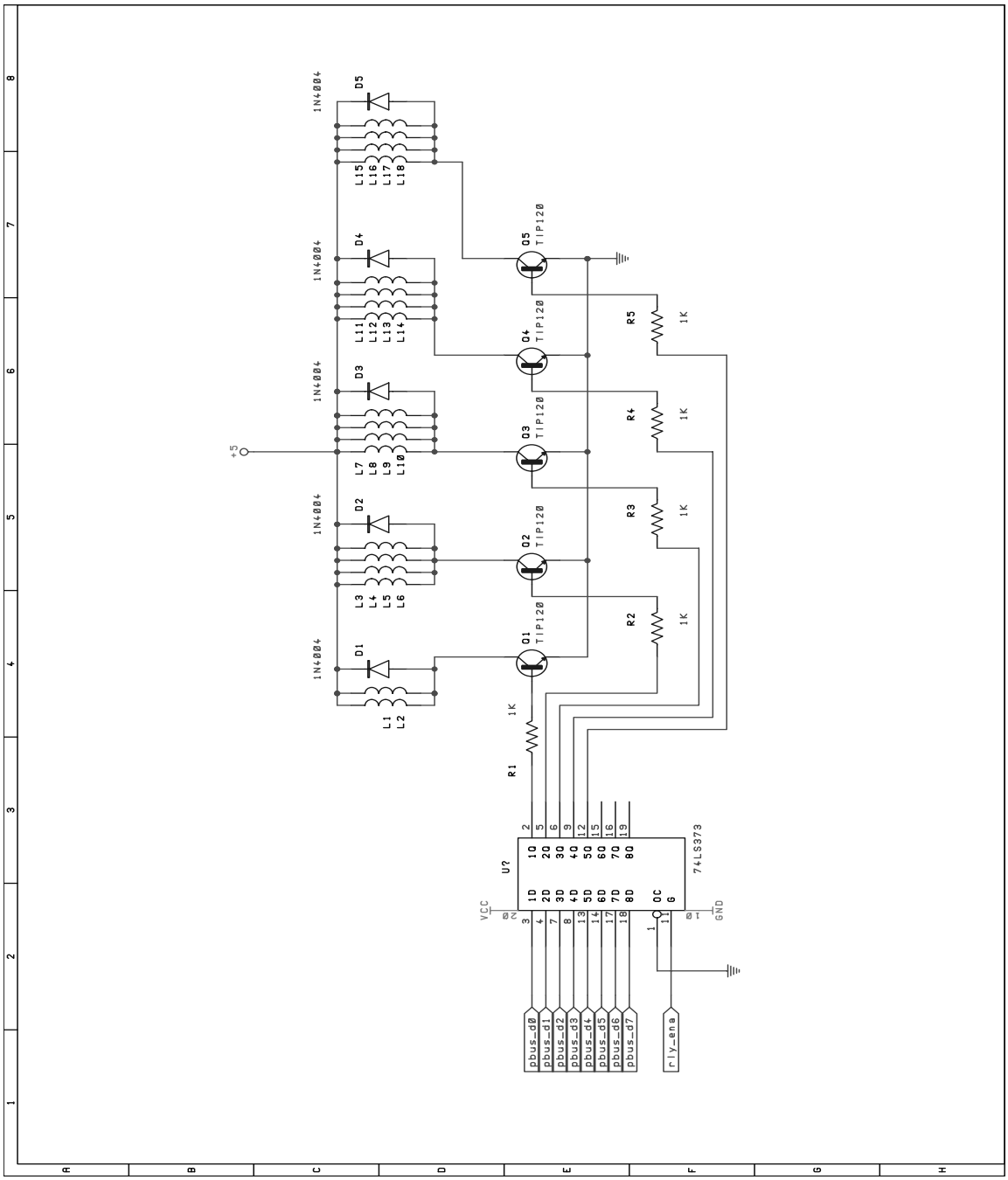


Figure 66: Bus switcher/combiner module, digital and relay driver schematic.

6.2.4 Digital VU Module

(Not yet fully designed; preliminary design on paper, not run through a schematic capture system yet.)

6.2.5 Audio Output Module

The audio output module is fairly straightforward in that it has no digital section. Two varieties of output modules exist: balanced and unbalanced.

The unbalanced output module uses 1/2 of an NE5532 dual op-amp as an output buffer/amplifier. This same op-amp also acts as the summing amplifier, to sum the signals from all of the mixer chips together. Figure 67 shows two channels of the unbalanced output stage in schematic form.

The balanced output modules uses 1/2 of an NE5532 dual op-amp as a summing amplifier, to sum the signals from all of the mixer chips together. Two additional op-amps are used to generate buffered and inverted signals, to provide a balanced (differential) output. Figure 68 shows two channels of the balanced output circuit.

NE5532 op-amps are rated to acceptably drive a 600 ohm load, thus conceivably making the system capable of driving 600 ohm terminated lines. However, research [1] suggests that "the popular NE5532 dual IC, which is found in unbuffered form as a 'transformerless' output amplifier in many low-cost designs, will show a significant drop in its maximum sustained output level when terminated. The same IC coupled to a pair of high-current output driver stages, while more expensive to manufacture, is a legitimate line power amplifier." This is something of a concern, but the specifications for this output module do not dictate that the unit be capable of driving 600 ohm lines, as the maximum output level is specified in dBu, not dBm. The beauty of the modular design of the DACS mixer shines through here, as an output module capable of more effectively driving 600 ohm loads could be designed and substituted for this module. For now, this is not a major concern, as the intended use is more for input into high-impedance devices such as amplifiers, equalizers, etc.

6.2.6 Microcontroller Module (pbus and LCD interfaces)

(to be converted from paper documentation from Axiom.. also need schematic capture of 8255-based Pbus controller, which is on paper only)

6.2.7 Power Supply

As specified, the mixer unit employs a linear-mode power supply. The choice of this type of supply over a switching supply stems from noise problems associated with switching supplies. Since one of the primary concerns with the mixer unit is eliminating sources of noise, this decision seemed fairly obvious.

Ideally, all components in the analog sections of the mixer would be capable of working with a single pair of power supplies, of an appreciable voltage (to maximize dynamic range and headroom). Unfortunately, the ICs used in the audio mixer modules can not be run outside of the ± 7VDC region. This wouldn't be a problem if it was not desirable for the mixer to be capable of handling large audio signals (extending up to the +24dBu range). Thus, a split supply scheme is used, as described in earlier sections. Input and output buffer op-amps run from ± 14VDC supplies, while the mixer chips (and some op-amps) run from ± 7VDC supplies.

In addition to the analog circuitry present in the mixer, several digital and hybrid ICs are used. Many of these ICs require a +5VDC supply, thus a separate supply for this is clearly necessary. In addition to simply requiring a different operating voltage, it makes sense from a noise standpoint to move the digital ICs on to their own supply. This way, any transients incurred

[1] found on `http://www.pre.com/News/airtip15.html`

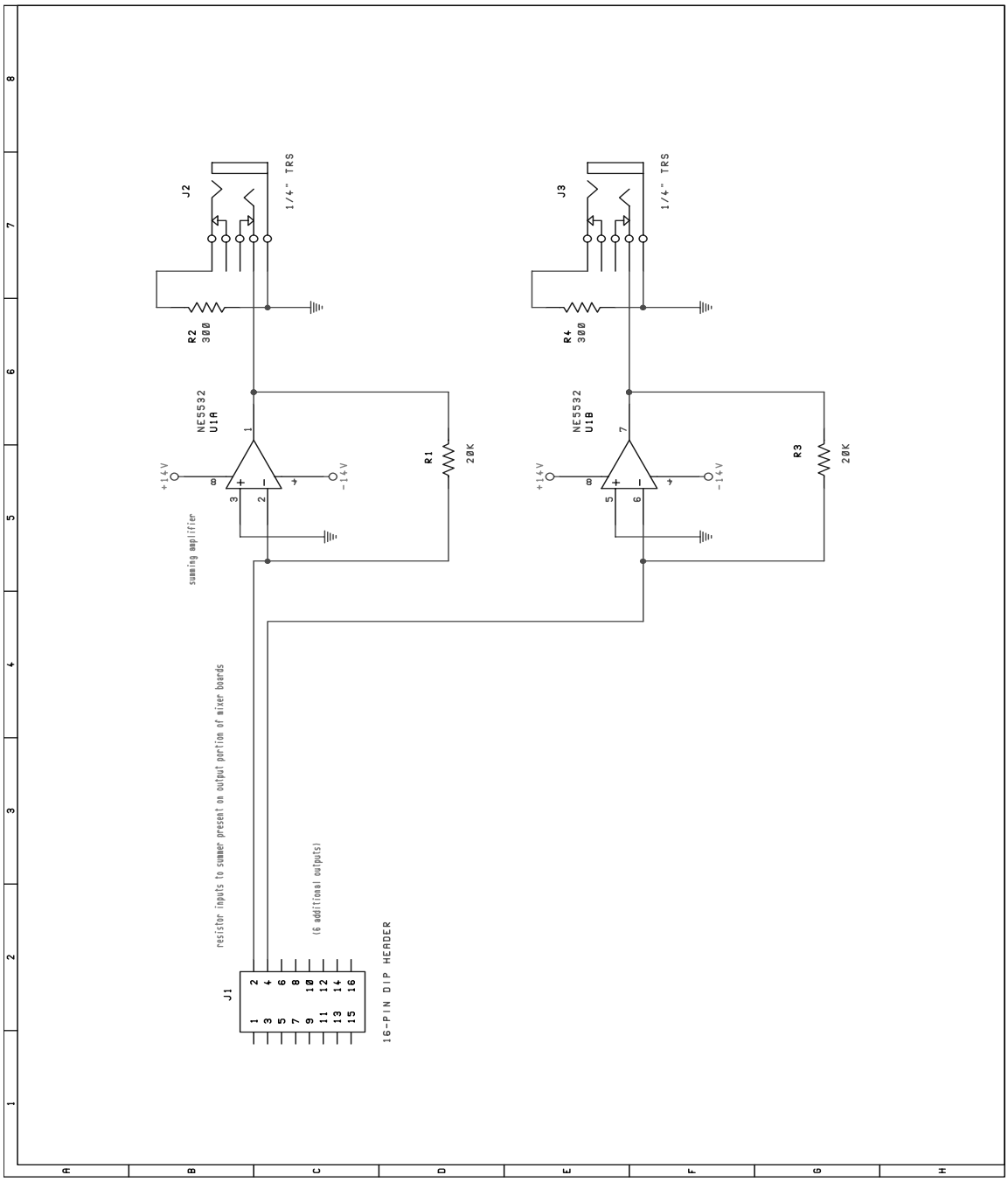


Figure 67: Audio output module, unbalanced driver schematic.

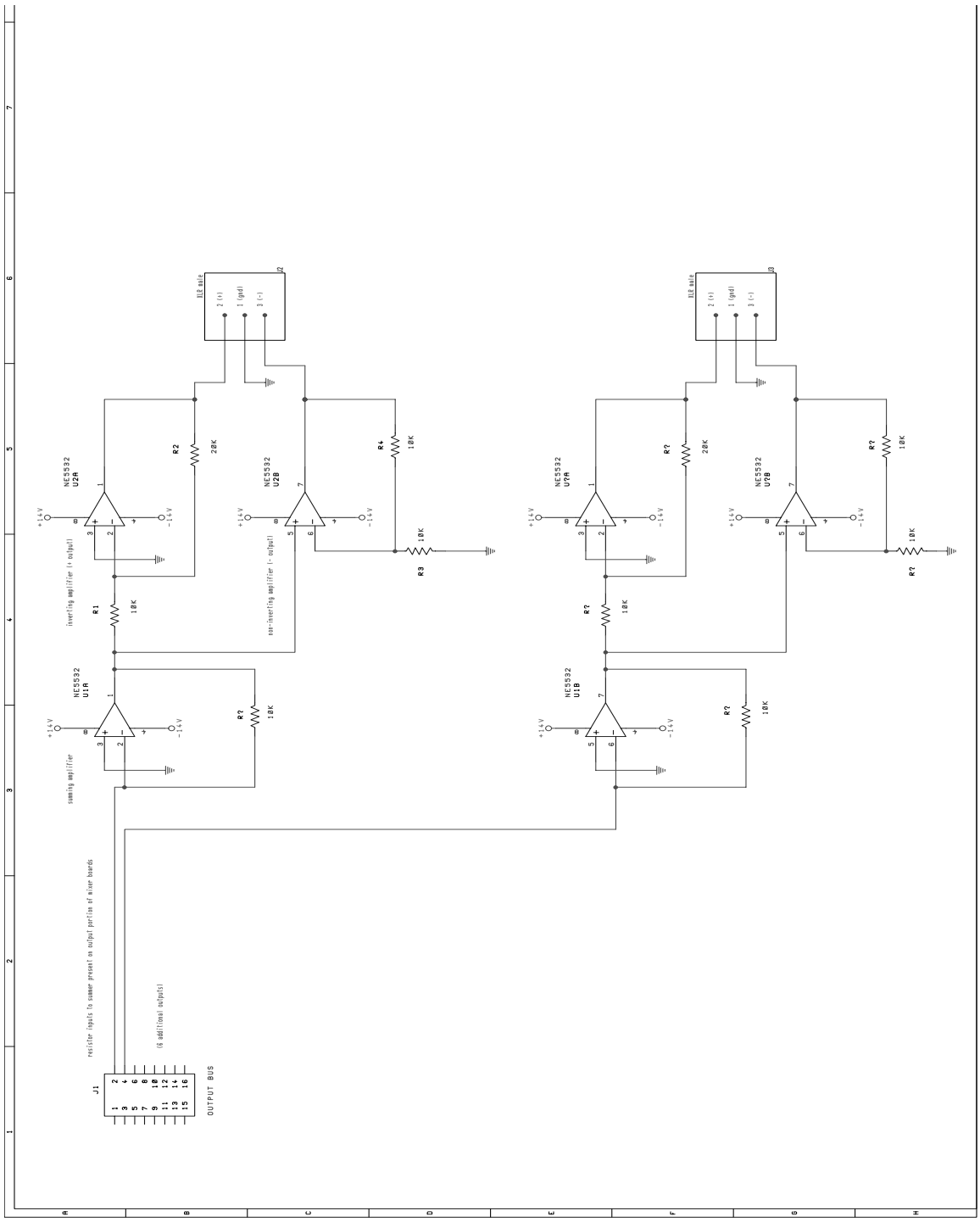


Figure 68: Audio output module, balanced driver schematic.

Voltage	Maximum Calculated Current (approx)
+14V	500mA†
-14V	500mA†
+7V	500mA‡
-7V	500mA‡

Table 1: Maximum calculated power supply currents. † 13mA maximum supply current for NE5532 dual op-amp. Assumes full system with all balanced input and output boards. ‡15mA maximum supply current for SSM2163 mixer IC. Assumes full system with sixteen audio mixer modules.

on the supply lines (reduced with copious amounts of bypass capacitance) do not make their way into any audio path.

The bus combiner module presents yet another problem. The relays chosen for the module have a very low coil resistance, on the order of 36 ω. With several of these running at once, a large amount of current will be drawn from the +5VDC supply. Thus, either the primary +5VDC supply must be capable of supplying approximately 3 amps of current, or a secondary +5VDC supply must be employed. At the time of this writing, a secondary high-current +5VDC supply is being planned.

The design uses standard LM317/LM337 adjustable regulators for the audio circuitry supplies, and the venerable LM7805 for the digital circuitry. The LM317 is a positive voltage three-terminal "adjustable" regulator. The LM337 is its sister part, and is a three-terminal negative voltage regulator. These components work by keeping a constant 1.25 volt difference between their output and adjust terminals. Equation 1 is used for determining component values for the output voltage.

$$V_{out} = 1.25((1 + (R2/R1)) \tag{1}$$

R1 and R2 refer to the resistors shown in the schematic in figure 69. Note that in the schematic, R2 is a trimmer potentiometer. The surround capacitors, C5 and C6 in the diagram, provide additional ripple rejection. The data sheet claims up to 80dB of ripple rejection using this scheme. Diode D2 acts to protect the regulator IC from capacitor discharges.

The remaining regulator circuits (utilizing U2, U3 and U4) are all similar. The circuits utilizing U3 and U4 are negative supplies.

Each LM317/337 is capable of supplying a regulated voltage at 1.5 amps. This is more than adequate for each supply voltage. Table 1 shows maximum calculated currents for each supply voltage. These figures were obtained from maximum supply current specifications from data sheets. Maximum supply current figures were then multiplied by the number of ICs of that type used on each power supply.

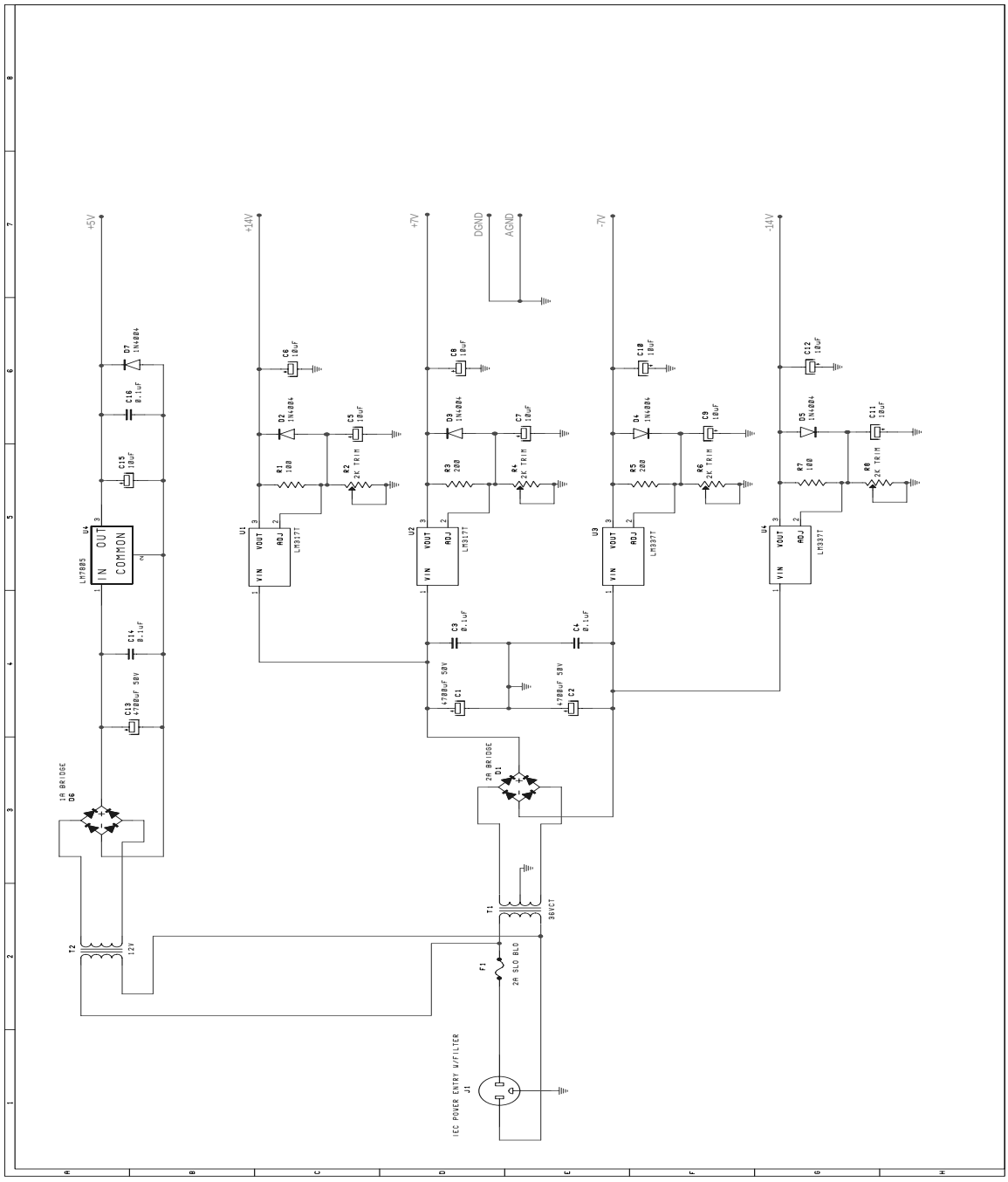


Figure 69: Audio mixer unit, power supplies for analog and digital circuitry.

7 Hardware Verification

7.1 Control Board

The control board is a relatively simple device, not involving any complex state machines or the like. Very little simulation was carried out, or, in the end, needed. The produced hardware functioned as designed.

For the VHDL designs present in the control board (and in the mixer unit), the Cypress Warp VHDL simulation tool was used. This simple tool allowed logical checks of the VHDL to be made before the GALs were burned. Figure 70 shows a screen shot of one such test.

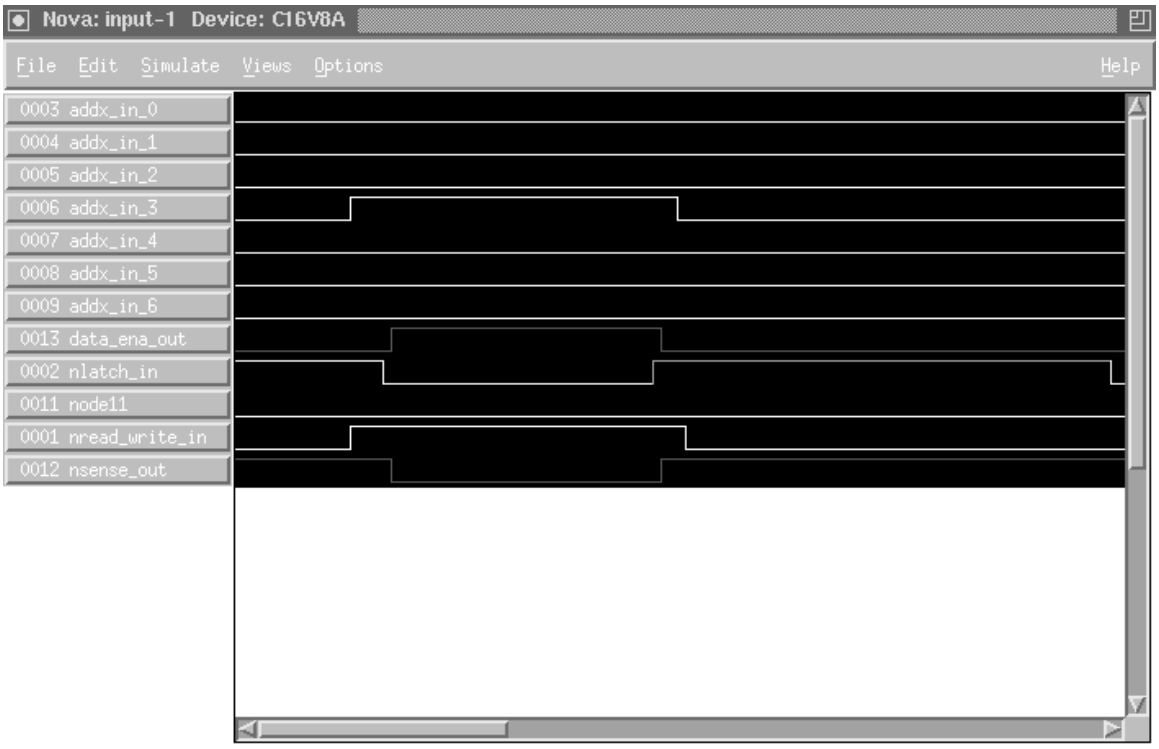


Figure 70: Screen shot from the Cypress Warp VHDL functional simulator.

7.2 Mixer Unit

(Note: the postscript output from the Spice simulation package was, for some reason, not compatible with LaTeX . No amount of reprocessing was able to make the files directly importable, thus, printouts from Spice are included at the end of the report and appropriately labeled.)

7.2.1 Balanced Audio Input Module

Transient, frequency, and noise analyses were performed on a PSpice model of the balanced audio input circuit.

The first simulation run was a transient response simulation. Plotting the pair of input voltages (sine waves, inverses of each other) and the output voltage shows that the circuit does, indeed, properly receive a balanced audio input signal. Also shown is the gain feature of the trim stage of the circuit, here set to a gain of 4. Refer to the attached PSpice plot entitled *balanced audio input - gain of 4.*

Since one stage of the input board is capacatively coupled to the next, to remove DC bias voltages, frequency response simulation was carried out, to ensure that the high-pass filter was designed properly. The specification for the mixer unit dictates that the unit be no more than 3dB down at 20Hz. The simulation shows that the low-end 3dB point of the input circuit is at around 17Hz. Refer to the attached plot, entitled *balanced audio input frequency response.*

Lastly, an input noise analysis was run. This was done to gain data for later noise analysis. Unfortunately, at present, an NE5532 model for PSpice is nowhere to be found. This noise data is for the 741 op-amp. Still, it provides at least some indication as to the noise present. Further analysis will be done when the proper Spice model has been obtained. Refer to the attached plot, entitled *balanced audio input, noise analysis.*

7.2.2 Unbalanced Audio Input

While there is no unbalanced audio input module in the system, per se, an unbalanced signal may be connected to the system. A transient analysis was done of this situation, to show that it does, indeed, work properly. The attached plot, entitled *balanced audio input board, w/unbalanced input - transient* shows the response.

7.2.3 Balanced Audio Output Module

(Refer to the attached PSpice schematic, entitled *balanced audio output schematic*)

The first simulation run was a transient response test. Plotting the input voltage (a sine wave in this case) against the pair of output voltages (the differential pair) shows that the circuit does, indeed, function as designed. Refer to the attached plot, entitled *balanced audio output - transient.*

A frequency sweep from 10Hz to 30KHz was performed, to observe the frequency response of the circuit. Predictions said that it should be relatively flat in this range, and indeed this was shown. While some rolloff starting around 8-10KHz is shown, the rolloff is in *thousandths* of volts. PSpice's automatic zoom feature makes this rolloff look a lot worse than it actually is. The attached plot, entitled *balanced output - freq response*, shows the response.

Lastly, a noise analysis (again, with the improper PSpice op-amp model) was run, to obtain data for later noise analysis.

7.2.4 Overall Noise Analysis

Overall system noise was "budgeted." It was understood that each component in the audio path contributed to end-to-end system noise. Care was taken to choose components with low noise specs, and use those components such that the noise specifications were met. Where possible, PSpice simulations were carried out to get a feel for the noise behavior of small subsystems.

There are two known noise problems with the prototype. There is a -50dB noise floor inherent in the system. Due to time constraints, no real investigation of the source of this noise was possible, though some theories have been proposed. In addition to this noise floor, digital "hash" from the internal microcontroller is being picked up by the system. This will likely be remedied with some shielding and more supply bypassing.

7.2.5 Total Harmonic Distortion Analysis

Rather than spending inordinate amounts of time performing THD analysis, as much care was taken along the design process to choose components with respectable THD specifications. To make sure these specifications were met, as much care as possible was taken to ensure that the parts were operating within specifications. Due to the time constraints on the project, it is felt that this was a reasonable method.

8 Hardware PC Board Designs

The PC boards were layed out directly from the schematics developed in previous sections. No auto-routing was used during the layout process, mostly because it was not entirely trusted. This is especially true in cases where audio signals and noisy digital signals share board space. Another point in favor of manual layout was that many of the parts had to be in specific positions on the PC board because of panel-mounting, etc. With many specific parts positions, some auto-routers become confused and produce routing of poor quality.

Many parts were chosen that were not in the default PowerPCB library. This required creating a custom library of parts. Once a stock of parts was on hand, a caliper and fine-scale ruler were used to take measurements of pins and case sizes. These measurements were used to construct a component layout for the custom parts library. To be sure the appropriate modeling was done, a real-size version of the part was printed and fitted with the actual part. Any variations or discrepancies were then fixed.

PADS Software's PowerPCB software was used for all board layouts. Once boards had been designed, proofed, and test-fitted, Gerber output was generated. Separate Gerber files are produced for top and bottom routing, solder masking, drilling, and silkscreening. All Gerber output was checked with GCPREVUE, a freeware Gerber viewer. Once convinced of the quality of the output, the Gerber files were sent to the PCB house via the Internet.

EP Circuits, a fabrication house located in Canada, offered the best price/performance ratio for board prototyping. It was decided that all boards in the mixer unit would be treated with solder mask, mainly because of the use of surface-mount components that were to be hand-soldered. All boards produced for the control board did not have solder masking applied. This was done in an attempt to save precious development funds.

8.1 Control Board

Three PC board designs were created and manufactured for the control board. The remaining boards, such as the Pbus controller, were hand-wired to save money.

8.1.1 Fader Module

Two fader modules are present in the prototype control board. They are identical boards with different GAL programming and jumper settings.

Figure 71 shows the silkscreen/assembly layer for the fader board. Table 2 shows the bill of materials for a single fader board. Figures 72 and 73 show the top and bottom routing layers, respectively.

8.1.2 Output Assign Module

A single output assignment module is present in the prototype control board. Figure 74 shows the silkscreen/assembly layer for the output assignment PC board. Table 3 lists the bill of materials for this board. Figures 75 and 76 show the top and bottom routing layers, respectively.

8.1.3 Transport Control Module

A single transport control module is present in the prototype control board. Figure 77 shows the silkscreen/assembly layer for the transport control PCB. Table 4 lists the bill of materials for this board. Figures 78 and 79 show the top and bottom routing layers, respectively.

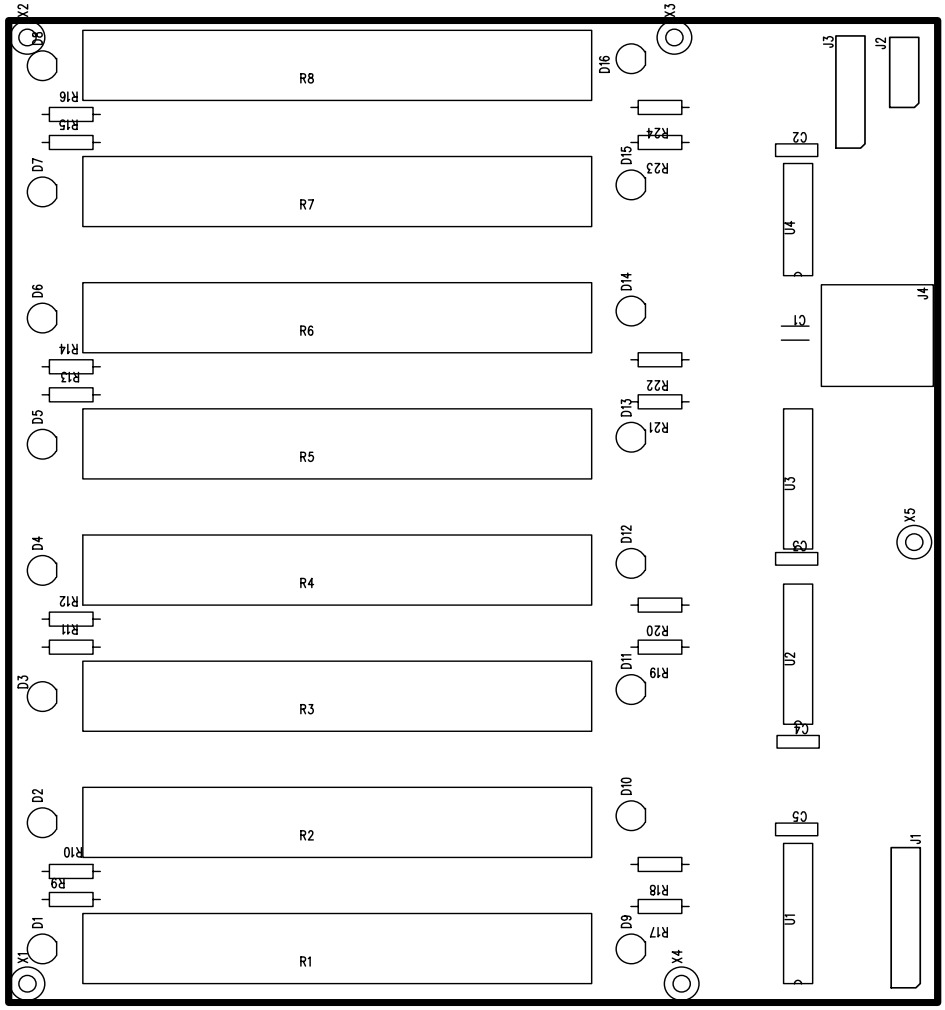


Figure 71: Fader module PCB, silkscreen/assembly drawing.

faders.pcb - Sat Mar 01 21:05:30 1997

REFDES	QTY	DESCRIPTION
R1 - R8	8	ALPHA 500K slide potentiometer
R9 - R24	16	300 ohm 1/8W
C1	1	10uF electrolytic
C2 - C5	4	0.1uF monolithic
D1 - D8	8	Green T1-3/4 LED
D9 - D16	8	Red T1-3/4 LED
U1	1	GAL16V8
U2, U3	2	74LS373
U4	1	CD4051
J1	1	20-pin DIP header
J2	1	10-pin DIP header
J3	1	16-pin DIP header or wire jumper
J4	1	4-pin 0.156in. power connector

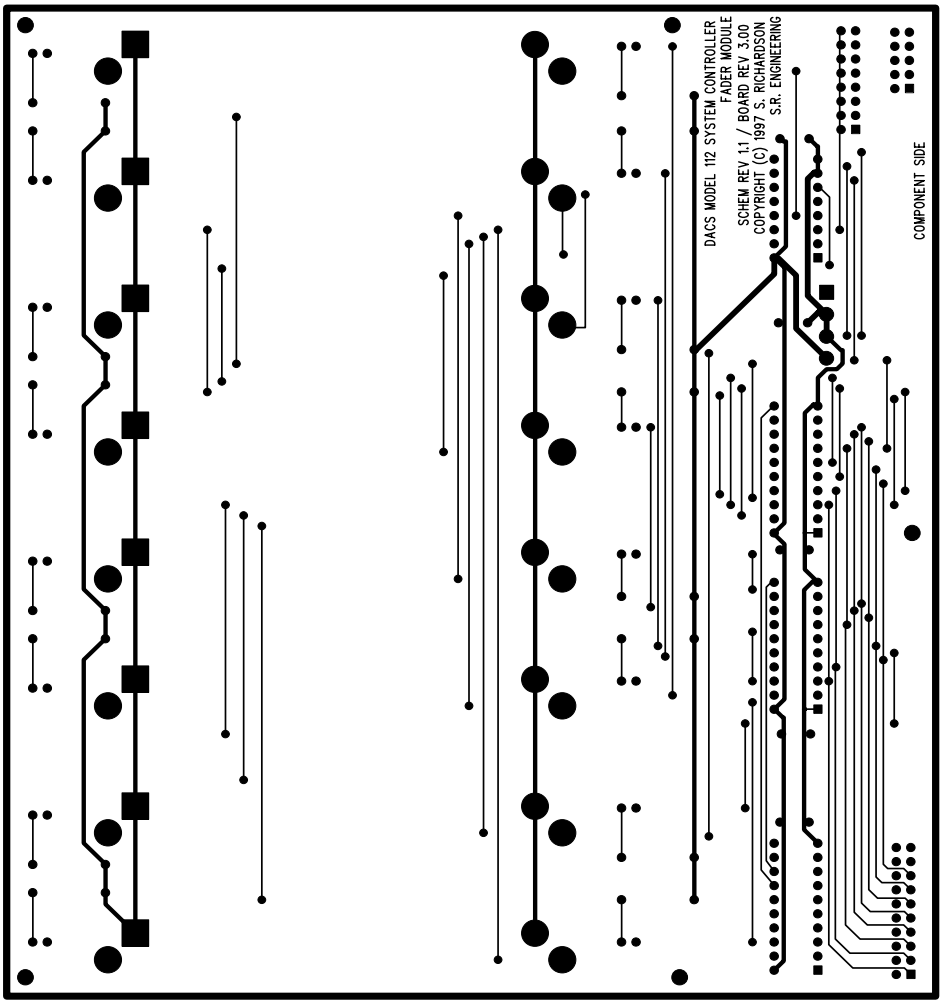
Table 2: Fader module, bill of materials.

