The Design and Implementation of an I/O Controller for the 386EX Evaluation Board

by

Marcus-Alan Gilbert

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology
May 1995

Author [illegible]
Department of Electrical Engineering and Computer Science
May 19, 1995

Certified by [illegible]
Richard D. Thornton
Supervisor

Accepted by [illegible]
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

The Design and Implementation of an I/O Controller for the 386EX Evaluation Board

by

Marcus-Alan Gilbert

Submitted to the
Department of Electrical Engineering and Computer Science

May 18, 1995

ABSTRACT

This report describes an extension of the development system for MIT's Microprocessor Design Lab, course 6.115. The current development system for 6.115 is the 386 EX evaluation board. The current development system has been extended by having the 68HC11 microcontroller control peripheral devices. The original intention was to develop such a system around the PowerPC line of processors, however the availability of the 386EX evaluation board, and its adoption into 6.115 led to its being the choice for the extension. The results of this initial development work are the creation of a microprocessor based system that is capable of processing data directly from an analog to digital converter under 68HC11's control. The software development has been done in C, 80x86 assembly, Max-Forth, and 68HC11 assembly. This report also includes sections that extend the prototype system to a PowerPC implementation.

Thesis Supervisor: Richard D. Thornton
Title: Professor Electrical Engineering and Computer Science

Table of Contents

List of Illustrations

Tables

Code Samples

Notational Convention Guide

Most of the text in this document will appear in the default text font however there will be certain instances where this is not the case. These include:

Register Names and Signal Names will appear in caps such as, REMAPCONFIG, AN1.
Active low signals will have the '/' character. For example, /WE.

Data values and memory addresses will use this font `0xFF`.
Hexadecimal numbers will appear as `0xdd`, where d is a hex digit.
Binary Numbers will be pre-pended with `0b` as in `0b1110`.

References to the World Wide Web will appear in italicized font,
http://www.powerpc.com.
When a World Wide Web Site appears in the reference section the following format will be used.
<Author's last name, Author's first name (if known)>. <Title as it appears at the top of the browser window> <*URL*>,<Year of last known modification>.

Usenet newsgroups will be designated by the following font: `comp.arch.embedded`

Source code will appear as such: `for(i=0; i<7;i++)`
When specific code variables are referred to in the text they appear in italicized font,
x_buffer.
Data types will appear as **`unsigned long`**

1 Introduction

This report describes current work toward a hybrid development system based on MIT's Microprocessor Project Lab, 6.115. The current development work focuses on extending the current development system, which is centered around the 386EX processor. The purpose is to use the microcontroller to pass data to the processor so that it can execute a data processing algorithm.

The current development system for 6.115 utilizes the 386 EX evaluation board. This has been a substantial improvement over the previous HP64000 system. The current system provides a contemporary development platform from which students can learn to use microprocessors. Current development has concentrated on a design objective that attempted to accomplish one goal, to create a system that is suitably generic to allow different host processors to be utilized in development. The first of these goals has been realized with the advent of the new development system. However, much of the work this term has focused on the second of these goals.

To create a system that is suitably generic to allow different host processors to be utilized, we have chosen to use the following approach. We have a low-cost microcontroller to handle I/O and data collection and have the main processor process the data (see figure 1). The original intent was to have the host processor be from the Power PC family. With the Power PC, we have a processor that is suitably fast to execute DSP algorithms comparably to DSP processors. However, because of its availability and it design for embedded applications, the 386 EX was chosen as the host processor. The 68HC11 provides the peripheral micro-controller functions for the new

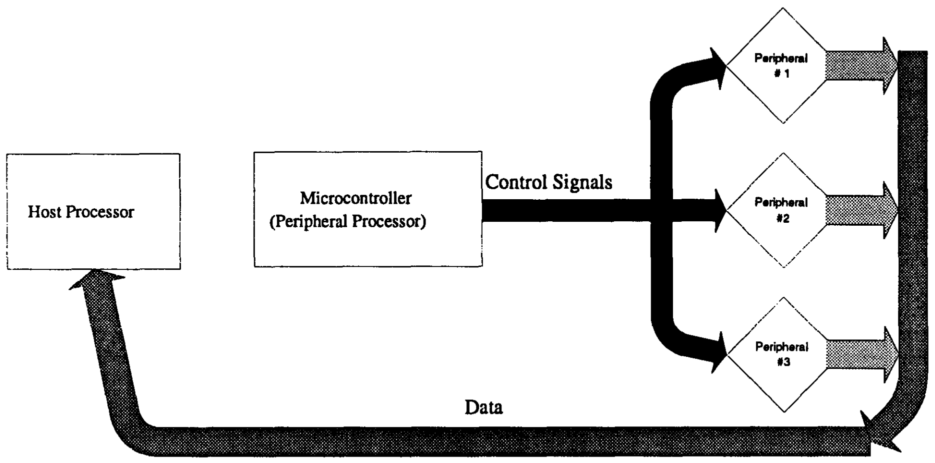


Figure 1: A Microcontroller Controlling Data Flow to the Host processor

development system. The 68HC11 was chosen for the abundance of software and cross development tools available. It was also chosen for the availability of on chip peripherals including RAM, ROM, asynchronous ports, and an Analog to Digital Converter (A/D). The 68HC11 Evaluation Board (EVB) that was used in the prototype development system contains an RS-232c level converter, several connectors, a small proto-board in addition to glue logic and system memory.

Originally there were supposed to be two modes of operation: code load, and data acquisition. In code load mode the peripheral micro-controller programs the host processors' ROM with instructions downloaded from the PC (see figure 2a). The host PC would connect to the 68HC11 via the comm port. With the advent of cross-development tools available for the PC it is feasible that the user can write and compile the program on the PC and then transfer the Intel Hex file from the PC to the development system even if the target is of a different processor family[1]. This mode does not appear in the current development system because of the code size necessary to hold the HEX file and calculate its checksum. (See System Enhancements)

[1] Alternatively, one could use a development system that has the same processing core as the target system. The main advantage to this being that the user need not worry about compatibility across platforms, and further run-time errors could be substantially reduced by running the program on the host computer.

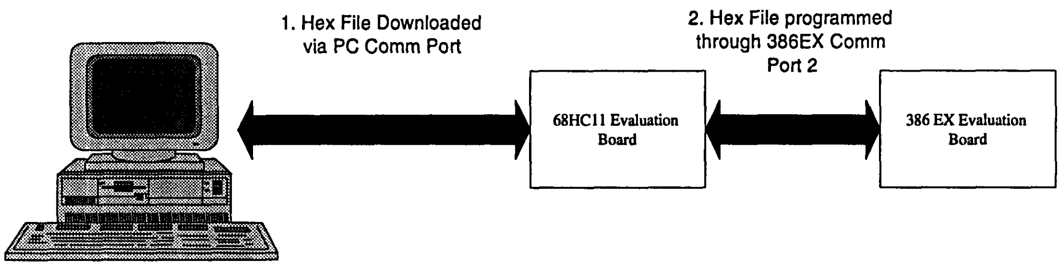

1. Hex File Downloaded via PC Comm Port
2. Hex File programmed through 386EX Comm Port 2
68HC11 Evaluation Board
386 EX Evaluation Board

Figure 2a: Mode 1, Code Load

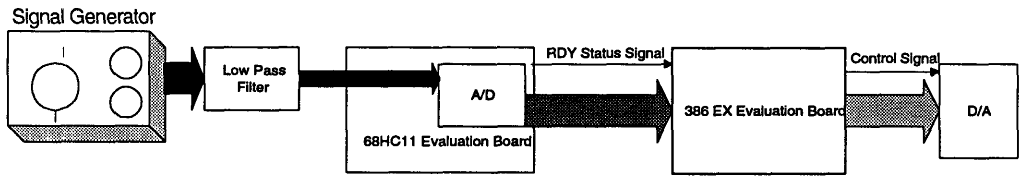


Figure 2b: Mode 2, Data Acquisition

In the data acquisition mode, the peripheral microcontroller provides data for the host microprocessor to manipulate (see figure 2b). The 68HC11 connects to the 386 EX evaluation board via the 386 EX expansion bus. It is conceivable that the host microcontroller would access other processors in a similar manner. For this implementation the data source is the Analog to Digital Converter contained within the 68HC11. The current design is meant to be flexible enough to allow the utilization of other peripherals and not meant to be restricted to the A/D. After the 386 EX has processed the data it sends the calculated output to the AD 7847 Digital to Analog Converter.

The following section further describes some of the background leading to the creation of the prototype system. Section 3, describes the operation of the 68HC11 evaluation board, section 4 details the anti-aliasing low pass filter. Section 5 discusses the implementation of the 386EX and its evaluation board and the operation of the 7847 under the control of the 386 EX, section 6 highlights the software written for the system and explains the DSP algorithms used to test the development system. Section 7 suggests possible alternatives and enhancements to the development system including those implementations that use the PowerPC 601. Section 8 mentions other microcontrolled applications in both industry and academia.

2 Background

The previous system under use in 6.115 lab was the HP-64000-UX Microprocessor Development Environment system. The system included the 68000 in-circuit emulators and diskless, networked HP workstations. The performance was suspect to say the least, although development was done the proven architecture of the 68000, the supporting workstations were slow and suffered from intermittent crashes. In addition, the former system was archaic and costly to maintain. The emulation pods were beginning to show signs of wear from repeated use. Also under the former system it was necessary for students to hand wire the peripheral components, which led to wasted time correcting wiring mistakes. This time could be better spent doing useful hardware and software development. In addition to wasting time doing mindless hand wiring, students were not exposed to microprocessors of contemporary design (e.g., 68030, '040, 386). The 68000 has been a formidable design that has been around for well over a decade, however students in the 6.115 class were not being exposed to current development tools and more recently designed microprocessor architectures[2]. At the end of 1993 it was clear that a solution utilizing current hardware was necessary. [1]

Although microprocessor designs and development tools have changed rapidly in the recent years, the objectives of the course have remained the same. The current and former objectives of 6.115 are, as reported in the Course plan: Hardware Engineering, Software Engineering, and Technical Skills. The current development system, the 386 EX evaluation board allows students to satisfy these goals while relieving students of the tedious tasks involved with hand wiring a system. The specific details of such a newer system are currently being investigated, however one of the essential

[2] It should be noted however, that the experiences with the HP development system are the result of our use of an out of date system and not a reflection of HP's inadequacies in the microprocessor development arena. In fact a newer version of the 64000 system has been proported to be more than adequate for development purposes. See the industry reference section.

components of an autonomous system is the capability to download programs from various sites, presumably PCs.

With this background, development work during the spring of 1994 utilized the existing HP64000 system to create a microprocessor based EPROM programmer capable of running programs downloaded from the PC. The evaluation system consisted of three sub-systems, the Microprocessor and lower memory, the multi-function peripheral, and the upper (EPROM) memory. The distinction between upper and lower memory is that lower memory contained instructions and data necessary to program the upper memory EPROM. Lower memory also contained the necessary vector to begin instruction execution at the upper EPROM's (memory mapped) address. It was originally planned to extend such a system from a 68000 based system to a 68040 (and possibly beyond) to even a 80x86 based system capable of downloading and subsequently running programs in embedded applications. [2]

During the fall of 1994, the microprocessor design lab received 386 EX evaluation boards and several Pentium systems. With the presence of an up to date development platform the design objective had to be changed from one of the construction of an immediate system to an objective that seeks to have an extendible architecture that can evolve to include newer microprocessors as the appear.

According to B. Furht and the W.A Halang in *A Survey of Real Time Systems* the next generation of real-time systems will be based on open systems. They assert:

> Open real-time computer concepts are based on hardware architectures which use off-the-shelf standard microprocessors, standard real-time operating systems, standard communications protocols, and standard busses.

With this perspective in mind, the author has sought to create, and illustrate how one might design a generic system. [3]

There still exists a need to continually improve the microprocessor project lab with current processors and allow for an open architecture for other processors. To this end, we contemplated doing development work with the PowerPC line of RISC architectures. The current set of RISC architectures and modern microprocessors can execute DSP algorithms suitably fast. Modern CISC architectures are also reported to perform native signal processing. [4][5] The first of the PowerPC processors, the 601, was released in 1992. There is beginning to be a abundance of development tools for the 601 (see Alternative Equipment). Specifically, the author would have liked to obtain the 601 System Design Kit with its associated peripherals. Unfortunately, time and finances did not allow us to use such a system. Future development work, should be aimed at obtaining such a system. [6] [7]

The focus here has been to extend the 386 EX evaluation board to a system in which a peripheral processor controls I/O, in the form of signal data. There are a large number of systems that utilize such an approach as to use one or more microcontrollers for peripheral processing and to utilize a more powerful CPU for core processing. It is hoped that the 386EX- 68HC11 system can be further developed into other systems such as a PowerPC-68HC11 system.

3 The Peripheral Processor

The peripheral processor section consists of a 68HC11 evaluation board and two LS244 Octal buffers. The 68HC11 evaluation board contains the necessary level converters and peripheral chips to allow connection to the PC (via a DB9 connector and serial cable) and to 386 EX expansion bus. The LS244 buffer provides the signal strength to drive data onto the 386 EX expansion bus.

3 .1 68HC11 EVALUATION BOARD - NMIX/T-0020 68HC11

The NMIX/T-0020 68HC11 evaluation board contains the 68HC11, RAM, level converters, glue logic, and a prototyping area. It also contains connectors for an LCD display and a 4x5 keypad interface. The board was purchased from New Micros, Inc. (see 68HC11 FAQ). The board has three 28 pin JEDEC (Joint Electronic Devices Engineering Council) sockets, and several connectors. The connectors that were used in this implementation were J6, and J5 (see schematics and figure 3).

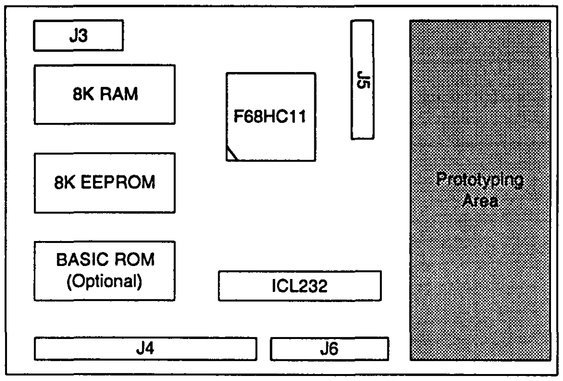


Figure 3: The NMIX-0020 Evaluation Board

The J6 connector is used for serial communication between the host PC and the 68HC11 evaluation board. The J6 connector is attached to the DB9 connector as shown in figure 4. Pins 5 and 6 provide serial out and serial in, respectively. Pin 8 provides the case and electrical grounds. The Clear to Send (CTS) and Request to Send (RTS) signals from the PC are tied together. The Data Terminal Ready (DTR) and Data Select Ready (DSR) from the PC are also tied together (see PC section).

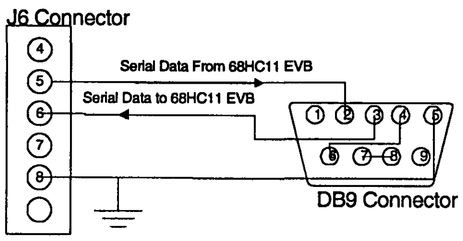


Figure 4: DB9 to J6 Connections

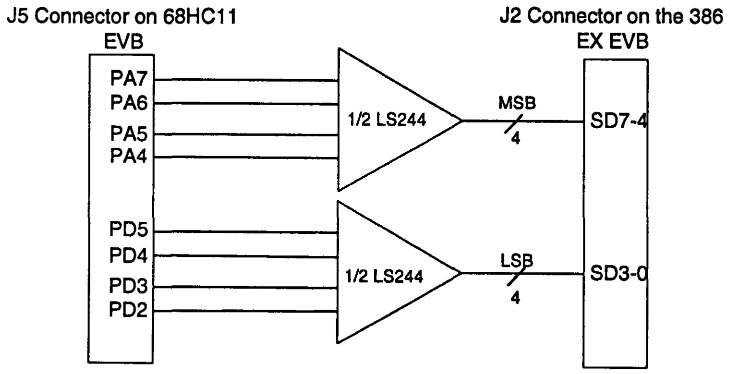


Figure 5: The J5 to LS244 to J2 Connections

The J5 connector provides data for the 386 EX evaluation board. Three of the 68HC11's Parallel ports are connected to the J5 connector. These parallel ports can be used as either inputs or outputs. These include Ports A, D, and E. Port D and A are used to provide data for the 386 EX evaluation board, in data acquisition mode. The connector pins 3-6 on J5 (which represent Port D) connect to the expansion bus of the 386 EX and provide the lower nibble of data. Connector pins 7, 8, 12, 13 , which represent Port A bit 3, 4, 6, and 7 respectively, form the higher nibble of data (see figure 5). Port A also provides the status signal for the indication that an A/D conversion has completed and the sample is ready. Port A pin 3 provides the active high, RDY signal. Port E pin 1, of the J5 provides the analog input signal for the internal A/D.

The output signals from the 68HC11 can only sink 1.6 μA to ground while preventing the package pin from rising above logical level zero, or .4 V. Similarly the 68HC11 can only source .8 μA at the logical one value of 4.5 V. These current levels are only sufficient enough to drive one TTL gate. Consequently an LS244 octal buffer has been added to provide adequate signal levels for the 386 EX evaluation boards.

Level conversion from the High Density Complementary Metal Oxide Semiconductor (HCMOS) signals to TTL levels is accomplished via the ICL232. The ICL232 provides the receiver/ transmitter pair for the serial in and serial out for connection to the PC. There is an additional receiver/transmitter pair that can be utilized for the serial communication (for Programming the Flash Memory) of the 386 EX EVB. For RS232C communication the logical one is represented by a -3V to -15V signal, conversely a logical zero is represented by a 3V to 15V signal. The ICL232 V_{dd} and V_{ss} supply voltages are 5V and 0V respectively, the higher voltages for RS232 communication are generated via an internal charge pump.

Glue logic provides the chip selects for the three 28 pin JEDECS. The three 28 Pin JEDECS occupy slots U3, U4, U5. Theses slots are used to provide the additional RAM, EPROM, and ROM respectively. The chip selects for the sockets are generated by the 74HC138. When the jumpers A and B are in the 8K position, all three address lines are brought to the decoder. This means that each of the eight generated chip selects represent a single 8k byte segment out of the 64K memory map. When jumpers A and B are in the 16K position, address lines A15 and A14 are brought to the decoder. The A13 address line is held high. This means that the upper four generated chip selects represent a single 16K byte segment out of the 64K byte memory map. When jumpers A and B are in the 32K position, address line A15 alone controls the part. The A14 and A13 lines are held high. Each of the two upper chip selects represent a 32K byte segments out of the 64K memory map. [8]

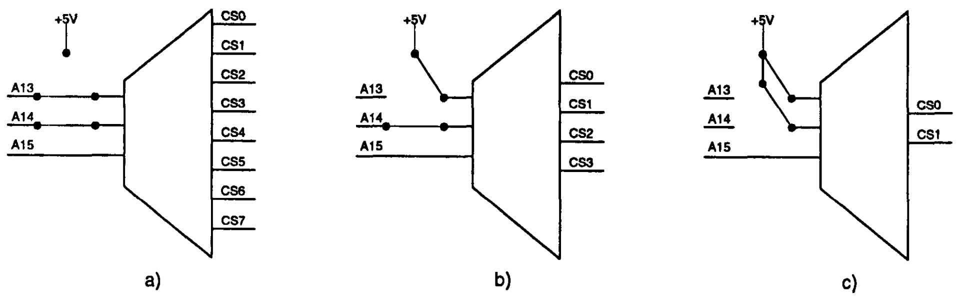


Figure 6: Effective Decoder Selections, with Jumpers A and B in: a) 8 K b) 16 K c) 32 K position

There are two other signals that control the decoder: Address Strobe (/AS) and the on board memory Disable (/MEMDIS). The /AS signal must be active low before any chip selects are enabled. This is the processor's signal indicating the address on the bus is valid for the off-chip memory. The /MEMDIS signal allows an off-board open collector source to disable the on board decoder, so off board components can usurp a memory segment from on board memory, even if the entire 64K is filled with RAM on the main board.