REFDES	QTY	DESCRIPTION
S1 - S18	18	E-Switch momentary PCB-mount pushbutton
R1 - R16	16	300 ohm 1/8W
R17 - R34	18	2.2K ohm 1/8W
C1 - C9	9	0.1uF monolithic
C10	1	10uF electrolytic
D1 - D16	16	T1-3/4 green LED
D17	1	dual 7-segment green LED display
U1 - U6	6	74LS373
U7, U8	2	74LS247
U9	1	GAL16V8
J1	1	20-pin DIP header
J2	1	4-pin 0.156in. power connector
J3	1	10-pin DIP header

Table 3: Output assign module, bill of materials.

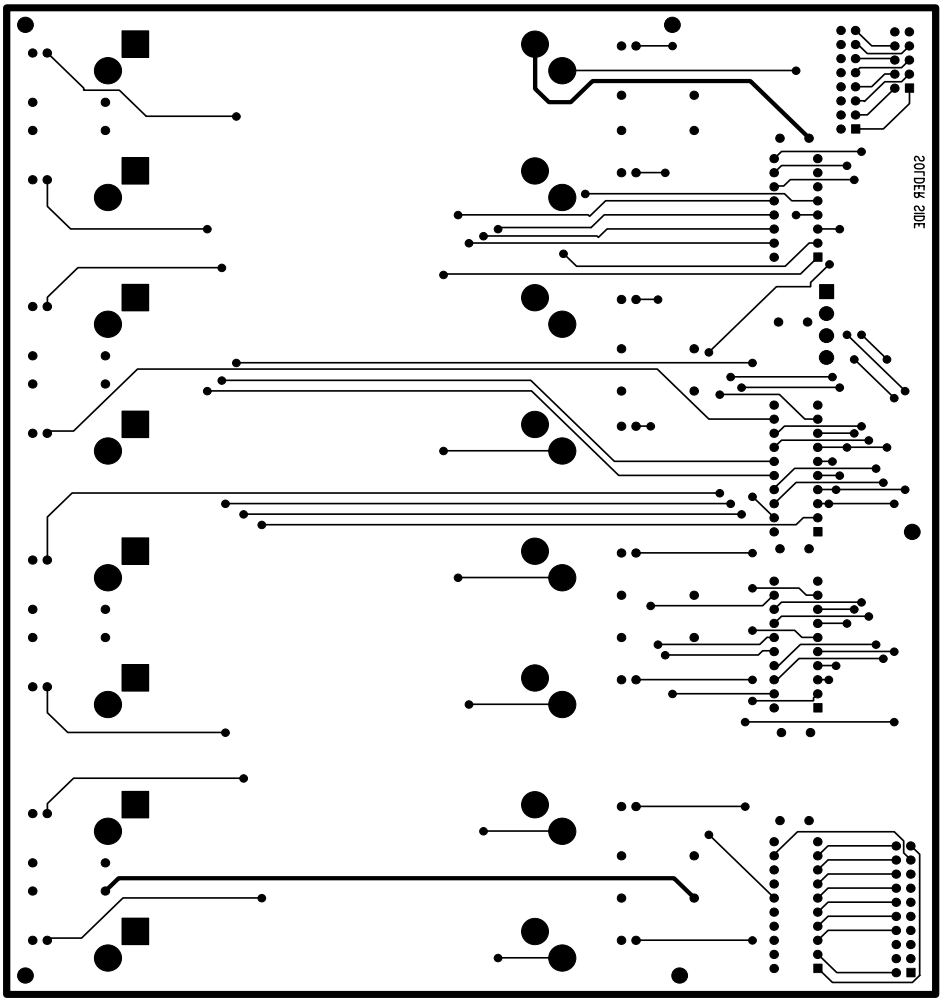
REFDES	QTY	DESCRIPTION
S1 - S7	7	E-Switch momentary PCB-mount pushbutton
R1 - R3, R11	3	300 ohm 1/8W
R4 - R10	7	2.2K ohm 1/8W
R1 - R13	13	0.1uF monolithic
D1 - D3	3	dual 7-segment green LED display
D4 - D6	3	T1-3/4 green LED
U1	1	GAL16V8
U2 - U7	6	74LS373
U8 - U13	6	74LS247

Table 4: Transport control module, bill of materials.



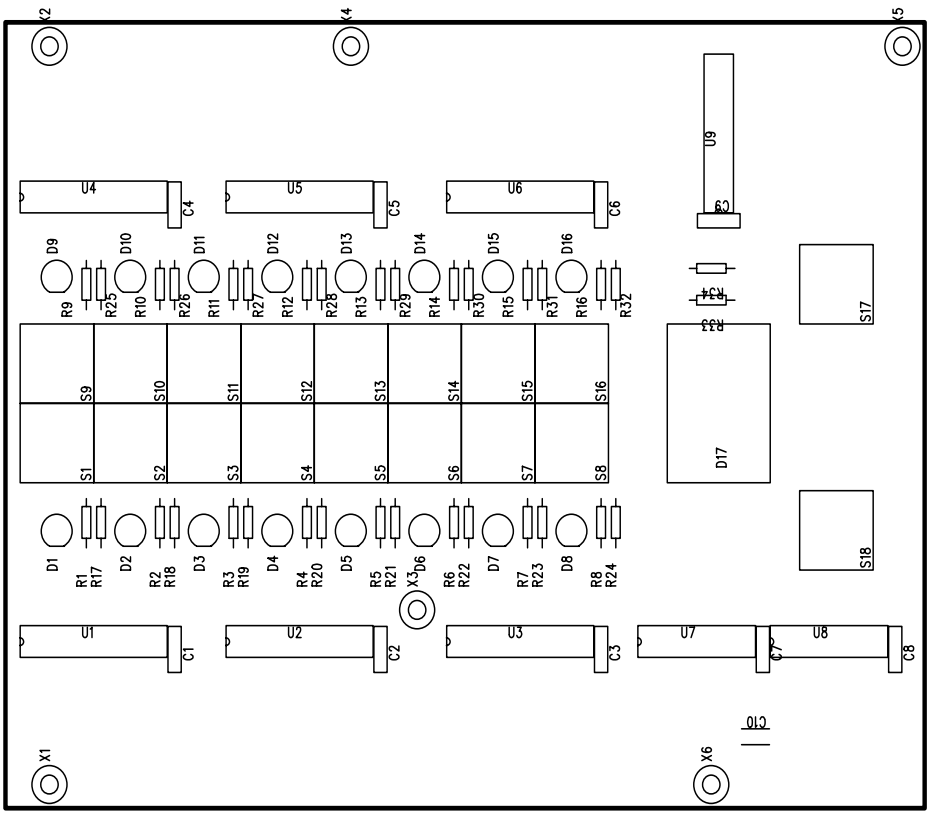
faders.pcb - Sat Mar 01 21:02:23 1997

Figure 72: Fader module PCB, component-side routing.



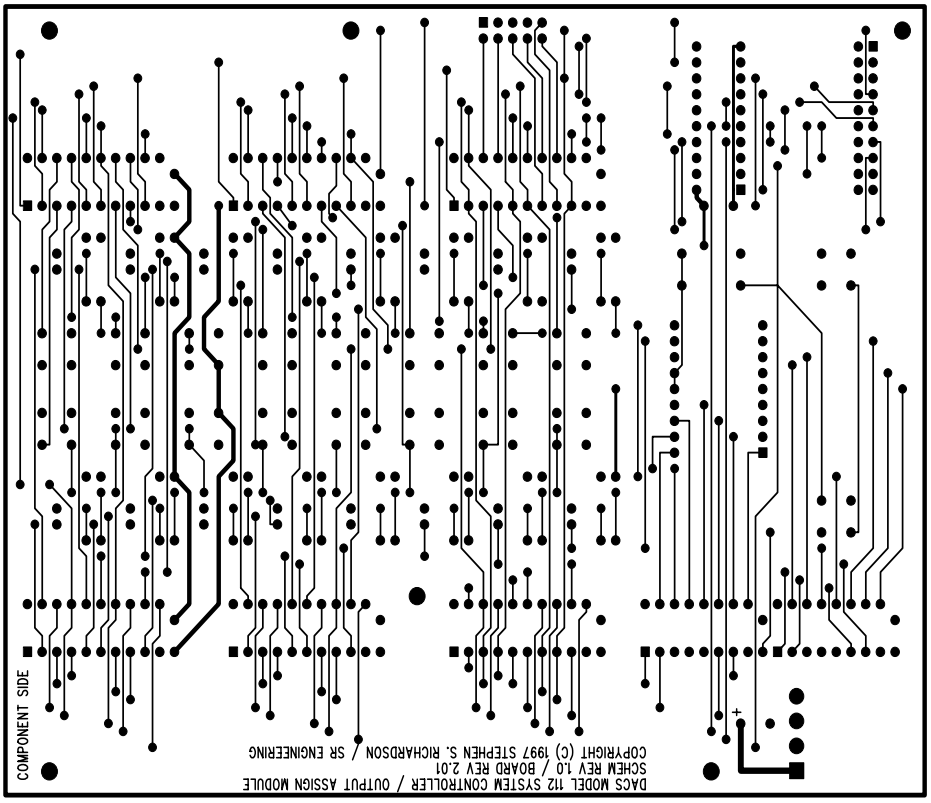
faders.pcb - Sat Mar 01 21:03:59 1997

Figure 73: Fader module PCB, solder-side routing.



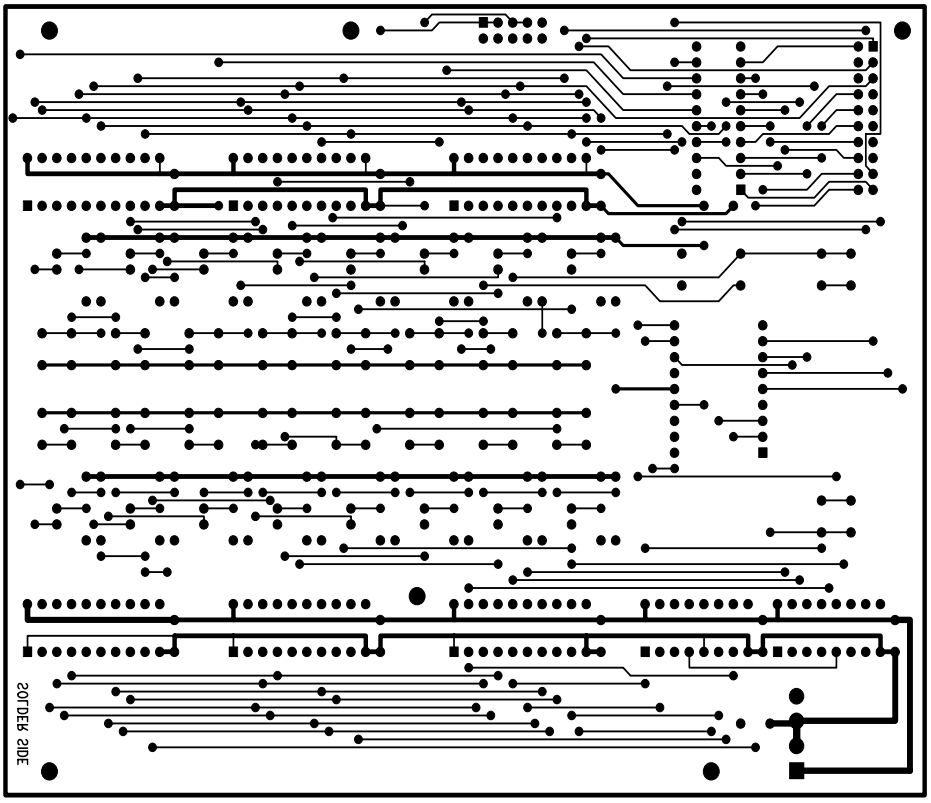
outassn.pcb - Sat Mar 01 20:47:43 1997

Figure 74: Output assign module PCB, silkscreen/assembly drawing.



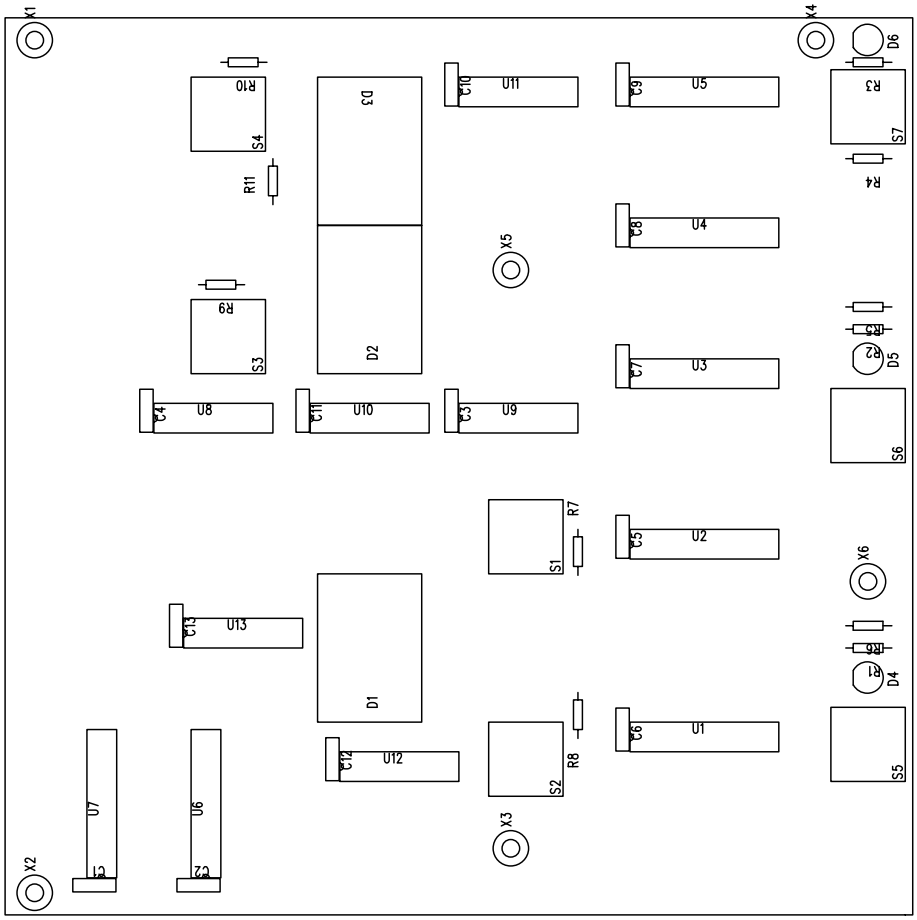
outassn.pcb - Sat Mar 01 20:44:43 1997

Figure 75: Output assign module PCB, component-side routing.



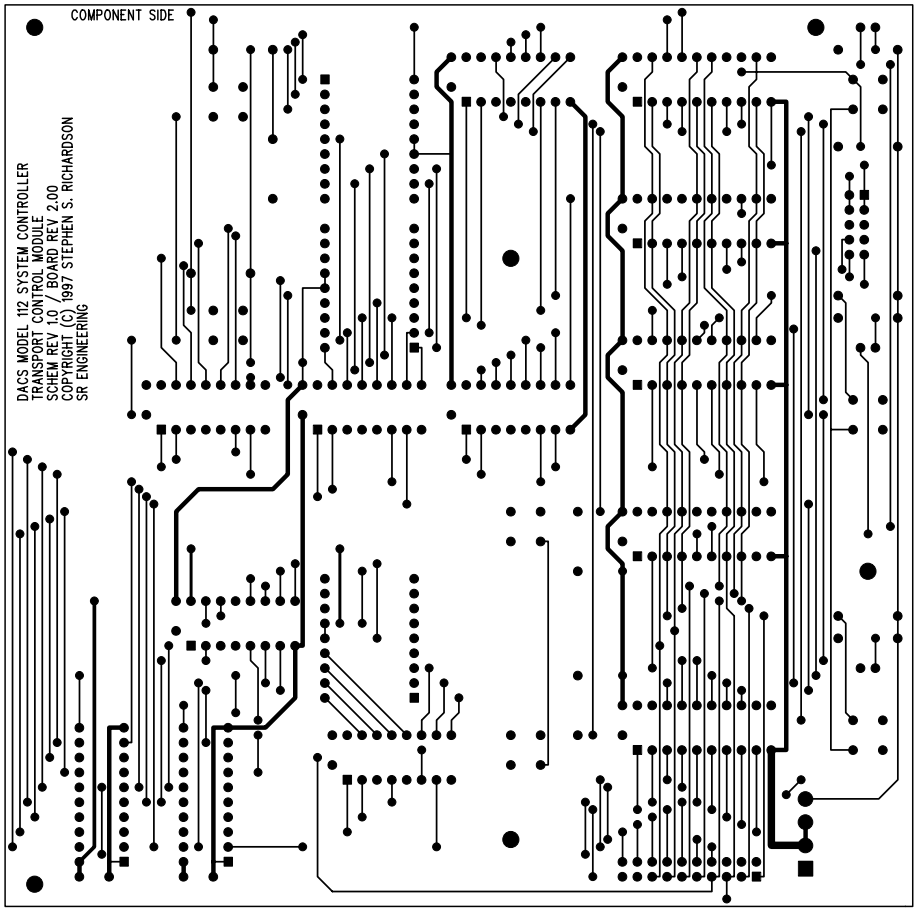
outassn.pcb - Sat Mar 01 20:46:05 1997

Figure 76: Output assign module PCB, solder-side routing.



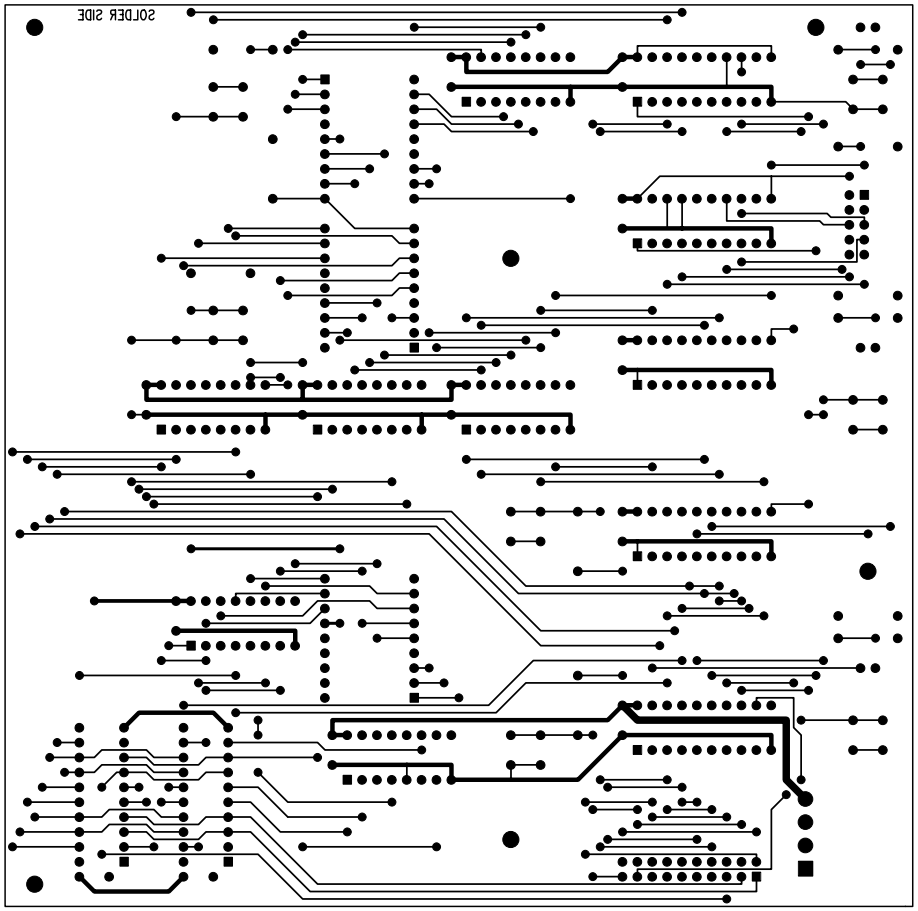
transport.pcb - Sat Mar 01 20:58:28 1997

Figure 77: Transport control module PCB, silkscreen/assembly drawing.



transport.pcb - Sat Mar 01 20:53:55 1997

Figure 78: Transport control module PCB, component-side routing.



transport.pcb - Sat Mar 01 20:55:17 1997

Figure 79: Transport control module PCB, solder-side routing.

REFDES	QTY	DESCRIPTION
R1 - R32,R34,R37,R40,R43, R46,R49,R52,R55	40	20K 1% metal film
R33,R35,R36,R38,R39,R41,R42, R44,R45,R57,R48,R50,R51,R53, R54,R56	16	10K 1% metal film
C1 - C8	8	0.47uF metallized polyester
C9 - C33	24	0.1uF monolithic
C34 - C38	5	10uF electrolytic
U1 - U12	12	NE5532
U13 - U16	1	DS1800 (not used, mis-design)
U17	1	GAL16V8
U18	1	74LS373
J1 - J8	8	Re'an 1/4 inch PCB mount switched jack
J9	1	16-pin DIP header
J10	1	Molex Mini-Fit Jr. 8-circuit right-angle
J11	1	20-pin DIP header

Table 5: Audio input module, bill of materials.

REFDES	QTY	DESCRIPTION
R1 - R8	8	10K 1% metal film
C1 - C10	10	0.1uF monolithic
C11,C12	2	10uF electrolytic
U1 - U4	4	SSM2163 (SOIC-28)
U5	1	GAL16V8
U6	1	74LS373
U7	1	74LS04
J1 - J4, J7 - J9	8	16-pin DIP header
J5	1	Molex Mini-Fit Jr. 8-circuit right angle
J6	1	20-pin DIP header

Table 6: Audio Mix Module, bill of materials.

8.2 Mixer Unit

Four PC board designs were created and manufactured for the mixer unit. The Pbus controller and power supply were hand-wired to save money.

8.2.1 Audio Input Module

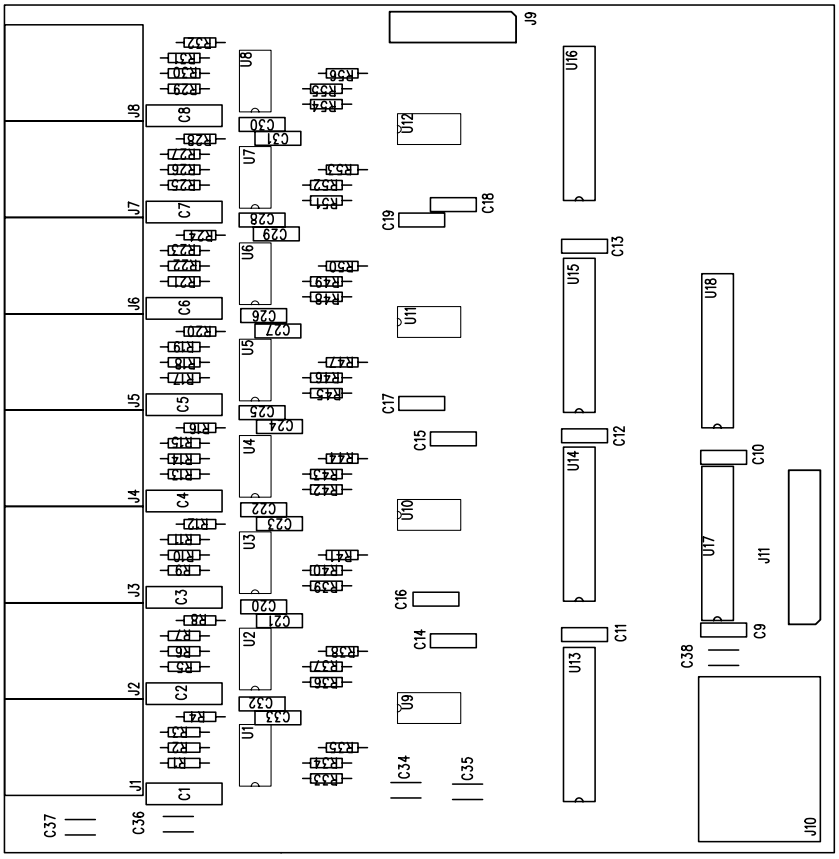
Four audio input modules are used in the prototype mixer unit. Figure 80 shows the silkscreen/assembly layer for the audio input PC board. Table 5 shows the bill of materials for this board. Figures 81 and 82 show the top and bottom routing layers, respectively.

It should be noted that, due to an oversight in design, the volume trim portion of this board is not functional. The DS1800 IC's are replaced with fixed resistor values in the prototype.

8.2.2 Audio Mix Module

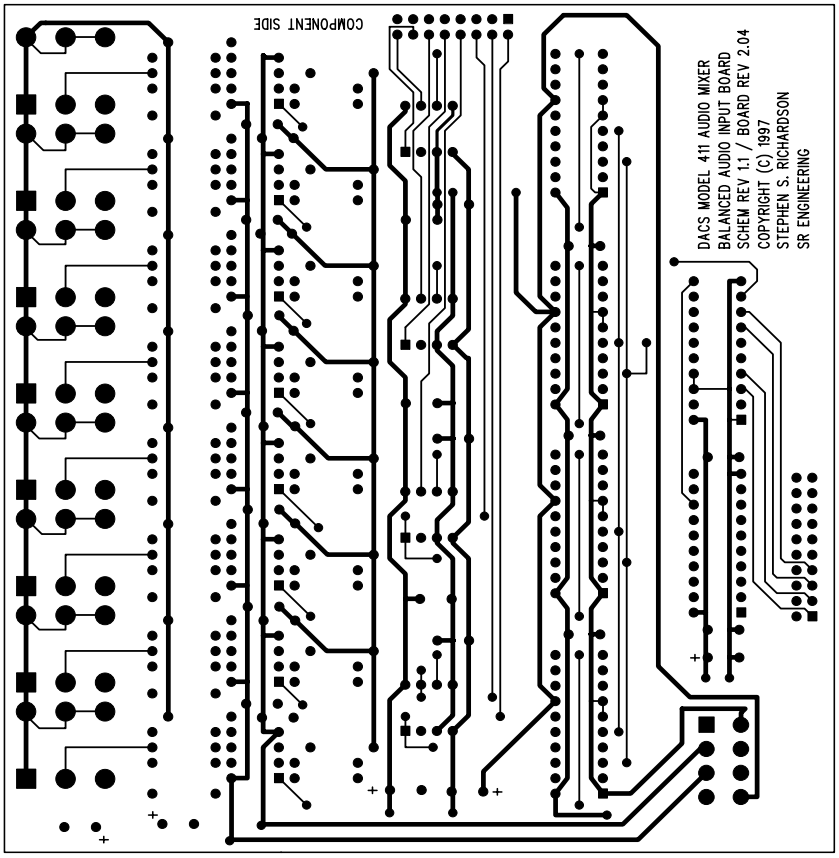
Four audio mix modules are used in the prototype mixer unit. Figure 83 shows the silkscreen/assembly layer for this PC board. Table 6 shows the bill of materials for the audio mix board. Figures 84 and 85 show the top and bottom routing layers for this board.

Note the presence of surface-mount components on the top routing layer.



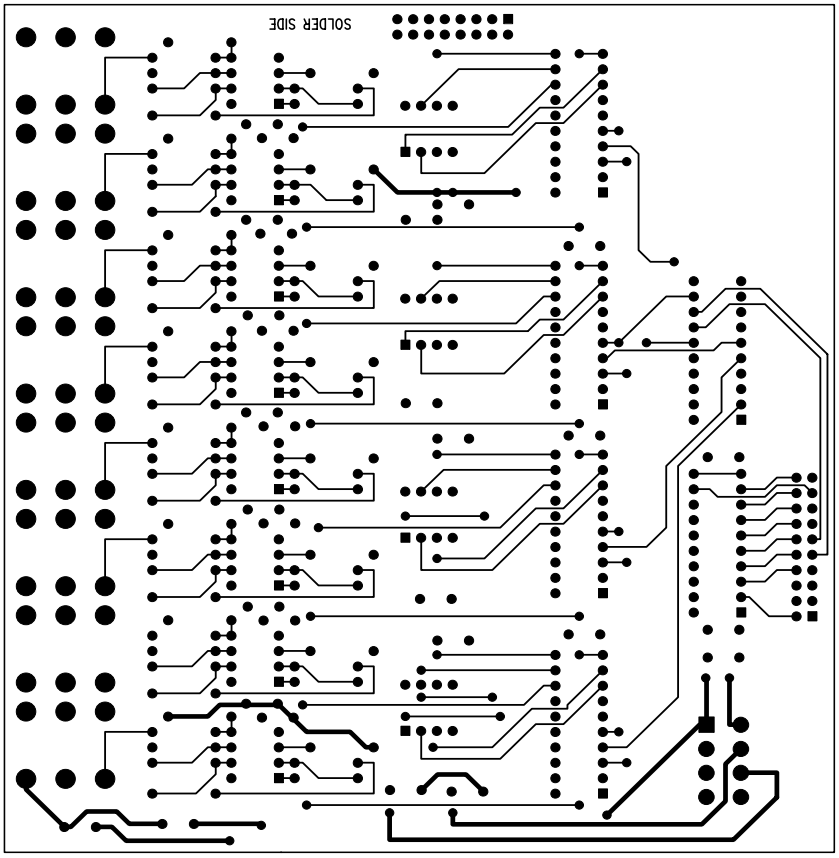
mix-bal-input.pcb - Sat Mar 01 20:29:47 1997

Figure 80: Audio input module PCB, silkscreen/assembly drawing.



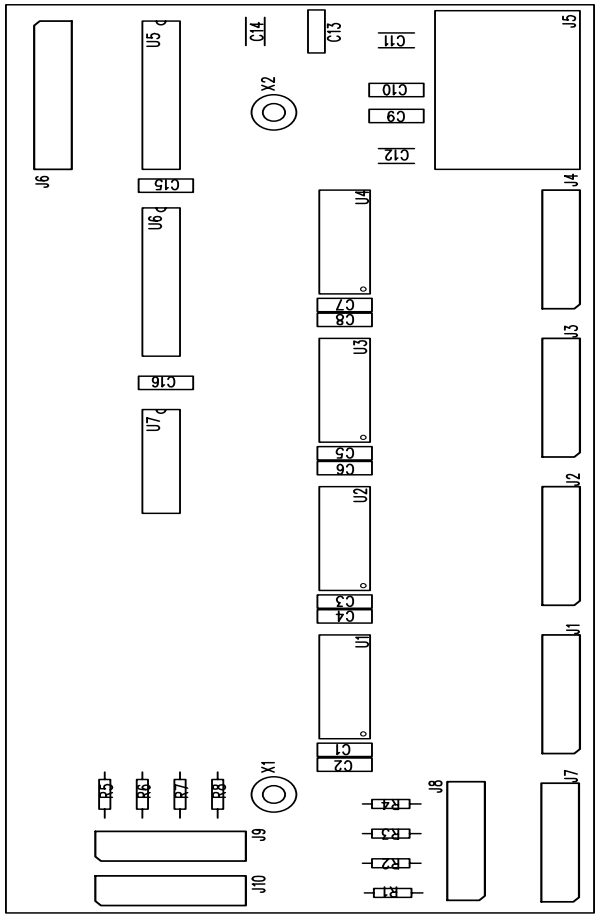
mix-bal-input.pcb - Sat Mar 01 20:26:46 1997

Figure 81: Audio input module PCB, component-side routing.



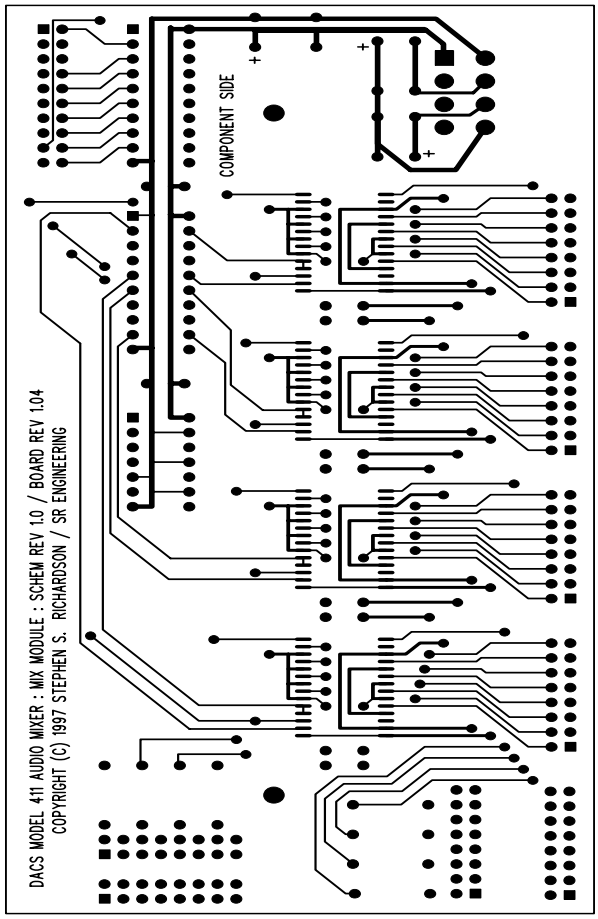
mix-bal-input.pcb - Sat Mar 01 20:28:17 1997

Figure 82: Audio input module PCB, solder-side routing.



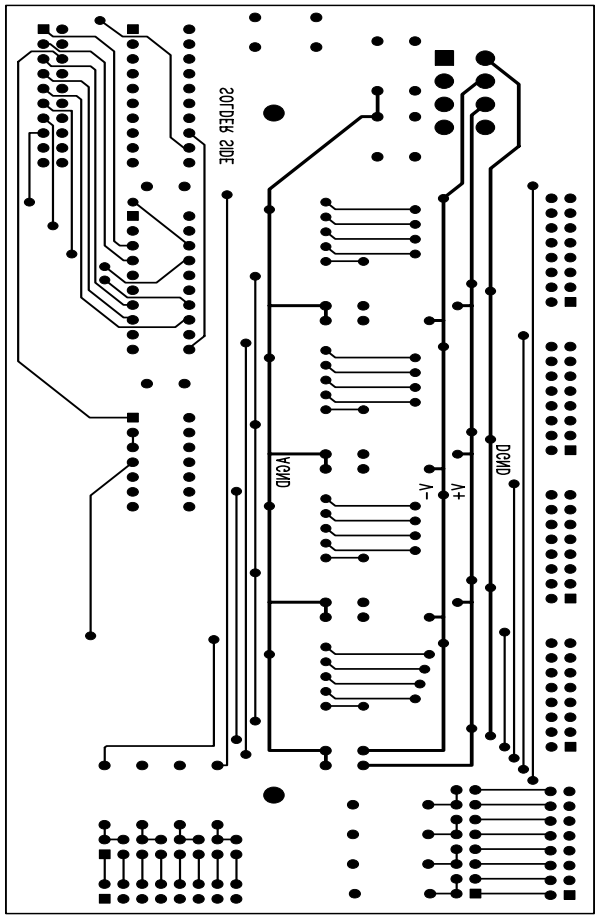
mixer-board.pcb - Sat Mar 01 20:34:42 1997

Figure 83: Audio mix module PCB, silkscreen/assembly drawing.



mixer-board.pcb - Sat Mar 01 20:31:42 1997

Figure 84: Audio mix module PCB, component-side routing.



mixer-board.pcb - Sat Mar 01 20:32:30 1997

Figure 85: Audio mix module PCB, solder-side routing.