For this implementation, the default 8K addressing scheme was utilized, which required A and B jumper settings to be set in the 8K configuration. It was decided that the Forth language would be used, thus nullifying the need for external ROM memory. This configuration allows a user program to reside in EEPROM memory.

The 68HC11 EVB communicates with the PC through the comm port as shown in see figure 7.

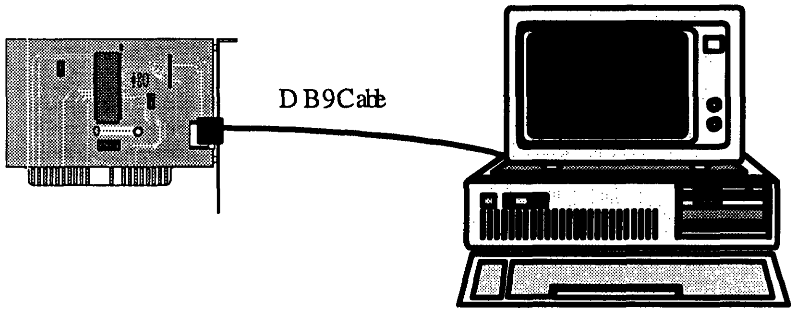


Figure 7: DB9 Connector attachment from PC to Eval Board

Signal from DB9	Abbreviation	DB9 Pin #	J6 Pin #
Data Carrier Detect	DCD	1	
Receive Data	Rx	2	7
Transmit Data	Tx	3	6
Data Terminal Ready	DTR	4	
Signal Ground	GND	5	8
Data Set Ready	DSR	6	
Request To Send	RTS	7	
Clear To Send	CTS	8	
No Connection	NC	9	--

Table 1: DB9 to J6 Connections and Signal Names

3 .2 THE 68HC11 MICROCONTROLLER

The 68HC11 microcontroller was chosen as a host processor because of its simple instruction set its availability of on-board peripherals, specifically an on-board analog to digital converter (A/D). There is also an abundance of hardware and software development tools available for this microcontroller. The 68HC11 is an 8-bit data, 16-bit address microcontroller. The instruction set is similar to earlier 68xx designs. Depending on the flavor the 68HC11 (denoted by the suffix or prefix, i.e. 68HC11<u>F1</u>) there are a variety of on chip peripherals that are available. These include EEPROM/OTPROM (One Time programmable read only memories) , RAM, digital I/O, timers, synchronous and asynchronous communication channels and an A/D converter. The board from New Micros uses the F68HC11FN. The 68HC11 Frequently Asked Questions (FAQ) describes the different flavors of the 68HC11. The F68HC11FN was produced by Motorola specifically for New Micros. The F68HC11FN is functionally equivalent to the 68HC11E9. This flavor of the 68HC11 contains 8 Kbytes ROM which contain the Max-Forth Kernel, 256 bytes of internal RAM, an 8 bit 8 channel A/D, 1 synchronous serial port, and 1 asynchronous serial port. [9]

There are four modes of operation of the 68HC11, these include: single chip, expanded, multiplexed, special bootstrap and special test. Single-chip mode allows the microcontroller to run with out external data paths. The New Micros Board has configured the 68HC11 to run in expanded, multiplexed mode. This mode allows the microcontroller to access off-chip devices. It provides an external data and address bus. The other two modes are special variations on the first two modes.

3 .2.1 The 68HC11 Functional Units

The functional units of the 68HC11 include: the Synchronous Peripheral Interface (SPI), the Serial Communications Interface (SCI), the A/D, EEPROM, the Parallel ports and the Central Processing Unit. The SPI provides the protocols necessary for synchronous communication with the peripherals. The Serial Communications Interface could provide the signals for serial communication with the 386 EX during code load mode and is also used by the Max-Forth Kernel. The parallel ports provide the data and control signals for the 386 EX during data acquisition mode. The A/D digital converter converts the continuous time input waveform to discrete time digital signals. The Central Processing Unit would provides the arithmetic ability necessary for the calculation of the HEX file checksum during the code load mode, and executes the instructions necessary to control the A/D during data acquisition mode. The following sections provide a brief overview of each functional unit and its associated registers.

3 .2.2 The Synchronous Serial Peripheral interface Unit (SPI)

The synchronous serial peripheral interface is one of the two serial communications channels on the 68HC11. The subsystem allows the 68HC11 to communicate with synchronous peripherals. The SPI provides fully duplex transfers as data can be both shifted in and shifted out on the two different serial lines. For this implementation, the SPI is not currently used. However, the SPI unit could be used to transfer data from a synchronous serial A/D such as the Analog Devices 7869 (see Section 7). This would utilize the MISO (Master In Slave Out) and MOSI (Master Out Slave In) pins which are also the Port D pins PD3 and PD2. The SCK and /SS pins which are pin PD0 and PD1, would also be used in this sort of implementation.

The serial peripheral unit is basically a shift register. The SPI synchronous clock register (SCK) controls how the bits are shifted out at what phase relative to the clock . There are four different combinations of the serial clock phase and polarity. These include the four combinations of clock high/low and polarity/zero. If clock high is selected, the data is transmitted in (out) on the low to high transition of SCK, conversely if clock low is selected data is transmitted on the high to low transition. The polarity designation has to do with the placement of the bits relative to the assertion of the /SS signal.

3 .2.3 The Serial Communications Unit (SCI)

The serial communications unit provides asynchronous communications with the 68HC11. The SCI is a fully duplex UART, that uses NRZ (non-return to zero) format. The NRZ format utilizes one start bit, eight or nine data bits and one stop bit. The synchronous communications unit, is controlled by five registers: BAUD, SCCR1, SCCR2, SCSR, and SCDR. Port D lines 1 and 0 provide the transmit (Tx) and receive (Rx) signals respectively, for the serial communication unit.

The BAUD register selects the baud rate for asynchronous communications and contains a couple of bits for factory testing.

The SCCR1, or SCI control register 1 has three bits associated with the 9 bit data transmit format. The WAKE bit selects in which manner the receiver will wake up. The remaining 4 bits are reserved and have no function.

The SCCR2, SCI control register 2, contains most of the SCI controls. The most significant four bits control the enabling of interrupts for the SCI interrupts. An interrupt can occur when a transmission buffer is empty, a transmission completes, when the receive buffer has been filled or the Rx line has become idle. Idle is defined as the line staying at a logic level 'high' for the duration of the transmission.

The SCI status register provides information with regard to conditions in the reciever/transmitter. The SCI status register tells whether or not: the transmit data register is empty, if the transmission has completed, if the receive data register is full, an idle line has been detected, or whether data has overrun the receive register. Framing errors and noise detection are also reported in the status register.

The SCI data register is a dual function register that provides a location to write data to be transmitted and read data to be received. Writing to this particular address, 0xB024, accesses the Transmit Data register, whereas a read to this address accesses the Receive Data register.

The Max-Forth kernel uses the SCI for communication with the PC. It enables the SCI and sets the Baud Rate to 9600 baud. It is therefore necessary for the I/O routines for the Max-Forth Kernel to be remapped in order to serially communicate with the 386 EX in mode 1. It is also necessary to reset the baud rate to 19200. (See section 7)

3 .2.4 The Analog to Digital Converter

The 8-bit, 8-channel Analog to digital converter on the 68HC11 provides the data for the 386 EX to process. The A/D is a successive approximation type. The A/D timing is synchronized with either the system E clock or with the internal oscillator. The A/D conversion takes about 32 E clocks. This implementation uses the E clock with a frequency of 2 MHz. At that clock speed the A/D has a sampling frequency of $\frac{2MHz}{32} =$ 62.5 kHz. By the Nyquist criterion, this yields a maximum input signal frequency of 31.25 kHz. To ensure that high frequency signals do not enter the A/D, the input signal is placed on the sampling channel, AN1 after passing through an anti-aliasing low pass filter (see Anti-Aliasing Filter Section). The Analog to Digital converter is controlled by the A/D control/status register, ADCTL. The ADCTL, selects the mode of operation for the

A/D, selects the channel(s) to be used for conversion, and indicates whether or not a conversion has completed. The operation of the A/D is also effected by the OPTION register. The outputs of the A/D are contained in registers ADR3-0.

The are two different scan modes, and two different channel modes in which the A/D can operate. The first scan mode is continuous in which the A/D continually performs conversions. The second scan mode, is single. In this mode the A/D performs four conversions and then sets the conversion complete flag. No other conversions are performed until the A/D control register is written. The first channel mode is single. In single channel mode conversions are made only from a single channel. In multiple channel mode the conversions are taken from a group of 4 different channels.

In the current implementation, the A/D runs in continuous mode and the data output registers ADR3-ADR0 are read successively to obtain the signal values. The data signal is represented as a unsigned fractional result of a comparison with V_{RH} and V_{RL}. A signal whose voltage of equal value to V_{RH}, would yield a result of 0xFF in one of the ADR registers. Similarly, a signal whose voltage measured the same as V_{RL} would yield a result of 0x00 in the ADR registers. On the NMIX-0020 board, the V_{RH} signal is connected to V_{dd} (5V) and the V_{RL} signal is connected to ground. After a signal has been sampled the output value from the ADR registers is passed on to the Port D and Port A data bus (see following Section).

The OPTION register also affects the operation of the A/D. The option register enables or disables the A/D circuitry. This register also selects whether the internal oscillator or the system E clock will be used. In addition this register can select whether or not to have a delay after the A/D starts converting. This delay is used to allow the internal oscillator to stabilize. However this system uses the external E clock and the internal oscillator is not used.

3 .2.5 The Parallel Ports

The 68HC11 has 40 I/O pins, grouped as five 8-bit ports. These five ports have multiplexed functionality. This functionality includes, timers, serial channel, A/D functionality, as well as general purpose input/output port functions. The functionality of these pins is determined by a register block. At start up this register block is located at address 0x1000, however the Max-Forth operating system remaps this to address 0xB000. There are nine registers that control the five ports (A-E), they reside in locations 0xB000-0xB00A .

The first port, Port A can be used for general purpose I/O or for use with the timer as a combination of input compares, output captures and a pulse accumulation. When Port A functions as general I/O, the first 3 pins PA0-2 are inputs only, while pins PA4-6 are dedicated outputs. Pin 7 and Pin 3 can be configured as either an inputs or an outputs. The values of these port pins can be accessed by reading the Port A register. Similarly the values on the Port A outputs can be altered by writing to the Port A register. For, this implementation Pin 7 and Pin 3 were configured as outputs.

The Port A pins connected to the J5 connector. The output pins PA7, PA6, PA5, PA4, and provide the higher nibble of the byte wide result from the A/D. PA3, provides the RDY signal for the 386EX to acknowledge (see figure 5).

The next port, Port B can have all of its pins set as general purpose outputs. However in the expanded/multiplexed mode of operations the 68HC11 uses this port for the high byte of the address bus. Consequently Port B is not connected to any of the port connectors and is not used for general purpose I/O.

The pins of Port C, are capable of being configured to be either inputs or outputs. The direction of these pins in controlled by the Data Direction register, DDRC. In expaned/multiplexed mode Port C takes on the multiplexed role of either the low byte of the address bus or it provides the data bus. Therefore, Port C is also not used for general purpose I/O.

Port D can be used for either general purpose I/O or it can be used for asynchronous or synchronous serial communications. The direction of the 6 lower bits is determined by the Port D data direction register, DDDR. The upper two bits of Port D are used for handshaking control for Ports B and C, these bits are unavailable for general I/O. These pins are not connected to J5. The lower two bits PD1, and PD0 provide the transmit (Tx) and receive (Rx) signals respectively for the serial control interface. Because Max-Forth uses the SCI for its I/O routines when communicating with the terminal, the two bits PD1 and PD0 are also unavailable for general I/O, however they are connected to J5. In this implementation the remaining 4 bits, PD2-PD5 provide 4 bits of data for the sampled value obtained from the A/D.

Port E pins provide either analog or digital inputs to the 68HC11. These pins can function as either input channels for the internal A/D or they can be used as general purpose digital inputs which can be read by accessing the Port E register. All of the Port E pins are connected to J5. The current prototype system uses Port E pin 1 (also known as AN1), for the analog signal input to the A/D.

3 .2.6 The Central Processing Unit

The 68HC11 CPU includes two 8 bit general purpose registers (A & B) that can be grouped into one 16 bit accumulator. It also includes two 16 bit index registers, which are used to access memory in index addressing mode. There is also a stack pointer, a program counter and a condition code register. Each instruction consists of an 8-bit opcode followed by one or more bytes of address/data information. The instruction set of the 68HC11 includes bit manipulation instructions, 16 bit divide instructions, and exchange instructions.

The A/D access routine was coded in 68HC11 assembler to take advantage of the performance increase from using assembly language and forego the overhead associated with using Max-Forth. (See software section) [10]

3 .3 THE PC CONNECTION

The PC connects with the ICL232 Level converter via a DB9 connector and cable (See figure 7). The cable connects to the serial port[3] of the host PC. A non-handshaking protocol, in which the CTS (Clear to Send) and RTS (Request to Send) are tied together, has been chosen to transfer data from the PC to the 68HC11 EVB. In this protocol, DTD (Data Terminal Detect), DTR (Data Terminal Ready), and DSR (Data Set

[3] Although this implementation used the 9 Pin RS232 port, it is also possible to use the 25 Pin connection with the appropriate DB25 connector.

Ready) are also tied together. The Max-Forth Operating System does not make use of these signals. The integrity and validity of the data can be verified by the Max-Forth system response.

The program commands were transferred from the PC through the comm port using the Maxtalk communications package. The command line would appear like this:

```
        maxtalk2 program.4th
```

However, any communications package which is capable of uploading and downloading files and can transfer data at 9600 baud will be sufficient for communications with the EVB. The above example shows how Maxtalk uploads files. Maxtalk can also save a log of the terminal session by using the following redirection:

```
        maxtalk2 > program.log
```

There are a couple of more things to note however. The communications package used must have the ability to wait for each echoed character and be able to wait at the end of line for the 'Line Feed' character (ASCII 0x0A or Ctrl-J)[11]

4 Analog Anti-Aliasing Low Pass Filter

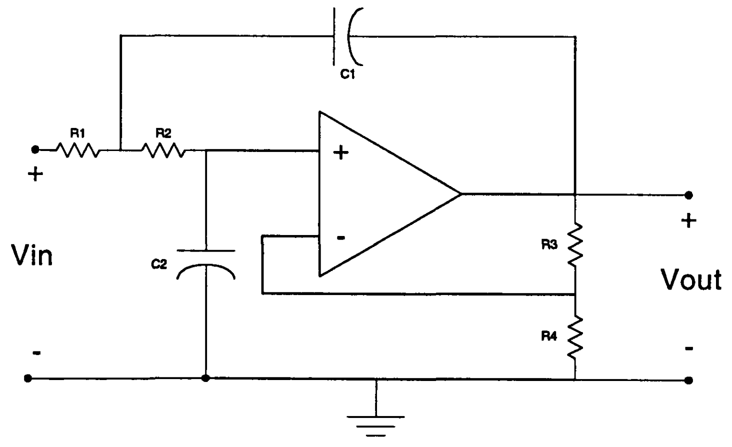


Figure 8 : The Sallen-Key Circuit

In order to prevent high frequency components present in the analog signal from aliasing the sampled signal, it is necessary to construct a low pass filter. This filter consists of a Sallen Key circuit as shown in figure 8. This is a type of second order Butterworth filter.

This unit gain low pass filter has a cutoff frequency, f_c of 7 kHz, when the passive elements have the following values: C1=C2=.015μF, R3=R4=7kΩ, R2=2.4kΩ, R1=1kΩ. 7 kHz was chosen as the cutoff frequency because the A/D (as it is run in this implementation) can only sample signals up to 7 kHz without aliasing. The active element in the circuit is the low noise, low offset voltage operational amplifier (OP27). It is suggested, however due to the variability of these resistor values that 10K and 1 K Pots be used to tune the filter. The passive element values were chosen using the techniques of Hilburn and Johnson. [12]

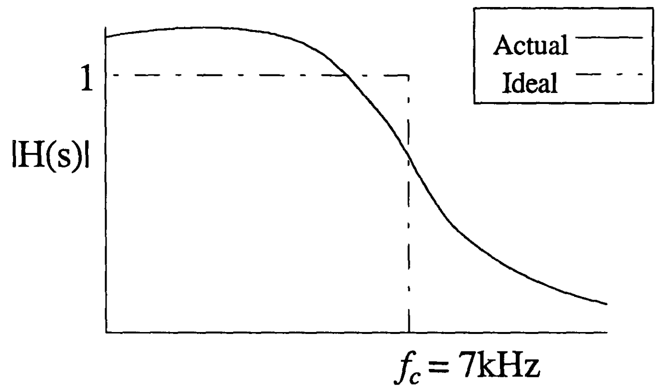


Figure 9 : Desired Frequency Response

5 The Host Processor Section

The host processor sections consists of the Intel 386 EX evaluation board from Intel and the 7847 Digital to Analog converter from Analog Devices. The 386 EX evaluation board includes memory, peripheral headers, programmable logic devices, in addition to the 386 EX microprocessor. (See figure 10) An overview of the major components used in the implementation will be given, for a more detailed discussion of the functional units please see the 386 EX hardware reference and the 386 EX evaluation board manual. [13][14]

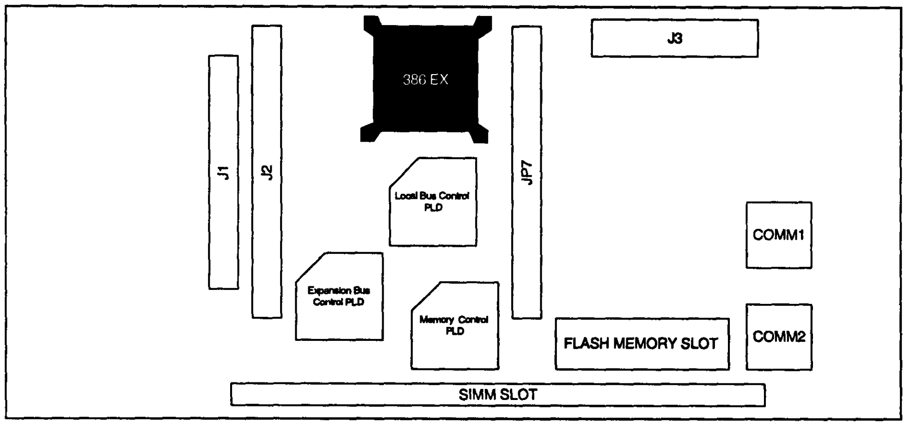


Figure 10: The 386 EX Board Functional Units

The Evaluation Board

5 .1 MEMORY

There are three different types of memory on the 386 EX Evaluation Board. These include SRAM, DRAM, and flash memory. The system SRAM is 32 Kbytes. The primary purpose use of the SRAM is to support the System Management Mode (See 386 EX Section). The SRAM can be used for other purposes, however care must be taken to ensure that the SRAM does not lie in the address space for the system DRAM. In this implementation neither the System Management Mode nor the SRAM is used.

The DRAM memory occupies slot US4 with a single sided 1 Mbyte SIMM. Additional Memory can be added by using single sided SIMMs with larger capacity, e.g. a 1 Mbyte x 32 SIMM and 4 Mbyte x 32 SIMM would increase the DRAM memory size to 4 Mbytes and 16 Mbytes respectively. The user must select single sided SIMMs as double sided SIMMs require two additional row address strobes, RAS that the board does not support. Access to the DRAM is provided by two wait states at the 25 MHz Operation.

The Flash memory on the evaluation board consists of the 512 Kbyte Flash 28F400 BX Memory. Access to the device is controlled by two chip selects /UCS and /C6. /UCS is used to access the entire 512 Kbyte array, while /C6 can only access the lower 256 Kbytes of the array. The 28400 BX is operated in a 16 bit wide (word) configuration by tying its /BYTE line high and tying its address A0 input to the 386 EX's A1 output. Access to the FLASH is also set for 2 wait states at 25 MHz operation.

5 .1.1 Programming Flash Memory

The Flash 28F400BX has the capability to be in-socket programmed. This means that it can be programmed from the comm port. This capability is provided by programming the 'boot block' section with a program that sets up the serial ports and provides instructions for programming the EPROM. The boot block is a special section of the EPROM that is write protected and cannot be in-socket programmed. The port 1.5 pin (/PRG_EN) controls the voltage on the V_{pp} pin of the EPROM. When /PRG_EN is active (low) the Voltage on the V_{pp} goes to 12 V. The RAM_WE signal which is connected to the write enable pin (/WE) on the EPROM also controls the writing of the EPROM.

The Flash Memory actually consists of seven separately erasable and programmable blocks, including the 16K write protected boot block. The Command User Interface (CUI) provides the interface between the microprocessor and the internal flash memory. Program commands and be executed by writing to the CUI. In addition to the /WE signal there is also /OE, /CE, and /BYTE. The /OE signal allows the driving of output signals. The chip enable signal activates the internal logic, decoders, and sense amplifiers. The /BYTE signal indicates whether or not the Flash operates in byte wide mode. On the evaluation board the BYTE signal is wired high indicating that the FLASH is configured for 16 bit wide operation. In addition to these signals the /RP reset/ deep power down signal controls whether or not the any block EPROM can be written to, and it also locks the boot block. On the evaluation board, this signal is wired to +12V indicating that the boot block is write protected. For more information about the 28F400 Flash memory refer to its data sheet. [15]

5 .2 PERIPHERAL CONNECTORS

The peripheral connectors on the evaluation board include two 32x2 D type connectors, J2 and JP7, and one 20x2 connector, J1. The J1 and J2 connectors are of the PC/104 standard. One of the larger connectors, JP7 is used to allow the connection of the D/A to

Parallel Ports 1 and 3, similarly the other large connector J2, provides the lower byte of the expansion bus and is connected to the Port A and D lines from the 68HC11 (see schematics or figure 5).

5 .3 PC/104 BUS

The PC/104 bus has been designed for embedded applications that allow a high degree of compatibility with existing Industry Standard Architecture (ISA) bus designs. The J1 and J2 connectors have the same logic signals as the ISA architecture present on the PC/AT. The arrangement of the connectors as shown on figure 10 occupies less space and consumes less power than the AT ISA implementation, and allows the cards to be stacked. This arrangement is geared towards embedded applications where power and space constraints are of concern.

5 .4 PROGRAMMABLE LOGIC DEVICES

The programmable logic devices provide control signals for the EVB and the 386 EX. There are 3 different PLDs. The Local Bus PLD controls the operation of the local bus, generation of the $\Phi 1$ clock signal, and generation of the data and expansion bus enables. The Memory Control PLD generates access signals for the Flash Memory and DRAM, including the /RAM_WE signal which connects to the Flash and the SRAM. The Expansion Bus PLD decodes memory and I/O accesses. It provides the correct strobes for these addresses that correspond to the expansion bus.

5 .5 ASYNCHRONOUS PORTS

There are two serial port connectors present on the 386 EX evaluation board. COM1 is a male DB9 connector that is configured for Data Carrier Equipment (DCE) . It has the RS232 functionality of a modem or serial port with all the status and control signals connected to the processor.

COM2, is a female DB9 connector that is configured for Data Terminal Equipment (DTE) operation. It has uses only two signals, Rx and Tx. The Rx and Tx lines have been swapped to allow communication with the PC or other peripheral device with a serial connection (see below). This port is used for debugging.

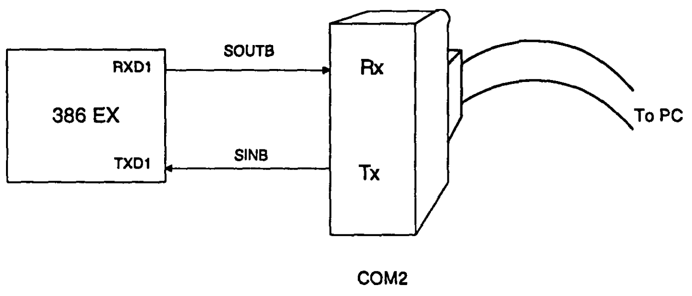


Figure 11 : DB9 with the Rx and Tx connections swapped.

5 .6 386 EX

The 386 EX consists of the 386 SX processing core and several on chip peripheral units that have been added to facilitate the creation of embedded applications. These include: the DMA Unit, Bus interface unit, Chip Select Unit, JTAG test Unit, Clock and Power Management Unit, DRAM Refresh Unit, Watchdog Timer Unit, Serial Communications Unit, Timer Unit, and the Interrupt Control Unit (ICU), as shown below. Design considerations may warrant the use of the Chip-select Unit, the DMA unit, ICU, or the Synchronous Serial I/O Unit (see Section 7). The Asynchronous Serial Unit is also used by the iRMX Operating System, for debugging purposes.

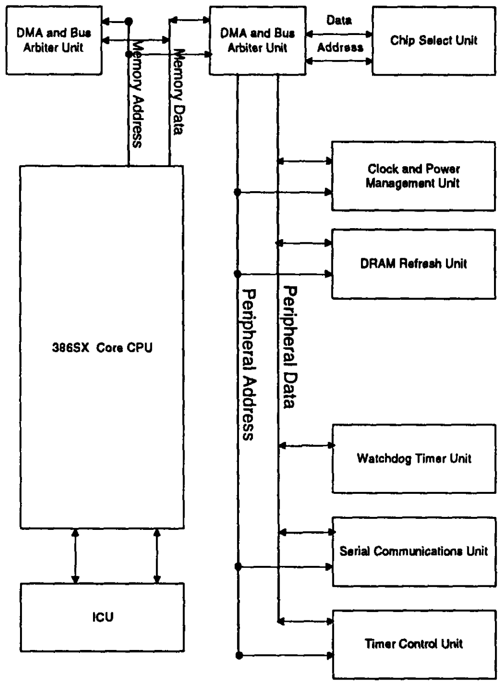


Figure 12 : The 386 EX Functional Components

5 .6.1 Parallel Ports

The 386 EX has 3 general purpose I/O ports, consisting of eight pins each. However, most of these I/O pins are multiplexed with other functions such as asynchronous serial I/O. In fact, the asynchronous communications clock, COMCLK is multiplexed with Port 3 pin 7. COMCLOCK connects to the 1.8432 MHz generator and cannot be used for general purpose I/O. In this implementation Port 3 pins 0-6 are used to provide the 7 bits of data for the Analog Devices 7847 Digital to Analog converter (D/A). Port 1 pin 7 provides the MSB of data to the D/A. After the byte of data has been placed on the input pins of the 7847, the data is latched by asserting the write signal. Port 1 pin 6 provides the write signal for the 7847.

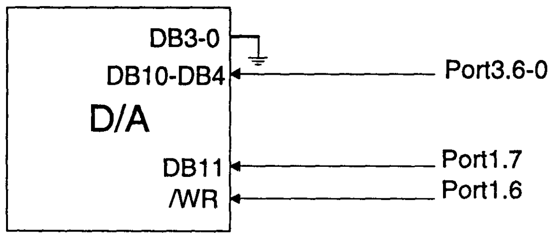


Figure 13: Parallel Port Connection to the D/A

5 .6.2 DMA Unit

The DMA unit has two channels, 0 and 1, which operate independently of each other. Each channel can function in one of several operating modes, and can transfer data of byte or word widths. The bus arbiter is also a part of the DMA unit. It receives requests from the DRAM refresh controller, and the external Bus Master, in addition to the two DMA channels.

The channel consists of three items: the Requester, the Target and the Byte count. These items are identified by the registers that define the memory or I/O device requesting DMA service (the Requester). The Requester is the only item that can initialize or end a DMA process. The Target (address) is the area of memory with which the requester wishes to exchange data. The Byte count indicates the amount of data to be transferred.

The DMA Unit includes features not found in the 8237 DMA family, however it can be configured to behave in a 8237 like manner.

5 .6.3 Chip Select Unit

The Chip Select Unit can generate up to eight different chip selects. A chip select is configured by assigning a series of addresses to a given chip select. These addresses are the higher order 11 bits of a 26 bit address, the lower 15 bits are masked out. Thus, each chip select is assigned a block of memory. Whenever the selected region of memory is referenced by the 386 EX address lines, the chip select for that region is asserted. This can be used to memory map I/O or other peripheral devices without the need for external glue logic.

5 .6.4 Interrupt Control Unit

The Interrupt Control Unit (ICU) consists of two cascaded 8259A modules. The ICU supports up two 16 interrupt requests (8 internal, 8 external). Interrupt requests are put in the Interrupt Request Register which contains one bit for each interrupt request signal. A corresponding bit is set for each interrupt request. If a conflict exists, the priority resolver chooses which request has the highest priority. The priority resolver controls the interrupt request line to the CPU. Additional interrupt requests can be serviced by the addition of externally cascaded 8259A's. In the original implementation the ICU was configured to service an interrupt initiated by the INT5 signal. However interrupt latency limited signal responses to 8 kHz, without invoking the DSP routine. A polling approach was subsequently chosen to initiate the transfer of data between the 68HC11 and the 386 EX processor board.

5 .6.5 The Analog Devices 7847

The 7847 is a 12 bit parallel D/A. It consists of two separate D/A's, channel A and B. The D/A has two ways of formatting the results. A digital conversion can have either a bi-polar 2's complement arrangement or the 12 bits of data can represent an unsigned number. Because the nature of the 68HC11 A/D is uni-polar, it has been chosen to operate the D/A in uni-polar mode as shown in figure 14. Also since the A/D in the 68HC11 is 8-bit wide the 7847 is in fact used as a 8-bit D/A with the four least significant bits grounded.

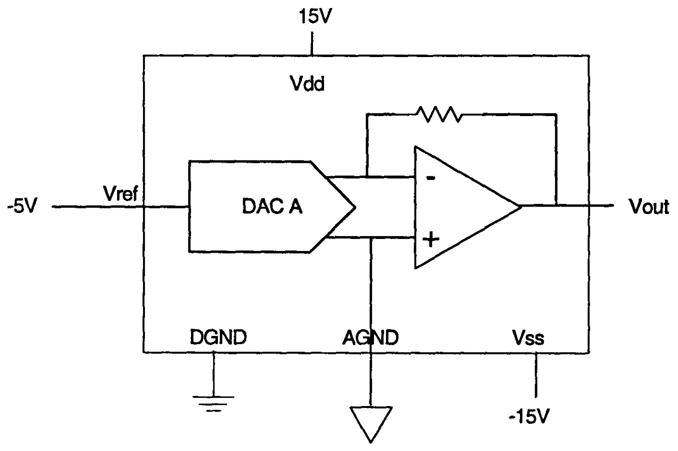


Figure 14: 7847 In Uni-polar Mode

Data can be latched in one of two ways. Either the /WR signal can be held low and a low to high transition on the corresponding Chip Select (/CSA or /CSB) will cause data to be latched. Similarly, the Chip select can be held low and a /WR signal transition from low to high will cause data to be latched, as shown below.

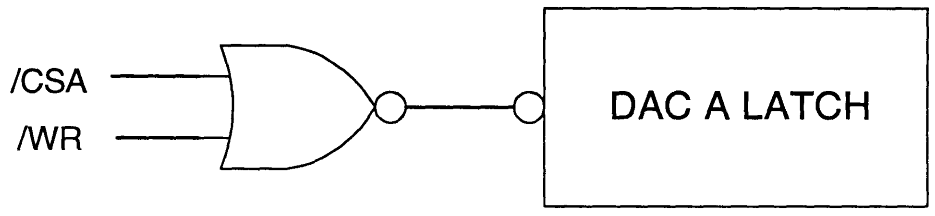


Figure 15: Latch Control Logic for the 7847

In this implementation, the chip selects for A or B are driven low, and a conversion is initiated by transitioning the write signal, /WR. At this time data is then latched, and a conversion is begun. The /WR signal is controlled by Port 1 pin 6 from the 386 EX as shown in figure 13. The bandwidth of the 7847, is 110 kHz. This is significantly higher than the A/D, however there is some latency involved with the 386EX and the separated parallel port interface. However the majority of the time between outputs is spent calculating the filtered output. In lab tests, with the split byte arrangement (the arrangement where the input data is spread across Ports 1 and 3 of the 386 EX) the signal could be changed at a frequency of 341 kHz. However, this was without the calculation of DSP algorithm, the true bandwidth of the system is much lower.

The V_{dd} and V_{ss} power supplies for the 7847 are +15 V and -15V respectively. For this implementation it was chosen to make V_{ref} -5V since the uni-polar mode of operation makes $V_{out} = \frac{<bit\_value>}{4096} \times V_{ref}$. This gives a signal with a range from 0 to 5V, which also matches the operation of the 68HC11's A/D. [16]

6 System Software and Algorithms

The system software for the 68HC11 performs register initialization functions. These functions include initializing the port I/O registers, and configuring the A/D on the 68HC11. These initialization functions are coded directly in Forth and subsequently interpreted by the resident Forth interpreter. The 68HC11 software must also read the samples from the A/D and send them to the parallel ports. However, because the timing of the section is critical, this software is coded in 68HC11 assembler.

The system software for the 386 performs two functions. The first of which will be to set up the necessary registers for the polling algorithm, and the I/O pins. The second function of the software will be to execute a Digital Signal Processing (DSP) algorithm. Both of these functions are programmed in C, with in-line assembler where necessary.

The DSP algorithm used to test the development system consists of three separate filters. After the A/D converter maps the signal to the discrete-time domain, the signal is filtered using one of three filters. The three filters consist of low, mid-band, and high pass filters. These originally consisted of 33-point finite impulse response (FIR) filters. The original low pass filter was the same filter used in the 6.115 class and the other filters were designed using Matlab. However, the processing time associated with the 33 point FIR filter was found to be to large and the filter size we first reduced to 27 points and subsequently to 13 points. The three (FIR) filters were then redesigned using Matlab.

6 .1 68HC11 ROUTINES

The 68HC11 from New Micros, contains the MaxForth Kernel in its ROM memory. The MaxForth kernel contains an interpreter to allow Forth commands to be typed in interactively or downloaded through the PC communications program. The Forth interpreter resident in the 68HC11 kernel is adequate for tasks which are not time critical, so the register initialization is written directly in Forth code.

Forth was created in the early '70s by Charles Moore after he became frustrated with conventional languages inability to allow control over a computer system that executed some real time task[4].

The Forth language differs from most languages in that it contains elements from both compilation and interpretation. Forth also combines elements from assembly and high level language. Forth differs from other languages in that its functions ('words') are defined based on definitions of other words. This is opposed to constructing new functions by simply listing other functions. Also Forth is a stack based language. Parameters passed to words are explicitly placed on the stack, in what has come to be known as post fix or reverse polish notation. Definitions can be in terms of other words or as we will see later, machine code. When defined in terms of other words these words in turn must be looked up to find the machine code that corresponds to that word. Forth jumps from one word definition to the next since the word definition consists essentially of the addresses of other words. [17] [18] [19]

When Max-Forth interacts with the terminal it operates in interpretive mode. The definitions of words can be interactively entered into the interpreter. However, after the definition of these words has been entered into the dictionary, the words can be subsequently used without the need to retype the definition.

The Forth programs written for this system perform two actions. First the programs perform register initializations and then they also collect data from the A/D and subsequently send this data across the Ports to the 386 EX.

The Forth register initialization code does four things. It sets Ports A and D to transfer the byte of data from the A/D, and it powers up and configures the A/D. The initial block of registers sits at 0x1000, however the Max-Forth kernel remaps these addresses to the 0xB000 offset. The initialization of the registers uses the `C!`, command. This command is a byte write to memory. The user first puts the byte value on the stack then puts the address to which the byte is to be written on the stack and then the user issues the `C!` command. For example, the following code selects single, continuous mode and channel AN1 for A/D operation:

```
        21 B030 C!
```

The A/D control register, ADCTL is located at address 0xB030 in the 68HC11 memory map. The Hex value of 0x21, indicates that bit positions 5 and 1 should be set. When bit

[4] This task incidentally was the control of Radio Telescopes

position 5 is set continuous mode is selected. The lower nibble value of 1 indicates that first channel should be used for conversion. Other register initializations use the same `C!` word command.

```
88 B026 C! ( Set Bits 7 and 3 to outputs)
FF B009 C! ( Set Port D 0-7 for outputs)
21 B030 C! ( Continuous, single channel set AN1)
80 B039 C! ( Turn on A/D use external clock)
```

Example 1: 68HC11 Register Initialization

For a more detailed discussion of the registers involved please refer to the Appendix section.

The performances requirements of A/D control routine necessitate that the routine be coded in 68HC11 assembler. The ADR1 register must be successively read to obtain the successive A/D values when running in continuous mode. Unfortunately, there is no way to know when one register contains valid data and when the next register is being filled. So the approach used is to get the data out of the first register and then wait until 64 cycles later to again get data out of the first register . This increases the sampling period by a factor of 4 from 16 μs to 64 μs, thus the maximum allowable input frequency is now $\frac{31.25}{4} = 7.8125$ kHz. It is desirable to obtain frequency samples at a rate that would correspond to the A/D bandwidth, however that rate at which samples appear on the port output lines has to be slow enough to allow the 386 EX to calculate the filtered response. In the original implementation, the four registers were read successively. However, because of the number of instructions in the acquisition loop exceeded thirty two, after several samples had been collected the acquisition routine would fall behind the A/D and the next sample read would actually reflect the incorrect sample. For example, suppose that when the A/D was on sample 10 and the acquisition routine was 4 samples behind the A/D, at sample 6. After a while, when the A/D is on sample 17 and the acquisition routine may be on sample 12. Thus the distance between the A/D has changed. If this happens three more times the values output from the acquisition routine will no longer represent sequential samples as the A/D has overlapped the acquisition routine and overwritten the next sample in the sequence.

In order to read these values at 128 clock intervals it is necessary to use 68HC11 assembler. Unfortunately, the Max-Forth kernel does not have an in-line assembler as do other popular implementations of Forth. For example consider the two following code segments:

```
CODE-SUB GA2D
        LDX #$B030
LABEL:  LDAA $01,X
        STAA $B000
        TAB
        ANDB $OF
        LSLB
        LSLB
        STAB $B008
        INCX
        XGDX
        ANDB $F3
        XGDX
        BRA LABEL
        RTS
END-CODE
```

```
CODE-SUB GA2D
  CE C, B030 ,    ( LDX
#$B030)
  A6 C, 01 C,     (
LABEL:LDAA $01,X)
  B7 C, B000 ,    ( STAA
$B000)
  16 C,           ( TAB)
  C4 C, 0F C,     ( ANDB
$OF)
  58 C,           ( LSLB)
  58 C,           ( LSLB)
  F7 C, B008 ,    ( STAB
$B008)
  08 C,           ( INCX)
  8F C,           ( XGDX)
  C4 C, F3 C,     ( ANDB
$F3)
  8F C,           ( XGDX)
  20 C, EC C,     ( BRA
LABEL)
  39 C,           ( RTS)
END-CODE
```

Example 2: A Forth word definition a) with a built in assembler b) without an assembler

With Max-Forth code the user must explicitly specify the op-codes involved with the assembler routine, as shown in Example 2B. In the original implementation the code appeared as above, however when the need for a down sampled A/D arouse the code had to be modified with a more time consuming acquisition loop. The current configuration pads the loop with store instructions.[5]

6 .2 386 EX REGISTER INITIALIZATION

In order to use the functional units on the 386 EX processor it is necessary to write to registers that initialize and configure the units. The first step in accomplishing this is to

[5] The STAA (store accumulator A) instruction was chosen as opposed to the standard NOP because it consumes more time (4 clocks as opposed to 2 for the NOP) , thus there are less instructions to be added. Less instructions added reduces the likelihood of making errors. See the code listings in the Appendix

enable the expanded I/O in the memory space. The expanded memory space contains all of the registers necessary for utilization of the parallel ports, interrupt control unit, and the DMA unit. In order to enable the expanded I/O space, the following assembly code is necessary :

```
    mov ax, 8000h /* This will be REMAPCONFIG */
    xchg al, ah
    out 23h, al   /* high byte to 23h */
    xchg al, ah
    out 22h, al     /* low byte to 22h  */
    out 22h, ax     /* Word to 22h */
```

Example 3: Enabling the Expanded I/O Addresses

The REMAPCONFIG register controls the access to the expanded I/O space. (See Appendix) Most of the code for the initialization of the registers was written in assembler by using the following series of commands:

```
    mov dx, <Port Address>
    mov  ax, <Port, value>
    out  ax, dx// Places the Word value contained in ax at
<Port Address>
```

The values can also be read out of the registers with the following instructions:

```
    mov dx, <Port Address>
    in ax,dx      // ax now contains the value at <Port
Address>
```

The first unit that we must set up is the parallel ports. In order to set up the parallel ports it is necessary to write to three registers, P*x*LTC, P*x*DIR, P*x*CFG, where *x* is a number from 1 to 3 indicating the port number. The P*x*CFG register controls whether or not the Port is controlled by an internal peripheral or configured for I/O mode and subsequently controlled by the P*x*LTC and P*x*DIR registers. The P*x*DIR register selects whether each pin on the port is an output or a high-impedance input. The P*x*LTC, data latch register controls the values driven to the pin. If the pin has been configured as an output, writing a one or a zero to the corresponding bit in the register changes the value on the pin accordingly.