REFDES	QTY	DESCRIPTION
R1 - R5	5	1K ohm 1/8W
C1	1	10uF electrolytic
C2,C3	2	0.1uF monolithic
K1 - K18	18	PED DPDT 5V coil relay
D1 - D5	5	1N4004
Q1 - Q5	5	TIP120
U1	1	GAL16V8
U2	1	74LS373
J1 - J9	9	16-pin DIP header
J10	1	Molex Mini-Fit Jr. 8-circuit right-angle
J11	1	20-pin DIP header

Table 7: Bus combiner/switcher, bill of materials.

REFDES	QTY	DESCRIPTION
R10,R12,R14,R16, R18,R20,R22,R24, R1 - R8,R33 - R40	24	10K ohm 1% metal film†
R9,R11,R13,R15, R17,R19,R21,R23	8	20K ohm 1% metal film†
C1,C2	2	10uF electrolytic
C3 - C22	20	0.1uF monolithic
U1 - U8	8	NE5532†
U9 - U12	4	NE5532
J1 - J8	8	Re'an right-angle PCB mount XLR †
J9 - J16	8	Re'an PCB mount 1/4 inch switched ‡
J17	1	16-pin DIP header
J18	1	Molex Mini-Fit Jr. 8-circuit right-angle

Table 8: Audio output module, bill of materials. † For balanced configuration only. ‡For unbalanced configuration only.

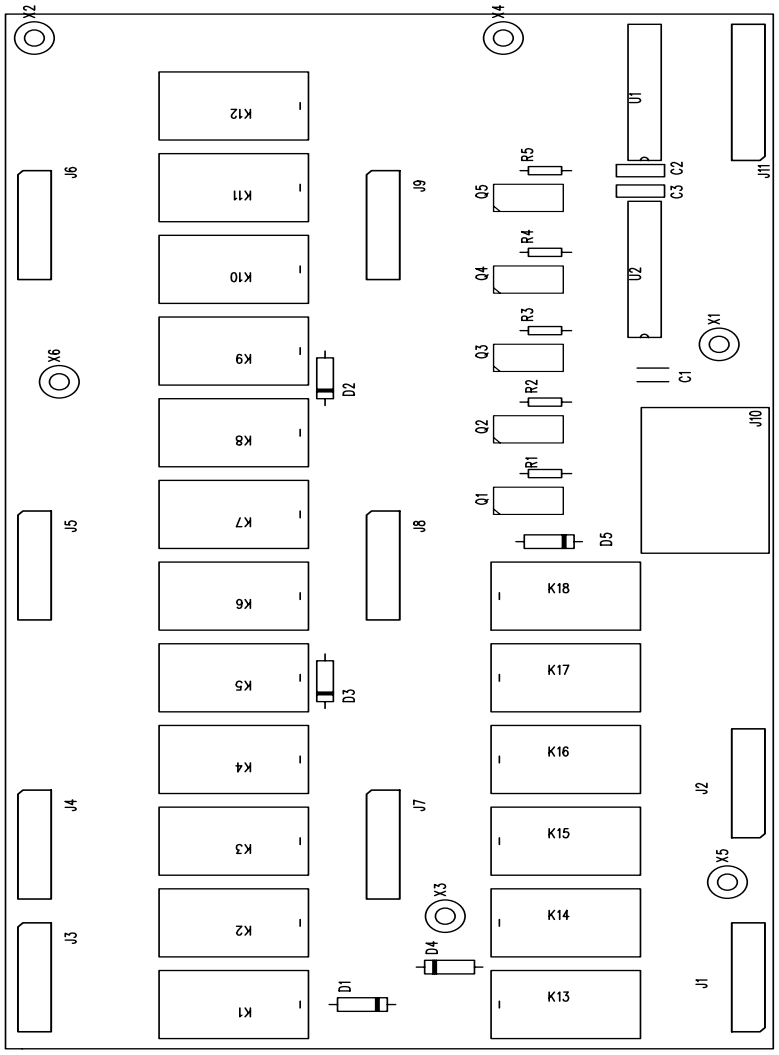
8.2.3 Bus Combiner and Switcher Module

There is a single bus combiner board present in the prototype mixer unit. Figure 86 shows the silkscreen/assembly layer for this PC board. Table 7 shows the bill of materials for the bus combiner board. Figures 87 and 88 show the top and bottom routing layers.

8.2.4 Audio Output Module

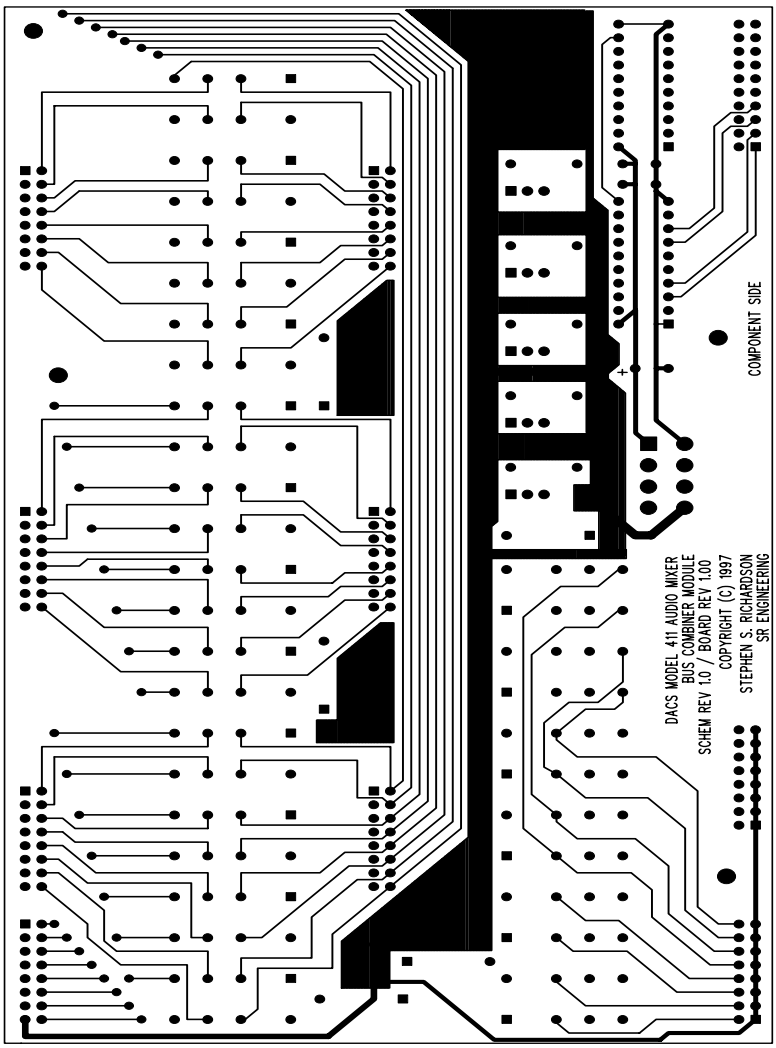
There are two audio output modules present in the prototype mixer unit. In the prototype, one board is configured for balanced audio while the other is configured for unbalanced audio. One single PC board design is used for both of these. Pads are available for 1/4 inch jacks and XLR jacks. Unbalanced boards require less components. This design was the most efficient possible for this situation.

Figure 89 shows the silkscreen/assembly layer for this PC board. Table 8 shows the bill of materials for the audio output board. Figures 90 and 91 show the top and bottom routing for this board, respectively.



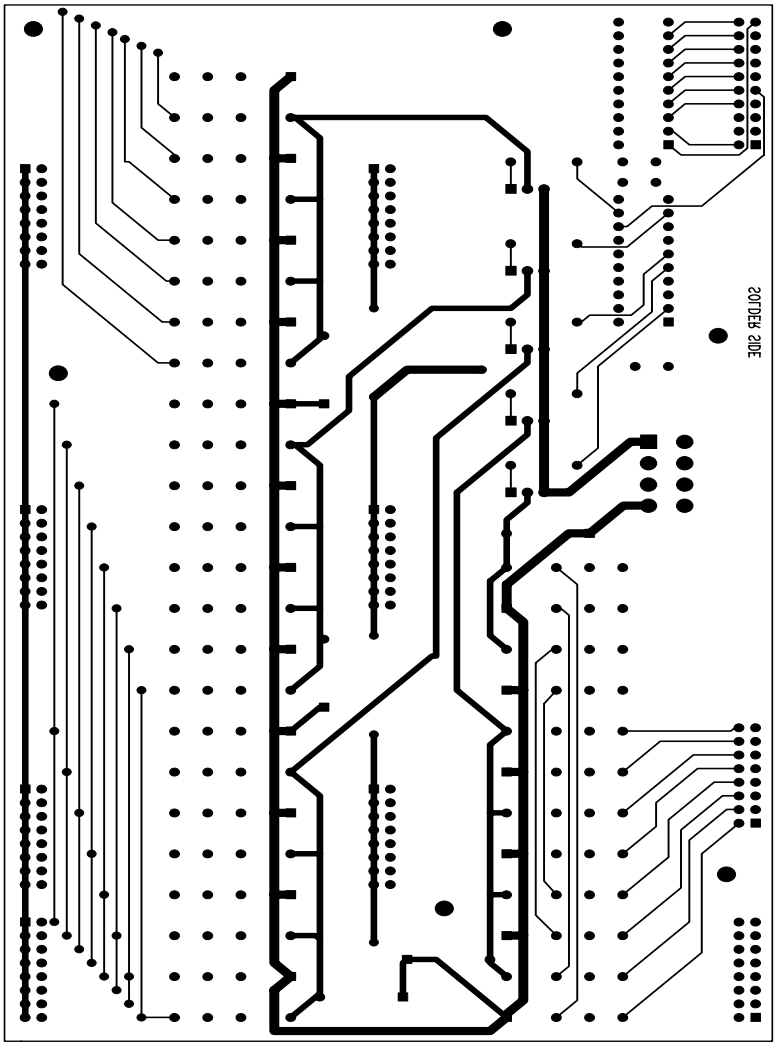
bus-comb.pcb - Sat Mar 01 20:22:05 1997

Figure 86: Audio bus switcher/combiner PCB, silkscreen/assembly drawing.



bus-comb.pcb - Sat Mar 01 20:19:44 1997

Figure 87: Audio bus switcher/combiner PCB, component-side routing.



bus-comb.pcb - Sat Mar 01 20:20:35 1997

Figure 88: Audio bus switcher/combiner PCB, solder-side routing.

output-balunbal.pcb - Sat Mar 01 20:15:16 1997

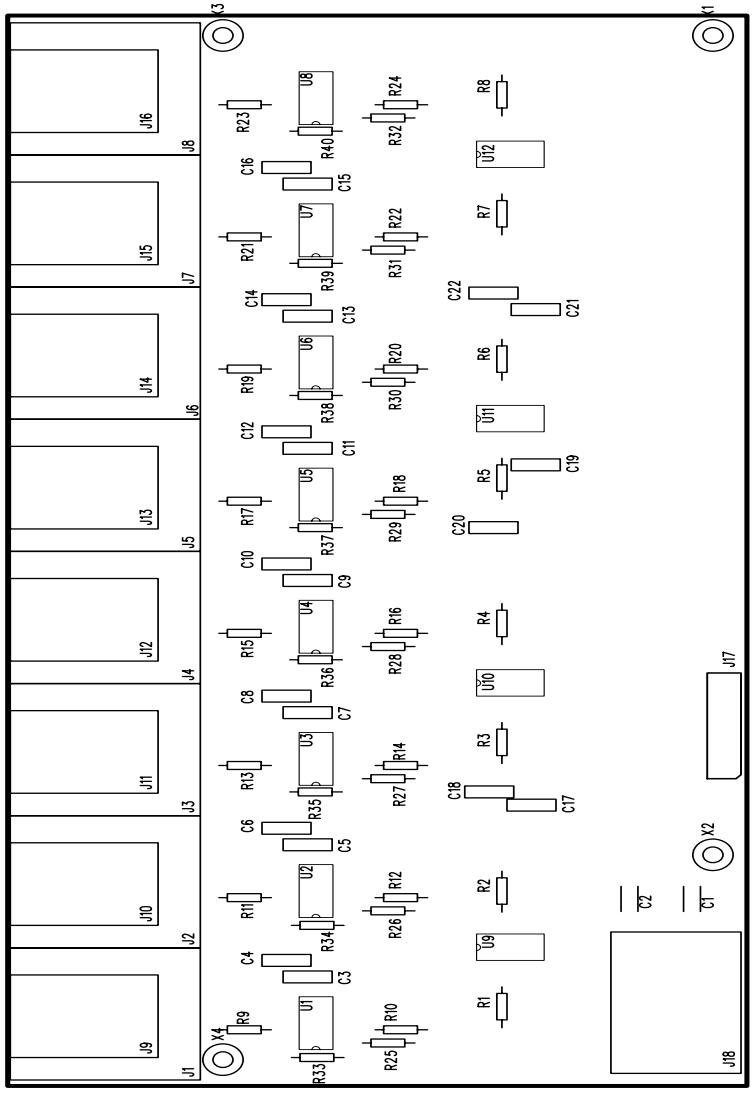


Figure 89: Audio output module PCB, silkscreen/assembly drawing.

output-balunbal.pcb - Sat Mar 01 19:58:13 1997

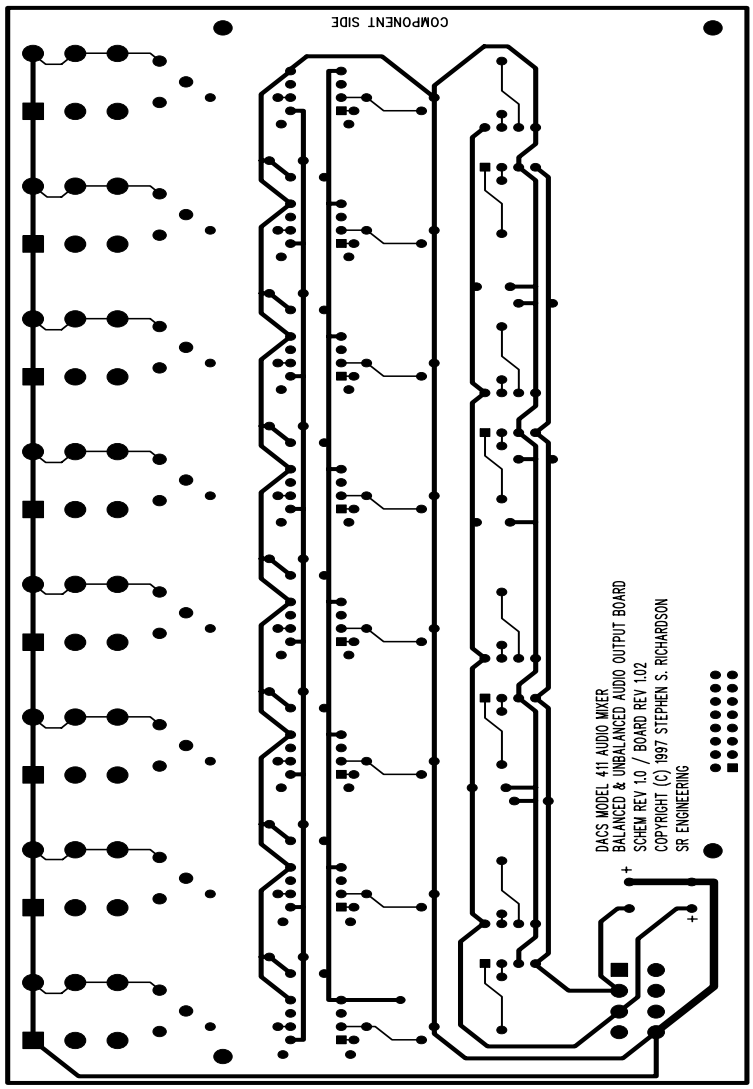
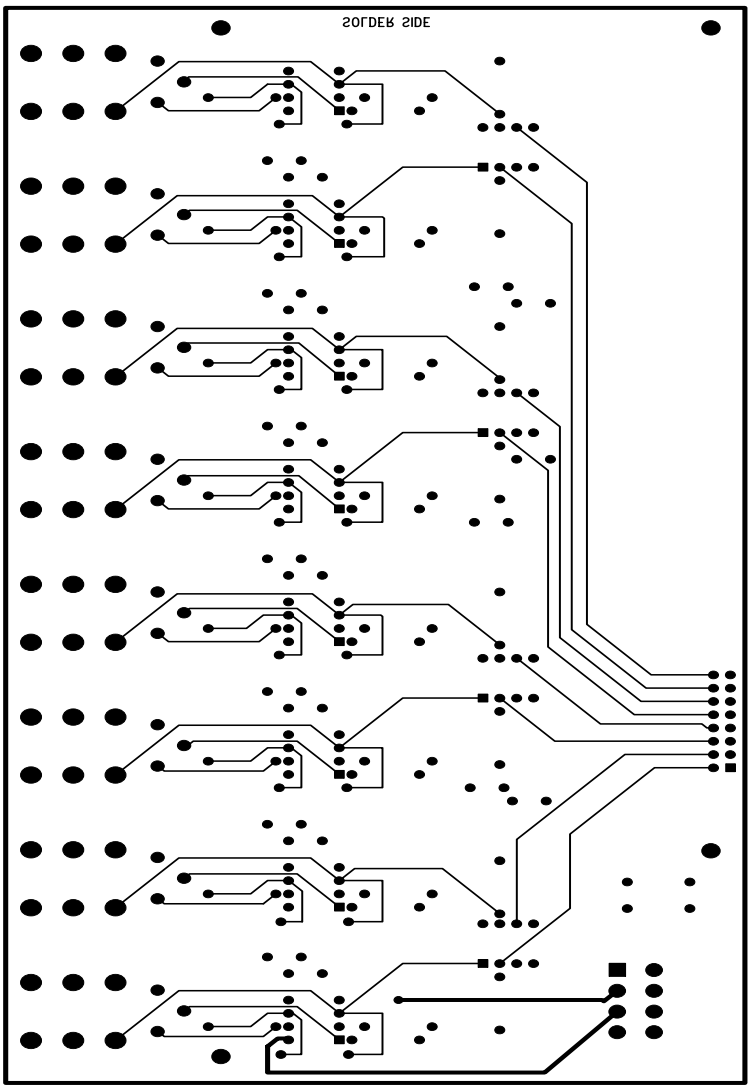


Figure 90: Audio output module PCB, component-side routing.



output-balunbal.pcb - Sat Mar 01 20:00:31 1997

Figure 91: Audio output module PCB, solder-side routing.

9 Hardware Chassis

Since the prototype hardware was constructed, layouts for the chassis had to be determined. The prototype hardware is not entirely well-laid out for volume manufacturing. The chassis designs are a compromise between mass-production designs and debuggable designs able to be constructed in a modest machine shop.

9.1 Control Board

The prototype control board is built around a 17 inch by 13 inch by 3 inch aluminum box. The fader, transport, and output PC boards are mounted to a piece of 1/8 inch acrylic, cut to fit over the top of the aluminum box. The top is 19 inches wide, providing rack mount ears. The panel jacks, CPU module, and Pbus controller board mount to the aluminum box.

Figure 92 shows the layout of the control board components.

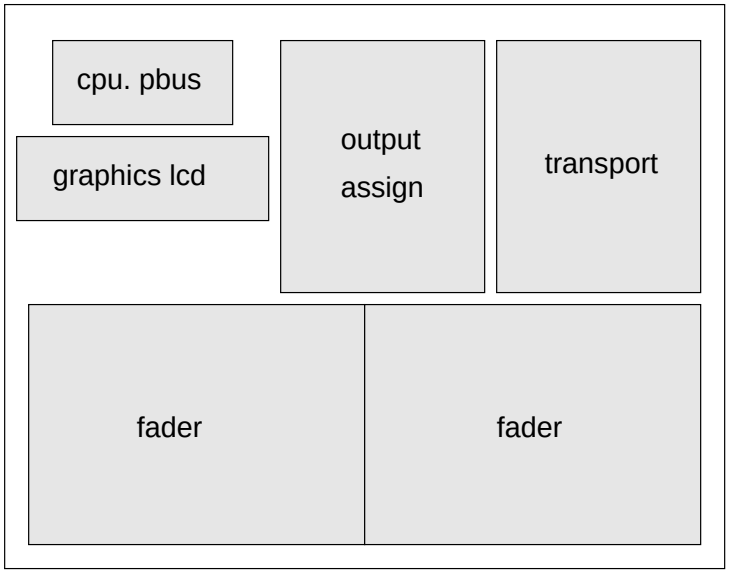


Figure 92: Control board chassis layout, top view.

9.2 Mixer Unit

The mixer unit is built around an off-the-shelf 3 rack-unit chassis. The rear panel of this chassis was removed and replaced with an aluminum frame to hold the removable audio input and output cards. A rectangular hole was cut in the front panel to mount the LCD. The display is protected by a piece of acrylic. Internal components firmly mount to the chassis.

Figure 93 shows the layout of the mixer unit components.

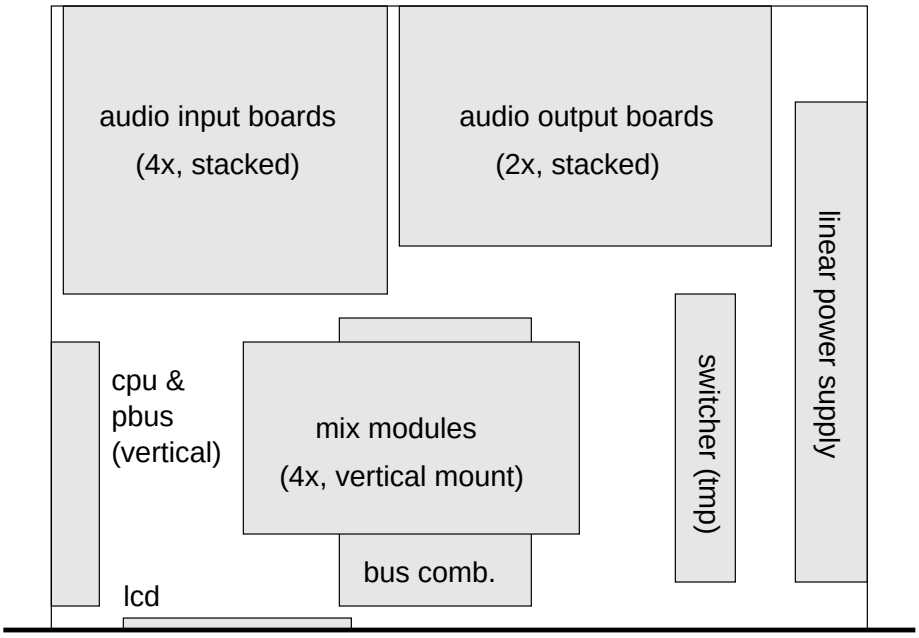


Figure 93: Mixer unit chassis layout, top view.

10 High-Level Firmware Design

Firmware is simply the name given to software running on embedded microprocessors or microcontrollers. What makes it "firm" is that it is typically loaded on to an EPROM or EEPROM, rather than a form of magnetic media, though this rule does not hold hard and fast. The exact function of firmware depends greatly upon the application. In standalone hardware, the firmware often contains elements of a user interface, application logic, etc. In peripheral or support hardware, the firmware often must handle communication with external devices in addition some of the aforementioned tasks.

Both the mixer unit and the control board require firmware to handle the functions of the underlying hardware. Since both units are peripheral-style hardware, the firmware needs to handle all of the functions mentioned above. The control board is largely a user-interface device, while the mixer unit is largely a task-oriented device. Both interface with a host PC running custom software.

The following subsections give an overview of the firmware design for the DACS control board and mixer unit. Later sections present the actual software architecture and source code.

10.1 Control Board

10.1.1 Design Overview

The firmware in the control board needs to handle serial communications, analog-to-digital conversion, Pbus communication, and LCD output. Additionally, it needs to use all of these to provide a user interface. Figure 94 shows a modular view of the required functionality.

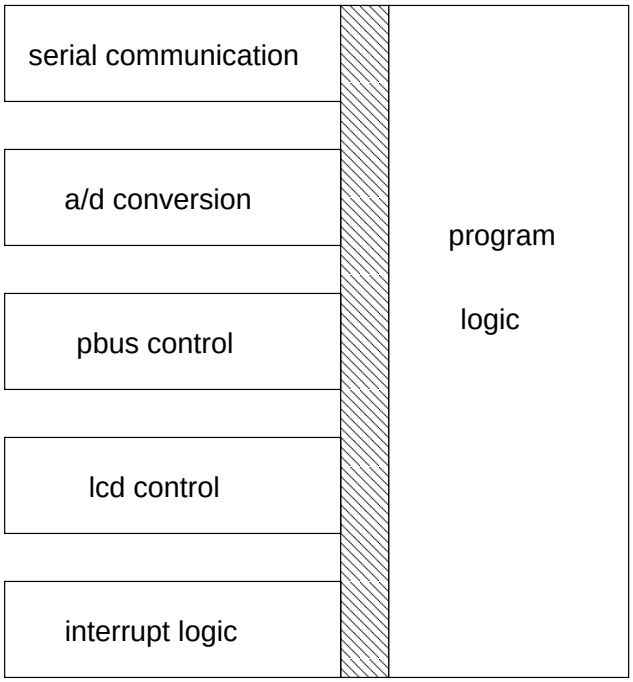


Figure 94: Control board firmware, modular overview.

While all of the low-level hardware communication functionality is relatively straightforward in concept, the user interface portion of the firmware is a bit more complex. For instance, a decision had to be made regarding exactly how much intelligence to embed into the controller itself, and how much to push to the software running on the host computer.

To aid further expansion and improvement of the user interface of the control board, a "master-slave" approach was taken. It was decided that low-level user interface primitives shall be coded into the firmware, but the use of these primitives shall be dictated by the software running on a host computer. The only function breaking this paradigm is a minimalist interface for controlling the DACS mixer unit in the absence of a host computer.

10.1.2 Overall Functional Design

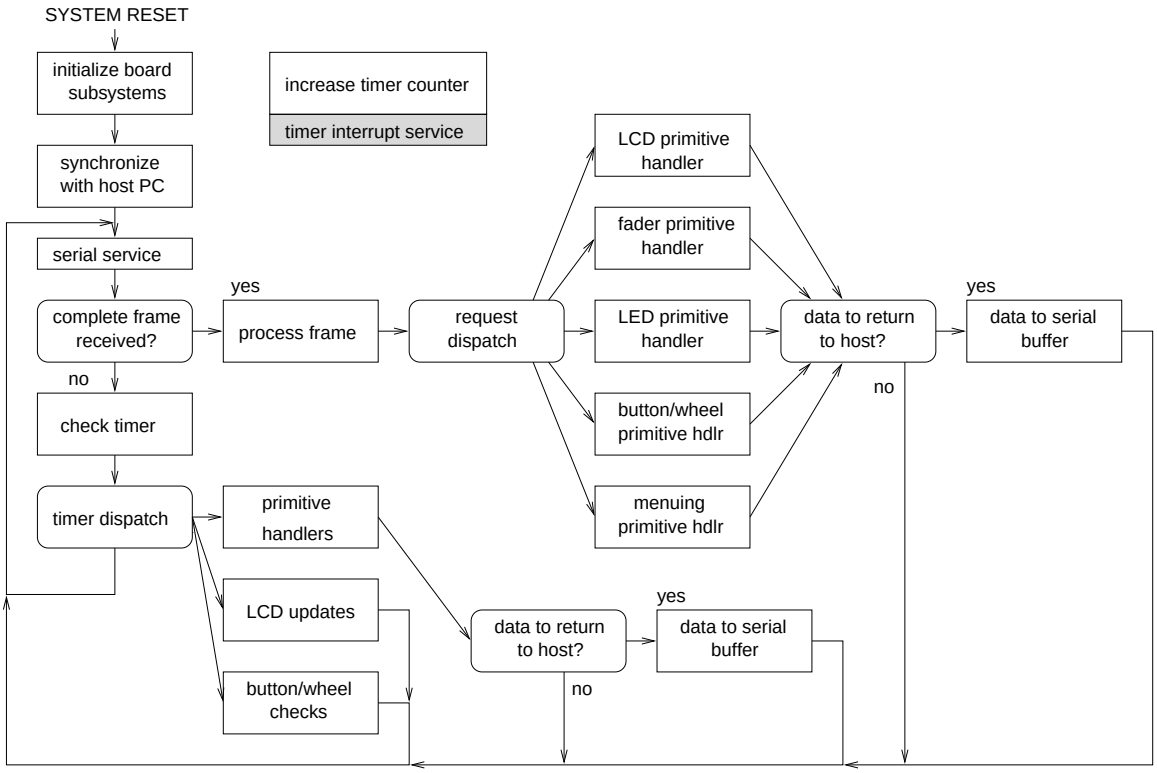


Figure 95: Control board firmware, intelligent mode functional flow diagram.

Figure 95 depicts a functional flow diagram for the intelligent mode of the control board firmware. At system reset, all components of the system are initialized. At that point, a host synchronization mode is entered, during which the host PC and the control board attempt to communicate and correctly exchange information. Upon successful completion of this synchronization, the control board sends configuration information to the host PC describing installed options, etc.

At this time, the main program loop is entered. The serial devices are serviced, thereby processing incoming and outgoing data. Should a complete frame of incoming data be received, that frame is processed and interpreted. Assuming it contains valid command data, the command is dispatched to the appropriate user interface primitive handler. For example, if a command is received to generate a menu and return the chosen value to the host computer, the appropriate action is taken.

In some cases, the primitive handlers immediately return data to the host computer, such as in the case of reading fader values, etc. In these cases, the data are immediately inserted into the serial buffer.

After processing incoming frame data, or after no data have been received from the serial subsystem, the timer counter is checked. If this timer counter has reached particular cutoff

values, functions are executed. Some user interface primitives require periodic servicing, such as the menu handling primitives.

After the timer routines are serviced, or if no timer routines are to be serviced, control loops back to the top, and the serial devices are again serviced. This loop continues until the unit is powered off.

Should, at any time, either the host PC or the control board become "confused," the confused party shall re-enter synchronization mode. Both the firmware and the host PC software shall recognize, in normal operating mode, synchronization data coming from the opposite device. This shall cause that device to also enter synchronization mode, beginning the cycle again.

The initial firmware developed for the board, and indeed, the only presently complete firmware, implements a "dumb mode" of operation. Figure 96 shows a functional flow diagram of this operating mode.

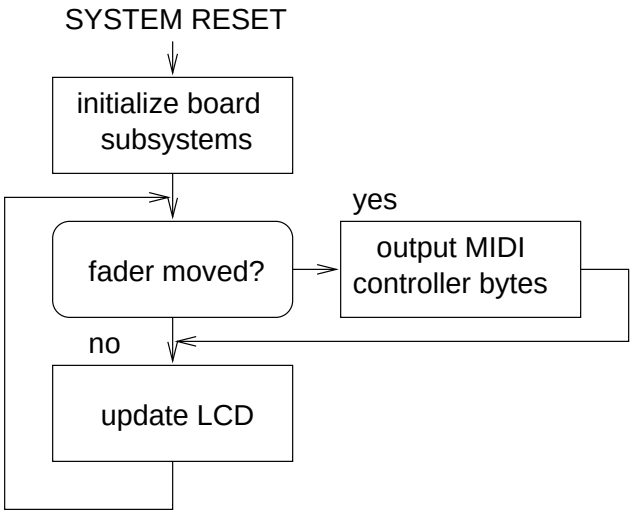


Figure 96: Control board firmware, dumb mode functional flow diagram.

Upon system reset, the various subsystems are initialized. At this point, the main loop is entered. In this loop, all of the faders are polled. If any of these faders have changed since the last poll, MIDI-like controller data are output on the serial port. This loop repeats until the system is powered off.

This mode of operation is exceptionally limited, however, due to time constraints, the complete firmware system for the control board was not able to be implemented as designed. It is believed that the design is a good one, however.

10.1.3 Component Functional Design

User interface primitives perform functions such as changing the state of LEDs, changing the contents of LCDs, displaying menu options on an LCD, and reading buttons or sliders. These functions, called by the host computer over a serial linkup, work together to form the logic of the user interface. Table 9 lists the user interface primitives.

10.2 Mixer Unit

10.2.1 Design Overview

The firmware in the mixer unit needs to handle serial communications, Pbus communication, and LCD output. Figure 97 shows a modular view of the required functionality.

primitive	input data	output	description
send fader values	none	fader value bytes (immediate)	returns all faders
send button values	none	button value bytes (immediate)	returns all buttons
send wheel counts	none	wheel count ints (immediate)	returns all wheel counts since last poll
set LED values	byte masks	none	sets LEDs on or off based on masks
set 7-segment	display id and value	none	sets 7-segment display to value
LCD string write	LCD id, string, x, y	none	displays string at x,y on LCD
LCD bitmap display	LCD id, sx, sy, data x, y	none	displays sx*sy bitmap on LCD at position x,y
menu handler	menu items, type, x, y	item id (when selected)	scrollable menu handler allows host to create menu options. user selects, host is notified of choice.
text data entry	string length, x, y	string (when done)	allows user to enter string 'arcade style'

Table 9: Control board user interface primitives.

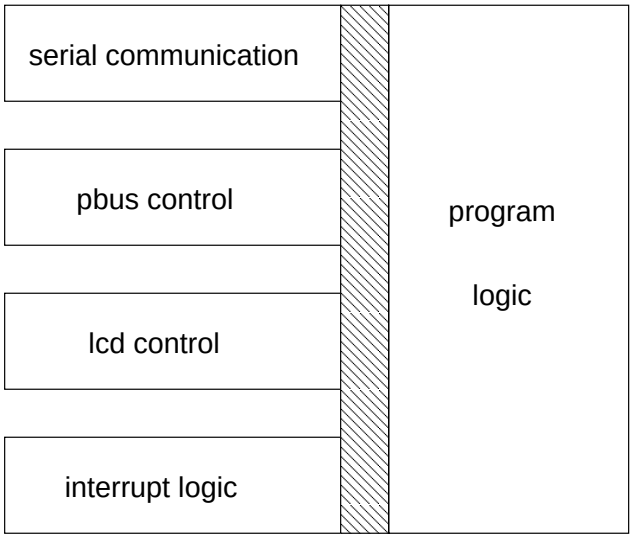


Figure 97: Mixer unit firmware, modular overview.

In addition to handling the functions mentioned above, the firmware of the mixer must service requests for mixer jobs. These requests arrive from the host computer via the serial communications module, and must be processed and executed.

10.2.2 Overall Functional Design

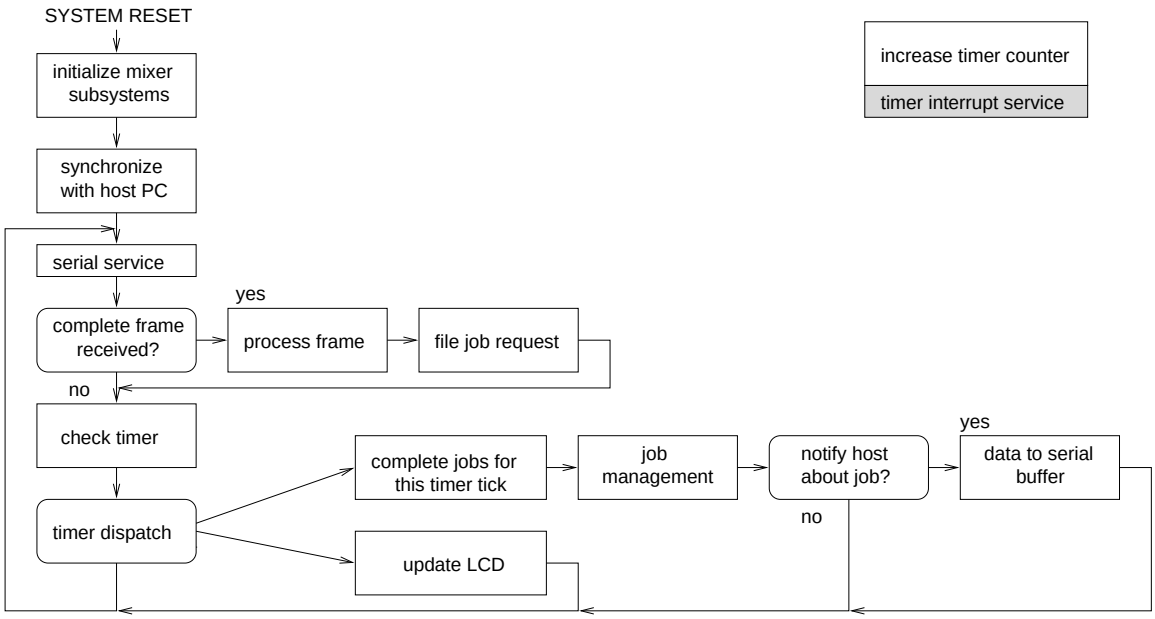


Figure 98: Mixer unit firmware, intelligent mode functional flow diagram.