In this implementation Port 3, pins 0-6 were selected as outputs. The author would have like to use pin 7 as well, however on the Evaluation Board pin 7 is connected to the COMCLOCK which is used for asynchronous communications, therefore it cannot be used for bi-directional I/O. Port 1 pins 6 and 7 were also used as outputs, and Port 2 pin 1 was used as an input. Pin 7 was used to drive the most significant data bit on the D/A. Pin 6 was used to drive the /WR line of the D/A. Because of the unavailability, of Port 3 pin 7, it is necessary to distribute the byte of information calculated by the DSP routine across Port 3 and Port 1. Also care must be taken to ensure that the Port 1 pin 6 bit does not get written at inopportune times because it controls the start of a conversion of the D/A. In order to accomplish this spread of the byte, the byte is written to the Port 3's data register, P3LTC, unaltered. Then the signal byte is ANDed with 0x80 and subsequently written to the P1LTC data register. This masks out the least significant 6 bits allowing the most significant bit of data to pass through. It also sends the Port 1, pin 6 bit low. The reader will recall that D/A conversion is begun on a low to high transition, thus it is necessary to have the /WR signal low before a conversion can be initiated. One would have to be cautious about transitioning the write signal and the most significant bit at the same time, however in lab tests the correct values were written to the D/A. This is due to the fact that the actual write of data occurs during the low to high transition of the /WR signal. The write is accomplished by sending the Port 1 pin 6 write signal high by ORing the value 0x40 with the value currently contained in the Port 1 data register. The OR is necessary because the value on the DB11 (Port 1 pin 7) should not change when the /WR signals transitions to the high value. This process continues as other bytes are written.

6 .3 THE DSP ALGORITHMS

The DSP algorithm used to test the development system consists of three separate filters. The A/D converter can map the signal to the discrete-time domain, with a sampling frequency of 62.5 kHz. By the Nyquist criterion, input signals up to 31.25 kHz can be successfully converted without aliasing. However, the calculation of the DSP algorithm takes processing time, this processing time limits the rate at which data can be presented to the 386 EX. Originally, it was thought that the 386 EX could compute the convolution in the 16 μs sample time. However experimentation revealed that the original C algorithm (See eval.c in the code listings) took 390 μs to compute an output. The C algorithm was abandoned and the routine coded in assembler and the size of the filter reduced to 27 points, this reduced the output calculation latency to 120 μs. It was then decided to further reduce the filter size to 13 points so that the latency time would be ~60 μs, which would correspond to the 64 μs reduced sampling period of the 68HC11. This resulted in a maximum input frequency of 7.8125 kHz.

The signal is filtered using one of three filters. The three filters consist of a low, band, and high pass filters. These are 13-point finite impulse response filters. The discrete time frequency responses are shown in figures 16 a-c, the cor responding continuous time domain frequency responses are shown in figure 17 a-c.

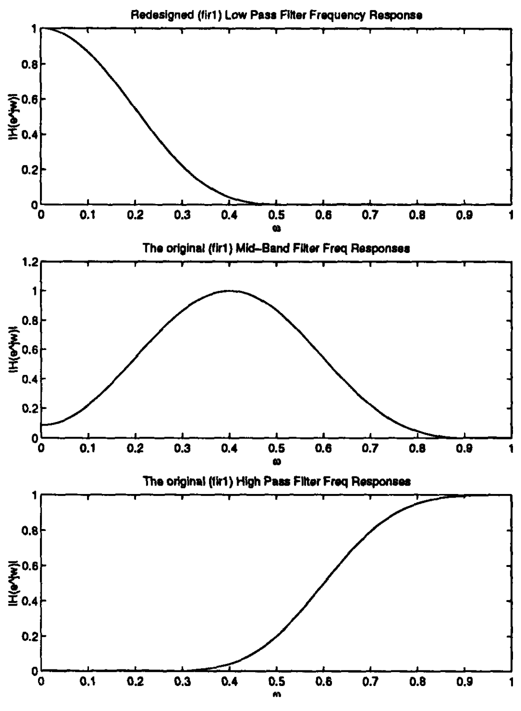


Figure 16 a-c Discrete Time Frequency Responses

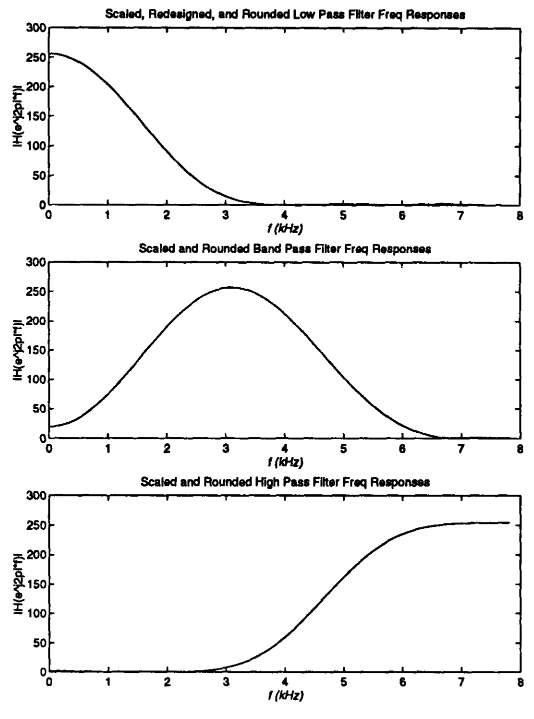


Figure 17 a-c Continuous Time Frequency Responses

The coefficients for most of these filters were designed using the Matlab function `fir1()` which creates a filter based on the Windowing algorithm. The original low pass filter was the filter used in 6.115 during the fall of 1994. As the reader will notice the low pass filter has a gain of .8, and passes signals up to .2 π (discrete time, 1.6 kHz continuous time) in frequency. The band pass filter passes filters from .2 to .6 π (1.6 to 4.6 kHz) with unity gain, and the high pass filter passes the .6 to π (4.6 to 7.8 kHz) frequencies with unity gain. The coefficients for these filters are all fractional values, which requires either a floating point algorithm or a fixed-point algorithm that includes scaling and rounding. The actual filter values are shown on the next page in figure 16 a-c.

[20] [2

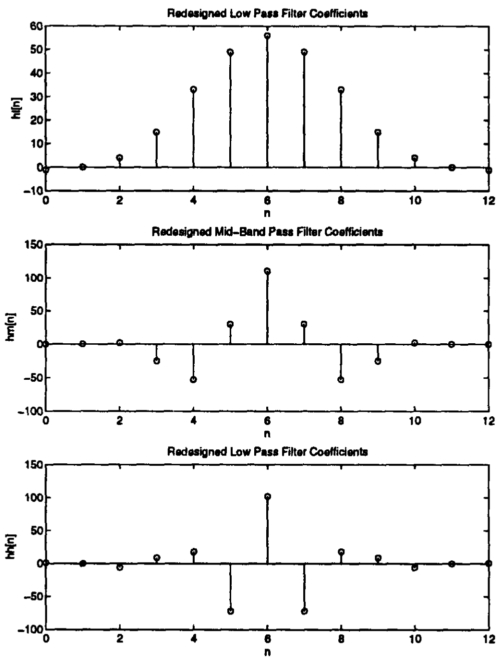


Figure 18: a-c Fixed Point Filter Coefficients

The first approach that was used was the floating point algorithm, in which the appropriate filter is chosen and its fractional coefficients are represented by a decimal value. Matlab has calculated these coefficients out to the fourth decimal place (the original low-pass filter has six significant digits). The fractional values are held in array of **floats**, which represent the filter values to the fourth decimal point, as shown below.

```
float hm[N] =
     { // floating pt. filter weights for mid-range filter
     -.0006,  .0000,  .0009,  -.0059,  -.0088,  .0030,
.0000,
      -.0058,  .0330,  .0440,  -.0138,   .0000,  .0250,
-.1504,
      -.2360,  .1145,  .3995,   .1145,  -.2360, -.1504,
.0250,
       .0000, -.0138,  .0440,   .0330,  -.0058,  .0000,
.0030,
      -.0088, -.0059,  .0009,   .0000, -.0006
};
```

Example 4: A Array of Floating point coefficients

The second approach used was to use a fixed point representation. With this approach, it was necessary to scale the filters by 256 (=2^8) thus making the fractional values greater than one. After the values were scaled they were then rounded to allow a representation of the values as discrete integers. The low-pass filter was redesigned because it originally had a higher number of significant digits. The original filter was redesigned using the same Windowing algorithm as the band and high pass filters. The resulting unity-gain filter was then scaled and rounded as the other filters. The scaled filter values are represented as an array of **shorts**, which represents the filter values as fixed-point quantities as shown below.

```
short hm[N] =
        {// fixed pt. filter weight factors for mid-band
filter
-1,    0,    1,   -6,   -9,   3,    0,   -6,   34,  45, -14,
  0,  26, -154, -242,  117,  409, 117, -242, -154,  26,
0,
-14,  45,   34,   -6,    0,    3,   -9,  -6,    1,    0,
-1};
```

Example 5: A Array of fixed point coefficients

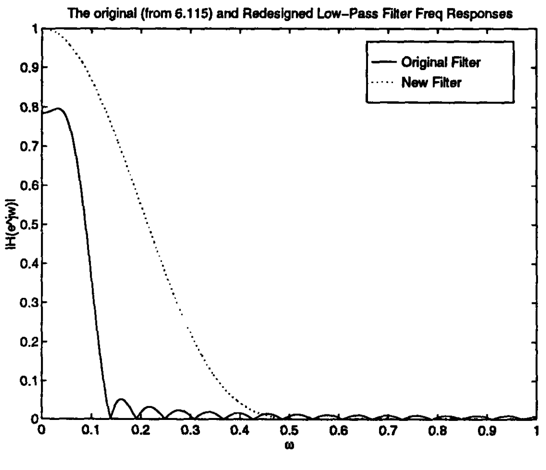


Figure 19: Redesigned Low Pass Filter Frequency Response

Fixed point was pursued as an alternative, because the 386 EX (whose CPU is functionally equivalent to the 386 SX) does not have a floating-point unit. As this is the case, it then becomes necessary to either have a non-floating point algorithm or to set a flag on the compiler to generate floating point emulation code. In either situation a certain degree of precision is lost, however using a floating point emulating code tends to increase code size, which would hamper performance. In the first few tests it took time 450 μs to calculate one filter output with the fixed-point algorithm. This subpar performance was from calculating a 33-point FIR and functional overhead. Also most of the code at this time was still in C, with only the D/A routine in assembler (this was later reduced to 350 μs at which time it was decided to abandon C). The floating point algorithm was only in DOS only tests because they would have added even more overhead to the DSP algorithm. It is doubted that it would have offered any performance improvement over fixed point as the EX has no native floating point unit. What the floating point algorithms did offer however, was provide a basis from which to judge the accuracy and completeness of the outputs from the fixed point algorithms. This suspect performance was the bottleneck of the system as the 68HC11 was producing A/D data at a rate of 16 μs. [22]

6 .3.1ACCOMPLISHING FILTERING THROUGH THE CONVOLUTION

The filtering was accomplished by calculating the convolution sum, $\Sigma x_n \times k_{T-n}$,for each output value. After the input filter value (represented by an **`unsigned char`**) has been passed into the function this value is cast into a signed **`short`** and goes into the *x_buffer*. Originally the *x_buffer* was an array of **`shorts`** that contained the 33 most recent filter values. Later the number of samples was reduced to 27, to decrease the amount of processing time spent calculating the output. And then subsequently reduced to further to 13. The newest value is in the 13th position and the oldest is in the 1st. For the fixed point algorithms the filter coefficients had to be scaled and rounded to be represented as integers. A scale factor of 256 was chosen to allow an integer representation. This value made it convenient to adjust the result after the convolution had been computed. After the necessary multiplies and adds have been accumulated, the value is divided by 256. However this not totally correct as a divide by 256 might not give the correct answer (see next section).

```
for (k = 32, acc = 0; k > -1; k--)
   {
switch(fil_typ)
     {
          case 'l':
               acc += hl[k] * x_buffer[k];
                   /* compute next output */
               break;
          case 'm':
               acc += hm[k] * x_buffer[k];
               break;
          case 'h':
               acc += hh[k] * x_buffer[k];
               break;
     }
```

Example 6: Accomplishing the Multiplies and Adds in the Convolution

6 .3.2 Correcting for Errors in the Output Signal

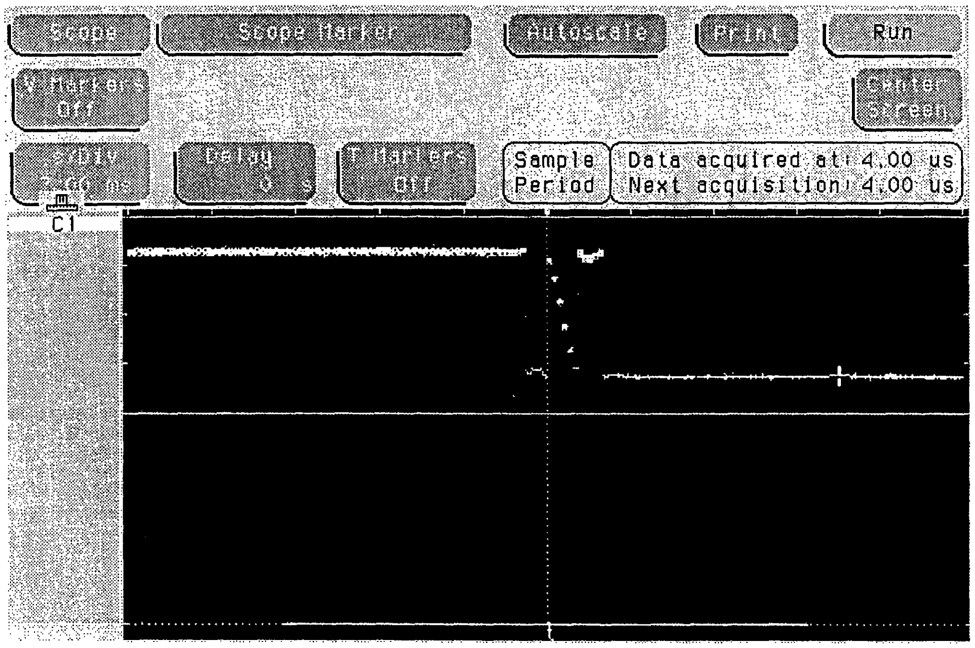
As was mentioned before the various filters in the previous sections filter different spectra contained within the bandlimited 7.8125 kHz signal. However after the filtering has been accomplished a few adjustments have to be made with regard to the outputs. First it must be ensured that the output filter values are unsigned integers with a value in the range from 0 to 255[6]. The output values must be formatted in this way because in the current configuration an unsigned 8 bit signal goes in to the filter and unsigned 8 bit signal comes out.[7] In the lowpass filter case this problem manifests itself in the form of over and

[6] This was a design decision. One could just as easily operated the 7847 D/A in bi-polar mode, however you would still have the dynamic range of 255 only your output values can now be signed values from -127 to 127. This is a bit confusing however, as the input signal does not directly correspond to the output signal. If this you choose to do this then you have to remember that the input DC signal value of 2.5V (0x77) when passed through the low pass filter would give an output value of zero.

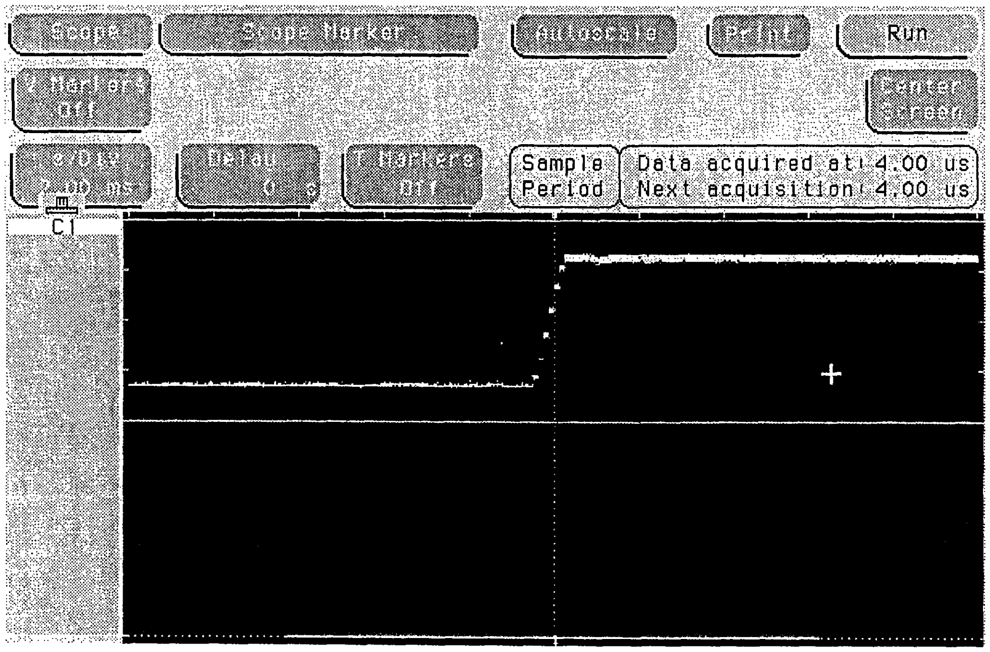
undershoot that appears within the signal, if left unchecked this can map the output signal into undesirable domains. The problem is that when overshoot or undershoot occurs the values held in the extended precision signed **`long`** variable gets erroneously truncated. For example if the variable value is a slightly negative number say -1 then it will get mapped into 255 as, -1 = 0b11111111111111111111111111 (32 1's in the binary representation) when truncated gets translated to 0b11111111 (an unsigned 8 bit binary representation of 255. Similar results happen with numbers greater than 255. The figure below illustrates the effects of under and over shoot with out preventative measures. To calculate the output from the low pass filter, the following algorithm is used. When the program detects that a **`long`** value is negative this filter output value is rounded to zero. When the program encounters a number greater than 255 the output is set to 255.

The mid band and high pass filters also suffer from these endpoint problems however because of the nature of the input signal they have to be handled differently. They have to handled it differently because the output **`long`** value values look drastically different from their lowpass counterparts. The input signal is unsigned which means it has an inherent DC component. After this signal passes through either of the midband and high pass filters the DC component has been removed. Thus the output signal has no offset and will have a signed value that will oscillate about zero. To maintain consistency between the input and output signals it is necessary to rectify this output value before it is passed to the D/A. Ordinarily, this can be accomplished by adding 127 to the result of filtering, however the mid and the band pass filters are also susceptible to overshoot and undershoot errors. It is therefore possible that the output filter value have a range beyond {127, -127}. When these values are greater than 127 and less than -127 they are rounded to 255 and 0 respectively.

[7] Alternatively you could have an extended precision D/A, with a larger supply voltage which would allow signals larger than 5 Volts to appear on the output. This would also require sending a block of data that was more than a byte wide.



a)



b)

Figure 20: a) An example of clipped over and undershoot b) Output after adjustment

6 .3.3 Testing the DSP Algorithms

In order to test the frequency responses, that were calculated by Matlab and entered into the DSP program, we had a series of test vectors. The first series of test vectors represented signals that alternated from -1 to 1, and consequently did not have a DC component. However the first test vector in that series was a vector of all ones, which is a DC signal. With these bi-polar and DC values the validity was tested for the filter values. At this time, floating point programs were being run directly as the software was being tested on the host PC, which was either a 486 DX2 or a Pentium which include a floating point unit. Thus, we could get decimal values in our results.

The next step was to present more realistic test vectors to the DSP engine. Since, the values would have come out of an uni-polar A/D, it was decided that the test vectors should reflect that fact. The values would also range from 0 to 255, and any fractional (less than one output) would need to be counted as zero. The test vectors were then changed to range from 0 to 255 and changed the program to handled output rounding. At this time, floating and fixed point representations were both utilized in the testing process. It was also necessary to modify the program to reflect the fact that we were doing filtering of signals that contained DC components. And further that this DC component should be present in the output signal.

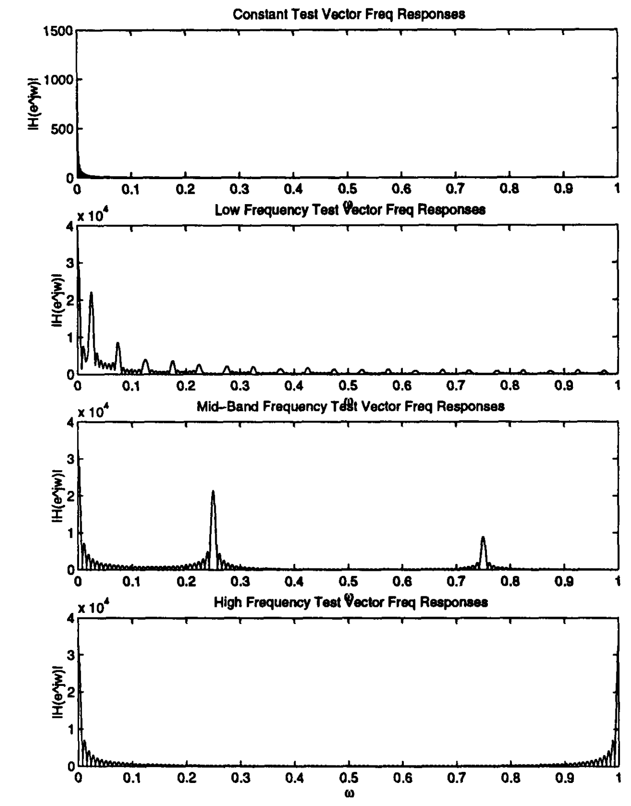


Figure 21: a-d The Frequency Responses of the Uni-polar (DC Added) Test Vectors

It then was then time to run the programs on the evaluation board. The first series of tests were run as MS-DOS programs. In order to run the program on the evaluation board it became necessary to remove the user interface functions such as `printf()` and `scanf()`. We also had to compile the floating point programs using floating point emulation. The input and output values could be accessed by using the Concurrent Sciences Softscope Debugger.

The Softscope Debugger is a MS-Windows based program that runs on the host machine and communicates with the target (386 EX EVB) board through the communications port. It allows for trace executions, breakpoints, memory dumping and other debugging functions. [C1]

After the algorithm results had been verified using the debugger, the D/A functions were added to allow the filtered values to appear as analog outputs. An oscilloscope was then used to verify the results of filtering.

The following tables show the different test programs used for debugging the DSP algorithm.

Test Name	Polarity of Test Vectors	Representation of Test Vectors	Representation of Filter Coefficients	Miscellaneous
fir1.c	Bi-Polar	Floating Point	Floating Point	Low Pass Filter only
fir2.c	Bi-Polar	Floating Point	Floating Point	All 3 Filters
fixfilt	Uni-Polar	Fixed Point	Fixed Point	Output scaling
floatfilt	Uni-Polar	Fixed Point	Floating Point	Required rounding

Table 2: Software Tests without Evaluation Board

Test Name	Polarity of Test Vectors	Test Vectors	Filter Coefficients	Miscellaneous
fireval.c	Bi-Polar	Floating Point	Floating Point	Same as fir1 without printf and scanf functions
eval2.c	---	---	---	D/A routines only
new.c	---	---	---	D/A routines only
evalwbrd.c	Uni-Polar	Fixed Point	Floating Point	Output scaling
eval.c	Uni-Polar	Fixed Point	Fixed Point	Required rounding*

Table 3: Software Tests with Evaluation Board

Test Name	Polarity of Test Vectors	Test Vectors	Filter Coefficients	Miscellaneous
lowpv2.c	Unipolar A/D samples	Fixed Point	Fixed Pt. 13 Pt FIR	Low Pass Filter Output adjustments*
midbpv2.c	UniPolar A/D samples	Fixed Point	Fixed Pt. 13 Pt. FIR	Mid-Band Filter Output adjustments*
highpv2.c	UniPolar A/D samples	Fixed Point	Fixed Pt. 13 Pt. FIR	High Pass Filter Output adjustments*

*Listing available in the Appendix

Table 4: Software Tests with Evaluation Board Connected to the 68HC11

6 .4 PUTTING IT ALL TOGETHER

After the DSP algorithms were debugged it was now time to add the functionality to accept signals from the 68HC11. This functionality includes accessing the expansion bus and polling the I/O lines for the assertion of the RDY signal.

In order to get the data at the correct times, it is necessary to configure one of the I/O ports[8] for input. This implementation uses a polling approach to detect the assertion of the RDY signal. The RDY signal connects to Port 2 pin 1. As shown in the figure below, the program waits until a zero is detected before searching for the one (RDY signal asserted). It does this to ensure that the current sample is the next sample in the sequence, not the same sample as before.

[8] Alternatively the high byte on the expansion bus could be used.

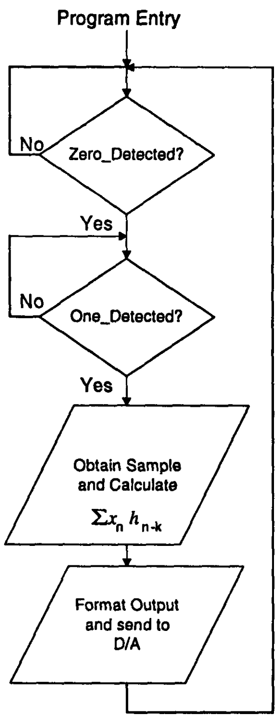


Figure 23: The 386 EX Program Flowchart

According to the 386 EX Evaluation board Manual in order to access the expansion bus it is necessary to access address 0x0100000 or higher. In order to generate this large address it is necessary to have this region of memory defined by a segment contained in the 386 EX global descriptor table (GDT). To accomplish this, the system call `rqe_create_discriptor` is called with the appropriate parameters to create an entry in the GDT. This function returns a selector, this selector is used to indicate a segment. When it is time to access data on the expansion bus the appropriate segment must be used, as the segment containing 0x0100000 is not a part of the code and data segments. To do this the ES register is loaded with the selector returned by the system call. The memory reference is then made by using an offset for that selector as follows:

```
mov al, es:0  // Get a byte of data at offset of the segment
```

The data is now ready to be read into the *x_buffer* and the resulting filtered output can be calculated and sent to the D/A. [23][24]

7 System Enhancements

This section suggests specific designs that could be beneficial to performance or architecture of the prototype system. It includes both specific components and alternative implementation strategies for an improved system.

7 .1 ANALOG TO DIGITAL AND DIGITAL TO ANALOG ENHANCEMENTS

The A/D converter in the 68HC11 is adequate but the use of a higher performance A/D might be beneficial to the overall design. The Analog Devices AD7869 and AD7870 both offer improved performance[9]. The 7869 features a synchronous serial interface that eliminates the need for parallel wires, and offers 83 kHz full power bandwidth. It is a complete A/D and D/A system. The 7869 offers improved 14 bit resolution. The A/D and D/A can operate independently of each other allowing access from different processors. The serial interface of the 7869 can be accessed in either asynchronous or synchronous manners.

The 7870 is a 12 bit A/D, that provides a 50 kHz full power bandwidth. It also offers the added versatility of being used in either a serial or parallel fashion.

Either of these configurations would necessarily require the use of control signals from the 68HC11. However, since the 68HC11 no longer needs to provide the byte wide sampling data the I/O port lines are free to provide control signals in either asynchronous or synchronous (with the use of the 68HC11's SPI) fashion. The choice of mixed signal components need not be restricted to these. The 7870 and 7869 are only two examples of possible implementations.

7 .2 PERIPHERAL PROCESSOR ENHANCEMENTS

Although its popularity and ease of use has made the 68HC11 the choice for the implementation of the development system, there are other microcontroller's that offer other advantages. These include the Motorola 68HC16 and the 8051 from Intel.

[9] The author does not intend to endorse Analog Devices' products, it is the intention of the author to illustrate alternative approaches. These two devices are products about which the author is familiar.

7 .2.1 The 68HC16

The Motorola 68HC16 is a 16 bit architecture that offers a full set of 16 bit instructions. It also features a higher resolution, higher performance, A/D, than its HC11 8 bit predecessor. The A/D can produce a result in 4 (2 MHz) clock periods which results in a 8 μs sample time. This results in a 62.5 kHz bandwidth. The 68HC16 also features a larger on-chip memory capacity. The HC16 offers increased performance with its instruction prefetch and pipelined execution. For DSP applications, the 68HC16 contains multiply and accumulate (MAC) registers and a 36 bit accumulator. Depending upon the application the 68HC16 might offer a good enough performance, not to merit the use of a host processor.

7 .2.2The 8051

The 8051 is an 8 bit microcontroller originally developed by Intel. It is now manufactured by different companies. Its popularity exceeds that of the 68HC11, well over 126 million units were shipped in 1993. Like the 68HC11 the 8051 comes in various flavors, with differing combinations of memory, general purpose I/O, and serial ports. Some variants of the 8051 operate at clock speeds of up to 40 MHz and perform 10 MIPs. There are two advantages to using the 8051. First, it handles interrupts efficiently and quickly. Secondly, it has a boolean processor which is beneficial in applications were bit control is needed. [25]

7 .2.3 The Need for Two Asynchronous Ports

In order to offer the ability to communicate directly with the PC and the Host processor at the same time, it is imperative that the peripheral processor have two asynchronous ports. The ports would be used simultaneously without the need for multiplexing the serial channel, which requires coordination from the host processor. With the added capacity offered by the extra port, the peripheral processor could conceivably be able to download the HEX or S19 file, perform checksums, and transmit the file to the peripheral processor without the need to wait until the entire file has been transmitted from the PC. Also in a debugging environment it would be beneficial to have some way to observe the state of the peripheral processor while the communication between it and the host processor is taking place.

7 .3 HOST PROCESSOR ENHANCEMENTS

As indicated by the mediocre performance in the calculation of a filtered output, the Host processor is the bottleneck of the system. One of the limitations of the current system is the processing core as evidenced by its performance in executing a DSP algorithm[10]. Alternative processing cores would offer higher performance in the calculation of DSP outputs. Specifically, the PowerPC line of processors from Motorola.

7 .3.1 The PowerPC 601

The PowerPC is a pipelined, superscalar RISC processor. It offers branch prediction, floating point unit, and shared memory access. The 601 is the first in the line of PowerPC processors from IBM and Motorola. The 601 includes floating point operations that include a multiply-add mechanism that is particularly beneficial for DSP applications. Accessing to the system interface for the 601 is provided by an external arbitration mechanism. The 601 also includes support for shared memory in the form of the Modified Exclusive Shared Invalid (MESI) protocol and external control of the on-chip cache. The PowerPC 601 offers several control signals that allow for bus arbitration. In a microcontrolled application it is suggested that the peripheral processor use the data arbitration signals: /DBG data bus grant, /DBWO data bus write only, and the data bus busy DBB, status signal in addition to other signals which support data bus tenure. For more information please refer to the 601 User's Manual.

Motorola and IBM have announced embedded control version PowerPCs. These include the 403 (from IBM) and the 515 (from Motorola). Also, there are, circa March 1995, new flavors of the 601, the 603e and the 602. The 602 was initially was originally

[10] This is not exactly fair, the purpose of the 386 EX is not DSP applications. Rather it was intended to be used in embedded applications.

marketed as an embedded version of the 601. But its current features illustrate its ability as a fully fledged general purpose processor. There is also the 403 from IBM which is an embedded control version. The embedded control versions of the PowerPC line are favored for a microprocessor project lab for their reduced complexity. Details of these events can be found at the PowerPC World Wide Web site and obtained via the PowerPC newsletter. [26]

7 .4 OTHER ISSUES

7 .4.1 DMA

In this implementation the DMA, could have been used to allow the 68HC11 to place data in the memory of the 386 EX. The purpose of using the DMA unit will be to allow the 68HC11 to transfer data from its A/D directly into the 386 EX's memory. DMA accessing would avoid the overhead of interrupt processing. However the 386 EX would have to be aware of when new sampled data has arrived. A suggested implementation, would be to have the 68HC11 request a transfer from DREQ0 using the RDY signal and receive the /DACK0, acknowledge signal on one of the port input pins. It could be done using the general purpose I/O ports of the 68HC11. Even though Furht and Halang renounce the use of DMA in their *Survey of Real-Tme Systems*, because it alters system behavior, DMA could be a viable alternative to the polling or interrupt driven implementation.

7 .4.2 Serial as Opposed to Parallel Operation

As suggested in the Analog to Digital section there could be architectural reasons why a serial connection (asynchronous or synchronous) between the host and peripheral processors might be beneficial. Although the 386 EX evaluation board required the use of the comm port to program the EPROM it need not be that way, the host processor might require a parallel connection to the PC. In any event, having one parallel or one serial connection might be beneficial in that the user may not have enough parallel ports for parallel communications or enough serial ports for serial communication. For more information about the discussion of parallel vs. serial computation, the reader is encouraged to read M. Boasson's *Dreams and Realities in Distributed Real-Time Systems.* [27]

7 .4.3 Improving the Software

Most of the code for the critical sections of the program in which the filtered output was calculated and subsequently sent to the D/A was coded in assembler. It is doubted that any significant performance increases would come using the same algorithm and coding in a different manner or language. However, the calculation of a filtered output using a 13-point FIR and the direct time domain computation of the convolution may have not been the best algorithm. Other approaches to filtering could be investigated such as decimation-in-time, decimation-in frequency, or even direct computation of the Fast Fourier Transform (FFT). The implications surrounding these methods are beyond the scope of this thesis however there has been mention of calculation of the FFT via an 80x86 machine. [28]

7 .4.4 Programming the EPROM

It was noted in the Introduction that the ability to program the host processor's program memory would be very beneficial. It is possible that the user could dynamically change the algorithm or program associated with the host processor. It would also be of importance, to be able to debug and alter programs without having to remove the host processor's connection to the peripheral processor. The ability to change and alter programs, in the host processor's program memory, whether it be RAM or ROM is paramount for any practical system.

With regard to the 386 EX Flash memory, the programming of this EPROM requires a fairly lengthy program, that can poll the comm port, send the appropriate FLASH programming commands, and have the ability to calculate the checksum. This information must be transmitted at 19200 baud. To program this amount of detail would be difficult and rather mundane in assembler and the use of the Forth language might incur unacceptable performance. The interested reader is encouraged to examine the `iboot.exe` file at the Intel BBS for C programs that perform the above operations on the 80x86 platform. [C2]

8 Alternative Methods

This section describes alternative strategies to the prototype system just described. It is divided into two sections: Academia and Equipment. The Academia section describes other classes and teaching methodologies at other universities for microprocessor and embedded control. The Equipment section focuses on other hardware and software development tools that could be used to teach embedded control or implement embedded control systems.

8 .1 ACADEMIA

There are a variety of methods for teaching the concepts involved with embedded control. However by nature of the subject much of the coursework emphasizes application over theory. A few of the examples of teaching are presented here.

8 .1.1 University of Alberta

The university of Alberta been developing a microprocessor development laboratory based on the 68000. They had found that a Senior level course, which used the 6809 microprocessor to be inadequate. For the final project in this course, students were required to build an application based on the 6809 system they had constructed thoughout the term. Apparently this was frustrating and non-constructive as they spent much of the time debugging the base system. The department of electrical engineering then decided to use the more powerful 68000 microprocessor. If the students would have been led to build a base system around the 68000 the class would have been even more unsuccessful due to the increased complexity of the 68000. So it was decided that the electrical engineering department design their own base system and distribute kits containing the base system to the students. This system uses the VME bus. [29]

8 .1.2 Rensselaer Polytechnic Institute

At RPI, there is a course for engineering majors called Laboratory Introduction to Embedded Control (LITEC). In this course the basic concepts of embedded control are taught and students are required to counteract an embedded control system and are encouraged to add their own contributions to the system. The development platform for this class are the MacIntosh Quadras and the target system is the Motorola 68HC11. They are currently exploring the use of the Neuron microcontroller family for future implementation in the class. [P1]

8 .1.3 A Simulator for Teaching

Eugene Styer, with the Department of Math, Statistics and Computer Science at Eastern Kentucky University has developed a simulator designed to allow students to work with simulated I/O devices as part of a larger simulator. The creation of the simulator came about as the result of the Styer's attempt to provide a broader view of the computer with input/output devices and not strictly a CPU and assembly code based simulator. The machine that the system simulates is a 16 bit, with 16 registers and a 64Kbyte address space.[30]

8 .2 EQUIPMENT

8 .2.1 Hardware

HP Systems

It must be mentioned that much of the motivation for this project came about as a result of our frustration associated with the older HP64000 Development System. However, according to Mark Brindle the more recent set of HP development tools are quite useful. He reports that the HP64700 emulators have an ANSI-C that is geared specifically toward embedded development. These systems emulate the 68000 and the embedded control 683xxx series processors, and include the standard debugging tools such as trace, memory emulation, and state/timing analysis. [P2]

Hallmark Reference Designs

Hamilton Hallmark's Reference Technical Support Center does work with various manufacturers to produce reference designs based on the latest technologies. The most recent design was the H4T that utilized the Intel 386 EX. Hamilton Hallmark provides the design documentation, including schematics free of charge. According to Wayne Diener, the H4T is a battery operated hand held device with an LCD display, FLASH Memory and an embedded DOS application. Hamilton Hallmark is also planning to make an embedded application reference design with the PowerPC. [P3]

PC/104 Bus

As was mentioned earlier the PC/104 bus is an emerging standard in the arena of embedded applications. There are a number of vendors who offer various cards conforming to the PC/104 bus standard. Perhaps some implementation of the 6.115 development system could be placed on one of more of these cards with the host processor and peripherals on one card and the core CPU on the other.

PC/104 is a consortium started by Ampro, and contains well over 100 companies now. The PC/104 cards measure 3.6" x 3.8", and has signals which are the logical equivalent of a 16 bit ISA bus. There is a multitude of processor, networking and I/O boards available.
[C3]

8 .2.2 Alternative Software

Concurrent Sciences

The operating system used on the 386 EX evaluation board was the Intel iRMX multi-tasking system. In conjunction with the iRMX system, the Softscope Debugger was used. Softscope also features a native debugger that does not require the Intel iRMX OS operating system. This would be an excellent selection for those users who do not require

the use of a multi-tasking operating system. The Csi-Monitor for this version takes up only 8Kbytes of memory in the 386 EX Flash EPROM.
[P4]

QNX OS

QNX is a realtime operating system that offers a 10Kbyte microkernel. It has built-in distributed processing and a scaleable OS that allows for the coordination of hundreds of processors or a minimum size configuration for embedded applications. QNX offers a host of TCP/IP utilities and supports NFS, remote procedure call and Simple Network Management Protocol (SNMP). What is particularly attractive about QNX is it would allow the 386EX system to boot from a network. This provides isolation as it is not connected to the PC comm port. [C4]

9 Results and Conclusion

The focus of this project has been to illustrate the use of microcontrollers to control system peripherals and handle system I/O while allowing a more powerful processor to manipulate and process this data. A further goal has been to show how the current microprocessor Lab, 6.115 might evolve over the years by offering alternatives to the current development system. The current development system could be extended by using off the shelf components and readily available parts. To this end the author has illustrated an extendible architecture by implementing a system that uses a microcontrolled unit for the collection of data, and a more powerful processor for the manipulation of that data. The task for the processor was the filtering of data using a DSP algorithm. The performance of this specific implementation of the processing system is adequate to show that the 386 EX evaluation board can be extended. However the performance was by no means stellar. The processing system was able to perform filtering on signals bandlimited to approximately 8 kHz. The type of filtering it was able to perform was reasonable. At the 8 kHz band limit it could perform low, mid range, and high pass filtering via a 13 point type I FIR. An example of a filtered output appears below, note that the lowpass filter does not pass the high frequency components at the edge of a square wave. Consequently, the output signal is not as sharp as the input. The mediocre performance of the system is not do to an inadequacy of design but rather the result of using a general purpose processor for a specific task. However, the goal was to illustrate how one might use a reconfigurable architecture to accomplish some task. And this goal has been accomplished more than satisfactorily.