Figure 98 depicts a functional flow diagram for the intelligent mode of operation for the mixer unit. At system reset, all internal subystems are initialized. System hardware is polled to determine what options are installed in the unit.

At this time, the unit enters a host synchronization mode. During this time, the mixer unit attempts to establish communications with the host PC. Once communications have been established, the mixer unit informs the host PC of its configuration and enters the main program loop.

The main loop begins with a serial device service routine. If a complete frame has been successfully received, this frame is processed. Assuming the data are valid, the job request from the host PC is entered into the job queue.

Execution continues by checking the timer value. If the appropriate amount of time has passed, all pending mixer jobs for the current time slice are serviced. Maintenance of the job tables is then completed to remove completed jobs and updating time-to-live values in unfinished jobs. Should a job ending require notifying the host PC, the data are output to the serial buffer at this point.

The status LCD is updated with current information periodically. This action is serviced by the timer dispatch. After timer processing, execution loops back to the top. This cycle repeats until the unit is powered off or reset.

Should, at any time, either the host PC or the mixer unit become "confused," the confused party shall re-enter synchronization mode. Both the firmware and the host PC software shall recognize, in normal operating mode, synchronization data coming from the opposite device. This shall cause that device to also enter synchronization mode, effectively resynchronizing both devices.

To facilitate testing, and produce a functional demonstration, an extremely stripped down "dumb" mode of operation was implemented. Unfortunately, it is still the only functioning piece of firmware at the moment, due to time constraints. Figure 99 shows the functional flow for this mode of operation.

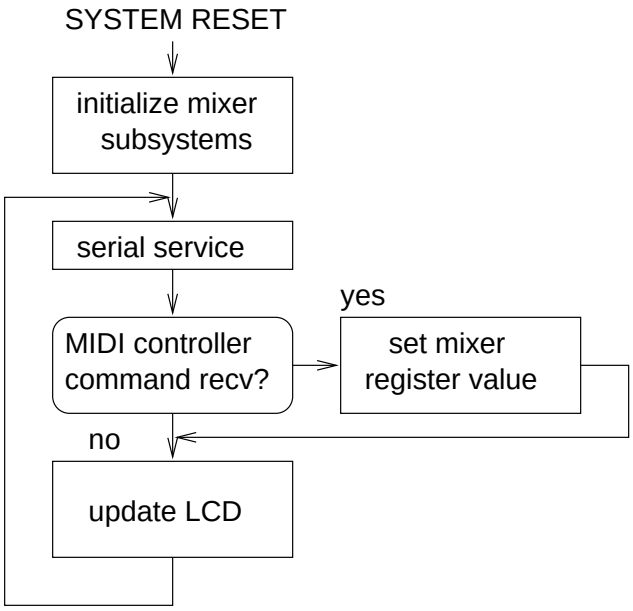


Figure 99: Mixer unit firmware, dumb mode functional flow diagram.

Upon system reset, the underlying hardware is initialized. Immediately, the system enters a loop waiting for serial data. If a valid MIDI controller command is received, channel and value data from that command are used to immediately set mixer registers. The system then returns to waiting for another valid MIDI controller command. Periodically, the system LCD is updated with simple status information. This loop continues until the device is powered off.

10.2.3 Component Functional Design

Mixer jobs may be requested by the host computer. These jobs complete in a finite amount of time. They can range from simply setting a particular mixer register to a value instantaneously to slowly fading sets of registers up or down over time. Several "slots" for jobs shall be available, allowing multiple concurrent jobs that start and end at different times. Figure 100 diagrammatically shows the mixer job engine. Note that the earliest free slot is always used for an incoming job (the position of job 4 shows this well). The time scale used is of arbitrary granularity, and should be chosen for convenience. A value of between 1/10th and 1/100th of a second should suffice for this application. The specific granularity should be chosen with an eye towards human perceptibility and microcontroller loading.

Several different types of jobs may be requested from the host, as shown in table 10. Some of the jobs allow for a variable time-to-completion. This allows variable-length audio fades to occur. Some jobs, mainly task management jobs, complete in a single time tick.

The mixer unit also has provisions for an add-on digital VU meter. This dictates that a means for the retrieval of the VU data be available. In addition, information about the installed modules in the mixer unit may be requested by the host computer.

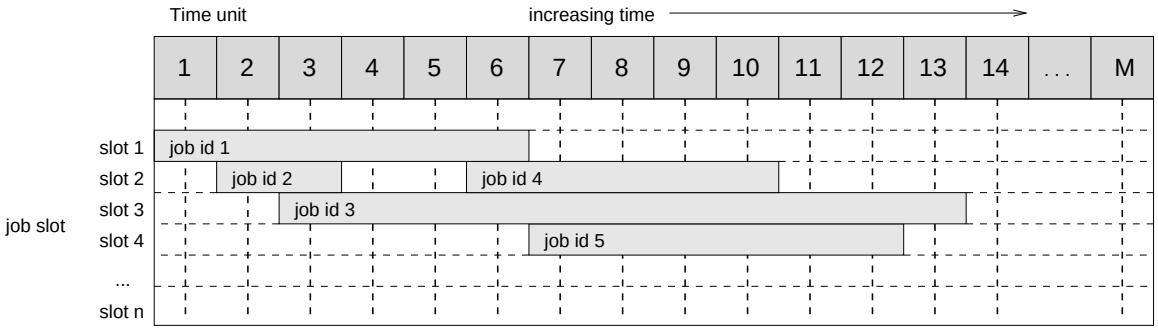


Figure 100: Mixer jobs, job slot vs. time.

Job	Time Units	Description
mixer reset	1	Reset mixer unit to default state; Cancel all jobs.
stop running job	1	Stop a particular running job, return value to initial state.
fade without notify	n	Over n time units, fade mixer register from initial value to ending value, interpolating as necessary.
fade with notify	n	Over n time units, fade mixer register from initial value to ending value, interpolating as necessary. Notify host upon completion.

Table 10: Mixer job table.

10.3 The DACS Firmware Library

Since both pieces of hardware use the same microcontroller and share some underlying hardware structure, it makes sense to build a library to provide support for the shared functionality between devices.

Both devices use serial ports, Pbus controllers, LCDs, and analog-to-digital conversion. This particular subset of functions was well-suited for inclusion in a firmware library.

Figure 101 depicts a data flow and function interface overview for the firmware library, dubbed `dacslib`.

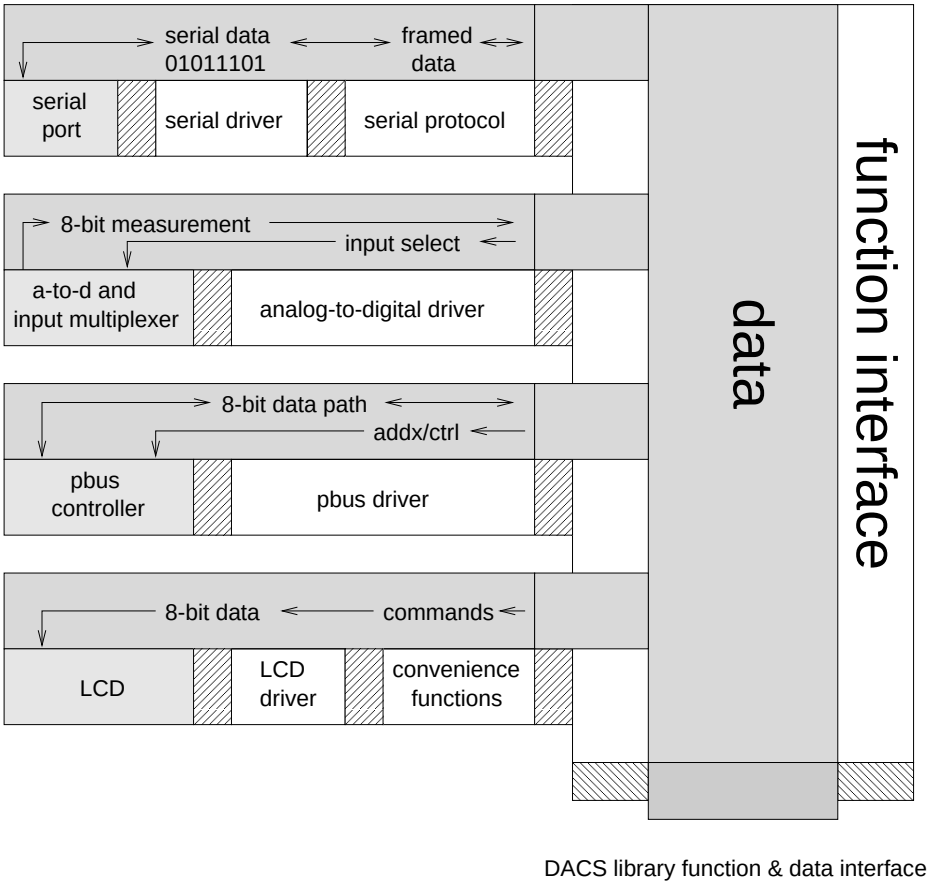


Figure 101: DACS firmware library, data flow and function interface overview.

11 Firmware Protocols

11.1 Serial Communications Protocol

To facilitate communication between the embedded microcontrollers and the host PC, a serial communications protocol was developed. The protocol is relatively simple in design, but effective.

The protocol consists of three layers: physical, framing, and application-specific. The physical layer relates to the actual medium and means of transmission. Framing relates to encapsulation and transmission/receipt of data packets. The application-specific layer defines messages that specific applications use to communicate.

11.1.1 Physical Layer

This protocol is intended to be used with a standard asynchronous serial port running at 9600 or 19200 baud, 8 data bits, no parity, one stop bit (8-N-1). Electrical characteristics of the serial line may follow EIA 232C, EIA 422 or EIA 485 specifications.

Support for the 68HC11 internal serial communications interface (SCI) shall be provided, as well as for the R65C51 Asynchronous Communications Interface Adapter (ACIA).

11.1.2 Framing

Before a raw data stream may be sent, it must be framed. A stream is of variable length, and may contain full 8-bit data. No error detection or correction is employed at this level of the protocol, however simple CRC (Cyclic Redundancy Check) calculation could be added in at a later date.

Three bytes are defined to facilitate framing the data. Most importantly, *the data link escape* (DLE) character, defined as `0x10`, tells the serial receiver that the next character received has special meaning and is not simply data. The two framing characters STX and ETX (*Start Transmission* and *End Transmission*, respectively) are used to signify the beginning and end of a frame when preceded by a DLE. STX and ETX are defined as `0x02` and `0x03` respectively. Figure 102 depicts a raw data stream properly framed by DLE, STX and ETX.

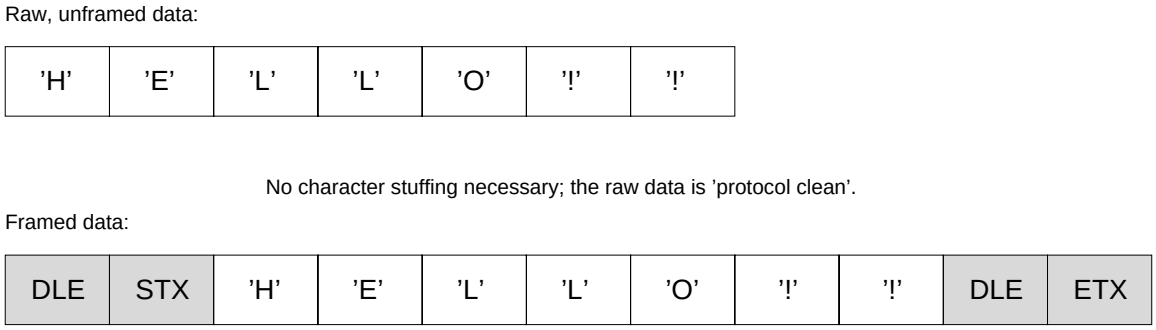


Figure 102: Serial protocol, 'clean' data framed for transmission.

Figure 102 shows a raw data stream with no embedded DLE bytes. However, it is certainly possible that the raw data stream may contain one or many DLE bytes. In this case, the sender must *character stuff*, or *escape* the embedded DLE characters. It simply doubles them up as it comes across them. The receiver merely strips the extra DLE off upon receipt of the frame. Figure 103 shows an example of a character stuffed frame.

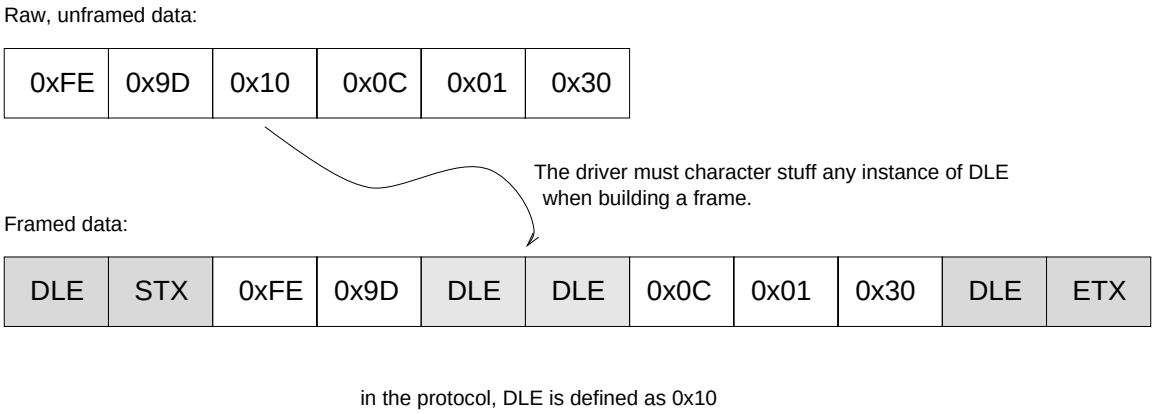


Figure 103: Serial protocol, 'unclean' data character stuffed and framed for transmission.

12 Firmware Modules

12.1 Firmware Library

The DACS firmware library provides a relatively rich set of functions to a firmware program. Figure 104 gives an overview of these functions.

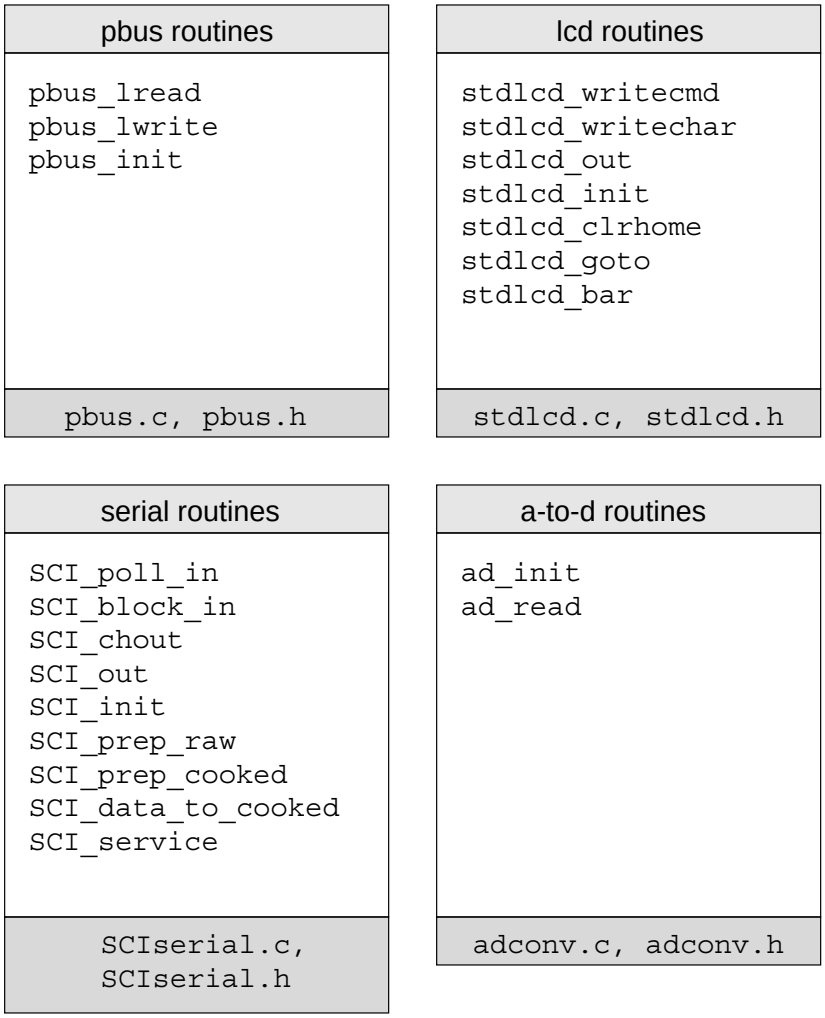


Figure 104: DACS firmware library, module view with functions.

The code, listed in the following section, provides reasonable documentation for the use of each of these functions in the comment block above each function.

12.2 Control Board

Figure 105 depicts the module layout for the "dumb" mode of the control board firmware. The shaded elements are modules from the firmware library. The remaining elements are specific to the control board application.

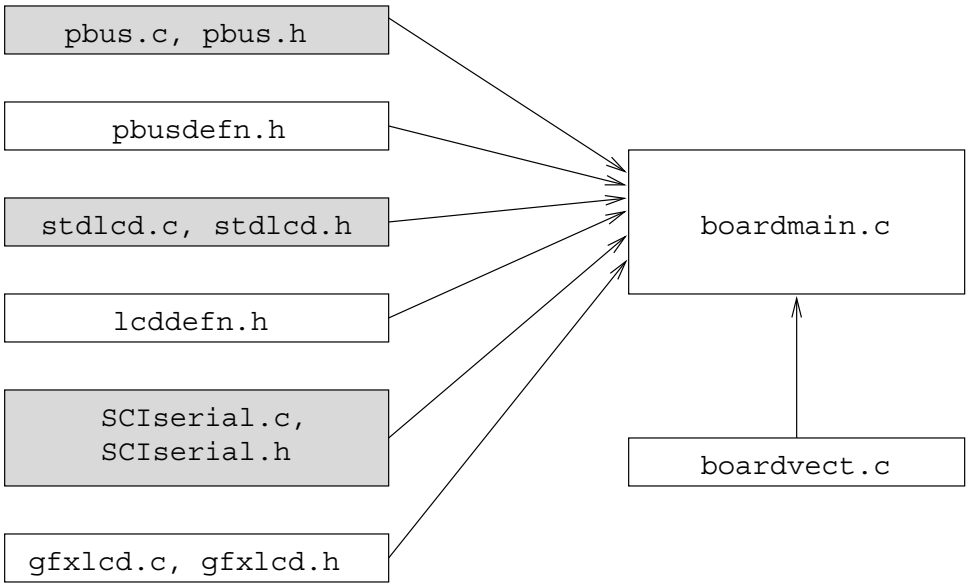


Figure 105: Control board firmware, module view.

12.3 Mixer Unit

Figure 106 depicts the module layout for the "dumb" mode of the mixer unit firmware. The shaded elements are modules from the firmware library. The remaining elements are specific to the mixer unit application.

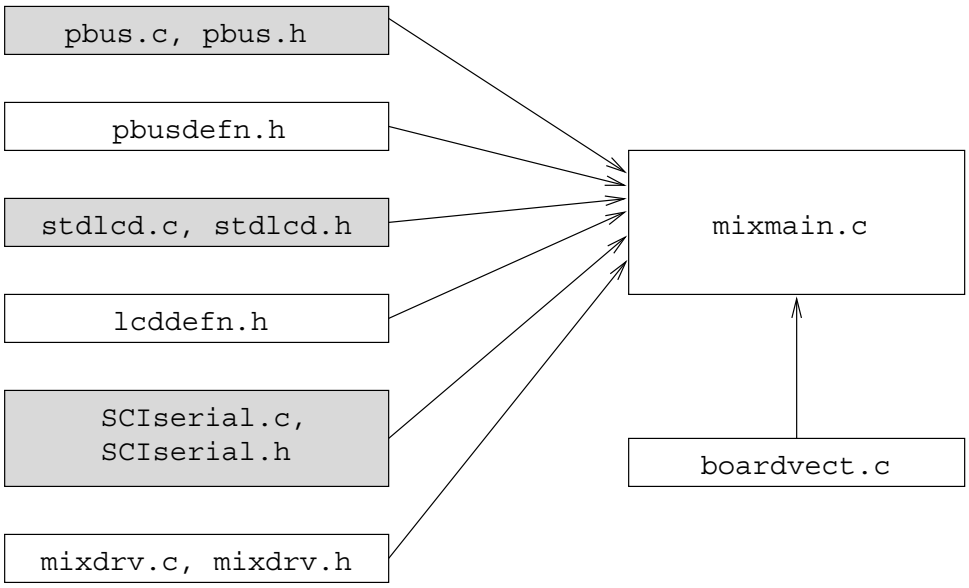


Figure 106: Mixer unit firmware, module view.

13 Firmware Code

This section presents all of the code presently written for the hardware developed. It is written in ANSI C, and is meant to be compiled using the ImageCraft 68HC11 cross-compiler `icc11`. Two versions were used during the development of the firmware, 3.6 and 4.0. No significant differences were noted between these versions, and all code should build under either version.

The Linux version of the `icc11` compiler was used. A build environment was tailored to the specific configurations of the embedded microcontrollers used. Appropriate unix shell modifications should be made to set these environment variables.

```
ICC11=/u/samba/mqp/68hc11/tools/icc11
ICC11_INCLUDE=$ICC11/include
ICC11_LIB=$ICC11/lib
ICC11_LINKER_OPTS="-dheap_size:0 -btext:0xE000 -dinit_sp:0x2FFF -bdata:0xB800"
```

13.1 Firmware Library, `dacslib`

13.1.1 SCI Serial Driver Header, `SCIserial.h`

The following header file provides function prototypes for the 68HC11 SCI serial driver, structure definitions for data frames, payload size definitions, and data link control byte definitions.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: SCIserial.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: SCIserial.h,v 1.1 1997/07/12 18:02:00 prefect Exp prefect $
 *
 *****************************************************************************/

#ifndef _SCIserial
#define _SCIserial

/* frame markers */
#define DLE 0x10     /* data link escape */
#define STX 0x02     /* start transmit */
#define ETX 0x03     /* end transmit */

/* buffers and lengths */

#define MAXPAYLOAD 128                    /* maximum length of the payload */
#define MAXFRLEN MAXPAYLOAD+MAXPAYLOAD+4  /* max frame length */
#define INBUFLEN MAXFRLEN+1               /* size of input buffer */
```

```
#define OUTBUFLEN MAXFRLEN+1              /* size of output buffer */


/* errors */

#define ERRTOOBIG  10         /* frame done; was too big! */

/* structures */

struct cooked_frame {          /* character stuffed frame */
  unsigned int len;
  unsigned int cur;
  unsigned char done;
  unsigned char data[MAXFRLEN];
};

struct raw_frame {             /* unstuffed frame */
  unsigned int len;
  unsigned int cur;
  unsigned char done;
  unsigned char dleflag;
  unsigned char stflag;
  unsigned char data[MAXPAYLOAD];
};

int SCI_poll_in (unsigned char *c);
char SCI_block_in (void);
void SCI_chout (unsigned char ch);
void SCI_out (char *s);
void SCI_init (void);
void SCI_prep_raw (struct raw_frame *raw);
void SCI_prep_cooked (struct cooked_frame *cooked);
void SCI_data_to_cooked (char *data, int len, struct cooked_frame *cooked);
int SCI_service (struct raw_frame *inraw, struct cooked_frame *outfr);

#endif
```

13.1.2 SCI Serial Driver Code, `SCIserial.c`

The following code provides serial driver functions for the 68HC11 SCI serial interface. Frame handling functions are provided as well as simple serial access routines.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: SCIserial.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
```

```
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************
 * Source code control:
 *
 * $Id: SCIserial.c,v 1.3 1997/07/13 01:23:16 prefect Exp prefect $
 *
 ******************************************************************************/

#include <hc11.h>
#include "SCIserial.h"

#ifdef _MIXER
#include "pbusdefn.h"
#endif


/******************************************************************************
 * SCI_prep_raw
 *
 * initializes a raw frame
 ******************************************************************************/
void SCI_prep_raw (struct raw_frame *raw)
{
  int i;

  raw->len=0;
  raw->cur=0;
  raw->done=0;
  raw->dleflag=0;
  raw->stflag=0;
  for (i=0;i<MAXPAYLOAD;i++) raw->data[i]=0;
}


/******************************************************************************
 * SCI_prep_cooked
 *
 * initializes a cooked frame
 ******************************************************************************/
void SCI_prep_cooked (struct cooked_frame *cooked)
{
  int i;

  cooked->len=0;
  cooked->cur=0;
  cooked->done=1;

  for (i=0;i<MAXFRLEN;i++) cooked->data[i]=0;
}


/******************************************************************************
```

```
 * SCI_data_to_cooked
 *
 * stuffs data into a cooked frame, setting parameters, etc.  output frame
 * is suitable to be directly output.
 ****************************************************************************/
void SCI_data_to_cooked (char *data, int len, struct cooked_frame *cooked)
{
  int i=0, j=0;
  char *p;

  /* start of frame: DLE+STX */
  cooked->data[j++]=DLE;
  cooked->data[j++]=STX;

  /* payload of frame (character stuffed) */
  p=data;

  for (i=0;i<len;i++) {
    if (*p == DLE) {
      cooked->data[j++]=DLE;
      cooked->data[j++]=DLE;
      p++;
    } else {
      cooked->data[j++]=*p;
      p++;
    }
  }

  /* end of frame: DLE+ETX */
  cooked->data[j++]=DLE;
  cooked->data[j++]=ETX;

  /* set length and other parameters */
  cooked->len=j;
  cooked->cur=0;
  cooked->done=0;
}


/****************************************************************************
 * SCI_service
 *
 * perform necessary SCI operations to manipulate incoming and outgoing
 * frames, using a raw frame as an input buffer
 ****************************************************************************/
int SCI_service (struct raw_frame *inraw, struct cooked_frame *outfr)
{
  unsigned char ch;

  /* outgoing serial data & sci output port !busy? */

  if ( (SCSR&0x80) && (outfr->done==0) ) {
```

```
#ifdef _MIXER
    PBC_PC^=0x08;
#endif
    SCDR=outfr->data[outfr->cur];  /* output the current byte */

    /* have we transmitted all of it yet? */
    if (++outfr->cur < outfr->len) {
      /* not yet.. */
    } else {
      /* yes */
      outfr->done = 1;
    }
#ifdef _MIXER
    PBC_PC&=~0x08;
#endif
  }

  /* incoming data on sci and unfinished inraw frame? */

  if ( (SCSR&0x20) && (inraw->done==0) ) {
    /* -- yes, get and fill in the next byte */

#ifdef _MIXER
    PBC_PC^=0x04;
#endif
    ch = SCDR;

    if (inraw->dleflag) {
      /* the last byte received was a DLE */

      if (ch == DLE) {
/* stuffed character */

if (inraw->stflag == STX) {
  /* save it if we're mid-frame */
  inraw->data[inraw->cur] = DLE;
  inraw->dleflag=0;

  if (++inraw->cur < MAXPAYLOAD) {
  } else {
    inraw->done = ERRTOOBIG;
  }
}
      } else if (ch == STX) {
/* start of frame */
inraw->done = 0;
inraw->cur = 0;
inraw->dleflag=0;
inraw->stflag=STX;

      } else if (ch == ETX) {
```

```
/* end of frame */
inraw->done = 1;
inraw->dleflag=0;
inraw->stflag=0;
      }

    } else {
      if (ch == DLE) {
/* first DLE */
inraw->dleflag = 1;
      } else {
/* other data */

if (inraw->stflag == STX) {
  /* save it if we're mid-frame */
  inraw->data[inraw->cur] = ch;
  inraw->dleflag=0;
  if (++inraw->cur < MAXPAYLOAD) {
  } else {
    inraw->done = ERRTOOBIG;
  }
}
      }
    }
#ifdef _MIXER
    PBC_PC&=~0x04;
#endif
  }
}


/*****************************************************************************
 * SCI_poll_in
 *
 * polled SCI input.  returns 1 if there was actually a character to get.
 *****************************************************************************/
int SCI_poll_in (unsigned char *c)
{
   if (SCSR&0x20) {
      *c=SCDR;
      return 1;
   } else {
      return 0;
   }
}


/*****************************************************************************
 * SCI_block_in
 *
 * blocking SCI input.  returns character.
 *****************************************************************************/
```

```
char SCI_block_in (void)
{
   while (!(SCSR&0x20));
   return (SCDR);
}


/*****************************************************************************
 * SCI_poll_in
 *
 * polled SCI input.  returns 1 if there was actually a character to get.
 *****************************************************************************/
void SCI_chout (unsigned char ch)
{
   while (!(SCSR&0x80));
   SCDR = ch;
}


/*****************************************************************************
 * SCI_out
 *
 * SCI output (blocking)
 *****************************************************************************/
void SCI_out (char *s)
{
   char *t=s;

   while (*t!=0) {
     while (!(SCSR&0x80));
     SCDR = *t++;
   }
}


/*****************************************************************************
 * SCI_init
 *
 * initialize SCI port, 9600 baud, 8-N-1
 *****************************************************************************/
void SCI_init (void)
{
   BAUD=0x30;
   SCCR2=0x2C;
}
```

13.1.3 ACIA Serial Driver Header File, `ACIAserial.h`

The following header file is very similar to `SCIserial.c`, except it provides declarations for R65C51 ACIA (Asynchronous Communications Interface Adapter) communications. This ACIA is present on the CMD-11A8 single-board computer used in the mixer unit.

```
/*****************************************************************************
```

```
 * DACS : Distributed Audio Control System
 *=============================================================================
 *         File: ACIAserial.h
 *       Author: Stephen S. Richardson
 * Date Created: 07.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library
 *=============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: ACIAserial.h,v 1.1 1997/07/23 22:12:31 prefect Exp prefect $
 *
 *****************************************************************************/

#ifndef _ACIAserial
#define _ACIAserial

/* ACIA port definitions (CMD11A8) */

#define ACIACTRL *(unsigned char *)(0xB5FB)
#define ACIACMD  *(unsigned char *)(0xB5FA)
#define ACIASTAT *(unsigned char *)(0xB5F9)
#define ACIADATA *(unsigned char *)(0xB5F8)


/* frame markers */
#define DLE 0x10     /* data link escape */
#define STX 0x02     /* start transmit */
#define ETX 0x03     /* end transmit */

/* buffers and lengths */

#define MAXPAYLOAD 128                     /* maximum length of the payload */
#define MAXFRLEN MAXPAYLOAD+MAXPAYLOAD+4   /* max frame length */
#define INBUFLEN MAXFRLEN+1                /* size of input buffer */
#define OUTBUFLEN MAXFRLEN+1               /* size of output buffer */


/* errors */

#define ERRTOOBIG  10         /* frame done; was too big! */

/* structures */

struct cooked_frame {          /* character stuffed frame */
  unsigned int len;
  unsigned int cur;
  unsigned char done;
```

```
  unsigned char data[MAXFRLEN];
};

struct raw_frame {              /* unstuffed frame */
  unsigned int len;
  unsigned int cur;
  unsigned char done;
  unsigned char dleflag;
  unsigned char stflag;
  unsigned char data[MAXPAYLOAD];
};

int ACIA_poll_in (unsigned char *c);
char ACIA_block_in (void);
void ACIA_chout (unsigned char ch);
void ACIA_out (char *s);
void ACIA_init (void);
void ACIA_prep_raw (struct raw_frame *raw);
void ACIA_prep_cooked (struct cooked_frame *cooked);
void ACIA_data_to_cooked (char *data, int len, struct cooked_frame *cooked);
int ACIA_service (struct raw_frame *inraw, struct cooked_frame *outfr);

#endif
```

13.1.4 ACIA Serial Driver Code, `ACIAserial.c`

The following code implements the ACIA serial driver code. It is very similar to `SCIserial.c`.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: ACIAserial.c
 *       Author: Stephen S. Richardson
 * Date Created: 07.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: ACIAserial.c,v 1.1 1997/07/23 22:12:36 prefect Exp prefect $
 *
 *****************************************************************************/

#include <hc11.h>
#include "ACIAserial.h"

#ifdef _MIXER
#include "pbusdefn.h"
#endif
```

```
/*****************************************************************************
 * ACIA_prep_raw
 *
 * initializes a raw frame
 *****************************************************************************/
void ACIA_prep_raw (struct raw_frame *raw)
{
  int i;

  raw->len=0;
  raw->cur=0;
  raw->done=0;
  raw->dleflag=0;
  raw->stflag=0;
  for (i=0;i<MAXPAYLOAD;i++) raw->data[i]=0;
}


/*****************************************************************************
 * ACIA_prep_cooked
 *
 * initializes a cooked frame
 *****************************************************************************/
void ACIA_prep_cooked (struct cooked_frame *cooked)
{
  int i;

  cooked->len=0;
  cooked->cur=0;
  cooked->done=1;

  for (i=0;i<MAXFRLEN;i++) cooked->data[i]=0;
}


/*****************************************************************************
 * ACIA_data_to_cooked
 *
 * stuffs data into a cooked frame, setting parameters, etc.  output frame
 * is suitable to be directly output.
 *****************************************************************************/
void ACIA_data_to_cooked (char *data, int len, struct cooked_frame *cooked)
{
  int i=0, j=0;
  char *p;

  /* start of frame: DLE+STX */
  cooked->data[j++]=DLE;
  cooked->data[j++]=STX;
```

```
  /* payload of frame (character stuffed) */
  p=data;

  for (i=0;i<len;i++) {
    if (*p == DLE) {
      cooked->data[j++]=DLE;
      cooked->data[j++]=DLE;
      p++;
    } else {
      cooked->data[j++]=*p;
      p++;
    }
  }

  /* end of frame: DLE+ETX */
  cooked->data[j++]=DLE;
  cooked->data[j++]=ETX;

  /* set length and other parameters */
  cooked->len=j;
  cooked->cur=0;
  cooked->done=0;
}


/*****************************************************************************
 * ACIA_service
 *
 * perform necessary ACIA operations to manipulate incoming and outgoing
 * frames, using a raw frame as an input buffer
 *****************************************************************************/
int ACIA_service (struct raw_frame *inraw, struct cooked_frame *outfr)
{
  unsigned char ch;

  /* outgoing serial data & ACIA output port !busy? */

  if ( (ACIASTAT&0x10) && (outfr->done==0) ) {

#ifdef _MIXER
    PBC_PC^=0x08;
#endif
    ACIADATA=outfr->data[outfr->cur];  /* output the current byte */

    /* have we transmitted all of it yet? */
    if (++outfr->cur < outfr->len) {
      /* not yet.. */
    } else {
      /* yes */
      outfr->done = 1;
    }
#ifdef _MIXER
```

```
    PBC_PC&=~0x08;
#endif
  }

  /* incoming data on ACIA and unfinished inraw frame? */

  if ( (ACIASTAT&0x08) && (inraw->done==0) ) {
    /* -- yes, get and fill in the next byte */