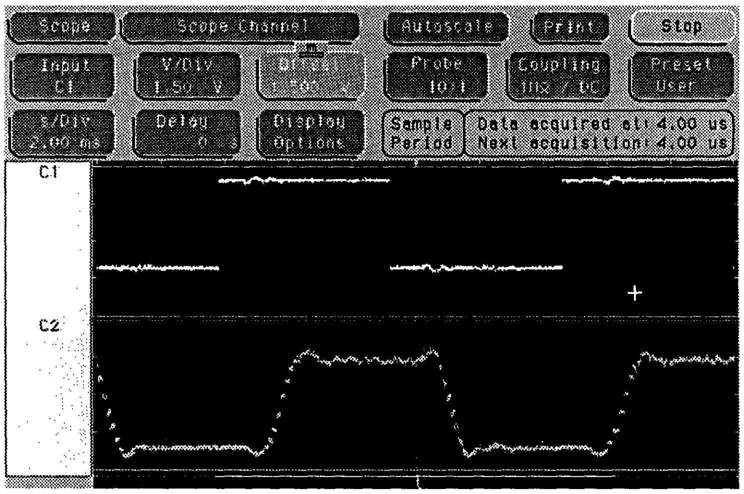


Figure 23: The Low Pass Filtered Output from a Square wave

10 Acknowledgments

Much of the material for the research of the embedded systems has come from the Internet. The usenet news groups `comp.arch.embedded`, `comp.sys.powerpc` have proved particularly helpful in finding material for this development system. The `comp.arch.embedded` was also beneficial in seeing what other universities and people in industry are doing with embedded systems and microprocessors. The PowerPC mailing list has provided an ample amount of information on the Power PC.

Special thanks goes to Aaron Schultz, a 6.115 T.A. for his tireless support of the 386EX development system and his useful instructions regarding its use. Special thanks, are also in order for Marty Hughes, who has been instrumental in providing support for the lab facilities. The author would also like to extend gratitude to Adi Askenazi for his instructions on the use of the Evaluation Board and the iRMX operating system, and Steve Jones for his assistance and encouragement with the 386 EX applications programming. Beth Frey has provided excellent technical support of Concurrent Science's Soft Scope Monitor. The author would also like to thank New Micros technical Software technical support for their assistance with the Forth Operating System.

11 Appendix

11.1 REFERENCES

11.2 CONTACT PERSONNEL

11.3 68HC11 REGISTER DESCRIPTIONS

11.4 386EX REGISTER DESCRIPTIONS

11.5 386EX MEMORY MAP

11.6 68HC11 MEMORY MAP

11.7 CODE LISTINGS

11.8 SCHEMATICS

References

[1] HP64000 Development System Reference, Hewlett-Packard, 1988.

[2] Gilbert, Marcus-Alan. *The Design and Implementation of an EPROM Programmer and Development System,* 1994.

[3] B. Furht, W.A. Halang. A Survey of Real-Time Computing Systems. *International Journal of Mini & Microcomputers*, Vol. 16 No. 3, 1994.

[4] *PCI Quick Reference Guide.* Intel Fax Back Doc 7279. Intel Corporation, 1994.

[5] *82430 FX PCI Chip Set Product Overview.* Intel Fax Back Doc 7089. Intel Corporation, 1994.

[6] PowerPC News:Power PC 601 . Information available at *http://power.globalnews.com/articles/4995.htm,* IBM Corporation, 1995.

[7] *PowerPC 601 Risc Microprocessor User's Manual.* Order Number MPC601UM/AD Rev 1, Motorola Corporation, 1995.

[8] *NMIX/T-0020 Hardware Manual.* New Micros, Inc. 1993

[9] Hersch, Russ. *The 68HC11 FAQ.* Posted at regular intervals to the following newsgroups: `comp.realtime, comp.robotics, sci.electronics, comp.answers` and `sci.answers.` 1995.

[10] *The Motorola M68HC11 Reference Manual*, Order Number M68HC11RM/AD Rev 3. Motorola, 1991.

[11] *The UMMAX Forth Version 3.3 Reference Manual.* New Micros, Inc. 1993

[12] Hilburn, John and David E. Johnson. *Manual of Active Filter Design.* McGraw-Hill, New York, 1993.

[13] *The 386 Embedded Microprocessor Hardware Reference*, Order No: 272485-000. Intel Corporation, 1994.

[14] *The 386 Evaluation Board Manual*, Order No. 272525-002. Intel Corporation, 1994.

[15] The 28F400BL-T/B, 28F400BL-T/B, 4 MBIT (256K x 16, 512K x 8) Low Power Boot Block Flash Memory Family. Order No. 290450-004. Intel

Corporation, 1994.

[16] *The AD7837/AD7847 Complete, Dual 12-Bit MDACs.* Analog Data, Vol II pg. 2-681-2-692 . Analog Devices, 1994.

[17] Matthews, John. *Forth Applications in Engineering and Industry.* Ellis Horwood Ltd, West Sussex, England. 1989.

[18] Kail, P.A.C. Forth: *A Complete Course in the Forth Programming Language.* Kogan Page Ltd. London. 1989.

[19] Tracy, Martin, Anita Andersen and Advanced MicroMotion, Inc. *Mastering Forth.* Simon & Schuster, Inc. 1989.

[20] Oppenheim, Alan v. and Ronald W. Schafer. *Discrete Time Signal Processing.* Prentice Hall, Englewood Cliffs, NJ. 1989

[21] Siebert, William. McC. *Circuits, Signals and Systems.* McGraw-Hill Cambridge, MA 1985.

[22] Sigmon, Kermit. *Matlab Primer,* 4e. ISBN# 0-8493-9440. CRC Press, Inc. 1994

[23] *The Intel386 SX Microprocessor Programmer's Reference Manual.* Order No. 240331-002. Intel Corporation, 1991.

[24] *iRMX EMB Operating System, System Calls Dictionary.* Order No. 619204-001. Intel Corporation, 1994.

[25] Hersch, Russ. *The 8051 FAQ.* Posted at regular intervals to the following newsgroups: `comp.sys.intel, comp.realtime, comp.robotics, comp.lang.forth and sci.electronics.` 1995

[26] PowerPC News. Available at *http://www.power.globalnews.com* or via electronic mail at `add@power.globalnews.com`, 1995.

[27] Boasson, M. Dreams and Realities in Distributed Real-Time Systems. *International Journal of Mini & Microcomputers.* Vol. 17, No. 1, 1995.

[28] Dobbe, Iwan edited by Bruce Scheier. Faster FFTs. *Dr. Dobb's Journal.* Vol. I Issue 2. February, 1995

[29] Digital Labs. *http://nyquist.ee.ualberta.ca/digital/digital.html,* 1995.

[30] Styer, Eugene. On the Design and Use of a Simulator for Teaching Computer Architecture. *SIGCSE BULLETIN*. Vol. 26 No. 3 Sept. 1994.

Company Contacts & Contact Personnel

This section lists some of the contacts, the author has made while gathering information for this document.

[C1] Concurrent Sciences. Inc.
P.O. Box 9666
Moscow, Idaho 83843
(208) 882-9774

[C2] The Intel BBS. (916) 356-3600

[C3] The PC/104 Consortium.
849B Independence Avenue
Mountain View, CA 94043.
Phone: (415)903-8304
Fax: (415)967-0995

[C4] QNX Software Systems, Ltd.
175 Terrence Matthews Crescent
Kanata, Ontario, Canada K2M 1W8
(613) 591-0931 FAX (613) 591-3579

[P1] Paul Kulp
Doctoral Candidate, Computer Engineering
Rensselaer Polytechnic Institute
kulpp2@rpi.edu

[P2] Mark Brindle
Hewlett-Packard Little Falls Site
brinde@lf.hp.com

[P3] Wayne Diener - Technical Programs Manager
Hamilton Hallmark (an Avnet Company)
3011 South 52nd St.
Tempe, AZ 85282
wdiener@tsc.hh.avnet.com

[P4] Beth Frey
Technical Support
bf@consci.com

tel: 208-882-0445

Relevant 68HC11 Register Descriptions

A * indicates this bit position is reserved and not used.

Register Name | Bit Position | Hex Address

PortA 0xB000

7	6	5	4	3	2	1	0
PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

This register is used to read and write port A, the first 3 lines PA0-PA2 are configured for output only, the PA6 and PA5 lines are dedicated outputs and the PA7 and PA3 lines can be configured as either inputs or outputs.

PortD 0xB008

*	*	PD5	PD4	PD3	PD2	PD1	PD0

The Port D data register is used to read and write Port D. Bits 0-5 can be used for general I/O while the top bits are reserved.

DDRD 0xB009

*	*	Bit5	Bit4	Bit 3	Bit 2	Bit 1	Bit 0

The Port D data direction registers is used to indicate, whether an I/O pin is an input or an output. A zero in the corresponding bit position indicates that the Pin is an input and conversely a one designates an output.

PACTL 0xB026

DDRA7	PAEN	PAMOD	PEDGE	DDRA3	*	RTR1	RTR0

Bit 7 DDRA7 controls the direction of Port A pin 7. When this bit is set pin 7 is an output and when it is cleared pin 7 is an input. The PAEN, PAMOD, PEDGE control the Pulse accumulator. DDRA3 functions similar to DDRA7 as it controls the I/O status of Port A pin 3.

ADCTL 0xB030

CCF	*	SCAN	MULT	CD	CC	CB	CA

The CCF indicates that a conversion has completed when this bit is set. The SCAN bit selects one of the modes or operation. When the SCAN bit is set the A/D runs in continous mode and the registers are successively rewritten. When this bit is clear the A/D only makes four conversions. The MULT bit indicates whether there is one channel or a group of channel being converted. The CD-CA bits are used to select the channel or channels for conversion.

ADRx, x=1-4

0xB031-34

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

The A/D data registers, these registers contain the eight bit result of an A/D conversion.

OPTION

ADPU	CSEL	IRQE	DLY	CME	*	CR1	CR0

The ADPU, A/D power up determines whether the A/D is powered up or powered down. When this bit is set the A/D circuitry is enabled, and when ADPU is cleared the A/D is powered down.

The CSEL Bit determines whether or not the system E clock or the internal A/D oscillator will be used. When this bit is set the internal oscillator is used for A/D conversion. Conversely, when this bit is cleared the external clock is used.

386EX Register Descriptions

Register Name Bit Position

PxLTC

7	6	5	4	3	2	1	0
Bit7	Bit 6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

Port Latch Register, when the corresponding pin is configured as an output this bit controls what value is driven to that output. When, the pin is configured other wise written a one to this bit causes the Pin to be an input, and written a zero causes it to be a open collector output.

PxCFG

Pin7	Pin6	Pin 5	Pin 4	Pin 3	Pin 2	Pin	Pin 1

Pin configuration, designates whtehr or not the associated pin will be driven by an internal peripheral or controled by the PxLTC and PxDIR registers. Setting the bit causes the corresponding pin to be controlled by an internal peripheral, and clearing this bit causes the pin to be controlled by the PxLTC and PxDIR registers.

PxDIR

Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	Pin 1	Pin 0

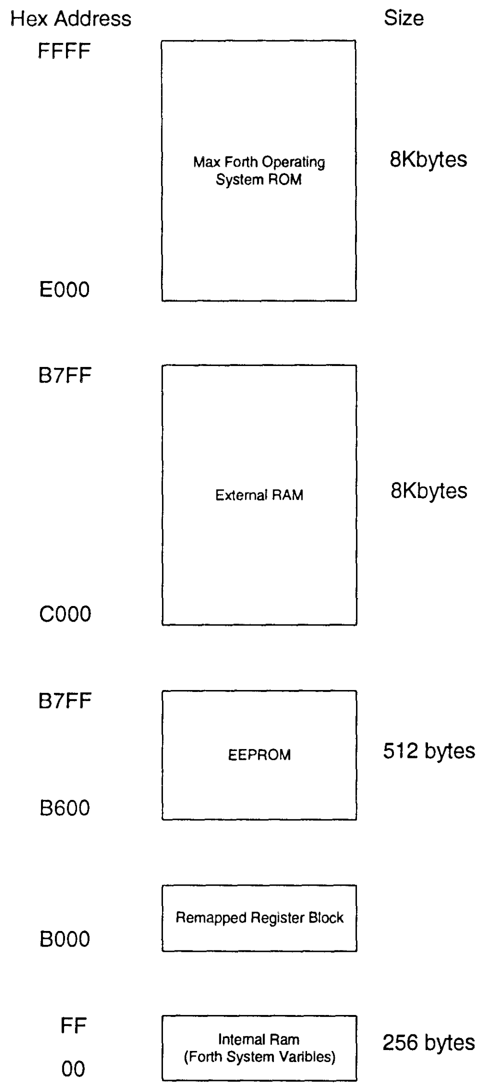
The port data registers designates whether or not the PIN will be an input or an output setting the corresponding bit selects and output and clearing the bit selects an input.

PxPIN

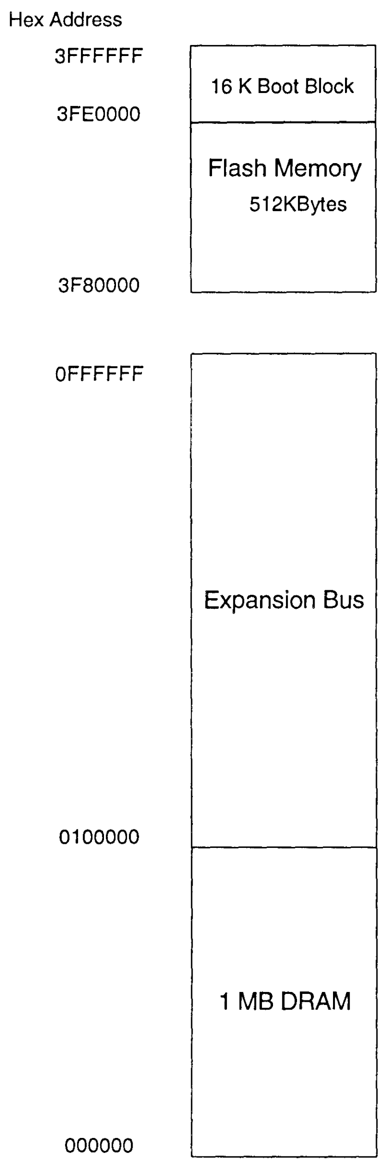
Pin 7	Pin 6	Pin 5	Pin 4	Pin 3	Pin 2	Pin 1	Pin 0

These read-only regisisters return the value on the pin associated with the bit.

68HC11 Memory Map



386 EX Memory Map



Code Listings

The following is a listing of the program files. These include c and Forth files.

```
( This code will run at exactly 128 clocks, = sampling frequency / 4)
HEX
CODE-SUB GA2D
  CE C, B030 ,   ( LDX #$B030)
  A6 C, 01 C,    ( LABEL: LDAA $01,X)
  16 C,          ( TAB)
  84 C, F0 C,    ( ANDA #$F0
  C4 C, 0F C,    ( ANDB #$0F; Set RDY signal low, mask high nibble for B)
  B7 C, B000 ,   ( STAA $B000 ; Send High nibble and Low RDY)
  58 C,          ( LSLB ; Shift them up two places)
  58 C,          ( LSLB )
  F7 C, B008 ,   ( STAB $B008 ; Store the low nibbles in Port D)
  8B C, 08 C,    ( ADDA #08 ; assert RDY bit)
  B7 C, B000 ,   ( STAA $B000 ; Send the active RDY)
  B7 C, B000 ,   ( 1)
  B7 C, B000 ,   ( 2)
  B7 C, B000 ,   ( 3)
  B7 C, B000 ,   ( 4)
  B7 C, B000 ,   ( 5)
  B7 C, B000 ,   ( 6)
  B7 C, B000 ,   ( 7)
  B7 C, B000 ,   ( 8)
  B7 C, B000 ,   ( 9)
  B7 C, B000 ,   ( 10)
  B7 C, B000 ,   ( 11)
  B7 C, B000 ,   ( 12)
  B7 C, B000 ,   ( 13)
  B7 C, B000 ,   ( 14)
  B7 C, B000 ,   ( 15)
  B7 C, B000 ,   ( 16)
  B7 C, B000 ,   ( 17)
  B7 C, B000 ,   ( 18)
  B7 C, B000 ,   ( 19)
  B7 C, B000 ,   ( 20)
  B7 C, B000 ,   ( 21)
  B7 C, B000 ,   ( 22)
  B7 C, B000 ,   ( 23)
  B7 C, B000 ,   ( 24)
  20 C, 9F C,    ( BRA LABEL)
  39 C,          ( RTS)
 END-CODE
```

init.4th

```
( 4th program to initialize registers)

COLD
HEX
88 B026 C! ( Set Bits 7 and 3 as to an outputs)
FF B009 C! ( Set Port D 0-7 for outputs)
21 B030 C! ( Continous, single channel set AN1)
80 B039 C! ( Turn on A/D use external clock)
```

```
// eval.c This evaluates all of the filters by having different test vectors
// and the different filtering coefficients
// May 16, 1995 Marcus-Alan Gilbert

#define N 33    /* number of h weighting factors */

#include <stdio.h>
#include <rmxemb.h>
#include "reg_add.h"

  long next_yFIR(unsigned char, char, short[]);  // FIR function
  void send_to_d2a(long, char);                  // D2A function
  void init_386(void);                           // initialize function

  void main(void)

  char filter_type = '1', data_type = '1';
  int i;
  unsigned char a[256];
  long value;
  short x_buffer[N]; // buffer for storing N most recent inputs
    for (;;)
      {    // Generate the different test vectors
           // h - high frequency test vect, m - midband test vectors
           // 1 - low freqency test vect, c - DC test signal
                switch(data_type)
                  {
                  case 'h':
                    for (i = 0; i < 256; i++) a[i] = (i%2 ? 255 : 0); break;
                  case 'm':
                    for (i = 0; i < 256; i++) a[i] = ((i/4)%2 ? 255 : 0); break;
                  case 'l':
                    for (i = 0; i < 256; i++) a[i] = (i/200 ? 255 : 0); break;
                  case 'c':
                    for (i = 0; i < 256; i++) a[i] = 128; break;
                  default: assert_error_flag(); continue;
                  }
           init_386();
           for(i=0;;i++)
             {
               /* Since i is an unsigned char it should reset */
               /* after 255.  Which is nice because the filter */
               /* will think the signal is repeating, which it */
               /* is basically but we don't have to have an */
               /* infinite array */

               value = next_yFIR(a[i], filter_type, x_buffer);
               send_to_d2a(value, filter_type);
               if(i == 255) i = 0;  // Reset i
               }

         }



 long next_yFIR(unsigned char next_x, char fil_typ, short x_buffer[])
 {  /* FIR filter */
  /* compute next output from next input */
  short k;              /* index counter */
  long acc;             /* accumulator */
  char b;               /* reorder index */


  short h1[N] = {       filter weighting fators  for low pass filter
```

```
        -1,    -0,    2,    4,    6,   5,    0,  -10,  -21,  -28, -23,    0,
        41,    95,  149,  190,  205, 190,  149,   95,   41,    0, -23,  -28,
       -21,   -10,    0,    5,    6,   4,    2,    0,   -1};

       short hm[N] = {       /* filter weighting fators for mid-range filter */
        -1,     0,    1,   -6.   -9,   3,    0,   -6,   34,   45, -14,    0,
        26,  -154, -242,  117,  409, 117, -242, -154,   26,    0, -14,   45,
        34,    -6,    0,    3,   -9,  -6,    1,    0,   -1};

       short hh[N] = {       /* filter weighting fators for high pass filter */
         2,     0,   -3,    2,    3,  -8,    0,   15,  -13,  -17,  37,    0,
       -67,    59,   92, -307,  409, -307,  92,   59,  -67,    0,  37,  -17,
       -13,    15,    0,   -8,    3,    2,   -3,    0,    2};

       for(b=0; b< 33; b++) x_buffer[b] = x_buffer[b+1];
                                  // Shift every body to left
       x_buffer[32] = next_x;     // The newest goes in the 33rd position

       for (k = 32, acc = 0; k > -1; k--)
       {                          // I am pulling another trick here
       switch(fil_typ)            // You are supposed to multiply the
               {                 // newest x by the first h, but I am
                       case 'l': // multiplying the last h.  The first
                                 /* and last h's are the same because the */
                                 /* filter is symmetric */
                               acc += hl[k] * x_buffer[k];
                                break;
                        case 'm':
                                acc += hm[k] * x_buffer[k];
                                break;
                        case 'h':
                                acc += hh[k] * x_buffer[k];
                                break;

               }
       }
 return acc;                     // acc represents the sum of the partial multiplies
}

void send_to_d2a(long num, char fil_type)
 {
 long val;
 unsigned char go;
       val = num/256;            // DC rectification
       if(fil_type == 'l')
               {
               if(val < 0) go = 0; // In the low pass filter if val
               else go = val;      // is negative it is only slightly so
               }                   // set zero
       else
               {
               if(val > 127) go = 255;// If we are in the mid or band pass
               if(val < -127) go = 0; // filters then values go slightly below
               else                   // -127 or slighlty above 127
                       {              // If so round appropriately
                       val += 127;    // If not add the dc signal back
                       go = val;
                       }
               }
       _asm
```

```
                mov dx,P1LTC
                and al,80h  /* Mask Out lower bits leaving, only the top bit *
                out dx, al
                or al,40h  /* Send Write High */
                out dx, al
                }
 }

void init_386(void)
{
 _asm{
                        /* The first thing we have to do is to enable
                               expanded I/O space. */

                        mov ax, 8000h /* This will be REMAPCONFIG */
                        xchg al, ah
                        out 23h, al   /* high byte to 23h */
                        xchg al, ah
                        out 22h, al      /* low byte to 22h  */
                        out 22h, ax      /* Word to 22h */



                        /* Now set the P3 and latch values */

                        mov dx, P3LTC
                        mov al, 0
                        out dx, al

                        /* Now set I/O port direction for output. */
                        /* Register P3DIR and P1DIR */

                        mov dx, P3DIR
                        mov al, 00h
                        out dx, al

                        mov dx, P1DIR
                        mov al, 00h
                        out dx, al

                        /* Set Port Mode Configuration for Port 3 */
                        /* Register P3CFG */

                        mov dx, P3CFG
                        mov al, 80h   /* Bits 0-6 are for outputs Bit 7 is for the com cl
ck */
                        out dx, al

                        mov dx, P1CFG
                        mov al, 3Fh   /* Bits 6,7 are used for control, bit 1-5 are for pr
ogramming the PROM */
                        out dx, al

                }
 }
```

```
// High Pass Filter Version 2
// Program to produce output based upon a
// 13-pt FIR
// Marcus-Alan Gilbert 5/16/95

#include "reg_add.h"
#include <rmxemb.h>
#define N 13

void init_386(void);


main()
{
        short x_buffer[N], hh[N] = {      /* filter weighting fators for high pass filter
*/
                                  1, 0, -6, 9, 18, -72, 102, -72, 18, 9, -6, 0, 1
                                  };


        init_386();  // Initialize input pins
        _asm
                {
                zero_detect:
                        mov dx, P2PIN
                        in al,dx
                        and al, 0x02
                        jz one_detect
                        mov al, 0xFF
                        mov dx, P2LTC
                        out dx, al
                        jmp zero_detect

                one_detect:
                        mov dx,P2PIN
                        in al,dx  /* xor al, 01h */
                        and al,0x02
                        jnz sample              //   jnz sample        // Data is Ready
                        mov dx, P2LTC
                        mov al, 0xFE
                        out dx, al
                        jmp one_detect

                sample:
                        mov si, 0
                        mov bx, 0
                        mov cx, 0
                        for_loop1:
                                        cmp si, (2*N)
                                        je done
                                        mov ax, x_buffer[si+2]
                                        mov x_buffer[si],ax
                                        cmp si, (2*(N-1))
                                        jl next
                                        mov al,[es:0]
                                        mov ah, 0
                                        mov x_buffer[si],ax
                                next:
                                        imul hh[si]
                                        add cx,ax
                                        adc bx,dx         Neccessary to get over flow fr
m low bit

                                        inc si
                                        jmp for_loop1
                        done:
                                        test bx.0xFFFF          // Test to see if it is ne
gative
                                        jz pos                  // It is positive
                                        test ch,0x80            // It is too negative
                                        jnz norect
                                        mov al, 0               // Its too negative and need
 to be rectified
                                        jmp check
                        norect:
                                        and ch,0x7F
                                        mov al,ch
                                        jmp check
                        pos:
                                        mov cl,0      // Quick div by 256
                                        xchg cl,ch
                                        mov al,cl               // Now its time to send the
 data to the D/A
                                        test al,0x80
                                        jnz over                // It is too positive
                                        add al, 127
                                        jmp check
                        over:
                                        mov al, 255
                        check:
                                        mov dx, P3LTC
                                        out dx, al
                                        mov dx, P1LTC
                                        and al, 80h
                                        out dx, al
                                        or al,40h
                                        out dx, al
                                        jmp zero_detect
                }


}

void init_386(void)
{
 SELECTOR sel;   // Selector for the Segment that contains the address of the expansion bu
s
 UINT_16 exp_ptr; // Exception pointer


 sel = rqe_create_descriptor(0x0100000, 1, &exp_ptr);
                                                                    _asm {
                         mov ax, sel
                         mov es, ax
                        }

        _asm{
                        /* The first thing we have to do is to enable
                                expanded I/O space. */

                        mov ax, 8000h  /* This will be REMAPCONFIG */
                        xchg al, ah
                        out 23h, al    /* high byte to 23h */
                        xchg al, ah
                        out 22h, al      /* low byte to 22h  */
                        out 22h, ax      /* Word to 22h */
```

```
                       /* Now set the P2 latch values */

                       mov dx, P2LTC
                       mov al, 0
                       out dx, al

                       /* Now set I/O port direction for output. */
                       /* Register P2DIR */

                       mov dx, P2DIR
                       in al,dx
                       and al,0xFC   // Preserve direction of higher order bits
                       or al, 0x02     // Put P2.0 as output and P2.1 as input
                       out dx, al

                       /* Set Port Mode Configuration for Port 3 */
                       /* Register P3CFG */

                       mov dx, P2CFG
                       in al,dx
                       and al,0xFC   // And Mask to allow  P2.7-P2.2 to remain in same st
ate

                       out dx,al    // and put pin 1 and pin 0 in I/O mode
               }

 _asm{
                       /* The first thing we have to do is to enable
                               expanded I/O space. */

                       mov ax, 8000h /* This will be REMAPCONFIG */
                       xchg al, ah
                       out 23h, al   /* high byte to 23h */
                       xchg al, ah
                       out 22h, al     /* low byte to 22h  */
                       out 22h, ax     /* Word to 22h */



                       /* Now set the P3 and latch values */

                       mov dx, P3LTC
                       mov al, 0
                       out dx, al

                       /* Now set I/O port direction for output. */
                       /* Register P3DIR and P1DIR */

                       mov dx, P3DIR
                       mov al, 00h
                       out dx, al

                       mov dx, P1DIR
                       mov al, 00h
                       out dx, al

                       /* Set Port Mode Configuration for Port 3 */
                       /* Register P3CFG */

                       mov dx, P3CFG
                       mov al, 80h   /* Bits 0-6 are for outputs Bit 7 is for the com clo
ck */

                       out dx, al
```

midbv2.c

```
// Mid Band Pass Filter  Version 2
// Program to produce output based for an AD7847 basedupon a
// a 13-pt FIR
// Marcus-Alan Gilbert 5/16/95

#include "reg_add.h"
#include <rmxemb.h>
#define N 13

void init_386(void);


main()
{
        short x_buffer[N], hm[N] =
                                { 0, 0, 2, -25, -53, 30, 111, 30, -53, -25, 2, 0};
        init_386();  // Initialize input pins
        _asm
                {
                zero_detect:
                        mov dx, P2PIN
                        in al,dx
                        and al, 0x02
                        jz one_detect
                        mov al, 0xFF
                        mov dx, P2LTC
                        out dx, al
                        jmp zero_detect

                one_detect:
                        mov dx,P2PIN
                        in al,dx  /* xor al, 01h */
                        and al,0x02
                        jnz sample               //   jnz sample        // Data is Ready
                        mov dx, P2LTC
                        mov al, 0xFE
                        out dx, al
                        jmp one_detect

                sample:
                        mov si, 0
                        mov bx, 0
                        mov cx, 0
                        for_loop1:
                                        cmp si, (2*N)
                                        je done
                                        mov ax, x_buffer[si+2]
                                        mov x_buffer[si],ax
                                        cmp si, (2*(N-1))
                                        jl next
                                        mov al,[es:0]
                                        mov ah, 0
                                        mov x_buffer[si],ax
                        next:
                                        imul hm[si]
                                        add cx,ax
                                        adc bx,dx        // Neccessary to get over flow fro
m low bit
                                        inc si
                                        inc si
                                        jmp for_loop1
                                        jz pos              // It is positive
                                        test ch,0x80        // It is too negative?
                                        jnz norect
                                        mov al, 0               // Its too negative and need
to be rectified
                                        jmp check
                                norect:
                                        and ch,0x7F
                                        mov al,ch
                                        jmp check
                                pos:
                                        mov cl,0        // Quick div by 256
                                        xchg cl,ch
                                        mov al,cl  // Now its time to send the data to the
D/A
                                        test al,0x80
                                        jnz over                // It is too positive
                                        add al, 127
                                        jmp check
                                over:
                                        mov al, 255
                                check:
                                        mov dx, P3LTC
                                        out dx, al
                                        mov dx, P1LTC
                                        and al, 80h
                                        out dx, al
                                        or al,40h
                                        out dx, al
                                        jmp zero_detect
                }

}

void init_386(void)
{
 SELECTOR sel;   // Selector for the Segment that contains the address of the expansion bu
s
 UINT_16 exp_ptr; // Exception pointer


 sel = rqe_create_descriptor(0x0100000, 0, &exp_ptr);
                  _asm {
                        mov ax, sel
                        mov es, ax
                       }

        _asm{
                       /* The first thing we have to do is to enable
                               expanded I/O space. */

                       mov ax, 8000h /* This will be REMAPCONFIG */
                       xchg al, ah
                       out 23h, al    /* high byte to 23h */
                       xchg al, ah
                       out 22h, al      /* low byte to 22h  */
                       out 22h, ax      /* Word to 22h */



                        /* Now set the P2 latch values */
```

```
                    mov al, 0
                    out dx, al

                    /* Now set I/O port direction for output. */
                    /* Register P2DIR */

                    mov dx, P2DIR
                    in al,dx
                    and al,0xFC   // Preserve direction of higher order bits
                    or al, 0x02     // Put P2.0 as output and P2.1 as input
                    out dx, al

                    /* Set Port Mode Configuration for Port 3 */
                    /* Register P3CFG */

                    mov dx, P2CFG
                    in al,dx
                    and al,0xFC   // And Mask to allow  P2.7-P2.2 to remain in same st
ate

                    out dx,al    // and put pin 1 and pin 0 in I/O mode
            }

 _asm{

                    /* The first thing we have to do is to enable
                            expanded I/O space. */

                    mov ax, 8000h /* This will be REMAPCONFIG */
                    xchg al, ah
                    out 23h, al   /* high byte to 23h */
                    xchg al, ah
                    out 22h, al     /* low byte to 22h  */
                    out 22h, ax     /* Word to 22h */



                    /* Now set the P3 and latch values */

                    mov dx, P3LTC
                    mov al, 0
                    out dx, al

                    /* Now set I/O port direction for output. */
                    /* Register P3DIR and P1DIR */

                    mov dx, P3DIR
                    mov al, 00h
                    out dx, al

                    mov dx, P1DIR
                    mov al, 00h
                    out dx, al

                    /* Set Port Mode Configuration for Port 3 */
                    /* Register P3CFG */

                    mov dx, P3CFG
                    mov al, 80h   /* Bits 0-6 are for outputs Bit 7 is for the com clo
ck */

                    out dx, al
```

lowpv2.c

```
// Low Pass Filter Version 2
// Program to produce output for the AD7847 based upon a
// a 13-pt FIR
// 5/16/95 Marcus-Alan Gilbert

#include "reg_add.h"
#include <rmxemb.h>
#define N 13

void init_386(void);


main()
{
        short x_buffer[N],
                hl[N] = {
                         -1, 0, 4, 15, 33, 49, 56, 49, 33, 15, 4, 0, -1
                        };

        init_386();  // Initialize input pins
        _asm
                {
                zero_detect:
                        mov dx, P2PIN
                        in al,dx
                        and al, 0x02
                        jz one_detect
                        mov al, 0xFF
                        mov dx, P2LTC
                        out dx, al
                        jmp zero_detect

                one_detect:
                        mov dx,P2PIN
                        in al,dx  /* xor al, 01h */
                        and al,0x02
                        jnz sample    // Ready to obtain sample
                        mov dx, P2LTC
                        mov al, 0xFE
                        out dx, al
                        jmp one_detect

                sample:
                        mov si, 0
                        mov bx, 0
                        mov cx, 0
                        for_loop1:
                                        cmp si, (2*N)
                                        je done
                                        mov ax, x_buffer[si+2]
                                        mov x_buffer[si],ax
                                        cmp si, (2*(N-1))
                                        jl next
                                        mov al,[es:0]
                                        mov ah, 0
                                        mov x_buffer[si],ax
                        next:
                                        imul hl[si]
                                        add cx,ax
                                        adc bx,dx
                                        inc si
                                        xchg cl,ch
                                        mov al,cl
                                        test bx,0xFFFF
                                        jz check
                                        jns pos
                                        mov al, 0  // Its negative
                                        jmp check
                        pos:
                                        mov al, 255
                        check:
                                        mov dx, P3LTC
                                        out dx, al
                                        mov dx, P1LTC
                                        and al, 80h
                                        out dx, al
                                        or al,40h
                                        out dx, al
                        jmp zero_detect
                }


}

void init_386(void)
{
 SELECTOR sel;// Selector for the Segment that contains the address
             // of the expansion bus
 UINT_16 exp_ptr; // Exception pointer


 sel = rqe_create_descriptor(0x0100000, 0, &exp_ptr);
                        _asm {
                         mov ax, sel
                         mov es, ax
                        }

        _asm{
                        /* The first thing we have to do is to enable
                                expanded I/O space. */

                        mov ax, 8000h /* This will be REMAPCONFIG */
                        xchg al, ah
                        out 23h, al   /* high byte to 23h */
                        xchg al, ah
                        out 22h, al     /* low byte to 22h  */
                        out 22h, ax     /* Word to 22h */



                        /* Now set the P2 latch values */

                        mov dx, P2LTC
                        mov al, 0
                        out dx, al

                        /* Now set I/O port direction for output. */
                        /* Register P2DIR */

                        mov dx, P2DIR
                        in al,dx
                        and al,0xFC // Preserve direction of higher order bits
                        or al, 0x02   // Put P2.0 as output and P2.1 as input
                        /* Set Port Mode Configuration for Port 3 */
                        /* Register P3CFG */

                        mov dx, P2CFG
                        in al,dx
                        and al,0xFC
                        out dx,al
                }

    _asm{

                        /* The first thing we have to do is to enable
                                expanded I/O space. */

                        mov ax, 8000h /* This will be REMAPCONFIG */
                        xchg al, ah
                        out 23h, al   /* high byte to 23h */
                        xchg al, ah
                        out 22h, al     /* low byte to 22h  */
                        out 22h, ax     /* Word to 22h */



                        /* Now set the P3 and latch values */

                        mov dx, P3LTC
                        mov al, 0
                        out dx, al

                        /* Now set I/O port direction for output. */
                        /* Register P3DIR and P1DIR */

                        mov dx, P3DIR
                        mov al, 00h
                        out dx, al

                        mov dx, P1DIR
                        mov al, 00h
                        out dx, al

                        /* Set Port Mode Configuration for Port 3 */
                        /* Register P3CFG */

                        mov dx, P3CFG
                        mov al, 80h
                        out dx, al

                        mov dx, P1CFG
                        mov al, 3Fh
                        out dx, al

                }
    }
```

```
% Matlab file for Frequency Response values based on the 68HC11 A/D,
% and the test vector frequency values.
%
% Marcus-Alan Gilbert              5/16/95

action1='print -deps filter1.eps;'
action2='print -deps filter2ac.eps;'
action3='print -deps filter3ac.eps;'
action4='print -deps filter4ac.eps;'
action5='print -deps filter5ad.eps;'
action6='print -deps filter6ac.eps;'

! rm filter.dia
diary filter.dia
diary on;
echo on;

figure(1); clf;

% Original Low Pass Filter coefficients from 6.115 course


hl = [-.0098732  -.0112578  -.0113833  -.0104106 -.0076728  -.0035061 .0025324   .0098510
.0186274  .0280183   .0379873   .0475351 .0565439 .0640174   .0699282   .0734527 .0748049
  .0734527   .0699282   .0640174 .0565439   .0475351   .0379873   .0280183 .0186274   .009
8510   .0025324 -.0035061 -.0076728  -.0104106  -.0113833  -.0112578 -.0098732];


% Frequency Responses

[HL, W] = freqz(hl, 1, 4096);

% The Redesign of the original Low Pass Filter to have the same number
% of significant digits and the same number of coeffiecients as the
% other fir1 designed filters

hl_red = fir1(12, .2);

[hrFT, W] = freqz(hl_red, 1, 4096);
Wk = 0:4095;
plot(W/pi, abs(HL), '-', W/pi, abs(hrFT), ':');
title('The original (from 6.115) and Redesigned Low-Pass Filter Freq Responses');
set(gca, 'Fontname', 'Symbol');
xlabel('w'); set(gca, 'Fontname', 'Helvetica');
ylabel('|H(e^jw)|');

legend('-', 'Original Filter', ':', 'New Filter');
eval(action1);

% Redesigned Filters based upon the Matlab fir1 functions
% and their corresponding frequency responses, to create a 13 pt. FIR
% filter

figure(2);clf;

hm = fir1(12, [.2 .6]);
[HM, W] = freqz(hm, 1, 4096);
hh = fir1(12, .6, 'high');
[HH, W] = freqz(hh, 1, 4096);
```