#ifdef _MIXER
    PBC_PC^=0x04;
#endif
    ch = ACIADATA;

    if (inraw->dleflag) {
      /* the last byte received was a DLE */

      if (ch == DLE) {
/* stuffed character */

if (inraw->stflag == STX) {
  /* save it if we're mid-frame */
  inraw->data[inraw->cur] = DLE;
  inraw->dleflag=0;

  if (++inraw->cur < MAXPAYLOAD) {
  } else {
    inraw->done = ERRTOOBIG;
  }
}
      } else if (ch == STX) {
/* start of frame */
inraw->done = 0;
inraw->cur = 0;
inraw->dleflag=0;
inraw->stflag=STX;

      } else if (ch == ETX) {
/* end of frame */
inraw->done = 1;
inraw->dleflag=0;
inraw->stflag=0;
      }

    } else {
      if (ch == DLE) {
/* first DLE */
inraw->dleflag = 1;
      } else {
/* other data */

if (inraw->stflag == STX) {
```

```
  /* save it if we're mid-frame */
  inraw->data[inraw->cur] = ch;
  inraw->dleflag=0;
  if (++inraw->cur < MAXPAYLOAD) {
  } else {
    inraw->done = ERRTOOBIG;
  }
}
      }
    }
#ifdef _MIXER
    PBC_PC&=~0x04;
#endif
  }
}


/*****************************************************************************
 * ACIA_poll_in
 *
 * polled ACIA input.  returns 1 if there was actually a character to get.
 *****************************************************************************/
int ACIA_poll_in (unsigned char *c)
{
   if (ACIASTAT&0x08) {
      *c=ACIADATA;
      return 1;
   } else {
      return 0;
   }
}


/*****************************************************************************
 * ACIA_block_in
 *
 * blocking ACIA input.  returns character.
 *****************************************************************************/
char ACIA_block_in (void)
{
   while (!(ACIASTAT&0x08));
   return (ACIADATA);
}


/*****************************************************************************
 * ACIA_poll_in
 *
 * polled ACIA input.  returns 1 if there was actually a character to get.
 *****************************************************************************/
void ACIA_chout (unsigned char ch)
{
```

```
   while (!(ACIASTAT & 0x10));
   ACIADATA = ch;
}


/*****************************************************************************
 * ACIA_out
 *
 * ACIA output (blocking)
 *****************************************************************************/
void ACIA_out (char *s)
{
   char *t=s;

   while (*t!=0) {
     while (!(ACIASTAT & 0x10));
     ACIADATA = *t++;
   }
}


/*****************************************************************************
 * ACIA_init
 *
 * initialize ACIA port, 9600 baud, 8-N-1
 *****************************************************************************/
void ACIA_init (void)
{
  ACIACTRL = 0x1E;
  ACIACMD = 0xCB;
}
```

13.1.5 pbus Driver Header File, `pbus.h`

The following is the header file for the Pbus driver code. It merely provides function prototypes for the Pbus functions.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: pbus.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: pbus.h,v 1.1 1997/07/23 21:31:54 prefect Exp prefect $
```

```
 *
 ******************************************************************************/

#ifndef _pbus
#define _pbus

void pbus_lwrite (unsigned char addx, unsigned char data);
unsigned char pbus_lread (unsigned char addx);
void pbus_init (void);

#endif
```

13.1.6 pbus Driver Code, pbus.c

The code that implements the Pbus driver is presented below.

```
/******************************************************************************
 * DACS : Distributed Audio Control System
 *=============================================================================
 *         File: pbus.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *=============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************
 * Source code control:
 *
 * $Id: pbus.c,v 1.1 1997/07/23 21:30:10 prefect Exp prefect $
 *
 ******************************************************************************/

#include <hc11.h>
#include "pbus.h"
#include "pbusdefn.h"


/******************************************************************************
 * pbus_init
 *
 * initializes pBUS controller, appropriate bits, etc.
 ******************************************************************************/
void pbus_init (void)
{
  PBC_PX=PBC_OX;                        /* pBUS in output mode */
  PBC_PA=0;                             /* pBUS data bus = 0 */
  PBC_PB=0;                             /* pBUS addx bus = 0 */
  PBC_PC|=PB_N_LATCH|PB_N_CLOCK;        /* pBUS latch- and clock- inactive */

}
```

```
/*****************************************************************************
 * pbus_lwrite
 *
 * does a standard pBUS latch write operation; writes data to pBUS addx.
 *****************************************************************************/
void pbus_lwrite (unsigned char addx, unsigned char data)
{
  PBC_PX=PBC_OX;                          /* pBUS in output mode */
  PBC_PC|=PB_N_LATCH|PB_N_CLOCK;          /* pBUS latch- and clock- inactive */

  PBC_PA=data;                            /* data on pBUS */
  PBC_PB=addx|PB_N_RD_WR;                 /* addx on pBUS, write mode */
  PBC_PC&=~PB_N_LATCH;                    /* pBUS latch- active */
  PBC_PC|=PB_N_LATCH;                     /* pBUS latch- inactive */
}


/*****************************************************************************
 * pbus_lread
 *
 * does a standard pBUS latch read operation; returns pBUS data from addx
 *****************************************************************************/
unsigned char pbus_lread (unsigned char addx)
{
  unsigned char c;

  PBC_PX=PBC_IX;                          /* pBUS in input mode */
  PBC_PC|=PB_N_LATCH|PB_N_CLOCK;          /* pBUS latch- and clock- inactive */

  PBC_PB=addx&~PB_N_RD_WR;                /* addx on pBUS, read mode */
  PBC_PC&=~PB_N_LATCH;                    /* pBUS latch- active */
  c=PBC_PA;                               /* data from pBUS */
  PBC_PC|=PB_N_LATCH;                     /* pBUS latch- inactive */
  return (c);
}
```

13.1.7 pbus Defines Template, `pbusdefn-template.h`

The following is a template `pbusdefn.h` file. In an application that uses the Pbus control code, this template is modified to reflect the memory locations of the controller ports. Additionally, this file provides a stub for the definition of application-specific Pbus addresses.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: pbusdefn.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
```

```
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************
 * Source code control:
 *
 * $Id: pbusdefn-template.h,v 1.2 1997/07/23 21:35:55 prefect Exp prefect $
 *
 ******************************************************************************/

#ifndef _pbusdefn
#define _pbusdefn

#define PBC_PA *(unsigned char *)(0xB580)  /* pBUS controller port A */
#define PBC_PB *(unsigned char *)(0xB581)  /* pBUS controller port B */
#define PBC_PC *(unsigned char *)(0xB582)  /* pBUS controller port C */
#define PBC_PX *(unsigned char *)(0xB583)  /* pBUS controller ctrl port */

#define PBC_IX 0x98   /* pBUS ctrl word: PA i / PB o / PC0-3 o / PC4-7 i */
#define PBC_OX 0x88   /* pBUS ctrl word: PA o / PB o / PC0-3 o / PC4-7 i */

/* pbus address definitions */
/* example: #define PBADX_MIX0  0x10    */

#endif
```

13.1.8 Alphanumeric LCD Driver Header, `stdlcd.h`

This header provides the function prototypes for the standard LCD driver.

```
/******************************************************************************
 * DACS : Distributed Audio Control System
 *============================================================================
 *         File: stdlcd.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************
 * Source code control:
 *
 * $Id: stdlcd.h,v 1.1 1997/07/23 22:33:23 prefect Exp prefect $
 *
 ******************************************************************************/

#ifndef _stdlcd
#define _stdlcd

void stdlcd_writecmd (unsigned char idx, unsigned char d);
void stdlcd_writechar (unsigned char idx, unsigned char d);
```

```
void stdlcd_out (unsigned char idx, char *s);
void stdlcd_init (unsigned char idx, int *s);
void stdlcd_clrhome (unsigned char idx);
void stdlcd_goto (unsigned char idx, unsigned char pos);
void stdlcd_bar (unsigned char idx, unsigned char pos, unsigned char part,
                 unsigned char total);
#endif
```

13.1.9 Alphanumeric LCD Driver Code, `stdlcd.c`

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: stdlcd.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/
 * Source code control:
 *
 * $Id: stdlcd.c,v 1.1 1997/07/23 22:33:29 prefect Exp prefect $
 *
 *****************************************************************************/

#include <hc11.h>
#include "stdlcd.h"
#include "lcddefn.h"

/*****************************************************************************
 * stdlcd_writecmd
 *
 * write a command byte to a standard lcd
 *****************************************************************************/
void stdlcd_writecmd (unsigned char idx, unsigned char d)
{
  unsigned char *o;

  while ((STDLCD_CMD+(idx*STDLCD_OFFSET))&0x80);
  o=&STDLCD_CMD+(idx*STDLCD_OFFSET);
  *o=d;
}

/*****************************************************************************
 * stdlcd_writechar
 *
 * write a data byte to a standard lcd
 *****************************************************************************/
void stdlcd_writechar (unsigned char idx, unsigned char d)
```

```
{
  unsigned char *o;

  while ((STDLCD_CMD+(idx*STDLCD_OFFSET))&0x80);
  o=&STDLCD_DATA+(idx*STDLCD_OFFSET);
  *o=d;
}

/*****************************************************************************
 * stdlcd_out
 *
 * write a string of data to a standard lcd
 *****************************************************************************/
void stdlcd_out (unsigned char idx, char *s)
{
  unsigned char *o;

  while (*s) {
    while ((STDLCD_CMD+(idx*STDLCD_OFFSET))&0x80);
    o=&STDLCD_DATA+(idx*STDLCD_OFFSET);
    *o=(unsigned char) *s;
    s++;
  }
}


/*****************************************************************************
 * stdlcd_init
 *
 * initialize a standard lcd
 *****************************************************************************/
void stdlcd_init (unsigned char idx, int *s)
{
  unsigned char *o;

  while (*s!=-1) {
    while ((STDLCD_CMD+(idx*STDLCD_OFFSET))&0x80);
    o=&STDLCD_CMD+(idx*STDLCD_OFFSET);
    *o=(unsigned char) *s;
    s++;
  }
}


/*****************************************************************************
 * stdlcd_clrhome
 *
 * clears standard lcd screen, homes cursor.
 *****************************************************************************/
void stdlcd_clrhome (unsigned char idx)
{
  stdlcd_writecmd(idx, 0x01);
```

```
  stdlcd_writecmd(idx, 0x00);
}

/*****************************************************************************
 * stdlcd_clrhome
 *
 * places the cursor on a standard lcd
 *****************************************************************************/
void stdlcd_goto (unsigned char idx, unsigned char pos)
{
  stdlcd_writecmd(idx, pos|0x80);
}


/*****************************************************************************
 * stdlcd_bar
 *
 * draws a percentage type bar graph on a standard lcd
 *****************************************************************************/
void stdlcd_bar (unsigned char idx, unsigned char pos, unsigned char *part,
                 unsigned char total)
{
  char txt[41], i;

  stdlcd_goto (idx, pos);

  for (i=0;i<total;i++) {
    if (i+1<part) txt[i]=0xDB;
    else txt[i]=' ';
  }

  txt[total]=0;

  stdlcd_out (idx, txt);

}
```

13.2 Control Board

13.2.1 Firmware Build, `Makefile`

The following GNU `Makefile` is used to build the firmware image for the control board. A standard Motorola S19 file is produced when the files are built. This image can be downloaded to the EEPROM on the embedded microcontroller board using special software.

```
##############################################################################
# DACS : Distributed Audio Control System
# Model 112 System Control Board / Firmware Build File
#
# The code, executables, documentation, firmware images, and all related
# material of DACS are
# Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
##############################################################################
```

```
CC = icc11
CFLAGS =
PROGS = links board
MAINDIR = .
S19DIR = $(MAINDIR)/s19
OBJDIR = $(MAINDIR)/obj
DACSLIB = ../dacslib

all: $(PROGS)

#
# set up links to DACSlib files, since icc11 3.6 can't do what i want..
# remember to change this if 'local' copies of the lib funcs are used
# (so they don't get overwritten)
# also change the make clean stuff if locals are used.
#
links:
rm -f SCIserial.h SCIserial.c pbus.h pbus.c stdlcd.h stdlcd.c
ln -s $(DACSLIB)/SCIserial.c SCIserial.c
ln -s $(DACSLIB)/SCIserial.h SCIserial.h
ln -s $(DACSLIB)/pbus.c pbus.c
ln -s $(DACSLIB)/pbus.h pbus.h
ln -s $(DACSLIB)/stdlcd.c stdlcd.c
ln -s $(DACSLIB)/stdlcd.h stdlcd.h

buildnum:
./buildnum.pl

board: links buildnum
$(CC) $(CFLAGS) -oboard.s19 *.c
mv board.s19 $(S19DIR)
mv *.o $(OBJDIR)

clean:
rm -f board.s19 *.o *~ $(S19DIR)/board.s19 $(OBJDIR)/*.o
rm -f SCIserial.h SCIserial.c pbus.h pbus.c stdlcd.c stdlcd.h
```

13.2.2 Control Board Main Code, `boardmain.c`

The following code implements a simple operation mode of the control board in which fader data is sent out as MIDI controller changes.

```
/*************************************************************************
 * DACS : Distributed Audio Control System
 *=======================================================================
 *         File: boardmain.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v3.6, 68HC11 target
 *        Build: library, not standalone
 *=======================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
```

```
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************/

#include <hc11.h>
#include "build.h"
#include "stdlcd.h"
#include "gfxlcd.h"
#include "lcddefn.h"
#include "SCIserial.h"
#include "pbus.h"
#include "pbusdefn.h"

unsigned char get_ad (unsigned char ch)
{
  ADCTL = ch;
  while (!(ADCTL & 0x80));
  return(ADR1);
}

void get_faders (unsigned char *list)
{
  unsigned char hc, m;
  int i;

  PBC_PX=PBC_OX;                          /* pBUS in output mode */
  PBC_PC^=PB_N_LATCH+PB_N_CLOCK;          /* pBUS latch- and clock- inactive */


  PBC_PA=0;                               /* data on pBUS */
  PBC_PB=PBADX_FADEMUX0^PB_N_RD_WR;       /* addx on pBUS, write mode */
  PBC_PC&=~PB_N_LATCH;                    /* pBUS latch- active */

  for (m=0;m<8;m++) {
    for (i=0;i<100;i++);
    PBC_PA=m;                             /* data on pBUS */
    ADCTL = 0;
    while (!(ADCTL & 0x80));
    list[m]=(ADR1+ADR2+ADR3+ADR4)/4;
  }
  PBC_PC^=PB_N_LATCH;                     /* pBUS latch- inactive */



  PBC_PA=0;                               /* data on pBUS */
  PBC_PB=PBADX_FADEMUX1^PB_N_RD_WR;       /* addx on pBUS, write mode */
  PBC_PC&=~PB_N_LATCH;                    /* pBUS latch- active */

  for (m=0;m<8;m++) {
    for (i=0;i<100;i++);
    PBC_PA=m;                             /* data on pBUS */
    ADCTL = 1;
    while (!(ADCTL & 0x80));
```

```
    list[m+8]=(ADR1+ADR2+ADR3+ADR4)/4;
  }
  PBC_PC^=PB_N_LATCH;                    /* pBUS latch- inactive */

}


void main (void) {
  int optrex_init[]=OPTREX16x1_INIT;

  unsigned char i,n,j;
  unsigned char faders[16], oldfaders[16];

  OPTION^=0x80;

  stdlcd_init(OPTREX16x1, optrex_init);
  stdlcd_goto(OPTREX16x1, 0);
  stdlcd_out(OPTREX16x1, "TRANSPOR");
  stdlcd_goto(OPTREX16x1, 0x40);
  stdlcd_out(OPTREX16x1, "T CONTRL");


  SCI_init();

  pbus_init();

  gfxlcd_init();

  for (i=0;i<16;i++) {
    faders[i]=0;
    oldfaders[i]=2;
  }

  while (1) {
    get_faders (faders);

    for (n=0;n<16;n++) {
      i=faders[n] / 5;
      j=faders[n] / 2;

      if ((oldfaders[n]/5)!=(faders[n]/5)) {
gfxlcd_btmbar (n, i);
SCI_chout (0xB0);
SCI_chout (n);
SCI_chout (j);
      }

      oldfaders[n] = faders[n];

    }
  }
}
```

13.2.3 Interrupt Vector Table, `boardvect.c`

The following code sets up the interrupt vector table for the 68HC11 used in the control board. Currently, no interrupts are used.

```
extern void _start(); /* entry point in crt11.s */

#pragma abs_address:0xffd6

void (*interrupt_vectors[])() =
{
/* to cast a constant, say 0xb600, use
   (void (*)())0xb600
 */
/* all interrupts reset the processor, which is probably not what
 * we want forever.. */
_start,/* SCI */
_start,/* SPI */
_start,/* PAIE */
_start,/* PAO */
_start,/* TOF */
_start,/* TOC5 */
_start,/* TOC4 */
_start,/* TOC3 */
_start,/* TOC2 */
_start,/* TOC1 */
_start,/* TIC3 */
_start,/* TIC2 */
_start,/* TIC1 */
_start,/* RTI */
_start,/* IRQ */
_start,/* XIRQ */
_start,/* SWI */
_start,/* ILLOP */
_start,/* COP */
_start,/* CLM */
_start /* RESET */
};
#pragma end_abs_address
```

13.2.4 Graphics LCD Driver Header, `gfxlcd.h`

This header contains the function prototypes for the Toshiba dot matrix LCD driver code.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: gfxlcd.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v3.6, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
```

```
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************/

#ifndef _gfxlcd
#define _gfxlcd

void gfxlcd_writecmd (unsigned char d);
void gfxlcd_writedata (unsigned char d);
void gfxlcd_cleartext (void);
void gfxlcd_textxy (unsigned char x, unsigned char y);
void gfxlcd_textout (char *s);
void gfxlcd_init (void);
void gfxlcd_btmbar (unsigned char pos, unsigned char height);

#endif
```

13.2.5 Graphics LCD Driver Code, `gfxlcd.c`

```
/******************************************************************************
 * DACS : Distributed Audio Control System
 *============================================================================
 *         File: gfxlcd.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v3.6, 68HC11 target
 *        Build: library, not standalone
 *============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************/

#include <hc11.h>
#include "gfxlcd.h"
#include "lcddefn.h"

/******************************************************************************
 * gfxlcd_writecmd
 *
 * write a command byte to a graphics lcd
 ******************************************************************************/
void gfxlcd_writecmd (unsigned char d)
{
  while (!(GFXLCD_CMD & GFXLCD_RDY));
  GFXLCD_CMD=d;
}

/******************************************************************************
 * gfxlcd_writedata
 *
 * write a data byte to a graphics lcd
 ******************************************************************************/
```

```
void gfxlcd_writedata (unsigned char d)
{
  while (!(GFXLCD_CMD & GFXLCD_RDY));
  GFXLCD_DATA=d;
}


/*****************************************************************************
 * gfxlcd_cleartext
 *
 * clears the text area of a graphics lcd
 *****************************************************************************/
void gfxlcd_cleartext (void)
{
  int i;

  gfxlcd_writedata (0x00);
  gfxlcd_writedata (0x10);
  gfxlcd_writecmd (0x24);     /* text home address cmd */

  for (i=0;i<320;i++) {
    gfxlcd_writedata (0);
    gfxlcd_writecmd (0xC0);
  }
}


/*****************************************************************************
 * gfxlcd_textxy
 *
 * positions the cursor in the text area of a graphics lcd
 *****************************************************************************/
void gfxlcd_textxy (unsigned char x, unsigned char y)
{
  unsigned int i;
  unsigned char hi, lo;

  i=(y*40)+x;

  hi=((i&0xFF00)>>8);
  lo=(i&0x00FF);

  gfxlcd_writedata (lo);
  gfxlcd_writedata (0x10+hi);
  gfxlcd_writecmd (0x24);     /* text home address cmd */
}


/*****************************************************************************
 * gfxlcd_textout
 *
 * output a string of text to the text area of a graphics lcd
```

```
 ****************************************************************************/
void gfxlcd_textout (char *s)
{
  while (*s) {
    gfxlcd_writedata ((unsigned char) *s - 32);
    gfxlcd_writecmd (0xC0);
    s++;
  }
}

/****************************************************************************
 * gfxlcd_btmbar
 *
 * make a graphics-mode bar graph at the bottom of the display
 * (looks like a fader knob)
 ****************************************************************************/
void gfxlcd_btmbar (unsigned char pos, unsigned char height)
{
  unsigned char ystop,hi,lo;
  unsigned int y;
  unsigned int a;

  gfxlcd_writecmd (0x9C);
  ystop=53-height;


  for (y=53;y>0;y--) {
      a=(y*40)+pos;

      hi=((a&0xFF00)>>8);
      lo=(a&0x00FF);


      gfxlcd_writedata (lo);
      gfxlcd_writedata (hi);
      gfxlcd_writecmd (0x24);

      if ((y==ystop)||(y==ystop-1)) {
gfxlcd_writedata (0x1F);
      } else {
gfxlcd_writedata (0x04);
      }
      gfxlcd_writecmd (0xC0);
  }

}

/****************************************************************************
 * gfxlcd_init
 *
 * initialize the Toshiba TLX-711A graphics LCD
 ****************************************************************************/
```

```
void gfxlcd_init (void)
{
  int i;

  gfxlcd_writecmd (0x80);  /* OR mode */

  gfxlcd_writedata (0x00);
  gfxlcd_writedata (0x00);
  gfxlcd_writecmd (0x42);

  gfxlcd_writedata (0x28);
  gfxlcd_writedata (0x00);
  gfxlcd_writecmd (0x43);

  gfxlcd_writedata (0x00);
  gfxlcd_writedata (0x10);  /* text home address set */
  gfxlcd_writecmd (0x40);

  gfxlcd_writedata (0x28);
  gfxlcd_writedata (0x00);  /* number of text areas set */
  gfxlcd_writecmd (0x41);

  gfxlcd_writecmd (0x9C);

  gfxlcd_writedata (0x00);
  gfxlcd_writedata (0x00);  /* graphic home address set */
  gfxlcd_writecmd (0x42);

  gfxlcd_writedata (0x00);
  gfxlcd_writedata (0x00);  /* address pointer set */
  gfxlcd_writecmd (0x24);

  for (i=0;i<2560;i++) {
    gfxlcd_writedata (0x00);
    gfxlcd_writecmd (0xC0);  /* write data */
  }


  gfxlcd_cleartext();

  gfxlcd_textxy (25,0);
  gfxlcd_textout ("DACS model 112");
  gfxlcd_textxy (0,7);
  gfxlcd_textout ("0123456789ABCDEF");
  gfxlcd_writecmd (0x9C);
}
```

13.2.6 Alphanumeric and Graphics LCD Driver Defines, `lcddefn.h`

The following header file defines ports and other important information for the alphanumeric and graphics LCDs present in the control board.

```
/*****************************************************************************
```

```
 * DACS : Distributed Audio Control System
 *=============================================================================
 *         File: lcddefn.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v3.6, 68HC11 target
 *        Build: library, not standalone
 *=============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef _lcddefn
#define _lcddefn

#define GFXLCD_CMD   *(unsigned char *)(0xB591)
#define GFXLCD_DATA  *(unsigned char *)(0xB590)

#define GFXLCD_RDY   0x03

#define STDLCD_CMD    *(unsigned char *)(0xB5F0) /* cmd port of LCD (base) */
#define STDLCD_DATA   *(unsigned char *)(0xB5F1) /* data port of LCD (base) */
#define STDLCD_OFFSET  0x02      /* multiplier for multiple LCDs */

/* the Optrex 16x1 is pretty weird - you tell the HD44780 that it's a
 * two-line display.. the first 8 chars are in the normal place, while the
 * last 8 start at location 40..
 */
#define OPTREX16x1   0
#define OPTREX16x1_INIT {0x3C,0x0C,0x06,0x01,-1}

#endif
```

13.2.7 pbus Driver Defines, pbusdefn.h

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *=============================================================================
 *         File: pbusdefn.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v3.6, 68HC11 target
 *        Build: library, not standalone
 *=============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef _pbusdefn
#define _pbusdefn
```

```
#define PBC_PA *(unsigned char *)(0xB580)  /* pBUS controller port A */
#define PBC_PB *(unsigned char *)(0xB581)  /* pBUS controller port B */
#define PBC_PC *(unsigned char *)(0xB582)  /* pBUS controller port C */
#define PBC_PX *(unsigned char *)(0xB583)  /* pBUS controller ctrl port */

#define PBC_IX 0x98   /* pBUS ctrl word: PA i / PB o / PC0-3 o / PC4-7 i */
#define PBC_OX 0x88   /* pBUS ctrl word: PA o / PB o / PC0-3 o / PC4-7 i */

#define PB_N_LATCH  (unsigned char) 0x01    /* bitmask for latch- line */
#define PB_N_CLOCK  (unsigned char) 0x02    /* bitmask for clock- line */
#define PB_N_RD_WR  (unsigned char) 0x80    /* bitmask for read-/write line */

/*****************************************************************************
 * pBUS device addresses
 *****************************************************************************/

#define PBADX_FADEMUX0  0x08
#define PBADX_FADEMUX1  0x09

#define PBADX_FADELED0  0x10
#define PBADX_FADELED1  0x11
#define PBADX_FADELED2  0x12
#define PBADX_FADELED3  0x13

#endif
```

13.3 Mixer Unit

13.3.1 Firmware Build, `Makefile`

The following GNU `Makefile` is used to build the firmware image for the mixer unit. A standard Motorola S19 file is produced when the files are built. This image can be downloaded to the EEPROM on the embedded microcontroller board using special software.

```
#############################################################################
# DACS : Distributed Audio Control System
# Model 411 Modular Automated Audio Mixer / Firmware Build File
#
# The code, executables, documentation, firmware images, and all related
# material of DACS are
# Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
#############################################################################

CC = icc11
CFLAGS =
PROGS = links mixer
MAINDIR = .
S19DIR = $(MAINDIR)/s19
OBJDIR = $(MAINDIR)/obj
DACSLIB = ../dacslib

all: $(PROGS)

#
```

```
# set up links to DACSlib files, since icc11 3.6 can't do what i want..
# remember to change this if 'local' copies of the lib funcs are used
# (so they don't get overwritten)
# also change the make clean stuff if locals are used.
#
links:
rm -f SCIserial.h SCIserial.c pbus.h pbus.c stdlcd.h
rm -f stdlcd.c ACIAserial.c ACIAserial.h
ln -s $(DACSLIB)/SCIserial.c SCIserial.c
ln -s $(DACSLIB)/SCIserial.h SCIserial.h
ln -s $(DACSLIB)/ACIAserial.c ACIAserial.c
ln -s $(DACSLIB)/ACIAserial.h ACIAserial.h
ln -s $(DACSLIB)/pbus.c pbus.c
ln -s $(DACSLIB)/pbus.h pbus.h
ln -s $(DACSLIB)/stdlcd.c stdlcd.c
ln -s $(DACSLIB)/stdlcd.h stdlcd.h

buildnum:
./buildnum.pl

mixer: links buildnum
$(CC) $(CFLAGS) -omixer.s19 *.c -D_MIXER
cp mixer.s19 /tmp
chmod a+r /tmp/mixer.s19
mv mixer.s19 $(S19DIR)
mv *.o $(OBJDIR)

clean:
rm -f mixer.s19 *.o *~ $(S19DIR)/mixer.s19 $(OBJDIR)/*.o
rm -f SCIserial.h SCIserial.c pbus.h pbus.c
```

13.3.2 Mixer Unit Main Code, `mixmain.c`

The following is the main code body for the mixer unit firmware. At present, it is a test bed for exercising the serial routines that were being developed at the time of this writing. A routine called `oldmain()` contains older code that was used to implement very basic functionality of the mixer unit. This code receives MIDI controller data on the serial port and adjusts mixer registers accordingly. Much of the designed firmware functionality is currently unimplemented.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: mixmain.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
```

```
 *
 * $Id: mixmain.c,v 1.1 1997/07/13 01:22:04 prefect Exp prefect $
 *
 ***************************************************************************/

#include <hc11.h>
#include "ACIAserial.h"
#include "pbus.h"
#include "pbusdefn.h"
#include "mixdrv.h"
#include "stdlcd.h"
#include "lcddefn.h"

struct mixer_unit_info unit;

void main (void)
{
  struct raw_frame in_raw;
  struct cooked_frame out_cooked;
  int n;

  ACIA_prep_raw (&in_raw);
  ACIA_prep_cooked (&out_cooked);

  mix_system_init();             /* initialize the mixer system */

  ACIA_data_to_cooked ("First", 5, &out_cooked);

  while (1) {
    ACIA_service (&in_raw, &out_cooked);

    if (in_raw.done==1) {

      stdlcd_clrhome(HITACHI40x2);   /* clear the LCD */
      stdlcd_out(HITACHI40x2, (char *) in_raw.data);

      ACIA_prep_raw (&in_raw);
    } else if (in_raw.done == ERRTOOBIG) {
      stdlcd_clrhome(HITACHI40x2);   /* clear the LCD */
      stdlcd_out(HITACHI40x2, "** Frame too large **");
      ACIA_prep_raw (&in_raw);
    }

    if ((out_cooked.done==1) && (n++>1000)) {
      n=0;
      ACIA_data_to_cooked ("Incoming...", 11, &out_cooked);
    }
  }
}


void oldmain (void)
```

```
{
  unsigned char ch, r, i, j;
  unsigned char chan, cmd;
  unsigned char dbytes;
  unsigned char brd, chip, minp, lev;
  unsigned char chipmasks[]={CHIP0,CHIP1,CHIP2,CHIP3};
  unsigned char data[10];
  unsigned char *dptr;
  int ticks,bytes;
  unsigned char spin[]={0xA5,'*','O','*'};
  unsigned char spincount;

  mix_system_init();      /* initialize the mixer system */
  stdlcd_clrhome(HITACHI40x2);   /* clear the LCD */
  /*                          12345678901234567890123456789012345678901234567890*/
  stdlcd_out (HITACHI40x2, "run [ ] load [                          ]");
  stdlcd_goto (HITACHI40x2, 0x40);
  stdlcd_out (HITACHI40x2, "mode [8x16 ] mixmodules [ 4] outs [u][b]");

  ticks=0;
  bytes=0;
  cmd=0;
  dbytes=0;
  spincount=0;

  while (1) {
    if (ACIA_poll_in (&ch)) {
      /* we got a character */

      PBC_PC^=0x04;  /* green link LED on */

      if (ch&0x80) {
/* MIDI command -- MSB is set */

if (ch == 0xFF) {
  /* reset the mixer to a known state */
  PBC_PC&=~0x04; /* green link LED off */
  PBC_PC^=0x08;  /* red link LED on */

  for (i=0;i<4;i++) {
    for (j=0;j<8;j++) {
      pbus_mwrite (PBADX_MIX0+i, 0x98+j, 0x98+j, 0x98+j, 0x98+j, CHIPALL);
      pbus_mwrite (PBADX_MIX0+i, 63, 63, 63, 63, CHIPALL);
    }
  }
  PBC_PC&=~0x08; /* red link LED off */
} else {
  /* set stuff for new command + data */
  cmd = ch&0xF0;  /* upper nybble is command */
  chan = ch&0x0F; /* lower nybble is channel */
}
```

```
dptr=data;
dbytes=0;
bytes++;  /* increase bytecount for 'load' graph */
ticks++;
      } else {
*dptr=ch;
dptr++;
dbytes++;
bytes++;  /* increase bytecount for 'load' graph */
ticks++;

if ( (cmd == 0xB0) && (dbytes == 2) ) {
  /* controller change - 1st datum=ctrl# 2nd datum=value */


  /* "address decoding" for figuring out where to write */

  brd = (data[0]&0x60)>>5;          /* mix board # */
  chip = (data[0]&0x18)>>3;         /* chip on the board */
  minp = (data[0]&0x07)+0x98;       /* input on chip + SSM2163 ofs */

  lev = 0x7F - ((data[1]/2)&0x7F);  /* attenuation level */


  /* write the address */
  pbus_mwrite (0x10+brd, minp, minp, minp, minp, chipmasks[chip]);
  /* write the level data */
  pbus_mwrite (0x10+brd, lev, lev, lev, lev, chipmasks[chip]);
}
      }
    } else {

      ticks++;

      if (ticks>=8192) {
/* update LCD and such, enough ticks have passed */

stdlcd_goto (HITACHI40x2, 5);
stdlcd_writechar (HITACHI40x2, spin[spincount]);

if (spincount<3) spincount++;
else {
  spincount=0;
}


/*stdlcd_bar (HITACHI40x2, 14, bytes/2, 25);*/

bytes=0;  /* zero byte counter */
ticks=0;  /* zero ticks counter */
      }
      PBC_PC&=~0x04;  /* green link LED off */
```

```
    }
  }
}
```

13.3.3 Interrupt Vector Table, `mixvect.c`

The following code segment is the interrupt vector table for the mixer unit. No interrupts are presently implemented in the firmware.