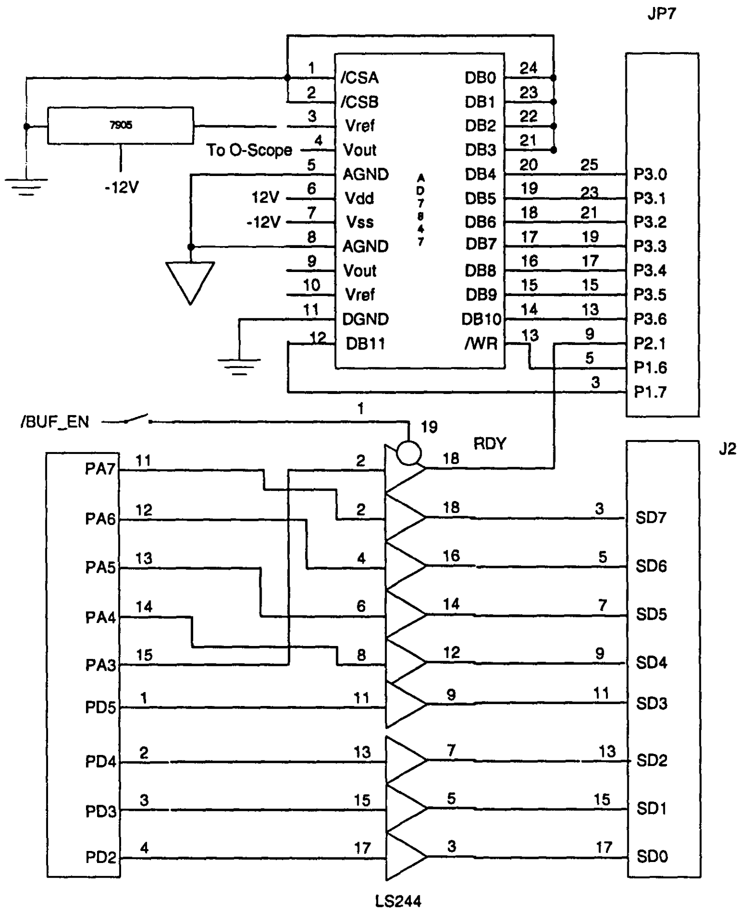
```
subplot(3,1,2);
plot(W/pi, abs(HM), '-');
title('The original (firl) Mid-Band Filter Freq Responses');set(gca,'Fontname','Symbol');x
label('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

subplot(3,1,3);
plot(W/pi, abs(HH), '-');
title('The original (firl) High Pass Filter Freq Responses');set(gca,'Fontname','Symbol');
xlabel('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

orient tall;
eval(action2);


% Shift Cut the bits of significance and round result for a fixed point
% computation.

hl_new = round(256 * hl_red);
hm_new = round(256 * hm);
hh_new = round(256 * hh);

hl_new
hm_new
hh_new

figure(3);clf;

% Scaled and rounded frequency responses

[hlnFT, W] = freqz(hl_new, 1, 4096);
[hmnFT, W] = freqz(hm_new, 1, 4096);
[hhnFT, W] = freqz(hh_new, 1, 4096);

subplot(3,1,1);
plot(W/pi, abs(hlnFT), '-');
title('Scaled, Redesigned, and Rounded Low Pass Filter Freq Responses');set(gca,'Fontname'
,'Symbol');xlabel('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

subplot(3,1,2);
plot(W/pi, abs(hmnFT), '-');
title('Scaled and Rounded Band Pass Filter Freq Responses');set(gca,'Fontname','Symbol');x
label('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

subplot(3,1,3);
plot(W/pi, abs(hhnFT), '-');
title('Scaled and Rounded High Pass Filter Freq Responses');set(gca,'Fontname','Symbol');x
label('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

% Check to make sure that rounded performance is satisfactory
% In the software for the 386EX these values will be divided by 256 =
% 2^8
orient tall;
eval(action3);


figure(4);clf;

% Corresponding Continuous Time Frequency Responses
% This helps us evaluate filter performance

subplot(3,1,1);
```

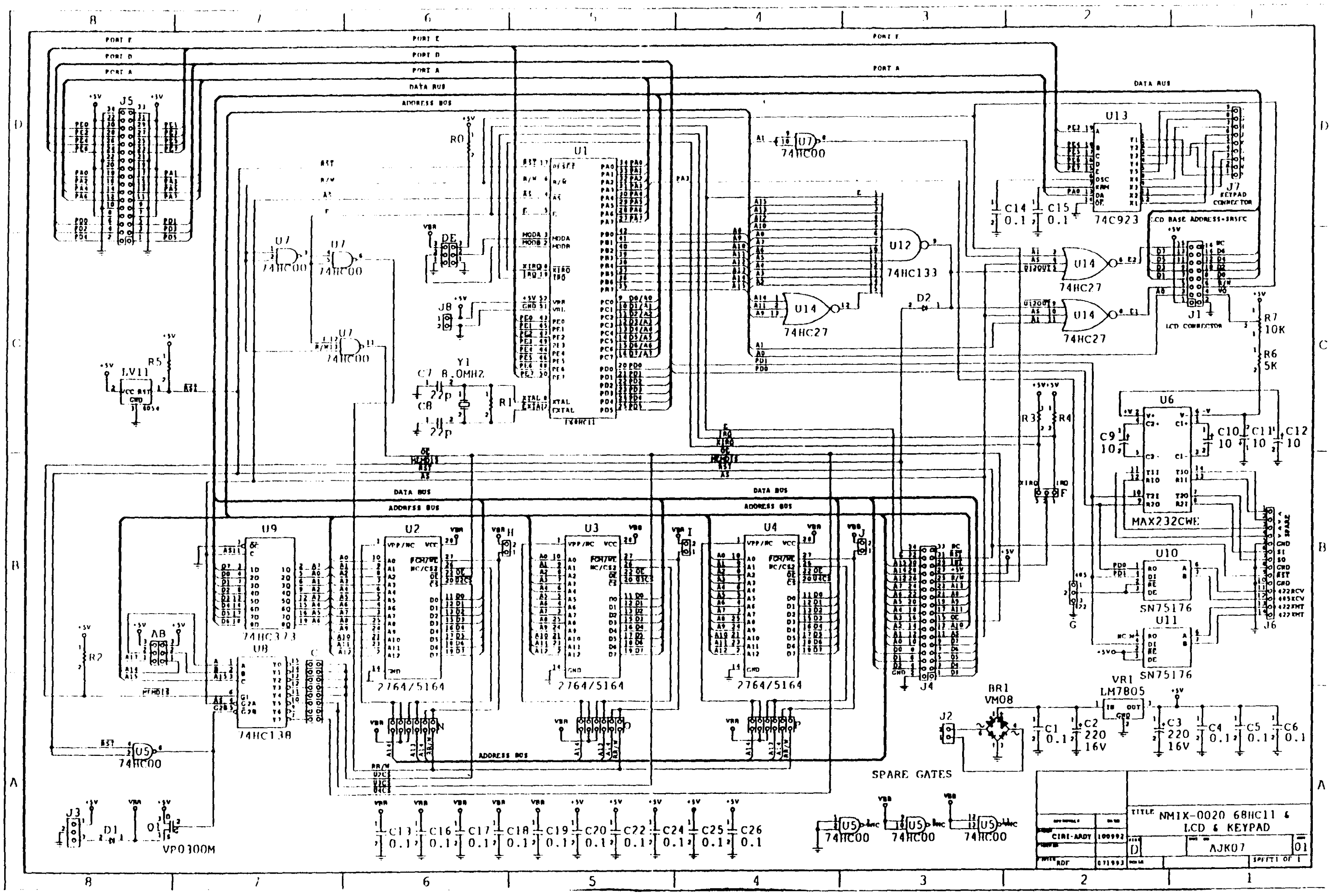
```
set(gca,'FontAngle','italic');xlabel('f (kHz)');
set(gca,'FontAngle','normal');ylabel('|H(e^j2pi*f)|');

subplot(3,1,2);
plot(W/pi*7.8125, abs(hmnFT), '-');
title('Scaled and Rounded Band Pass Filter Freq Responses');set(gca,'FontAngle','italic');
xlabel('f (kHz)');set(gca, 'FontAngle', 'normal');ylabel('|H(e^j2pi*f)|');
subplot(3,1,3);
plot(W/pi*7.8125, abs(hhnFT), '-');
title('Scaled and Rounded High Pass Filter Freq Responses');set(gca,'FontAngle','italic');
xlabel('f (kHz)');set(gca, 'FontAngle', 'normal');ylabel('|H(e^j2pi*f)|');

orient tall;
eval(action4);

figure(5);clf;

n = 0:12;

subplot(3,1,1);
stem(n, hl_new);
title('Redesigned Low Pass Filter Coefficients');xlabel('n');ylabel('hl[n]');

subplot(3,1,2);
stem(n, hm_new);
title('Redesigned Mid-Band Pass Filter Coefficients');xlabel('n');ylabel('hm[n]');

subplot(3,1,3);
stem(n, hh_new);
title('Redesigned Low Pass Filter Coefficients');xlabel('n');ylabel('hh[n]');

orient tall;
eval(action5);
figure(6);clf;

% Uni-polar Test Vector Frequency Responses

n = 0:255;
constftest = ones(32);
[cFT, W] = freqz(constftest, 1, 4096);
lowftest = round(255.^(cos(pi*floor(rem(n/40,2)))));
[lFT, W] = freqz(lowftest, 1, 4096);
midftest = round(255.^(cos(pi*floor(rem(n/4,2)))));
[mFT, W] = freqz(midftest, 1, 4096);
highftest = round(255.^(cos(pi*floor(rem(n,2)))));
[hFT, W] = freqz(highftest, 1, 4096);

subplot(4,1,1);
plot(W/pi, abs(cFT), '-');
title('Constant Test Vector Freq Responses');set(gca,'Fontname','Symbol');xlabel('w');set(
gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

subplot(4,1,2);
plot(W/pi, abs(lFT), '-');
title('Low Frequency Test Vector Freq Responses');set(gca,'Fontname','Symbol');xlabel('w')
;set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');

subplot(4,1,3);
plot(W/pi, abs(mFT), '-');
title('Mid-Band Frequency Test Vector Freq Responses');set(gca,'Fontname','Symbol');xlabel
('w');set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');
```

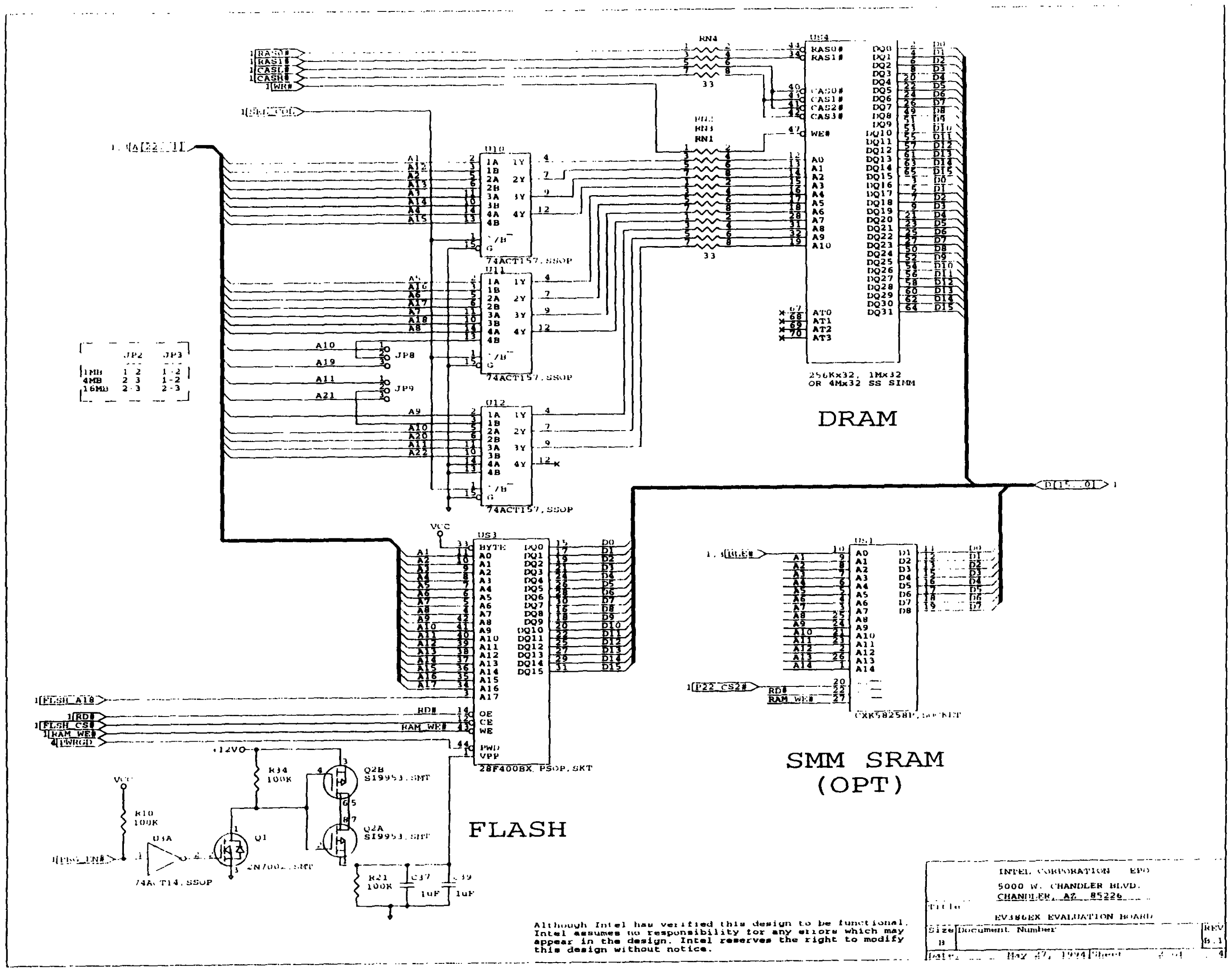
```
title('High Frequency Test Vector Freq Responses');set(gca,'Fontname','Symbol');xlabel('w'
);set(gca, 'Fontname', 'Helvetica');ylabel('|H(e^jw)|');
orient tall;
eval(action6);

echo off;
diary off;
```

Schematics

The schematic below illustrates the connections between the 386 EX EVB and the 68HC11 Evaluation Board. Detailed schematics of these board follow.

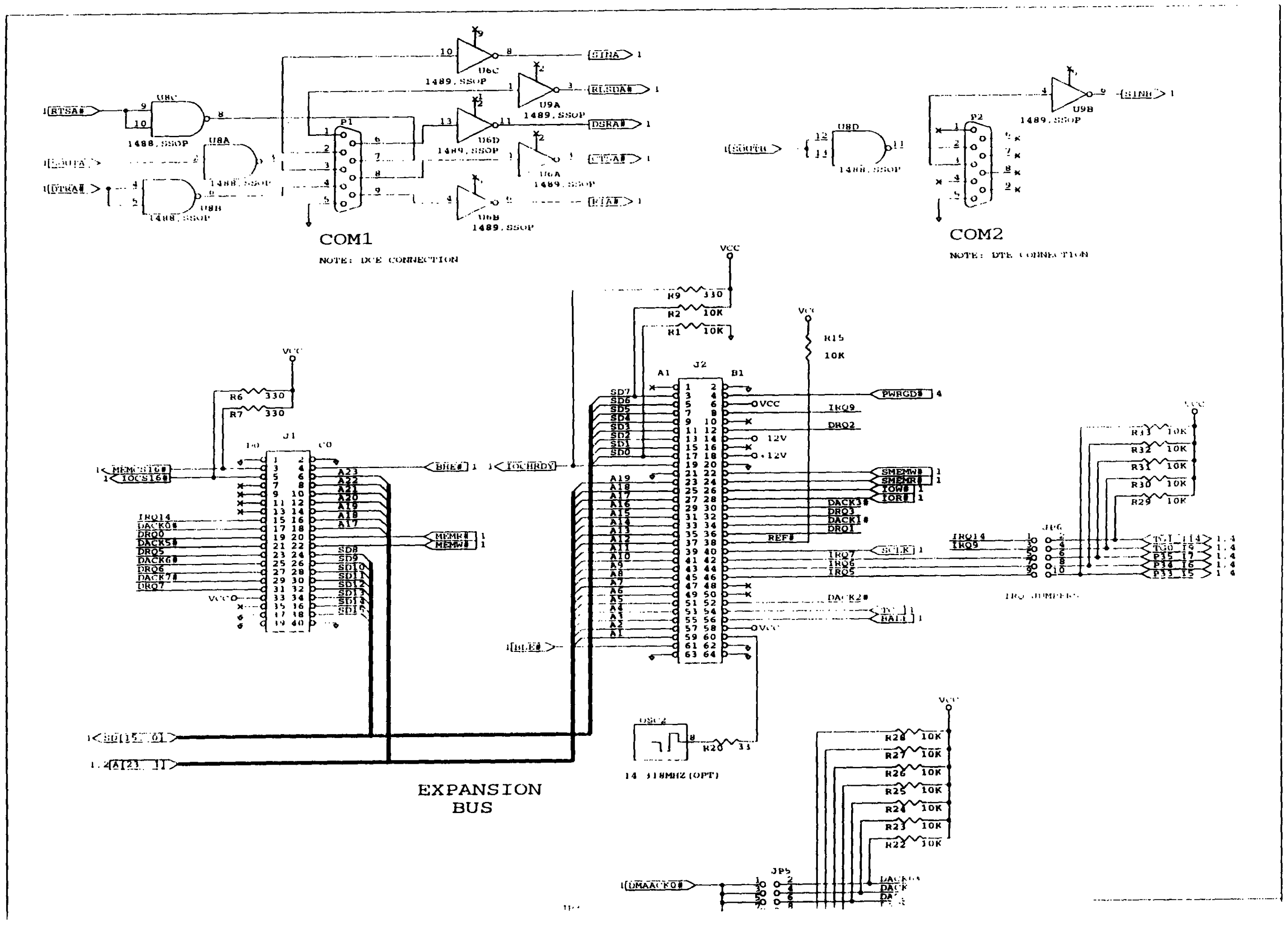



TITLE NMIX-0020 68HC11 & LCD & KEYPAD
D
AJK07
01
SHEET 1 OF 1
CIRI-ARDY 100992
RDF 071993
U1 68HC11
U2 2764/5164
U3 2764/5164
U4 2764/5164
U6 MAX232CWE
U8 74HC138
U9 74HC373
U10 SN75176
U11 SN75176
U12 74HC133
U13 74C923
U14 74HC27
U7 74HC00
U5 74HC00
VR1 LM7805
BR1 VM08
Y1 8.0MHZ
J1 LCD CONNECTOR
J7 KEYPAD CONNECTOR
LCD BASE ADDRESS=$B5FC
J2
J3
J4
J5
J6
J8
Q1 VP0300M
LV11
R7 10K
R6 5K
C7 22p
C8 22p
C1 0.1
C2 220 16V
C3 220 16V
C4 0.1
C5 0.1
C6 0.1
C9 10
C10 10
C11 10
C12 10
C13 0.1
C14 0.1
C15 0.1
C16 0.1
C17 0.1
C18 0.1
C19 0.1
C20 0.1
C22 0.1
C24 0.1
C25 0.1
C26 0.1
SPARE GATES
PORT E
PORT D
PORT A
DATA BUS
ADDRESS BUS



Although Intel has verified this design to be functional, Intel assumes no responsibility for any errors which may appear in the design. Intel reserves the right to modify this design without notice.

INTEL CORPORATION EPO 5000 W. CHANDLER BLVD. CHANDLER, AZ 85226		
Title	EV386EX EVALUATION BOARD	
Size B	Document Number	REV 6.1
Date: May 27, 1994	Sheet 2 of 4	


COM1
NOTE: DCE CONNECTION
COM2
NOTE: DTE CONNECTION
EXPANSION
BUS
1488.SSOP
1489.SSOP
U8C
U8A
U8B
U8D
U6C
U6D
U6A
U6B
U9A
U9B
P1
P2
J1
J2
A1
B1
C0
D0
R6 330
R7 330
R9 330
R2 10K
R1 10K
R15 10K
R20 33
OSC2
14.318MHZ (OPT)
JP6
JP5
IRQ JUMPERS
VCC
RTSA#
DTRA#
STRA
RLSDA#
DSRA#
CTSA#
RIA#
SINB
SOUTB
MEMCS16#
IOCS16#
BHE#
IOCHRDY
BLE#
SD[15..0]
A[23..1]
PWRGD#
SMEMW#
SMEMR#
IOW#
IOR#
SCLK
REF#
HALT
DMAACK0#
IRQ14
IRQ9
IRQ7
IRQ6
IRQ5
DRQ2
DACK3#
DRQ3
DACK1#
DRQ1
DACK2#
12V
-12V
MEMR#
MEMW#
R33 10K
R32 10K
R31 10K
R30 10K
R29 10K
R28 10K
R27 10K
R26 10K
R25 10K
R24 10K
R23 10K
R22 10K