```
extern void _start(); /* entry point in crt11.s */

#pragma abs_address:0xffd6

void (*interrupt_vectors[])() =
{
/* to cast a constant, say 0xb600, use
   (void (*)())0xb600
 */
/* all interrupts reset the processor */
_start,/* SCI */
_start,/* SPI */
_start,/* PAIE */
_start,/* PAO */
_start,/* TOF */
_start,/* TOC5 */
_start,/* TOC4 */
_start,/* TOC3 */
_start,/* TOC2 */
_start,/* TOC1 */
_start,/* TIC3 */
_start,/* TIC2 */
_start,/* TIC1 */
_start,/* RTI */
_start,/* IRQ */
_start,/* XIRQ */
_start,/* SWI */
_start,/* ILLOP */
_start,/* COP */
_start,/* CLM */
_start /* RESET */
};
#pragma end_abs_address
```

13.3.4 Mixer Driver Header, `mixdrv.h`

The following header defines many values for mixer unit configuration, defines configuration structures, and provides function prototypes.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: mixdrv.h
 *       Author: Stephen S. Richardson
```

```
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ****************************************************************************
 * Source code control:
 *
 * $Id: mixdrv.h,v 1.1 1997/07/13 01:22:23 prefect Exp prefect $
 *
 ****************************************************************************/

#ifndef _mixdrv
#define _mixdrv

/* masks for writing to the SSM2163's on the mixer module cards */
#define CHIP0    0xA8
#define CHIP1    0xA2
#define CHIP2    0x8A
#define CHIP3    0x2A
#define CHIPALL  0x00


#define MAXMIXBRDS 16
#define MAXINPBRDS 4
#define MAXOUTBRDS 2
#define MAXOPTBRDS 2

#define MIX_2163x4   1   /* ID byte for stock 4xSSM2163 mix module card */
#define MIX_2163x4V  2   /* ID byte for stock 4xSSM2163 with trim ctrl */

#define IN_ANALOG8   1   /* ID byte for 8-input analog input card */
#define IN_DIGITAL8  2   /* ID byte for 8-input digital input card (TBD) */

#define OUT_ANALOG8  1   /* ID byte for 8-output analog card */
#define OUT_DIGITAL8 2   /* ID byte for 8-output digital output card (TBD) */

#define OPT_BUSCOMB  1   /* ID byte for input/output bus combiner */
#define OPT_VU       2   /* ID byte for digital VU meter board */

#define NO_BRD       0   /* NO board installed */

struct mixer_unit_info {
  unsigned char id;         /* unit ID of this DACS mixer (from host) */
  unsigned char serno[12];  /* serial number of this mixer (factory) */
  unsigned char ver[30];    /* version number of firmware (compile) */

  unsigned char mix_brd[MAXMIXBRDS];  /* ID bytes for mixer boards */
  unsigned char mix_brds;             /* number of mixer boards installed */
  unsigned char inp_brd[MAXINPBRDS];  /* ID bytes for input boards */
```

```
  unsigned char inp_brds;             /* number of input boards installed */
  unsigned char out_brd[MAXOUTBRDS];  /* ID bytes for output boards */
  unsigned char out_brds;             /* number of output boards installed */
  unsigned char opt_brd[MAXOPTBRDS];  /* ID bytes for option boards */
  unsigned char opt_brds;             /* number of option boards installed */

  unsigned char hostname[17];   /* name of host unit is connected to */
};

extern struct mixer_unit_info unit;

void mix_get_system_config (void);
void mix_system_init (void);
void mix_reset_mixers (void);
void mix_reset_buscomb (void);
void mix_reset_vu (void);

void pbus_mwrite (unsigned char addx, unsigned char data0,
  unsigned char data1, unsigned char data2,
  unsigned char data3, unsigned char mask);



#endif
```

13.3.5 Mixer Driver Code, `mixdrv.c`

The following code provides the necessary underlying functionality to access the hardware of the mixer unit.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: mixdrv.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: mixdrv.c,v 1.1 1997/07/13 01:22:41 prefect Exp prefect $
 *
 *****************************************************************************/

#include <hc11.h>
#include "pbusdefn.h"
#include "mixdrv.h"
#include "build.h"
```

```
#include "lcddefn.h"
#include "stdlcd.h"
#include "serno.h"
#include "romtext.h"
#include "ACIAserial.h"


/*****************************************************************************
 * mix_get_system_config
 *
 * dynamically acquire the system configuration of the mixer unit by polling
 * for cards
 *
 * TODO: make this dynamic and not hardcoded.
 *****************************************************************************/
void mix_get_system_config (void)
{
  int i;

  unit.id=0;                          /* this needs to be set by the host */
  strcpy (unit.serno, UNIT_SERNO);
  strcpy (unit.ver, build_info);

  unit.mix_brd[0]=MIX_2163x4;
  unit.mix_brd[1]=MIX_2163x4;
  unit.mix_brd[2]=MIX_2163x4;
  unit.mix_brd[3]=MIX_2163x4;
  for (i=4;i<MAXMIXBRDS;i++) unit.mix_brd[i]=NO_BRD;
  unit.mix_brds=4;


  unit.inp_brd[0]=IN_ANALOG8;
  unit.inp_brd[1]=IN_ANALOG8;
  unit.inp_brd[2]=IN_ANALOG8;
  unit.inp_brd[3]=IN_ANALOG8;
  unit.inp_brds=4;

  unit.out_brd[0]=OUT_ANALOG8;
  unit.out_brd[1]=OUT_ANALOG8;
  unit.out_brds=2;

  unit.opt_brd[0]=OPT_BUSCOMB;
  unit.opt_brd[1]=NO_BRD;
  unit.opt_brds=1;
}


/*****************************************************************************
 * mix_system_init
 *
 * initialize the mixer hardware
 *****************************************************************************/
```

```
void mix_system_init (void)
{
  int i,d, brd;
  int hitachi_init[]=HITACHI40x2_INIT;
  unsigned char module,ch;

  stdlcd_init(HITACHI40x2, hitachi_init);           /* init 40x2 LCD */
  stdlcd_out(HITACHI40x2, (char *) lcdmsg_intro);   /* put up intro */

  ACIA_init();    /* initialize the ACIA subsystem */
  pbus_init();   /* initialize pBUS controller */

  PBC_PC^=0x08;

  mix_get_system_config();   /* poll the hardware and obtain config */

  mix_reset_mixers();        /* reset all mixers to max. attenuation */

  /* reset option boards */
  for (i=0;i<MAXOPTBRDS;i++) {
    switch (unit.opt_brd[i]) {
    case OPT_BUSCOMB:
      mix_reset_buscomb();       /* reset the bus combiner board */
      break;
    case OPT_VU:
      mix_reset_vu();            /* reset digital vu board */
      break;
    }
  }

  PBC_PC&=~0x08;

  /* wait around, and make the link LED green/black alternating */
  for (d=0;d<40000;d++) {
    PBC_PC^=0x04;
    PBC_PC&=~0x04;
  }

  stdlcd_clrhome(HITACHI40x2);                       /* clear/home the LCD */
}


/*****************************************************************************
 * mix_reset_mixers
 *
 * resets all of the mixer chips to maximum attenuation
 *****************************************************************************/
void mix_reset_mixers (void)
{
  int module, ch, p;

  /* turns on all mixer channels to maximum attenuation */
```

```
  for (module=0;module<4;module++) {
    for (ch=0;ch<8;ch++) {
      p=0x98+ch;
      pbus_mwrite (PBADX_MIX0+module, p, p, p, p, CHIPALL);
      pbus_mwrite (PBADX_MIX0+module, 63, 63, 63, 63, CHIPALL);
    }
  }
}


/*****************************************************************************
 * mix_reset_buscomb
 *
 * resets the bus combiner module
 *****************************************************************************/
void mix_reset_buscomb (void)
{
  pbus_lwrite (PBADX_BCOMB,0x04+0x08+0x10);  /* turn off relays */
}

/*****************************************************************************
 * mix_reset_vu
 *
 * resets digital vu module
 *****************************************************************************/
void mix_reset_vu (void)
{
  /* not currently implemented in hw */
}


/*****************************************************************************
 * pbus_mwrite
 *
 * writes to a DACS mixer module containing 4 SSM2163 mixer chips.
 *
 * four separate bytes are blasted in a parallel bitbanged-serial scheme.
 *
 *****************************************************************************/
void pbus_mwrite (unsigned char addx, unsigned char data0,
  unsigned char data1, unsigned char data2,
  unsigned char data3, unsigned char mask)
{
  unsigned char i;
  unsigned char dslice;
  unsigned char bitmask_msb[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};

  PBC_PA=0xAA;                          /* mix chips ld-/wr- inact */
  PBC_PC|=PB_N_LATCH|PB_N_CLOCK;        /* pBUS latch- and clock- inactive */

  PBC_PB=addx|PB_N_RD_WR;               /* addx on pBUS, write mode */
```

```
  PBC_PC&=~PB_N_LATCH;                  /* pBUS latch- active */

  for (i=0;i<8;i++) {
    dslice = (((data0&bitmask_msb[i])<<i)>>7)|
             (((data1&bitmask_msb[i])<<i)>>5)|
             (((data2&bitmask_msb[i])<<i)>>3)|
             (((data3&bitmask_msb[i])<<i)>>1);

    dslice |= mask;

    PBC_PC&=~PB_N_CLOCK;                  /* pBUS clock- active */
    PBC_PA=dslice;                        /* data slice on pBUS */
    PBC_PC|=PB_N_CLOCK;                   /* pBUS clock- inactive */
  }

  PBC_PA=0xAA;                           /* mix chips ld-/wr- inact */
  PBC_PC|=PB_N_LATCH;                    /* pBUS latch- inactive */
}
```

13.3.6 pbus Driver Defines, pbusdefn.h

The following header defines the locations of the Pbus controller and various Pbus addresses.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: pbusdefn.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef _pbusdefn
#define _pbusdefn

#define PBC_PA *(unsigned char *)(0xB580)  /* pBUS controller port A */
#define PBC_PB *(unsigned char *)(0xB581)  /* pBUS controller port B */
#define PBC_PC *(unsigned char *)(0xB582)  /* pBUS controller port C */
#define PBC_PX *(unsigned char *)(0xB583)  /* pBUS controller ctrl port */

#define PBC_IX 0x98   /* pBUS ctrl word: PA i / PB o / PC0-3 o / PC4-7 i */
#define PBC_OX 0x88   /* pBUS ctrl word: PA o / PB o / PC0-3 o / PC4-7 i */

#define PB_N_LATCH  (unsigned char) 0x01    /* bitmask for latch- line */
#define PB_N_CLOCK  (unsigned char) 0x02    /* bitmask for clock- line */
#define PB_N_RD_WR  (unsigned char) 0x80    /* bitmask for read-/write line */

/*****************************************************************************
```

```
 * pBUS device addresses
 ******************************************************************************/

#define PBADX_MIX0  0x10    /* audio mix module v1.0 addresses */
#define PBADX_MIX1  0x11
#define PBADX_MIX2  0x12
#define PBADX_MIX3  0x13    /* prototype hardware ends HERE */
#define PBADX_MIX4  0x14
#define PBADX_MIX5  0x15
#define PBADX_MIX6  0x16
#define PBADX_MIX7  0x17
#define PBADX_MIX8  0x18
#define PBADX_MIX9  0x19
#define PBADX_MIX10 0x1A
#define PBADX_MIX11 0x1B
#define PBADX_MIX12 0x1C
#define PBADX_MIX13 0x1D
#define PBADX_MIX14 0x1E
#define PBADX_MIX15 0x1F    /* these are spec'ed, included for completeness */

#define PBADX_INP0  0x08    /* audio input trim control v1.0 addresses */
#define PBADX_INP1  0x09
#define PBADX_INP2  0x0A
#define PBADX_INP3  0x0B

#define PBADX_BCOMB 0x30    /* bus combiner */

#endif
```

13.3.7 Alphanumeric LCD Driver Defines, `lcddefn.h`

The following header defines specifics for the alphanumeric LCD on the mixer unit.

```
/******************************************************************************
 * DACS : Distributed Audio Control System
 *=============================================================================
 *         File: lcddefn.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.14.97
 *  Environment: ICC11 v4.0, 68HC11 target
 *        Build: library, not standalone
 *=============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ******************************************************************************/

#ifndef _lcddefn
#define _lcddefn

#define STDLCD_CMD    *(unsigned char *)(0xB5F0) /* command port of LCD (base) */
#define STDLCD_DATA   *(unsigned char *)(0xB5F1) /* data port of LCD (base) */
```

```
#define STDLCD_OFFSET  0x02      /* multiplier for multiple LCDs */

#define HITACHI40x2  0
#define HITACHI40x2_INIT {0x3C,0x0C,0x06,0x01,-1}
#endif
```

14 High-Level Software Design

14.1 Overall System: The Distributed Concept

Thus far, very little of this design document has stressed the concept of distributed audio control. This is because much of what makes DACS distributed is implemented in the high-level system software.

DACS takes advantage of off-the-shelf PCs and peripheral hardware. These days, powerful PCs with network cards, CD-ROMs, and sound cards are commonplace. DACS can use any number of networked PCs to scale to almost any conceivable audio application. For example, a network of six PCs could be used in an audio application requiring a dozen audio CDs, six digital audio sound cards, four separate MIDI buses, two DACS mixer units, and a control board. While this is something of an extreme case, it merely serves to illustrate the possibilities.

To make DACS services truly distributed, a client/server approach is used. TCP/IP, a widely available network protocol, provides a reliable way to communicate over a wide variety of computer networks. The base model for DACS is communication over standard 10Mbit ethernet. This does not preclude the use of 100Mbit ethernet, ATM, or other physical media. TCP/IP even works on a single, non-networked host. This makes implementation easy, and allows the distributed paradigm to be scaled down to a single PC application.

Figure 107 shows a system view of the DACS software, with emphasis on the distribution of services. Note that where TCP/IP is seen as a communications means, the software components may be run on separate machines.

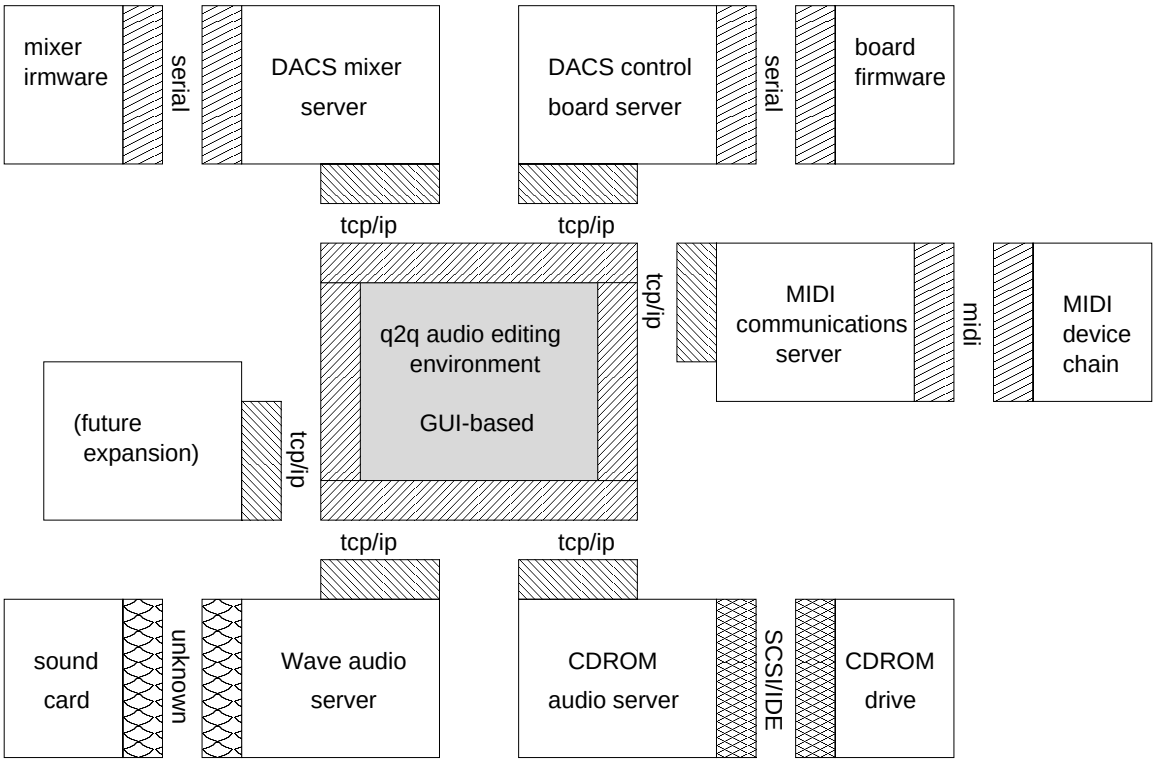


Figure 107: System view of DACS software components.

The central application, dubbed `q2q`, acts as a master to all of the various DACS components, wherever they are located in the DACS configuration. This GUI-based program allows complex audio scripts to be built using an intuitive interface. Editing and creation of these scripts may

also take place by using the control board standalone, or as a supplement to the GUI.

Audio scripts can be of two overall types, time-cued or user-cued. Time-cued scripts tend to be used in situations where SMPTE synchronization is desirable, such as in broadcast audio or studio recording. User-cued scripts tend to be used in situations where manually starting each cue is necessary, such as in a theatre setting. Manual cueing does not preclude the possibility of having time-subcued audio events.

Various DACS service providers allow the `q2q` software to access DACS subsystems. Currently, service providers for the mixer unit, control board, generic MIDI interfaces, generic PC soundcards, and generic PC CDROM drives are envisioned. Abstracting the underlying hardware from the higher-level software provides the distinct advantage of compatibility and simplicity from the top down. As new hardware with a similar functionality set comes along (such as a PC-based MiniDisc drive, for example), upper-level software does not need to change much to accomodate this new device. A generic class of transport-style devices can be created, all of which are accessed similarly from the `q2q` software.

Distributing the services among several programs also reduces the problem with `q2q` becoming an excessively monolithic program. This reduces memory consumption, size of the program, and allows a much more scalable system to be built.

14.2 Functional Design

14.2.1 Service Providers

Figure 108 depicts the functional flow of a typical DACS service provider. The exact nature of a provider depends on the specific application, but the overall structure is the same for all providers.

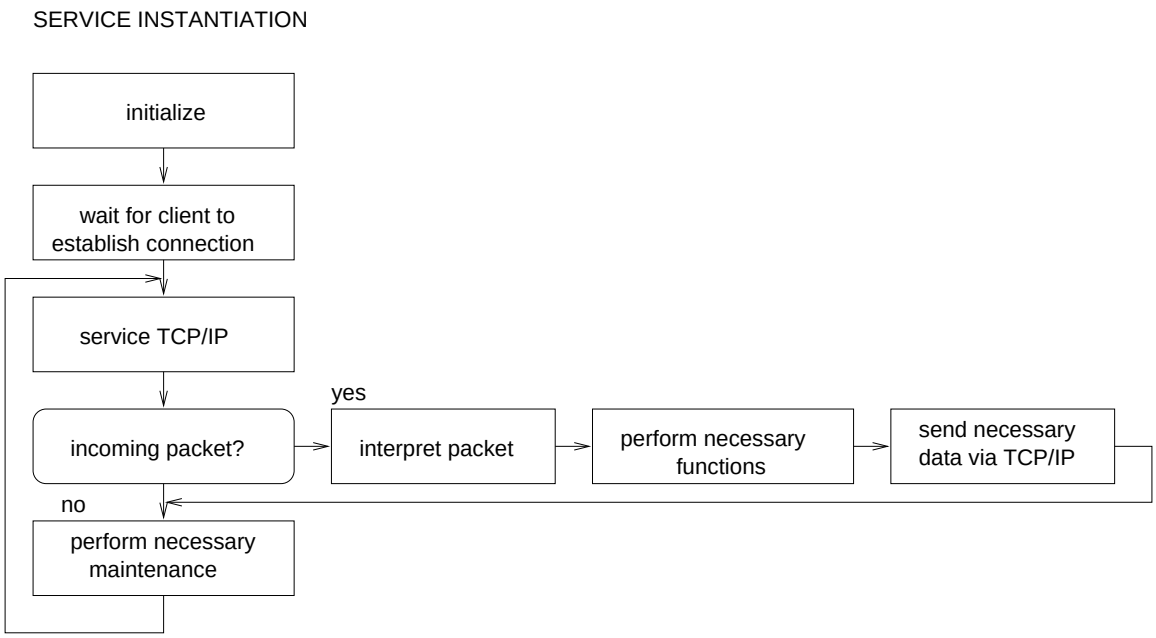


Figure 108: Functional flow diagram of a typical DACS service provider.

When the service is instantiated, it initializes itself and establishes itself as a TCP/IP server. Upon proper contact from the client, `q2q`, the program enters a loop, waiting for commands from the client and servicing requests. This loop continues until the client drops the connection. When the connection is dropped, service providers should return to the state in which they wait for a

connection from a client. All running functions should stop and return to a default state if this happens.

14.2.2 Main Application

When the `q2q` program is started, it initializes, and then immediately attempts to contact all service providers. Upon completion of this, it enters one of two states, selected by the user. Edit mode allows interactive editing of audio scripts, and they are executed in run mode. Figure 109 shows this top level of program flow.

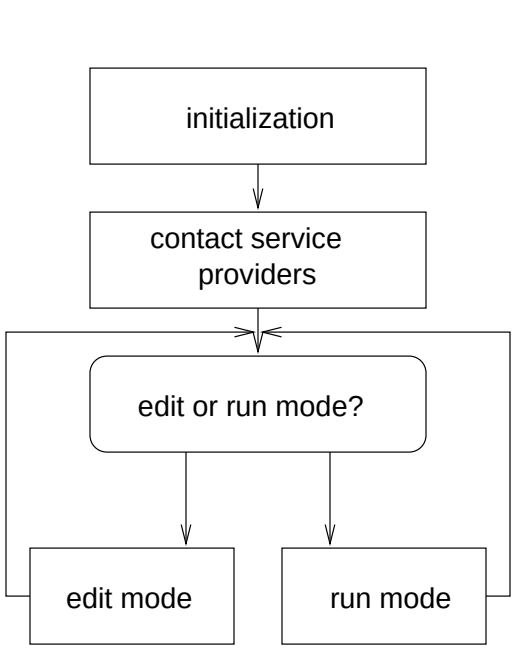


Figure 109: Functional flow diagram of `q2q` at the top level.

In the run level, the program waits for cue triggers, be they manual or timed. These triggers come from DACS service providers, such as the control board service, the MIDI service (for SMPTE clock over MIDI), etc. If it finds that a trigger has occurred, the appropriate commands for that audio script are dispatched to service providers. This loop continues until the user chooses to cancel, or the script completes. Figure 110 shows this flow diagram.

In the edit mode, the program allows the audio script to be edited. This editing can take place via the GUI or through the control board. The details of this editing are not covered in-depth here. A simple loop in which the editing takes place is run. This loop continues until the user enters the run mode. Figure 111 shows this flow diagram.

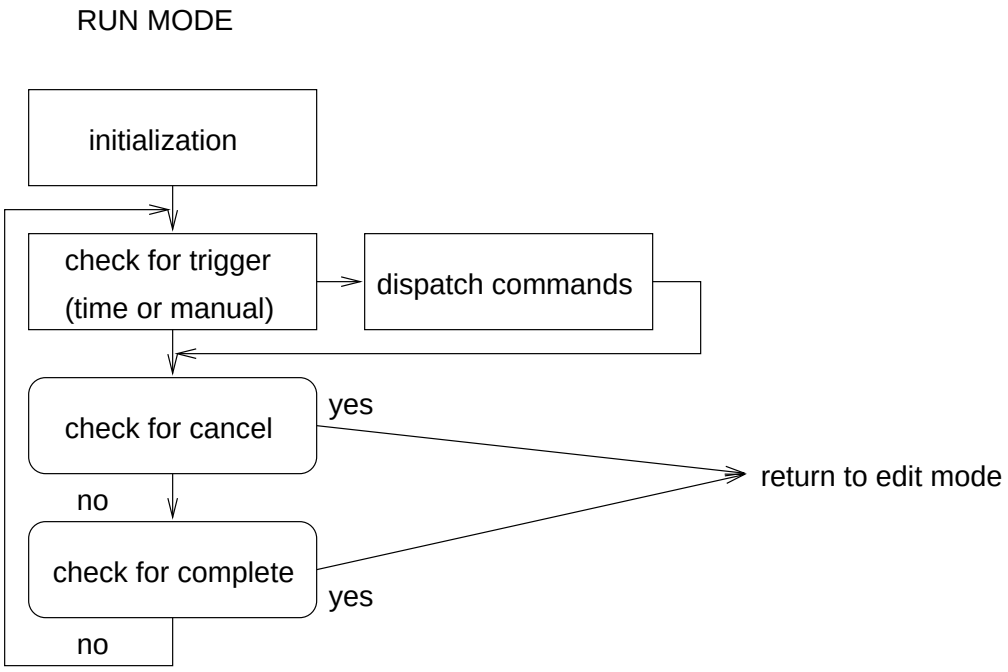


Figure 110: Functional flow diagram of `q2q` in the run mode.

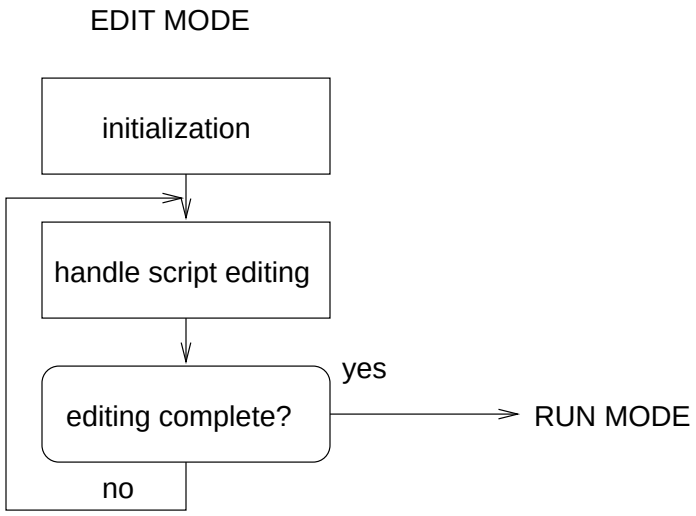


Figure 111: Functional flow diagram of `q2q` in the edit mode.

15 Software Graphical User Interface

The preliminary version of `q2q` is being built using the XForms library under a Linux/X Window environment. This application is far from complete, but some screen shots of the forms under development can help give a feel for what the application will look like.

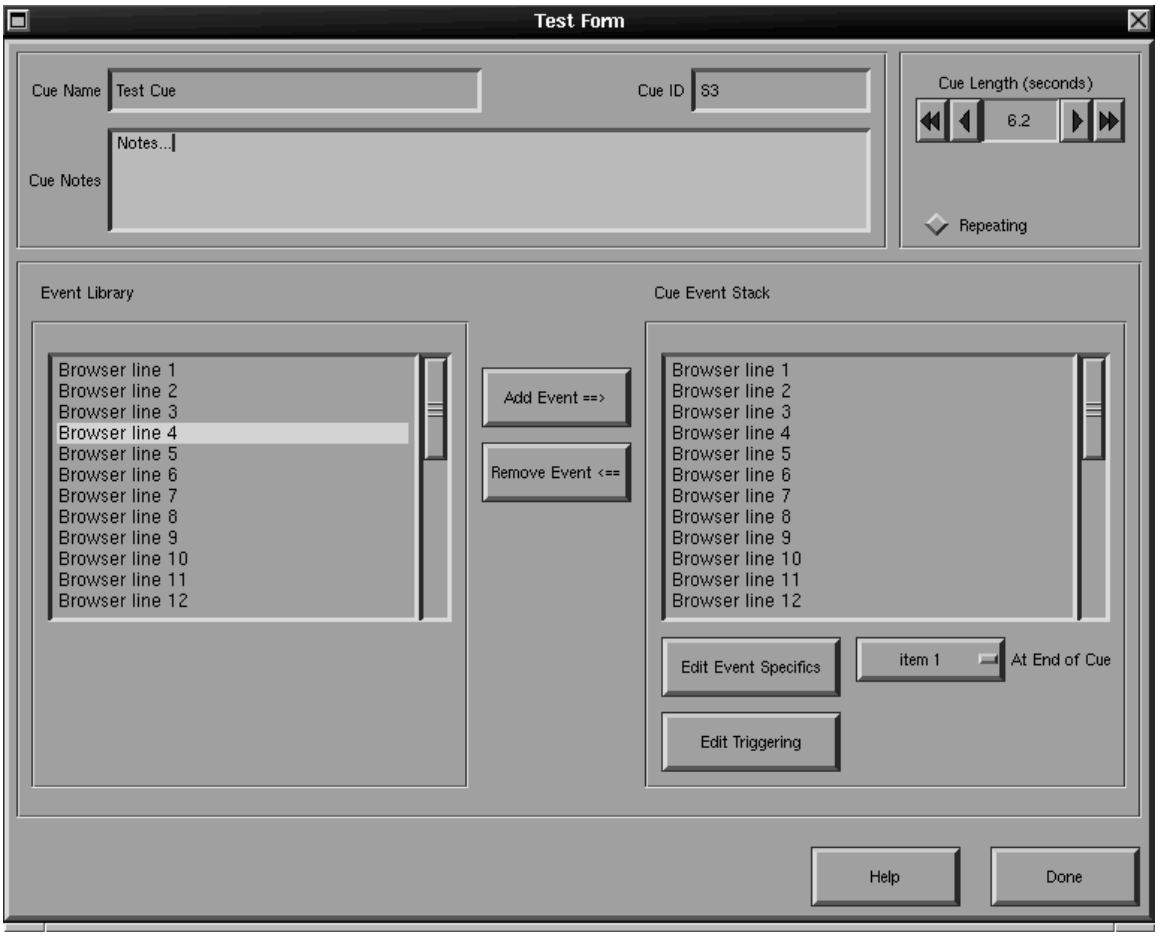


Figure 112: Preliminary form for "edit cue" function.

Figure 112 shows the form for the cue editing function. Controls are available to control the length of the cue, and whether or not the cue repeats. Audio events from the internal event library can be added and cued at different time intervals, adjustable by the user.

Figure 113 is the form used for editing event triggering options. Time delays and triggering after the completion of other events are possible.

Figure 114 shows the form used to build a performance stack. Cues previously defined are added to a chronological list at the right. This list, or performance stack, can be set up to be triggered manually or via time code.

While these forms are a far cry from a working application, they are a definite start in the right direction. Given more time, the `q2q` application will be built, and will likely look something like what these forms present.

Plans include later producing a Windows 32-bit version of `q2q`. This would have the distinct advantage of tying in well with existing Windows audio applications.

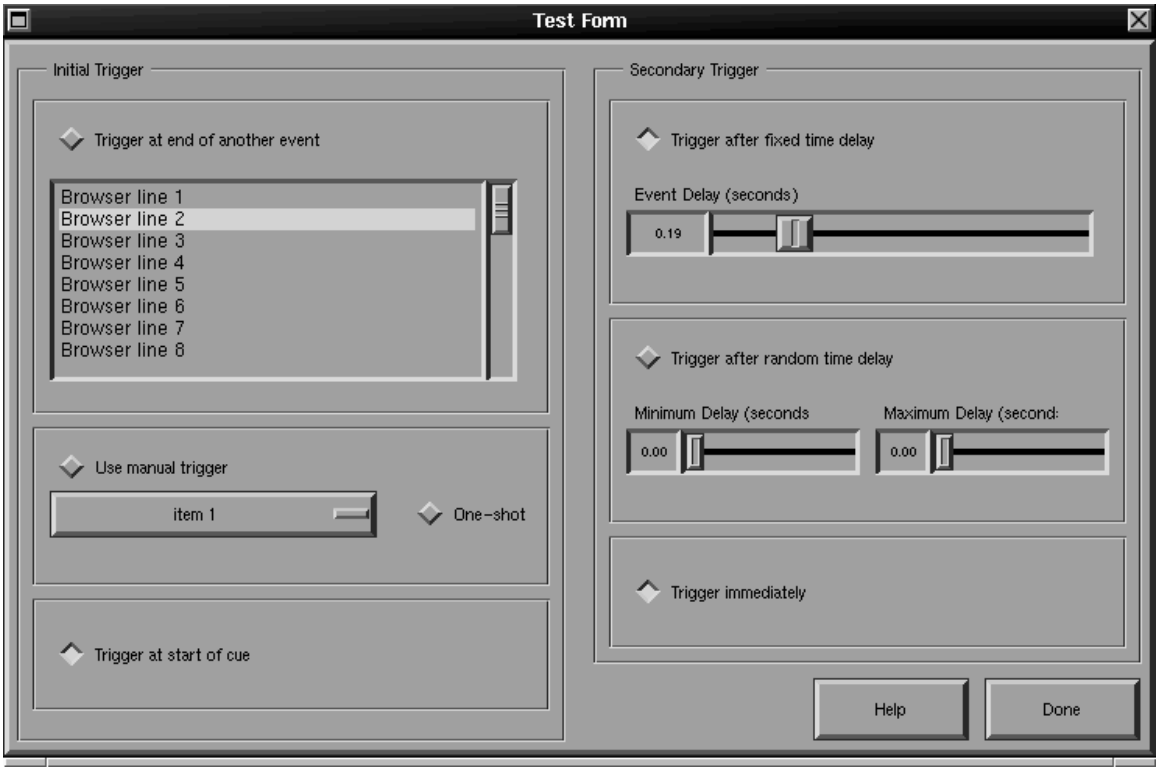


Figure 113: Preliminary form for "edit event trigger" function.

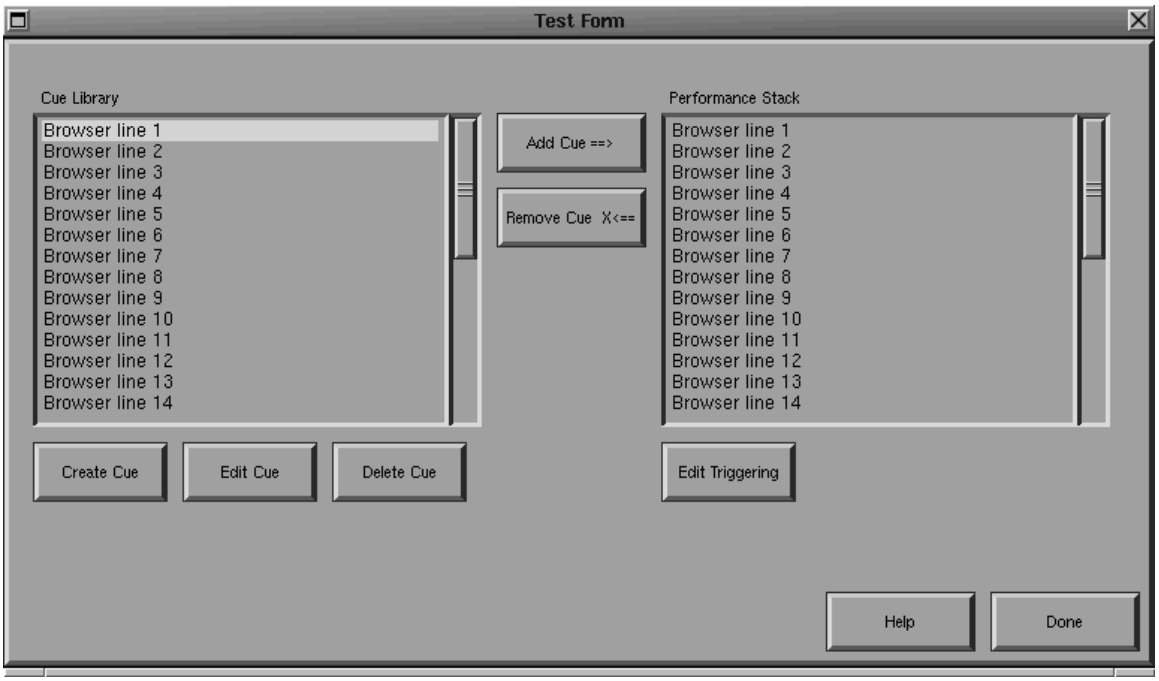


Figure 114: Preliminary form for "build performance stack" function.

16 Software Code

This section presents all of the code presently written to implement pieces of the software design. It is by no means complete, but it does provide a functional test-bed for a simple mode of operation. All code is written in ANSI C, and is intended to be compiled in a Linux environment using GCC 2.7.1 or above.

16.1 MIDI Controller

This section lists the Linux code used to implement a simple MIDI translator for the functions of DACS. It was mainly written to get a demo working for presentation day.

16.1.1 Main Program Code, `midictrl.c`

This code is the main MIDI control program. It simply exchanges MIDI-like information over serial ports between the DACS control board, mixer unit, and a separate host PC running a MIDI sequencing package and a special serial port MIDI driver.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: midictrl.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.18.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

/*
 * most of this code is a pretty ugly hack.. it was written hastily to get
 * a demo working for presentation day.
 *
 * implements "dumb" mode of control board, mixer unit, cdrom
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netdb.h>
#include "midi.h"
#include "mixer.h"
#include "client.h"
```

```
#include "cdaudio_comm.h"

void oops (char *mesg)
{
  perror (mesg);
  exit (1);
}

void main (void)
{
  struct midi_stream board, mixer, midi;
  struct mix_control mctrl;
  unsigned char ch, obuf[10];
  int r;
  int sock_cd;
  char buf_cd[CD_TCP_BUFSIZE];
  struct cdtype *cd;

  printf ("DACS : Distributed Audio Control System\nCopyright (C) 1997 Stephen S. Richardson / SR E

  /* connect with CD audio server */

  printf ("Contacting CD audio server...");
  sock_cd = tcpEstablishConn ("localhost", CD_TCP_PORT);
  if (sock_cd<0) oops ("tcpEstablishConn (cd audio)");

  printf ("OK.\n");

  cd=(struct cdtype *) buf_cd;


  /* open serial & MIDI devices */
  midi.fd=midi_openser("/dev/cua1", B38400);  /* Windows generic serial MIDI */
  mixer.fd=midi_openser("/dev/cua2", B9600);  /* DACS 411 mixer serial MIDI */
  board.fd=midi_openser("/dev/cua0", B9600);  /* DACS 112 ctrlr serial MIDI */

  if (!midi.fd) oops ("can't open midi device...");
  if (!mixer.fd) oops ("can't open mixer device...");
  if (!board.fd) oops ("can't open board device...");



  mixer_reset(&mixer);
  for (r=0;r<128;r++) mctrl.chipreg[r]=0xff;

  while (1) {

    if (midi_datawaiting(&midi)) {
      /* there's data waiting from the MIDI host */

      midi_readstream (&midi);   /* get it.. */
```

```
      if (midi.validdata) {
/* we've received a whole valid command */

switch (midi.cmd) {
case MIDICMD_CONTROL:

  /* it was a MIDI controller message */

  if (midi.chan==0x0F) {

    /* hardcoded hack: if it's channel 15, it's going to the mixer */

    printf ("MIDI controller %x to value %x on channel %x\n", midi.data[0], midi.data[1], midi.char

    /* change the controller table */

    mctrl.midictrl[midi.data[0]-1]=midi.data[1];

    /* call the controller translate routine */

    controller_translate (&mctrl, mixer.fd);
  }
  break;

case MIDICMD_PROGRAM:

  /* MIDI program change */

  if (midi.chan==0x0E) {

    /* another egregious hack, if it's on channel 14, we control the
       CD-ROM, and choose which disc to load */

    printf ("MIDI program change to %d on channel %x\n", midi.data[0], midi.chan);

    /* the cd data structure.. mangle appropriate elements */

    cd->disc=midi.data[0];
    cd->function=CD_OPEN;

    /* talk to CDROM server */

    tcpWriteBuffer (sock_cd, buf_cd, CD_TCP_BUFSIZE);
  }
  break;

case MIDICMD_NOTEON:

  /* MIDI note on command */

  if (midi.chan==0x0E) {
```

```
    /* yes, another hack for the demo.. note-on starts a track on
       the current cd */

    printf ("MIDI note %d on with velocity %d on channel %d\n", midi.data[0], midi.data[1], midi.ch

    cd->function=CD_PLAY;
    cd->track=midi.data[0]+1;
    cd->s_min=0;
    cd->s_sec=1;
    cd->d_min=73;
    cd->d_sec=0;
    tcpWriteBuffer (sock_cd, buf_cd, CD_TCP_BUFSIZE);

  }
  break;
}
      }
    } else {

      switch (midi.runstatus) {
      case MSTOP:
printf ("MIDI stop\n");

/* when the MIDI stop code is received, stop all of our running
   functions */

mixer_reset (&mixer);
midi.runstatus = 0;
cd->function=CD_STOP;
tcpWriteBuffer (sock_cd, buf_cd, CD_TCP_BUFSIZE);
cd->function=CD_CLOSE;
tcpWriteBuffer (sock_cd, buf_cd, CD_TCP_BUFSIZE);
break;
      case MSTART:
printf ("MIDI start\n");

/* when the MIDI start code is received, reset some stuff.. */

mixer_reset (&mixer);
midi.runstatus = 1;
break;
      case MCONT:
break;
      }

      if (midi_datawaiting(&board)) {

/* there's data waiting on the board's stripped-MIDI channel */

midi_readstream (&board);

if (board.validdata) {
```

```
  switch (board.cmd) {
  case MIDICMD_CONTROL:

    if (board.chan==0x00) {
      printf ("BOARD controller %x to value %x on channel %x\n", board.data[0], board.data[1], boar

      /* send this to the MIDI host on channel 15 */

      obuf[0]=0xBF;
      obuf[1]=board.data[0]+1;
      obuf[2]=board.data[1];

      write(midi.fd, obuf, 3);

    }
    break;

  }
}
      } else {
      }


    }

  }

}
```

16.1.2 MIDI Handler Header, `midi.h`

This is the header file for `midi.c`. It defines several command bytes from the MIDI protocol as well as gives function prototypes for the MIDI functions.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: midi.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.21.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef _midi
#define _midi

#define MIDIDATABUFSZ   10            /* size of the MIDI data buffer */
```

```
/* run status */
#define MSTART          0x01          /* "start" received */
#define MSTOP           0x02          /* "stop" received */
#define MCONT           0x03          /* "continue" received */


#define MIDICMD_NOTEOFF 0x80          /* note off */
#define MIDICMD_NOTEON  0x90          /* note on */
#define MIDICMD_AFTER   0xA0          /* aftertouch */
#define MIDICMD_CONTROL 0xB0          /* controller */
#define MIDICMD_PROGRAM 0xC0          /* program change */
#define MIDICMD_PRESS   0xD0          /* channel pressure */
#define MIDICMD_BEND    0xE0          /* pitch wheel */

#define MIDICMD_SYSEXST 0xF0          /* sysex start */
#define MIDICMD_SYSEXEN 0xF7          /* sysex end */

#define MIDICMD_MTCQF   0xF1          /* MIDI time code quarter frame */
#define MIDICMD_SPP     0xF2          /* song position pointer */
#define MIDICMD_SONGSEL 0xF3          /* song select */
#define MIDICMD_TUNEREQ 0xF6          /* tune request */

#define MIDICMD_CLOCK   0xF8          /* realtime: clock */
#define MIDICMD_START   0xFA          /* realtime: start */
#define MIDICMD_CONT    0xFB          /* realtime: continue */
#define MIDICMD_STOP    0xFC          /* realtime: stop */
#define MIDICMD_ASENSE  0xFE          /* realtime: active sense */
#define MIDICMD_RESET   0xFF          /* realtime: reset */


/* generic logical MIDI protocol handling structure */
struct midi_stream {
  int fd;                             /* file descriptor for device */
  unsigned char cmd;                  /* current command */
  unsigned char chan;                 /* current channel for command */
  unsigned char data[MIDIDATABUFSZ];  /* data for command */
  unsigned char obuf[MIDIDATABUFSZ];  /* output buffer for device */
  unsigned char *dptr;                /* pointer to current data */
  unsigned int dcount;                /* 'countdown' expected bytes for cmd */
  unsigned long int midiclock;        /* MIDI clock */
  unsigned char runstatus;            /* MIDI run status */
  unsigned char resetflag;            /* MIDI reset flag */
  unsigned char validdata;            /* is cmd valid? */
};

/* function prototypes */
int midi_openser (char *devnam, int bd);
int midi_openmidi (char *devnam);
void midi_readstream (struct midi_stream *ms);
int midi_datawaiting (struct midi_stream *ms);

#endif
```

16.1.3 MIDI Handler Code, `midi.c`

This code handles low-level MIDI streams. This code supports MIDI-over-serial as well as true MIDI ports in a Linux environment.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: midi.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.21.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include "midi.h"

/*****************************************************************************
 * midi_openser
 *
 * opens a serial port as a virtual MIDI device.
 *****************************************************************************/
int midi_openser (char *devnam, int bd)
{
  struct termios t;
  int fd;

  /* open device */
  fd=open(devnam, O_RDWR|O_NONBLOCK);

  if (fd<1) {
    fprintf (stderr, "Error opening device %s.\n", devnam);
    exit (1);
  }

  tcgetattr (fd, &t);

  /* set the port discipline */
  t.c_iflag=BRKINT|IGNPAR;
  t.c_oflag=OPOST;
  t.c_cflag=CS8|CREAD|CLOCAL;
```

```
  t.c_lflag=0;

  cfsetospeed (&t, (speed_t) bd);
  cfsetispeed (&t, (speed_t) bd);
  tcsetattr (fd, TCSANOW, &t);

  return (fd);
}

/*****************************************************************************
 * midi_openmidi
 *
 * opens a sound card's midi port as a midi device
 *****************************************************************************/
int midi_openmidi (char *devnam)
{
  int fd;

  fd=open (devnam, O_RDWR|O_NONBLOCK);

  if (fd==-1) {
    fprintf (stderr, "error opening MIDI device %s.\n", devnam);
    exit (1);
  }
  return (fd);
}


/*****************************************************************************
 * midi_readstream
 *
 * handles reading a MIDI stream rather inefficiently.
 *****************************************************************************/
void midi_readstream (struct midi_stream *ms)
{
  int r;
  unsigned char ch;

  do {
    r=read(ms->fd,&ch,1);
  } while (r<1);

  if (ch&0x80) {
    /* MIDI command */

    /*
     * first we check to see if it's a MIDI realtime message.
     *
     * from the MIDI spec:
     *
     * Each RealTime Category message (ie, Status of 0xF8 to 0xFF) consists
     * of only 1 byte, the Status. These messages are primarily concerned
```

```
     * with timing/syncing functions which means that they must be sent and
     * received at specific times without any delays. Because of this, MIDI
     * allows a RealTime message to be sent at any time, even interspersed
     * within some other MIDI message. For example, a RealTime message
     * could be sent inbetween the two data bytes of a Note On message. A
     * device should always be prepared to handle such a situation;
     * processing the 1 byte RealTime message, and then subsequently resume
     * processing the previously interrupted message as if the RealTime
     * message had never occurred.
     */

    if ( (ch >= 0xF8) && (ch <= 0xFF) ) {
      /* MIDI realtime */

      switch (ch) {

      case MIDICMD_CLOCK:
ms->midiclock++;
ms->validdata=0;
break;

      case MIDICMD_START:
ms->midiclock=0;
ms->runstatus=MSTART;
ms->validdata=0;
break;

      case MIDICMD_CONT:
ms->runstatus=MCONT;
ms->validdata=0;
break;

      case MIDICMD_STOP:
ms->runstatus=MSTOP;
ms->validdata=0;
break;

      case MIDICMD_ASENSE:
ms->validdata=0;
break;

      case MIDICMD_RESET:
ms->resetflag=1;
ms->validdata=0;
break;
      }
    } else if ( (ch==MIDICMD_SYSEXST) || (ch==MIDICMD_SYSEXEN) ) {
      /* system exclusive */



    } else {
```

```
      /* some other kind of MIDI command */

      ms->cmd = ch&0xF0;       /* upper nybble is command */
      ms->chan = ch&0x0F;      /* lower nybble is channel */

      switch (ms->cmd) {
      case MIDICMD_NOTEOFF:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_NOTEON:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_AFTER:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_CONTROL:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_PROGRAM:
ms->dptr=ms->data;
ms->dcount=1;
ms->validdata=0;
break;
      case MIDICMD_PRESS:
ms->dptr=ms->data;
ms->dcount=1;
ms->validdata=0;
break;
      case MIDICMD_BEND:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_MTCQF:
ms->dptr=ms->data;
ms->dcount=1;
ms->validdata=0;
break;
      case MIDICMD_SPP:
ms->dptr=ms->data;
ms->dcount=2;
ms->validdata=0;
break;
      case MIDICMD_SONGSEL:
```

```
ms->dptr=ms->data;
ms->dcount=1;
ms->validdata=0;
break;
      case MIDICMD_TUNEREQ:
ms->dptr=ms->data;
ms->dcount=0;
ms->validdata=1;  /* no data, so it's valid right away */
break;
      default:
ms->dptr=ms->data;
ms->dcount=0;
ms->validdata=0;
break;
      }

    }
  } else {
    /* MIDI data, not command */

    if (ms->dcount) {
      /* current MIDI command still has pending data, save it */

      *ms->dptr = ch;
      ms->dptr++;
      ms->dcount--;
    }

    /* did we get all of the data for this command? */
    if (!ms->dcount) ms->validdata=1;

  }
}


int midi_datawaiting (struct midi_stream *ms)
{
  int fd;
  fd_set fds;
  struct timeval tv;

  fd=ms->fd;

  bzero (&tv, sizeof (struct timeval));
  tv.tv_usec = 1;

  FD_ZERO (&fds);
  FD_SET (fd, &fds);
  while ((select (fd+1, &fds, NULL, NULL, &tv))==-1); /* make sure select */
                                                      /* works            */

  if (FD_ISSET (fd, &fds)) return (1);
```

```
  else return (0);
}
```

16.1.4 Mixer Mid-Level Driver Header, `mixer.h`

This is the header file for `mixer.c`. This contains a structure for setting mixer register values as well as function prototypes.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: mixer.h
 *       Author: Stephen S. Richardson
 * Date Created: 04.21.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef _mixer
#define _mixer

#include "midi.h"

#define NUMCHIPREGS 128

struct mix_control {
  unsigned char chipreg[NUMCHIPREGS];    /* chip registers */
  unsigned char midictrl[128];           /* MIDI controller registers */
};


void controller_translate (struct mix_control *mc, int fd);
void mixer_reset (struct midi_stream *ms);


#endif
```

16.1.5 Mixer Mid-Level Driver Code, `mixer.c`

This code handles DACS mixer mid-level functionality. This code is mostly responsible for handling single MIDI controller to multiple mixer register translations (e.g. pan knobs, etc.).

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: mixer.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.21.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
```

```
 *============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include "mixer.h"
#include "midi.h"

/****************************************************************************
 * controller_translate
 *
 * translate MIDI controllers to DACS 411 mixer register data.
 * kind of kludgy, but functional.
 ****************************************************************************/
void controller_translate (struct mix_control *mc, int fd)
{
  float lev;
  unsigned char pan, luleft[128], luright[128], newl, newr, obuf[10];
  unsigned char fl1,fl2;
  int i,r;

  for (i=0;i<64;i++) {
    luleft[i]=127;
    luright[i]=(i*2)+1;
  }
  for (i=64;i<128;i++) {
    luleft[i]=127-((i-64)*2);
    luright[i]=127;
  }

  fl1=0;
  fl2=0;

  for (i=0;i<32;i++) {

    lev=(float) mc->midictrl[i] / 127.0;
    pan=mc->midictrl[i+64];

    newl = (unsigned char) luleft[pan] * lev;
    if (newl != mc->chipreg[i]) {
      mc->chipreg[i] = newl;
```

```
      if (fl1) usleep(150);

      obuf[0]=MIDICMD_CONTROL;
      obuf[1]=i;
      obuf[2]=newl;

      r=0;
      do {
r+=write (fd, obuf, 3);
      } while (r<3);

      fl2=1;

    }

    newr = (unsigned char) luright[pan] * lev;
    if (newr != mc->chipreg[i+32]) {
      mc->chipreg[i+32] = newr;

      if (fl2) usleep (150);

      obuf[0]=MIDICMD_CONTROL;
      obuf[1]=i+32;
      obuf[2]=newr;

      r=0;
      do {
r+=write (fd, obuf, 3);
      } while (r<3);

      fl1=1;

    }

  }
}


/*****************************************************************************
 * mixer_reset
 *
 * resets the DACS mixer.
 *****************************************************************************/
void mixer_reset (struct midi_stream *ms)
{
  int r;
  char resbyte=0xFF;

  r=0;
  do {
    r=write (ms->fd, &resbyte, 1);
  } while (r<1);
```

```
}
```

16.1.6 CDROM Service Provider Communications Header, cdaudio_comm.h

This header file provides the structure used for communication with the CDROM service provider. It also provides command definitions. Any program using this service provider should include this header.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 * CDROM audio server
 *
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef CDAUDIO_COMM_
#define CDAUDIO_COMM_

#define CD_TCP_PORT  4202
#define CD_TCP_BUFSIZE 256

#define CD_PRG_EXIT 255

#define CD_CLOSE    1
#define CD_OPEN     2

#define CD_PLAY     10
#define CD_CUE      11
#define CD_PAUSE    12
#define CD_RESUME   13
#define CD_STOP     14
#define CD_EJECT    15

struct cdtype {
  unsigned char disc;            /* which disc */
  unsigned char function;        /* cdrom function */
  unsigned char track;           /* track# to perform function on */
  unsigned char s_min;           /* seek minute */
  unsigned char s_sec;           /* seek second */
  unsigned char d_min;           /* duration minute */
  unsigned char d_sec;           /* duration second */
};

struct discinfo {
  unsigned short int tsec[100];
};

#endif
```

16.1.7 CDROM Service Provider CD Function Header, cdaudio_func.h

This header file provides function prototypes and structures for accessing a CDROM for audio playback in a Linux environment.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 * CDROM audio server
 *
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#ifndef CDAUDIO_FUNC_
#define CDAUDIO_FUNC_

#include <linux/cdrom.h>
#include "cdaudio_comm.h"


/* prefix for cdrom device */
#define CDROM_DEV "/dev/scd"

#define SUCCESS            1
#define ERROR_CDROMHW      -50

struct cdhw_t {
    struct cdrom_tochdr tochdr;
    struct cdrom_tocentry tocentries[100];
    struct cdrom_subchnl subchnl;
};


extern struct cdhw_t *read_hw (int cdfile, int *err);
extern int cdrPlay (int track, int seekmin, int seeksec, int durmin, int dursec, int cdfile, struct
extern int cdrCue (int track, int seekmin, int seeksec, int durmin, int dursec, int cdfile, struct
extern int cdrPause (int cdfile);
extern int cdrResume (int cdfile);
extern int cdrStop (int cdfile);
extern int cdrEject (int cdfile);

#endif
```

16.1.8 CDROM Service Provider Main Code, cdaudio_daemon.c

This code establishes the server and handles requests from a client.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: cdaudio_daemon.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.22.97
```

```
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *============================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 ****************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include "server.h"
#include "cdaudio_func.h"
#include "cdaudio_comm.h"

void oops (char *mesg)
{
  perror (mesg);
  exit (1);
}

int discjob (struct cdtype *cmd, int cdfd, struct discinfo *di)
{
  int v=0;

  switch (cmd->function) {
  case CD_PLAY:
    printf ("[CD Play: Track %d seek to %2.2d:%2.2d for %2.2d:%2.2d]...\n", cmd->track, cmd->s_min,
    v=cdrPlay (cmd->track, cmd->s_min, cmd->s_sec, cmd->d_min, cmd->d_sec, cdfd, di);
    break;

  case CD_CUE:
    printf ("[CD Cue: Track %d seek to %2.2d:%2.2d for %2.2d:%2.2d]...\n", cmd->track, cmd->s_min,
    v=cdrCue (cmd->track, cmd->s_min, cmd->s_sec, cmd->d_min, cmd->d_sec, cdfd, di);
    break;

  case CD_PAUSE:
    printf ("[CD Pause]...\n"); fflush (stdout);
    v=cdrPause (cdfd);
    break;

  case CD_RESUME:
    printf ("[CD Resume]...\n"); fflush (stdout);
    v=cdrResume (cdfd);
```

```
    break;

  case CD_STOP:
    printf ("[CD Stop]...\n"); fflush (stdout);
    v=cdrStop (cdfd);
    break;

  case CD_EJECT:
    printf ("[CD Eject]...\n"); fflush (stdout);
    v=cdrEject (cdfd);
    break;
  }

  if (v==SUCCESS) printf ("OK\n");
  else printf ("Failed.\n");
  fflush(stdout);
  return (v);
}

int cdtracks (int cdfd, struct discinfo *di)
{
  struct cdhw_t *cd;
  int err, i;

  printf ("[Read disc info]... \n"); fflush (stdout);
  cd = read_hw (cdfd, &err);
  if (err!=SUCCESS) {
    printf ("Failed.\n");
    fflush(stdout);
    free (cd);
    return (ERROR_CDROMHW);
  } else {
    printf ("OK.\n");
    fflush(stdout);
    di->tsec[0]=0;
    for (i=cd->tochdr.cdth_trk0; i<=(cd->tochdr.cdth_trk1+1);i++) {
      di->tsec[i+1]=cd->tocentries[i].cdte_addr.msf.minute*60+
cd->tocentries[i].cdte_addr.msf.second;
      printf ("[Track %d starts %d seconds into disc.]\n", i, di->tsec[i]);
      fflush(stdout);
    }
    free (cd);
    return (SUCCESS);
  }

}

void main (void)
{
  int s, sfd, ex, hlt, v=0, cdstat, cdf=-1;
  struct ipaddx ip;
  struct cdtype *st;
```

```
  struct discinfo cdinfo;
  char buffer[CD_TCP_BUFSIZE];
  char dev[80];

  printf ("DACS : Distributed Audio Control System\nCopyright (C) 1997 Stephen S. Richardson / SR R
  fflush (stdout);
  printf ("Attempting to establish server... \n"); fflush (stdout);
  s=tcpEstablishServer ("localhost", CD_TCP_PORT);
  if (s<0) oops ("cdaudio_daemon (establish server)");
  hlt=0;
  printf ("OK.\n");
  fflush (stdout);

  printf ("Waiting for client connection... \n");

  while (!hlt) {
    while ((tcpDataWaiting(s)) != DATA) {
      usleep (100000);
      /* idle */
    }

    sfd=tcpAcceptConn (s, &ip);
    if (sfd<0) oops ("cd_daemon (accept connection)");
    printf ("Accepted.\n");
    fflush (stdout);

    ex=0;
    cdstat=CD_CLOSE;

    while (ex==0) {
      if (tcpDataWaiting (sfd)==DATA) {
v=tcpReadBuffer (sfd, buffer, sizeof (buffer));
if (v==ERROR_HANGUP) {
  printf ("[CD Stop]...\n");
         fflush (stdout);
  cdrStop (cdf);
    if (cdstat==CD_OPEN) {
      printf ("[CLOSE CDROM device]..."); fflush (stdout);
              fflush (stdout);
      close (cdf);
      if (cdf<0) {
printf ("Failed!\n");
                fflush (stdout);
      } else {
printf ("OK.\n");
                fflush (stdout);
      }
      cdstat=CD_CLOSE;
    }
  printf ("Client disconnect.\nWaiting for client connection... \n");
  fflush (stdout);
  ex=1;
```

```
}
else if (v==ERROR_READ) oops ("tcp_daemon (read buffer)");
else {
  st = (struct cdtype *) buffer;

  /**** HANDLE OPENING/CLOSING THE CDROM DEVICE ****/

  if (st->function == CD_OPEN) {
    if (cdstat==CD_CLOSE) {
      sprintf (dev, "%s%d", CDROM_DEV, st->disc);
      printf ("[OPEN CDROM device %s]...\n", dev); fflush (stdout);
      cdf=open(dev, O_RDONLY);
      if (cdf<0) {
printf ("Failed!\n");
                fflush (stdout);
cdstat=CD_CLOSE;
      } else {
printf ("OK.\n");
                fflush (stdout);
cdstat=CD_OPEN;
cdtracks (cdf, &cdinfo);
      }
    } else {
      close (cdf);
      sprintf (dev, "%s%d", CDROM_DEV, st->disc);
      printf ("[OPEN CDROM device %s]...\n", dev); fflush (stdout);
              fflush (stdout);
      cdf=open(dev, O_RDONLY);
      if (cdf<0) {
printf ("Failed!\n");
                fflush (stdout);
cdstat=CD_CLOSE;
      } else {
printf ("OK.\n");
                fflush (stdout);
  cdstat=CD_OPEN;
cdtracks (cdf, &cdinfo);
      }
    }
  } else if (st->function == CD_CLOSE) {
    if (cdstat==CD_OPEN) {
      printf ("[CLOSE CDROM device]...\n"); fflush (stdout);
      close (cdf);
      if (cdf<0) {
printf ("Failed!\n");
                fflush (stdout);
      } else {
printf ("OK.\n");
                fflush (stdout);
      }
      cdstat=CD_CLOSE;
    }
```

```
  } else if (st->function == CD_PRG_EXIT) {
    printf ("Exiting.\n");
    fflush(stdout);
    hlt=1;
  } else {
    if (cdstat == CD_OPEN) {
      v=discjob (st, cdf, &cdinfo);
    } else {
      printf ("[tried to perform a function without first opening CDROM]\n");
     fflush(stdout);
    }
  }
}
      } else {
/* idle */
usleep (2000);
      }
    }
  }
}
```

16.1.9 CDROM Service Provider CD Function Code, `cdaudio_func.c`

This code handles talking to the CD-ROM device driver. This code is relatively specific to the Linux environment.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: cdaudio_func.c
 *       Author: Stephen S. Richardson
 * Date Created: 04.22.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: make
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <linux/cdrom.h>
#include "cdaudio_func.h"

struct cdhw_t *read_hw (int cdfile, int *err)
{
```

```
  int i;
  struct cdhw_t *hw = malloc(sizeof(struct cdhw_t));

  /* read header */
  if ( ioctl(cdfile, CDROMREADTOCHDR, &(hw->tochdr)) == -1 ) {
    *err=ERROR_CDROMHW;
    return(NULL);
  }

  /* read individual tracks */
  for (i=hw->tochdr.cdth_trk0; i<=hw->tochdr.cdth_trk1; i++) {
    hw->tocentries[i-1].cdte_track = i;
    hw->tocentries[i-1].cdte_format = CDROM_MSF;
    if ( ioctl(cdfile, CDROMREADTOCENTRY, &(hw->tocentries[i-1])) == -1 ) {
      *err=ERROR_CDROMHW;
      return (NULL);
    }
  }

  /* read the lead-out track */
  hw->tocentries[hw->tochdr.cdth_trk1].cdte_track = CDROM_LEADOUT;
  hw->tocentries[hw->tochdr.cdth_trk1].cdte_format = CDROM_MSF;
  if ( ioctl(cdfile, CDROMREADTOCENTRY, &(hw->tocentries[hw->tochdr.cdth_trk1])) == -1 ) {
    *err=ERROR_CDROMHW;
    return (NULL);
  }

  /* read subchannel info */
  hw->subchnl.cdsc_format = CDROM_MSF;
  if ( ioctl(cdfile, CDROMSUBCHNL, &(hw->subchnl)) == -1 ) {
    *err=ERROR_CDROMHW;
    return (NULL);
  }

  *err=SUCCESS;
  return (hw);

}


int cdrPlay (int track, int seekmin, int seeksec, int durmin, int dursec, int cdfile, struct discin
{
  int t;
  struct cdrom_tochdr tochdr;
  struct cdrom_msf msf;
  struct cdrom_subchnl subchnl;

  subchnl.cdsc_format = CDROM_MSF;

  if (ioctl(cdfile, CDROMSUBCHNL, &subchnl) == -1) {
    return (ERROR_CDROMHW);
  }
```

```
  if (subchnl.cdsc_audiostatus == CDROM_AUDIO_PAUSED) {
    if ( ioctl(cdfile, CDROMRESUME) == -1 ) {
      return (ERROR_CDROMHW);
    }
  } else {
    if ( ioctl(cdfile, CDROMREADTOCHDR, &tochdr) == -1 ) {
      return (ERROR_CDROMHW);
    }

    t=(seekmin*60)+seeksec+di->tsec[track];

    msf.cdmsf_min0 = t/60;
    msf.cdmsf_sec0 = t-(t/60)*60;
    msf.cdmsf_frame0 = 1;

    if ( (durmin==0) && (dursec==0) ) {
      t=di->tsec[track+1];

      msf.cdmsf_min1 = t/60;
      msf.cdmsf_sec1 = t-(t/60)*60;
      msf.cdmsf_frame1 = 1;
    } else {
      t=(durmin*60)+(dursec)+di->tsec[track];

      msf.cdmsf_min1 = t/60;
      msf.cdmsf_sec1 = t-(t/60)*60;
      msf.cdmsf_frame1 = 1;
    }

    if ( ioctl(cdfile, CDROMPLAYMSF, &msf) == -1 ) {
      return (ERROR_CDROMHW);
    }
  }
  return (SUCCESS);

}

int cdrCue (int track, int seekmin, int seeksec, int durmin, int dursec, int cdfile, struct discinf
{
  int t;
  struct cdrom_tochdr tochdr;
  struct cdrom_msf msf;
  struct cdrom_subchnl subchnl;

  subchnl.cdsc_format = CDROM_MSF;

  if (ioctl(cdfile, CDROMSUBCHNL, &subchnl) == -1) {
    return (ERROR_CDROMHW);
  }

  if (subchnl.cdsc_audiostatus == CDROM_AUDIO_PAUSED) {
```

```
    if ( ioctl(cdfile, CDROMRESUME) == -1 ) {
      return (ERROR_CDROMHW);
    }
  } else {
    if ( ioctl(cdfile, CDROMREADTOCHDR, &tochdr) == -1 ) {
      return (ERROR_CDROMHW);
    }


    t=(seekmin*60)+seeksec+di->tsec[track];

    msf.cdmsf_min0 = t/60;
    msf.cdmsf_sec0 = t-(t/60)*60;
    msf.cdmsf_frame0 = 1;

    if ( (durmin==0) && (dursec==0) ) {
      t=di->tsec[track+1];

      msf.cdmsf_min1 = t/60;
      msf.cdmsf_sec1 = t-(t/60)*60;
      msf.cdmsf_frame1 = 1;
    } else {
      t+=(durmin*60)+dursec;

      msf.cdmsf_min1 = t/60;
      msf.cdmsf_sec1 = t-(t/60)*60;
      msf.cdmsf_frame1 = 1;
    }

    if ( ioctl(cdfile, CDROMPLAYMSF, &msf) == -1 ) {
      return (ERROR_CDROMHW);
    }

    if ( ioctl(cdfile, CDROMPAUSE) == -1 ) {
      return (ERROR_CDROMHW);
    }
  }
  return (SUCCESS);
}

int cdrPause (int cdfile)
{
  if ( ioctl(cdfile, CDROMPAUSE) == -1 ) {
    return (ERROR_CDROMHW);
  }
  return (SUCCESS);
}

int cdrResume (int cdfile)
{
  if ( ioctl(cdfile, CDROMRESUME) == -1 ) {
    return (ERROR_CDROMHW);
```

```
  }
  return (SUCCESS);
}

int cdrStop (int cdfile)
{
  if ( ioctl(cdfile, CDROMSTOP) == -1 ) {
    return (ERROR_CDROMHW);
  }
  return (SUCCESS);
}

int cdrEject (int cdfile)
{
  if ( ioctl(cdfile, CDROMEJECT) == -1 ) {
    return (ERROR_CDROMHW);
  }
  return (SUCCESS);
}
```

16.1.10 Serial Communications Header, `serial.h`

This header file defines functions and structures for the serial handling code that implements the protocol described in the firmware section of this document.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: serial.h
 *  Description: routines to handle serial communications with DACS components
 *       Author: Stephen S. Richardson
 * Date Created: 07.12.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: serial.h,v 1.1 1997/07/25 12:09:25 prefect Exp prefect $
 *
 *****************************************************************************/

#define SERBUFLEN 1024                          /* serial buffer size */

#define MAXPAYLOAD 128                          /* maximum payload size */
#define MAXFRSIZE  MAXPAYLOAD+MAXPAYLOAD+4      /* maximum frame size */

#define DLE  '!'                /* data link escape character 0x10 */
#define STX  '['                /* start transmission 0x02 */
#define ETX  ']'                /* end transmission 0x03 */
```

```
struct serbuf {
  char data[SERBUFLEN];          /* data in buffer */
  int len;                       /* current length of buffer */
  int cur;                       /* index to current data in buffer */
};


struct raw_frame {
  int len;                       /* length of frame */
  int cur;                       /* current index */
  char data[MAXPAYLOAD];         /* frame data */
  char done;                     /* done flag */
  char dleflag;                  /* DLE flag */
  char stxflag;                  /* STX flag */
};


void ser_init_frame (struct raw_frame *f);
void ser_init_serbuf (struct serbuf *b);
int ser_write_buf (struct raw_frame *fr, struct serbuf *ob);
int ser_open (char *devnam, int bd);
int ser_service (struct serbuf *ib, struct serbuf *ob,
 struct raw_frame *of, int serfd);
```

16.1.11 Serial Communications Code, `serial.c`

This code implements the protocol described in the firmware protocols section of this design document.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: serial.c
 *  Description: routines to handle serial communications with DACS components
 *       Author: Stephen S. Richardson
 * Date Created: 07.12.97
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: serial.c,v 1.1 1997/07/13 14:18:00 prefect Exp prefect $
 *
 *****************************************************************************/

#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include "serial.h"

void ser_init_frame (struct raw_frame *f)
{
  f->len=0;
  f->cur=0;
  f->done=0;
  f->dleflag=0;
  f->stxflag=0;

  memset (f->data, 0, MAXPAYLOAD);
}

void ser_init_serbuf (struct serbuf *b)
{
  b->len=0;
  b->len=0;
  memset (b->data, 0, SERBUFLEN);
}

int ser_write_buf (struct raw_frame *fr, struct serbuf *ob)
{
  int i, j=0;

  /* return 0 if the whole frame can't be put in the buffer. */
  if (ob->len + fr->len >= SERBUFLEN) return 0;

  /* start of frame */
  ob->data[j++]=DLE;
  ob->data[j++]=STX;

  /* data of frame */
  for (i=0;i<fr->len;i++) {
    if (fr->data[i]==DLE) {
      ob->data[j++]=DLE;
      ob->data[j++]=DLE;       /* character stuffing */
    } else {
      ob->data[j++]=fr->data[i];
    }
  }

  /* end of frame */
  ob->data[j++]=DLE;
  ob->data[j++]=ETX;
```

```
  ob->len+=j;

  return (1);
}

int ser_service (struct serbuf *ib, struct serbuf *ob,
 struct raw_frame *of, int serfd)
{
  int r, i, j;

  /* outgoing serial data? */

  if (ob->len) {
    r=write (serfd, ob->data, ob->len);

    if (r<0) {
      /* error writing. */

      fprintf (stderr, "error writing to serial device.\n");
      exit (1);
    } else {
      /* data were written */

      if (r==ob->len) {
/* all data were written */

ob->len=0;
      } else {
/* only some of the data were written.. shift buffer left by r */

memmove (ob->data, ob->data+r, ob->len-r);
ob->len -= r;
      }
    }
  }


  /* incoming serial data and enough space to store some of it? */

  if ((ser_datawaiting (serfd)) && (ib->len < SERBUFLEN)) {

    r=read (serfd, ib->data+ib->len, SERBUFLEN - ib->len);

    if (r<0) {
      fprintf (stderr, "error reading from serial device.\n");
      exit (1);
    } else {
      /* data were read */

      ib->len += r;
```

```
      /* process into a frame */

      for (i=0;i<ib->len;i++) {
if (ib->data[i] == DLE) {
  /* found a DLE */
  if (of->dleflag) {
    /* last char was a DLE */

    of->dleflag=0;

    if (of->stxflag) {
      of->data[of->cur++]=DLE;    /* it was a character stuffed DLE */
    }

  } else {
    /* last char was not a DLE */
    of->dleflag=1;
  }
} else {
  /* found something other than a DLE */
  if (of->dleflag) {
    /* last char was a DLE */

    of->dleflag=0;

    if (ib->data[i] == STX) {
      /* found start of packet */
      of->stxflag=1;
      of->done=0;
      of->len=0;
      of->cur=0;
    } else if (ib->data[i] == ETX) {
      /* found end of packet */
      of->stxflag=0;
      of->len=of->cur;
      of->done=1;
    }
  } else {
    /* last character was not DLE */
    if (of->stxflag) {
      of->data[of->cur++]=ib->data[i];   /* put the data into frame */
    }
  }
}
      }

      /* reset input buffer position */
      ib->cur=0;
      ib->len=0;
    }
  }
}
```

```
int ser_datawaiting (int serfd)
{
  fd_set fds;
  struct timeval tv;

  bzero (&tv, sizeof (struct timeval));
  tv.tv_usec = 1;

  FD_ZERO (&fds);
  FD_SET (serfd, &fds);
  while ((select (serfd+1, &fds, NULL, NULL, &tv))==-1); /* make sure select */
                                                         /* works            */

  if (FD_ISSET (serfd, &fds)) return (1);
  else return (0);
}


/*****************************************************************************
 * ser_open
 *
 * opens a serial port at a baud rate for NON-BLOCKING read/write
 *****************************************************************************/
int ser_open (char *devnam, int bd)
{
  struct termios t;
  int fd;

  /* open device */
  fd=open(devnam, O_RDWR|O_NONBLOCK);

  if (fd<1) {
    fprintf (stderr, "Error opening serial device %s.\n", devnam);
    exit (1);
  }

  tcgetattr (fd, &t);

  /* set the port discipline */
  t.c_iflag=BRKINT|IGNPAR;
  t.c_oflag=OPOST;
  t.c_cflag=CS8|CREAD|CLOCAL;
  t.c_lflag=0;

  cfsetospeed (&t, (speed_t) bd);
  cfsetispeed (&t, (speed_t) bd);
  tcsetattr (fd, TCSANOW, &t);

  return (fd);
}
```

16.1.12 TCP/IP Client Library Header, `client.h`

This header provides defines and function declarations for the TCP/IP client library.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: client.h
 *  Description: TCP/IP client routines, header file
 *       Author: Stephen S. Richardson
 * Date Created: 04.23.95
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: client.h,v 1.1 1997/07/25 12:15:48 prefect Exp prefect $
 *
 *****************************************************************************/

#ifndef CLIENT_
#define CLIENT_

#define DATA                1
#define NODATA              0

#define SUCCESS             1
#define ERROR_BADHOST       -10
#define ERROR_SOCKET        -11
#define ERROR_CONNECT       -12
#define ERROR_KILL          -13
#define ERROR_WRITEFAILED   -14
#define ERROR_HANGUP        -15
#define ERROR_READ          -16

extern int tcpEstablishConn (char *hostname, int port);
extern int tcpKillConn (int fd);
extern int tcpWriteBuffer (int fd, char *buffer, size_t bufsize);
extern int tcpDataWaiting (int sfd);
extern int tcpReadBuffer (int fd, char *buffer, int bufsize);

#endif
```

16.1.13 TCP/IP Client Library Code, `client.c`

The following code provides easy access to Berkeley TCP/IP sockets in a Unix (Linux) environment.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: client.c
 *  Description: TCP/IP client routines
 *       Author: Stephen S. Richardson
 * Date Created: 04.23.95
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: client.c,v 1.1 1997/07/25 12:15:44 prefect Exp prefect $
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>
#include <netdb.h>
#include <time.h>
#include <sys/time.h>
#include "client.h"

/*****************************************************************************
 attempts to connect to an ip and tcp port, returning an fd of the socket

 returns the file descriptor of the socket or

 ERROR_BADHOST - hostname or ip specified is bogus
 ERROR_SOCKET - the 'socket' command failed
 ERROR_CONNECT - the 'connect' command failed
*****************************************************************************/

int tcpEstablishConn (char *hostname, int port)
{
  struct sockaddr_in bba;
  struct hostent *hp;
  int s;

  memset (&bba, 0 , sizeof (bba));
  bba.sin_family = AF_INET;
  hp = gethostbyname (hostname);
```

```
  if (hp == NULL) return (ERROR_BADHOST);
  memcpy (&bba.sin_addr, hp->h_addr, hp->h_length);
  bba.sin_port = htons ((unsigned short int) port);

  s = socket (AF_INET,SOCK_STREAM,0);
  if (s == -1) return (ERROR_SOCKET);
  if (connect (s,(struct sockaddr *) &bba, sizeof (bba)) !=0)
    return (ERROR_CONNECT);

  return s;
}



/*****************************************************************************
 kills (disconnects) a socket

 returns SUCCESS if it worked or

 ERROR_KILL if it couldn't be disconnected
*****************************************************************************/

int tcpKillConn (int fd)
{
  int retcode;

  retcode = close (fd);
  if (retcode==-1) return (ERROR_KILL);
  else return (SUCCESS);
}



/*****************************************************************************
 writes out buffer of size bufsize to an open file descriptor (a socket)

 returns number of bytes written or

 ERROR_WRITEFAILED - write failed
*****************************************************************************/

int tcpWriteBuffer (int fd, char *buffer, size_t bufsize)
{
  int retcode;

  retcode=write (fd, buffer, bufsize);

  if (retcode==-1) {
    return (ERROR_WRITEFAILED);
  }
  else return (retcode);
}
```

```
/*****************************************************************************
 synchronous I/O multiplexer - detects if there's stuff waiting using select
 doesn't disturb the file descriptor set you send

 returns DATA if there's data, or NODATA if there's none

*****************************************************************************/

int tcpDataWaiting (int sfd)
{
  fd_set fds;
  struct timeval tv;

  bzero (&tv, sizeof (struct timeval));
  tv.tv_usec = 1;

  FD_ZERO (&fds);
  FD_SET (sfd, &fds);
  while ((select (sfd+1, &fds, NULL, NULL, &tv))==-1); /* make sure select */
                                                       /* works            */

  if (FD_ISSET (sfd, &fds)) return (DATA);
  else return (NODATA);
}



/*****************************************************************************
 reads bytes from a socket file descriptor, putting them into the buffer

 returns number of bytes read, or

 ERROR_HANGUP if the socket has been disconnected

*****************************************************************************/

int tcpReadBuffer (int fd, char *buffer, int bufsize)
{
  int count;

  count=read (fd, buffer, bufsize);
  if (count==0) {
    close (fd);
    return (ERROR_HANGUP);
  } else if (count==-1) {
    return (ERROR_READ);
  }
  else return (count);
}
```

16.1.14 TCP/IP Server Library Header, `server.h`

This header provides definitions and function prototypes for the TCP/IP server library functions.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: server.h
 *  Description: TCP/IP server routines, header file
 *       Author: Stephen S. Richardson
 * Date Created: 04.23.95
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: server.h,v 1.1 1997/07/25 12:15:41 prefect Exp prefect $
 *
 *****************************************************************************/

#ifndef SERVER_
#define SERVER_

#define DATA                1
#define NODATA              0

#define SUCCESS             1

#define ERROR_KILL          -13
#define ERROR_WRITEFAILED   -14
#define ERROR_HANGUP        -15
#define ERROR_READ          -16

#define ERROR_BADHOST       -100
#define ERROR_SOCKET        -101
#define ERROR_BIND          -102
#define ERROR_LISTEN        -103
#define ERROR_ACCEPT        -104


struct ipaddx {
  unsigned char first;
  unsigned char second;
  unsigned char third;
  unsigned char fourth;
} ipaddx;

extern int tcpEstablishServer (char *hostname, int port);
extern int tcpAcceptConn (int s, struct ipaddx *ip);
extern int tcpKillConn (int fd);
```

```
extern int tcpWriteBuffer (int fd, char *buffer, size_t bufsize);
extern int tcpDataWaiting (int sfd);
extern int tcpReadBuffer (int fd, char *buffer, int bufsize);

#endif
```

16.1.15 TCP/IP Server Library Code, `server.c`

The following code provides easy access to Berkeley sockets to set up a server in a Unix environment.

```
/*****************************************************************************
 * DACS : Distributed Audio Control System
 *===========================================================================
 *         File: server.c
 *  Description: TCP/IP server routines
 *       Author: Stephen S. Richardson
 * Date Created: 04.23.95
 *  Environment: GNU C Compiler (GCC) v2.7.1, Linux i486 v2.0.28
 *        Build: library
 *===========================================================================
 * The code, executables, documentation, firmware images, and all related
 * material of DACS are
 * Copyright (C) 1997 Stephen S. Richardson - ALL RIGHTS RESERVED
 *****************************************************************************
 * Source code control:
 *
 * $Id: server.c,v 1.1 1997/07/25 12:15:37 prefect Exp prefect $
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "server.h"

/*****************************************************************************
 establishes a tcp server on hostname at port

 returns a socket if it works, or

 ERROR_BADHOST if a bad host was specified
 ERROR_SOCKET if there's a socket error
 ERROR_BIND if there's a bind error
 ERROR_LISTEN if there's a listen error
 *****************************************************************************/
```

```
int tcpEstablishServer (char *hostname, int port)
{
  struct sockaddr_in saddr;
  struct hostent *hp;
  int s;

  memset (&saddr,0,sizeof (saddr));
  saddr.sin_family = AF_INET;
  hp = gethostbyname (hostname);
  if (hp == NULL) return (ERROR_BADHOST);

  bzero ((char *) &saddr.sin_addr, hp->h_length);

  saddr.sin_port = htons (port);

  s = socket (AF_INET,SOCK_STREAM,0);
  if (s== -1) return (ERROR_SOCKET);

  if (bind (s, (struct sockaddr *) &saddr, sizeof (saddr)) !=0)
    return (ERROR_BIND);

  if (listen (s,1) !=0) return (ERROR_LISTEN);  /* queue 1 request only */

  return (s);
}



/*****************************************************************************
 accepts a connection from a client, returning a socket file descriptor
 needs the port socket and an ipaddx pointer passed to it (it gets filled in
 with the ip address of the host connecting to the server)

 returns socket file descriptor, or

 ERROR_ACCEPT if accept errors out more than 255 times.
 *****************************************************************************/

int tcpAcceptConn (int s, struct ipaddx *ip)
{
  struct sockaddr_in addr;
  struct hostent *b;
  int sfd, addrlen, x=0;

  addrlen = sizeof (struct sockaddr_in);

  do {
    sfd = accept (s, (struct sockaddr *) &addr,&addrlen);
    x++;
  } while ((sfd<0)&&(x<=255));
  if (x>=255) return (ERROR_ACCEPT);
```

```
  b = gethostbyaddr ((char *)&addr.sin_addr.s_addr, sizeof (addr.sin_addr.s_addr),AF_INET);

  ip->first = addr.sin_addr.s_addr & 0x000000FF;
  ip->second = (addr.sin_addr.s_addr & 0x0000FF00)>>8;
  ip->third = (addr.sin_addr.s_addr & 0x00FF0000)>>16;
  ip->fourth = (addr.sin_addr.s_addr & 0xFF000000)>>24;

  return (sfd);
}


/*****************************************************************************
 kills (disconnects) a socket

 returns SUCCESS if it worked or

 ERROR_KILL if it couldn't be disconnected
*****************************************************************************/

int tcpKillConn (int fd)
{
  int retcode;

  retcode = close (fd);
  if (retcode==-1) return (ERROR_KILL);
  else return (SUCCESS);
}



/*****************************************************************************
 writes out buffer of size bufsize to an open file descriptor (a socket)

 returns number of bytes written or

 ERROR_WRITEFAILED - write failed
*****************************************************************************/

int tcpWriteBuffer (int fd, char *buffer, size_t bufsize)
{
  int retcode;

  retcode=write (fd, buffer, bufsize);

  if (retcode==-1) {
    return (ERROR_WRITEFAILED);
  }
  else return (retcode);
}


/*****************************************************************************
```

```
 synchronous I/O multiplexer - detects if there's stuff waiting using select

 returns DATA if there's data, or NODATA if there's none

*****************************************************************************/

int tcpDataWaiting (int sfd)
{
  fd_set fds;
  struct timeval tv;

  bzero (&tv, sizeof (struct timeval));
  tv.tv_usec = 1;

  FD_ZERO (&fds);

  FD_SET (sfd, &fds);
  while ((select (sfd+1, &fds, NULL, NULL, &tv))==-1); /* make sure select */
                                                       /* works            */

  if (FD_ISSET (sfd, &fds)) return (DATA);
  else return (NODATA);
}



/*****************************************************************************
 reads bytes from a socket file descriptor, putting them into the buffer

 returns number of bytes read, or

 ERROR_HANGUP if the socket has been disconnected

*****************************************************************************/

int tcpReadBuffer (int fd, char *buffer, int bufsize)
{
  int count;

  count=read (fd, buffer, bufsize);
  if (count==0) {
    close (fd);
    return (ERROR_HANGUP);
  } else if (count==-1) {
    return (ERROR_READ);
  }
  else return (count);
}
```

A VHDL Code

This appendix includes the VHDL code used to synthesize the logic for the GAL16V8 devices used as Pbus address decoders throughout the system. It is important to note that individualized addresses were required for each board in the systems. This required essentially identical VHDL code for duplicate boards, with the actual address changed. In these cases, only a representative piece of VHDL code is included. It should be clear to those familiar with VHDL how to change the actual decoded addresses. Each piece of code forces the pinouts, rather than allowing the compiler to handle those decisions. This was done because the PC boards were designed in advance of the GALs, with an assumed pinout.

All of the VHDL code is well commented, thus no additional description is given with the code. For each piece of VHDL code, the pin report is included for reference.

The VHDL code was compiled into standard JEDEC files through the use of the *Cypress Warp VHDL Synthesizer*, running on the author's Sun SPARCstation IPC. The Atmel GALs were programmed using a universal programmer in the WPI ECE Shop.

A.1 Control Board

A.1.1 Fader Module

The VHDL code for a fader module decode GAL is shown below.

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: fader-1.vhd
--   Date: 03.13.97
-- Target: Atmel ATF16V8B
--------------------------------------------------------------------------------
-- Fader board pbus address decoding GAL
-- first fader board
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x08         a/d mux n_ena     out
-- 0x10         led group 1 ena   out
-- 0x11         led group 2 ena   out
--------------------------------------------------------------------------------

ENTITY fade_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
mux_nena_out   :  OUT bit;
led_ena_2_out   :  OUT bit;
led_ena_1_out   :  OUT bit;
                nsense_out       :  OUT bit
            );

-- Force a package and a pinout
ATTRIBUTE part_name of fade_decode:entity is "C16V8";
```

```
ATTRIBUTE pin_numbers of fade_decode:entity is
  "nread_write_in:1 nlatch_in:2 addx_in(0):3 addx_in(1):4
           addx_in(2):5 addx_in(3):6 addx_in(4):7 addx_in(5):8
           addx_in(6):9 nsense_out:12 led_ena_1_out:13
           led_ena_2_out:14 mux_nena_out:15";

END fade_decode;


ARCHITECTURE behavior OF fade_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0001000" THEN

-- pbus 0x08

mux_nena_out <= '0';   -- mux nena ACTIVE
nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE

ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010000" THEN

-- pbus 0x10

mux_nena_out <= '1';   -- mux nena INACTIVE
nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '1';  -- led 1 ena ACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE

ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010001" THEN

-- pbus 0x11

mux_nena_out <= '1';   -- mux nena INACTIVE
nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '1';  -- led 2 ena ACTIVE
ELSE

-- not active

mux_nena_out <= '1';   -- mux nena INACTIVE
nsense_out <= '1';     -- pbus nsense INACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
END IF;
END PROCESS;
```

```
END behavior;
```

The pin report for the fader module decode GAL is shown below.

```
                      C16V8A

                  --------------------
 nread_write_in =| 1|            |20|* not used
      nlatch_in =| 2|            |19|* not used
      addx_in_0 =| 3|            |18|* not used
      addx_in_1 =| 4|            |17|* not used
      addx_in_2 =| 5|            |16|* not used
      addx_in_3 =| 6|            |15|= mux_nena_out
      addx_in_4 =| 7|            |14|= led_ena_2_out
      addx_in_5 =| 8|            |13|= led_ena_1_out
      addx_in_6 =| 9|            |12|= nsense_out
       not used *|10|            |11|* not used
                  --------------------
```

A.1.2 Output Assign Module

The VHDL code used to synthesize the logic for the output assignment module decode GAL is shown below.

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: outassn.vhd
--   Date: 03.13.97
-- Target: Atmel ATF16V8B
--------------------------------------------------------------------------------
-- Output assign board pbus address decoding GAL
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x14         led group 1 ena   out
-- 0x15         led group 2 ena   out
-- 0x16         7 segment ena     out
-- 0x30         button grp 1 nena in
-- 0x31         button grp 2 nena in
-- 0x32         button grp 3 nena in
--------------------------------------------------------------------------------

ENTITY outassn_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
btn_1_nena_out  :  OUT bit;
btn_2_nena_out  :  OUT bit;
btn_3_nena_out  :  OUT bit;
seg7_ena_out   :  OUT bit;
led_ena_2_out   :  OUT bit;
```

```
led_ena_1_out   :  OUT bit;
                nsense_out      :  OUT bit
            );

-- Force a package and pinout
ATTRIBUTE part_name of outassn_decode:entity is "C16V8";
ATTRIBUTE pin_numbers of outassn_decode:entity is
  "nread_write_in:1 nlatch_in:9 addx_in(0):2 addx_in(1):3
           addx_in(2):4 addx_in(3):5 addx_in(4):6 addx_in(5):7
           addx_in(6):8 nsense_out:12 btn_1_nena_out:13
           btn_2_nena_out:14 btn_3_nena_out:15 seg7_ena_out:16
           led_ena_2_out:17 led_ena_1_out:18";

END outassn_decode;


ARCHITECTURE behavior OF outassn_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010100" THEN

-- pbus 0x14

nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '1';  -- led 1 ena ACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE

ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010101" THEN

-- pbus 0x15

nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '1';  -- led 2 ena ACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010110" THEN

-- pbus 0x16
```

```
nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '1';  -- 7seg ena  ACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '0'
   AND addx_in = "0110010" THEN

-- pbus 0x32 (read)

nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '0'; -- btn 3 nena ACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '0'
   AND addx_in = "0110001" THEN

-- pbus 0x31 (read)

nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '0'; -- btn 2 nena ACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '0'
   AND addx_in = "0110000" THEN

-- pbus 0x30 (read)

nsense_out <= '0';     -- pbus nsense ACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '0'; -- btn 1 nena ACTIVE

ELSE
```

```
-- not active

nsense_out <= '1';     -- pbus nsense INACTIVE
led_ena_1_out <= '0';  -- led 1 ena INACTIVE
led_ena_2_out <= '0';  -- led 2 ena INACTIVE
seg7_ena_out <= '0';  -- 7seg ena  INACTIVE
btn_3_nena_out <= '1'; -- btn 3 nena INACTIVE
btn_2_nena_out <= '1'; -- btn 2 nena INACTIVE
btn_1_nena_out <= '1'; -- btn 1 nena INACTIVE
END IF;
END PROCESS;
END behavior;
```

The pinout for the output assignment module decode GAL is shown below:

```
                      C16V8A
                  -------------------
 nread_write_in =| 1|           |20|* not used
      addx_in_0 =| 2|           |19|* not used
      addx_in_1 =| 3|           |18|= led_ena_1_out
      addx_in_2 =| 4|           |17|= led_ena_2_out
      addx_in_3 =| 5|           |16|= seg7_ena_out
      addx_in_4 =| 6|           |15|= btn_3_nena_out
      addx_in_5 =| 7|           |14|= btn_2_nena_out
      addx_in_6 =| 8|           |13|= btn_1_nena_out
      nlatch_in =| 9|           |12|= nsense_out
       not used *|10|           |11|* not used
                  -------------------
```

A.1.3 Transport Control Module

The VHDL code for the transport control decode GAL is shown below:

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: transp.vhd
--   Date: 03.13.97
-- Target: Atmel ATF16V8B
--------------------------------------------------------------------------------
-- Transport control board pbus address decoding GAL
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x17         7 segment 1 ena   out
-- 0x18         7 segment 2 ena   out
-- 0x19         7 segment 3 ena   out
-- 0x20         led enable        out
-- 0x33         button nena       in
-- 0x50         encoder nena      in
--------------------------------------------------------------------------------
```

```
ENTITY transp_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
seg7_1_ena_out  :  OUT bit;
seg7_2_ena_out  :  OUT bit;
seg7_3_ena_out  :  OUT bit;
led_ena_out     :  OUT bit;
btn_nena_out    :  OUT bit;
enc_nena_out    :  OUT bit;
                nsense_out      :  OUT bit
            );

-- Force package and pinout
ATTRIBUTE part_name of transp_decode:entity is "C16V8";
ATTRIBUTE pin_numbers of transp_decode:entity is
  "nread_write_in:1 nlatch_in:2 addx_in(0):3 addx_in(1):4
           addx_in(2):5 addx_in(3):6 addx_in(4):7 addx_in(5):8
           addx_in(6):9 nsense_out:12 led_ena_out:13 seg7_3_ena_out:14
           enc_nena_out:15 btn_nena_out:16 seg7_2_ena_out:17
           seg7_1_ena_out:18";

END transp_decode;


ARCHITECTURE behavior OF transp_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010111" THEN

-- pbus 0x17

nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '1';  -- 7seg 1 ena ACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '0';     -- led ena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0011000" THEN

-- pbus 0x18

nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '1';  -- 7seg 2 ena ACTIVE
```

```
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '0';     -- led ena INACTIVE

ELSIF nlatch_in = '0' AND nread_write_in = '0'
   AND addx_in = "0110011" THEN

-- pbus 0x33 (read)

nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '0';    -- btn nena ACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '0';     -- led ena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '0'
   AND addx_in = "1010000" THEN

-- pbus 0x50 (read)

nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '0';    -- enc nena ACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '0';     -- led ena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0011001" THEN

-- pbus 0x19

nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '1';  -- 7seg 3 ena ACTIVE
led_ena_out <= '0';     -- led ena INACTIVE


ELSIF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0100000" THEN

-- pbus 0x20
```

```
nsense_out <= '0';      -- pbus nsense ACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '1';     -- led ena ACTIVE

ELSE

-- not active

nsense_out <= '1';      -- pbus nsense INACTIVE
seg7_1_ena_out <= '0';  -- 7seg 1 ena INACTIVE
seg7_2_ena_out <= '0';  -- 7seg 2 ena INACTIVE
btn_nena_out <= '1';    -- btn nena INACTIVE
enc_nena_out <= '1';    -- enc nena INACTIVE
seg7_3_ena_out <= '0';  -- 7seg 3 ena INACTIVE
led_ena_out <= '0';     -- led ena INACTIVE

END IF;
END PROCESS;
END behavior;
```

The pinout for the transport control decode GAL is shown below:

```
                    C16V8A
                 ------------------
 nread_write_in =| 1|          |20|* not used
      nlatch_in =| 2|          |19|* not used
      addx_in_0 =| 3|          |18|= seg7_1_ena_out
      addx_in_1 =| 4|          |17|= seg7_2_ena_out
      addx_in_2 =| 5|          |16|= btn_nena_out
      addx_in_3 =| 6|          |15|= enc_nena_out
      addx_in_4 =| 7|          |14|= seg7_3_ena_out
      addx_in_5 =| 8|          |13|= led_ena_out
      addx_in_6 =| 9|          |12|= nsense_out
       not used *|10|          |11|* not used
                 ------------------
```

A.2 Mixer Unit

A.2.1 Audio Input Module

The VHDL code used to synthesize the logic for the audio input module address decode GAL is shown below:

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: input-1.vhd
--   Date: 03.13.97
```

```
-- Target: Atmel ATF16V8B
--------------------------------------------------------------------------------
-- Input board trim control pbus address decoding GAL
-- First input board
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x08         latch ena         out
--------------------------------------------------------------------------------

ENTITY input_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
data_ena_out    :  OUT bit;
                nsense_out      :  OUT bit
             );

-- Force part and pinout
ATTRIBUTE part_name of input_decode:entity is "C16V8";
ATTRIBUTE pin_numbers of input_decode:entity is
  "nread_write_in:1 nlatch_in:2 addx_in(0):3 addx_in(1):4
           addx_in(2):5 addx_in(3):6 addx_in(4):7 addx_in(5):8
           addx_in(6):9 nsense_out:12 data_ena_out:13";

END input_decode;


ARCHITECTURE behavior OF input_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0001000" THEN

-- pbus 0x08

nsense_out <= '0';     -- pbus nsense ACTIVE
data_ena_out <= '1';   -- data latch ena ACTIVE

ELSE

-- not active

nsense_out <= '1';     -- pbus nsense ACTIVE
data_ena_out <= '0';   -- data latch ena INACTIVE

END IF;
END PROCESS;
END behavior;
```

The pinout for the input module decode GAL is shown below:

```
                    C16V8A
                  ----------------
 nread_write_in =| 1|        |20|* not used
      nlatch_in =| 2|        |19|* not used
      addx_in_0 =| 3|        |18|* not used
      addx_in_1 =| 4|        |17|* not used
      addx_in_2 =| 5|        |16|* not used
      addx_in_3 =| 6|        |15|* not used
      addx_in_4 =| 7|        |14|* not used
      addx_in_5 =| 8|        |13|= data_ena_out
      addx_in_6 =| 9|        |12|= nsense_out
       not used *|10|        |11|* not used
                  ----------------
```

A.2.2 Mix Module

The VHDL code used to synthesize the logic for the mix module address decode GAL is shown below:

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: mix-1.vhd
--   Date: 03.13.97
-- Target: Atmel ATF16V8B
--------------------------------------------------------------------------------
-- Mix board pbus address decoding GAL
-- first mix board
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x10         latch ena         out
--------------------------------------------------------------------------------

ENTITY mix_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
data_ena_out    :  OUT bit;
                nsense_out      :  OUT bit
             );

-- force part and pinout
ATTRIBUTE part_name of mix_decode:entity is "C16V8";
ATTRIBUTE pin_numbers of mix_decode:entity is
  "nread_write_in:1 nlatch_in:2 addx_in(0):3 addx_in(1):4
           addx_in(2):5 addx_in(3):6 addx_in(4):7 addx_in(5):8
           addx_in(6):9 nsense_out:12 data_ena_out:13";

END mix_decode;
```

```
ARCHITECTURE behavior OF mix_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0010000" THEN

-- pbus 0x10

nsense_out <= '0';     -- pbus nsense ACTIVE
data_ena_out <= '1';   -- data latch ena ACTIVE

ELSE

-- not active

nsense_out <= '1';     -- pbus nsense ACTIVE
data_ena_out <= '0';   -- data latch ena INACTIVE

END IF;
END PROCESS;
END behavior;
```

The pinout for the mixer module decode GAL is shown below:

```
                    C16V8A
                  ---------------
 nread_write_in =| 1|       |20|* not used
      nlatch_in =| 2|       |19|* not used
      addx_in_0 =| 3|       |18|* not used
      addx_in_1 =| 4|       |17|* not used
      addx_in_2 =| 5|       |16|* not used
      addx_in_3 =| 6|       |15|* not used
      addx_in_4 =| 7|       |14|* not used
      addx_in_5 =| 8|       |13|= data_ena_out
      addx_in_6 =| 9|       |12|= nsense_out
       not used *|10|       |11|* not used
                  ---------------
```

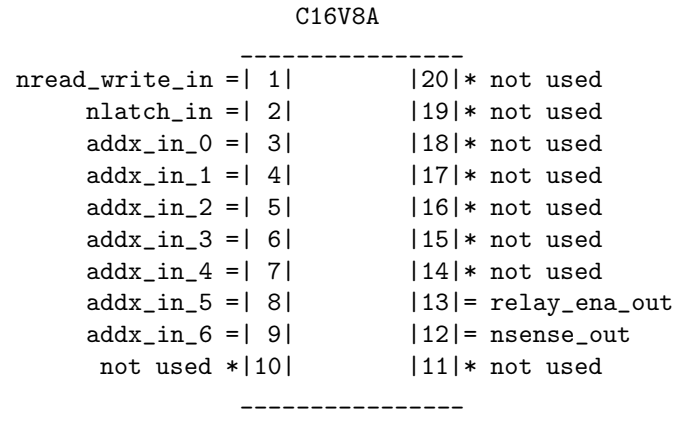
A.2.3 Bus Combiner Module

The VHDL code used to synthesize the logic for the bus combiner address decode GAL is shown below:

```
--------------------------------------------------------------------------------
-- DACS : Distributed Audio Control System
--
-- Copyright (C) 1997 Stephen Scott Richardson
--------------------------------------------------------------------------------
--   File: buscomb.vhd
--   Date: 03.13.97
-- Target: Atmel ATF16V8B
```

```
--------------------------------------------------------------------------------
-- Bus combiner pbus address decoding GAL
--
-- pbus addx    function          dir (from uC)
-- ================================================
-- 0x30         latch ena         out
--------------------------------------------------------------------------------

ENTITY buscomb_decode IS
PORT (
addx_in         :  IN bit_vector (6 DOWNTO 0);
nlatch_in :  IN bit;
nread_write_in :  IN bit;
relay_ena_out   :  OUT bit;
                nsense_out      :  OUT bit
            );

-- force part and pinout
ATTRIBUTE part_name of buscomb_decode:entity is "C16V8";
ATTRIBUTE pin_numbers of buscomb_decode:entity is
  "nread_write_in:1 nlatch_in:2 addx_in(0):3 addx_in(1):4
           addx_in(2):5 addx_in(3):6 addx_in(4):7 addx_in(5):8
           addx_in(6):9 nsense_out:12 relay_ena_out:13";

END buscomb_decode;


ARCHITECTURE behavior OF buscomb_decode IS
BEGIN
PROCESS (nlatch_in, addx_in, nread_write_in)
BEGIN
IF nlatch_in = '0' AND nread_write_in = '1'
   AND addx_in = "0110000" THEN

-- pbus 0x30

nsense_out <= '0';     -- pbus nsense ACTIVE
relay_ena_out <= '1';  -- data latch ena ACTIVE

ELSE

-- not active

nsense_out <= '1';     -- pbus nsense ACTIVE
relay_ena_out <= '0';  -- data latch ena INACTIVE

END IF;
END PROCESS;
END behavior;
```

The pinout for the bus combiner decode GAL is shown below:


C16V8A
nread_write_in =| 1|
nlatch_in =| 2|
addx_in_0 =| 3|
addx_in_1 =| 4|
addx_in_2 =| 5|
addx_in_3 =| 6|
addx_in_4 =| 7|
addx_in_5 =| 8|
addx_in_6 =| 9|
not used *|10|
|20|* not used
|19|* not used
|18|* not used
|17|* not used
|16|* not used
|15|* not used
|14|* not used
|13|= relay_ena_out
|12|= nsense_out
|11|* not used

B Axiom MC68HC11 Single Board Computer References

Memory Range	Size	Function
0xFFFF-0xE000	8K	Program EEPROM
0xDFFF-0xB800	10.2K	ROM, RAM or EEPROM
0xB7FF-0xB600	512 bytes	68HC11 Internal EEPROM
0xB5FF-0xB5F8	8 bytes	R65C51 ACIA 0xB5FC-0xB5FF not used 0xB5FB control register 0xB5FA command register 0xB5F9 status register 0xB5F8 data register
0xB5F7-0xB5F4	4 bytes	82C55 ports 0xB5F7 control register 0xB5F6 port c 0xB5F5 port b 0xB5F4 port a
0xB5F3-0xB5F0	4 bytes	LCD interface 0xB5F2-0xB5F3 not used 0xB5F1 data register 0xB5F0 command register
0xB5EF-0xB5E0	16 bytes	CS6 (onboard addx decoding)
0xB5DF-0xB5D0	16 bytes	CS5
0xB5CF-0xB5C0	16 bytes	CS4
0xB5BF-0xB5B0	16 bytes	CS3
0xB5AF-0xB5A0	16 bytes	CS2
0xB59F-0xB590	16 bytes	CS1
0xB58F-0xB580	16 bytes	CS0 (mixer Pbus controller) 0xB584-0xB58F not used 0xB583 control port 0xB582 port C 0xB581 port B 0xB580 port A
0xB57F-0x8000	13.7K	ROM, RAM or EEPROM
0x7FFF-0x1040	28.6K	RAM

Table 11: Axiom CMD-11A8 MC68HC11-based single board computer, memory map (1/2).

0x103F-0x1000	64 bytes	68HC11 internal registers 0x103F CONFIG 0x103D INIT 0x103C HPRIO 0x1039 OPTION 0x102F SCDR 0x102E SCSR2 0x102D SCCR2 0x102C SCCR1 0x102B BAUD 0x1029 SPSR 0x1028 SPCR 0x1026 PACTL 0x1025 TFLG2 0x1024 TMSK2 0x1020 TCTL1 0x100F TCNT 0x100E TCNT 0x100D OC1D 0x100C OC1M 0x100B CFORC 0x100A PORTE 0x1009 DDRD 0x1008 PORTD 0x1007 DDRC 0x1005 PORTCL 0x1004 PORTB 0x1003 PORTC 0x1002 PIOC 0x1000 PORTA
0x0FFF-0x00FF	3.8K	RAM
0x00FE-0x0000	256 bytes	68HC11 internal RAM

Table 12: Axiom CMD-11A8 MC68HC11-based single board computer, memory map (2/2).

Memory Range	Size	Function
0xFFFF-0xE000	8K	Program EEPROM
0xDFFF-0xB800	10.2K	external memory select
0xB7FF-0xB600	512 bytes	68HC11 Internal EEPROM
0xB5FF-0xB5F4	12 bytes	CS7
0xB5F3-0xB5F0	4 bytes	LCD interface 0xB5F2-0xB5F3 not used 0xB5F1 data register 0xB5F0 command register
0xB5EF-0xB5E0	16 bytes	CS6 (onboard addx decoding)
0xB5DF-0xB5D0	16 bytes	CS5
0xB5CF-0xB5C0	16 bytes	CS4
0xB5BF-0xB5B0	16 bytes	CS3
0xB5AF-0xB5A0	16 bytes	CS2
0xB59F-0xB590	16 bytes	CS1 (graphics LCD) 0xB592-0xB59F not used 0xB591 command register 0xB590 data register
0xB58F-0xB580	16 bytes	CS0 (board Pbus controller) 0xB584-0xB58F not used 0xB583 control port 0xB582 port C 0xB581 port B 0xB580 port A
0xB57F-0x8000	13.7K	external memory select
0x7FFF-0x6000	8K	RAM (mirror)
0x2000-0x3FFF	8K	RAM (mirror)

Table 13: Axiom CMM-11A8 MC68HC11-based single board computer, memory map (1/2).

0x103F-0x1000	64 bytes	68HC11 internal registers 0x103F CONFIG 0x103D INIT 0x103C HPRIO 0x1039 OPTION 0x102F SCDR 0x102E SCSR2 0x102D SCCR2 0x102C SCCR1 0x102B BAUD 0x1029 SPSR 0x1028 SPCR 0x1026 PACTL 0x1025 TFLG2 0x1024 TMSK2 0x1020 TCTL1 0x100F TCNT 0x100E TCNT 0x100D OC1D 0x100C OC1M 0x100B CFORC 0x100A PORTE 0x1009 DDRD 0x1008 PORTD 0x1007 DDRC 0x1005 PORTCL 0x1004 PORTB 0x1003 PORTC 0x1002 PIOC 0x1000 PORTA
0x0FFF-0x00FF	3.8K	Not Used
0x00FE-0x0000	256 bytes	68HC11 internal RAM

Table 14: Axiom CMM-11A8 MC68HC11-based single board computer, memory map (2/2).

C References

C.1 Books

1. Comer, Douglas E. and David L. Stevens. *Internetworking with TCP/IP Volume III.* Prentice Hall, New Jersey, 1993.
2. Herniter, Marc E. *Schematic Capture with PSpice.* Merrill/Macmillan College Publishing, New York, 1994.
3. Horowitz, Paul and Winfield Hill. *The Art of Electronics, second edition.* Cambridge University Press, New York, 1989.
4. Pellerin, David and Michael Holley. *Practical Design Using Programmable Logic.* Prentice Hall, New Jersey, 1991.
5. Schneiderman, Ben. *Designing the User Interface, Second Edition.* Addison-Wesley, Massachusetts, 1992.
6. Stevens, W. Richard. *Advanced Programming in the Unix Environment.* Addison-Wesley, Massachusetts, 1992.
7. Toth, Viktor. *Visual C++ 4 UNLEASHED.* Sams Publishing, Indiana, 1996.
8. Triebel, Walter A. and Avtar Singh. *The 8088 and 8086 Microprocessors.* Prentice Hall, New Jersey, 1991.

C.2 Data Books and Data Sheets

1. Axiom Manufacturing, *CMD-11A8 HC11 Single Board Computer.* Axiom Manufacturing, Texas, 1995.
2. Axiom Manufacturing, *CMM-11A8 68HC11 Single Board Computer.* Axiom Manufacturing, Texas, 1995.
3. Cypress Semiconductor, *Warp VHDL Reference Manual.* Cypress Semiconductor, California, 1995.
4. Hitachi, *HD44780 Dot Matrix LCD Controller.* Hitachi.
5. ImageCraft, *ICC11 68HC11 C Compiler and Development Environment Version 4.0.* ImageCraft, California, 1997.
6. Motorola, *MC68HC11A8 Technical Data*, Motorola.
7. National Semiconductor, *CMOS Logic Databook, 1988 Edition.* National Semiconductor, California, 1988.
8. Toshiba Corporation, *TLX-711A-E0 Dot Matrix LCD Module Data Sheet.* Toshiba Corporation, Japan, 1988.

C.3 Web Sites

1. Analog Devices, `http://www.analog.com/`
2. Dallas Semiconductor, `http://www.dalsemi.com/`
3. National Semiconductor, `http://www.national.com/`
4. NE5532 data, `http://www.pre.com/News/airtip15.html`
5. Philips (Signetics), `http://www.philips.com/`
6. XForms, `http://bragg.phys.uwm.edu/xforms/`