Lab Manual
for
Single- and Multiple-Chip Microcomputer Interfacing

Peter Song
University of Texas

G. Jack Lipovski
University of Texas

Prentice Hall
Englewood Cliffs, New Jersey 07632

© 1988 by Motorola Inc.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

This document contains information on a new product. Specifications and information herein are subject to change without notice. Motorola reserves the right to make changes to any products herein to improve functioning or design. Although the information in this document has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.

Motorola, Inc. general policy does not recommend the use of its components in life support applications where in a failure or malfunction of the component may directly threaten life or injury. Per Motorola Terms and Conditions of Sale, the user of Motorola components in life support applications assumes all risk of such use and indemnifies Motorola against all damages.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-811605-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface		vii
1	**Introduction to the Buffalo Monitor**	**1**
	1.1 MC68HC11 Modes	1
	1.2 Using Buffalo for Program Development	2
	1.3 Buffalo Commands	6
	1.4 A Peek into the Buffalo ROM	8
	1.5 A Complete Three-chip Computer	11
2	**The Buffalo Monitor**	**12**
3	**The Greatest Common Divisor**	**15**
4	**A Random Number Generator**	**19**
5	**Internal Sorting**	**25**
6	**Linked Lists**	**29**
7	**The Huffman Code**	**32**
8	**A 6811 CPU Emulator**	**36**
9	**A 6811 Assembler**	**43**
10	**A Floating-Point Adder**	**51**
11	**Memory Systems**	**57**
12	**A Traffic-Light Controller**	**63**
13	**An IC Tester**	**67**
14	**A Logic Analyzer**	**71**
15	**A Bar-Code Reader**	**79**
16	**A Magnetic Card Code Reader**	**87**
17	**The Keyboard and LED Display**	**91**
18	**A DC and RMS Digital Voltmeter**	**98**
19	**A Thermometer**	**104**
20	**An Alarm Clock**	**108**
21	**Local Networks**	**116**
22	**A Floppy Disk Drive Controller**	**126**
23	**Using the MC68HC11A2 Chip**	**132**
Appendix A: Parts List		**144**
Appendix B: Motorola Ordering Information		**147**

List of Figures

Figure Title **Page**

1.1 Hardware diagram for the single-chip mode 11
10.1 Single-precision IEEE floating-point format 52
10.2 Range of single-precision floating-point numbers 53
11.1 Logic diagram of the memory systems 59
12.1 Traffic light arrangement 63
12.2 Control of an LED 64
13.1 Combinational circuit model 67
13.2 Sequential circuit model 68
13.3 Block diagram of an IC tester 69
14.1 Simplified block diagram of the logic analyzer 71
14.2 Block diagram of the logic analyzer 75
14.3 Timing diagram of the write cycle 76
14.4 State diagram of the logic analyzer 76
14.5 Logic Diagram of the address trigger 78
15.1 Number 642 in 2-of-5 format 80
15.2 Bar-code reader to the MC68HC11A8 interface 81
16.1 American Magnetics MagStripe^{TM} Card Reader signals 88
16.2. Credit card code 88
16.3 Magnetic card code ceader to the MC68HC11A8 interface 89
17.1 A 64-key virtual keyboard 92
17.2 A common-cathode 7-Segment LED 93
17.3 Logic diagram of the MC14499 control of a 4-Digit LED display 94
17.4 Multiplexer-decoder logic for keyboard scanning 95
17.5 Design of hexadecimal digits on 7-Segment LED display 96
17.6 Hardware for software multiplexing of a 4-Digit LED display 97
18.1 A Ladder Digital-to-Analog Converter 99
18.2 A Level Shifter 100
19.1 Temperature-voltage nonlinearity of a thermistor 105
19.2 A Temperature Sensing Circuit 105
20.1 Pitch duration symbols 111
20.2 Block diagram of the digital alarm clock 111
20.3 Block diagram of the alarm clock for the optional part 112
20.4 State diagram of the alarm clock 113
21.1 Frame format 117
21.2 Network interface logic 118
21.3 Manchester encoding of data 1011 119
21.4 Short frame format 119
21.5. CRC-16 generating function 120
21.6 Unbalanced configuration 120
21.7 The three functional units of an Ethernet server 122
22.1 Magnetic coating with grains of floppy disk 126
22.2 Magnetic flux change a on recorded bit stream 127
22.3 Double-density floppy disk format 128
22.4 File organization in OS-9 129
23.1 A Layout for an MC68HC11A2 quad surface mount chip 135

List of Tables

Table Title **Page**

12.1 Traffic light sequence descriptor 64
15.1 2-of-5 code 80
20.1 Pitch scale 109
20.2 An array of notes denoting (duration,pitch) 115

Preface

This lab manual describes twenty one experiments with the MC68HC11 single-chip microcomputer. Nine are essentially software experiments, and twelve are hardware interfacing experiments. The manual is designed to accompany the text *Single- and Multiple-Chip Microcomputer Interfacing* by G.J. Lipovski, Prentice-Hall, 1987. It provides essential hands-on experience that reinforces the concepts taught in that text. The experiments are developed in three levels: standard, optional, and extra parts. They can be implemented on the MC68HC11A8 chip, the M68HC11EVB board, and the M68HC11EVM board. The standard part of each experiment is recommended for all students, and is designed to fit in the MC68HC11A8 EEPROM memory. The optional part is recommended for good students, and the extra part is suggested for advanced students. These can be implemented on the M68HC11EVB board, and the M68HC11EVM board. Although all three environments can be used for these experiments, the M68HC11EVB board is best suited to developing these student experiments. Also, although the MC68HC11A8 chip, the M68HC11EVB board and the M68HC11EVM board require only a power supply and a dumb terminal, an IBM PC or a Macintosh can be used as smart terminals to make the experiments more pleasant and more efficient in teaching concepts.

The first chapter describes the Buffalo monitor, which is the monitor in the MC68HC11A8 chip and the M68HC11EVB board. It is provided as a reference for all students. Further information is available in the M68HC11EVB Evaluation Board User's Manual, provided with that board. Similar information on the M68HC11EVM is available in the M68HC11EVM Evaluation Board User's Manual, provided with that board.

The first experiment described in chapter 2 shows how to get the MC68HC11A8 chip running in single-chip mode. It is quite easy to do, and students with a reasonable hardware background, having handled integrated circuits before, can expect to get their own computer running in a matter of hours. This experiment may be skipped for those students using the M68HC11EVB board and the M68HC11EVM board.

The second experiment described in chapter 3 shows how to use subroutines, handle numbers, and use input-output routines. The third experiment described in chapter 4 expands on these ideas. The fourth experiment described in chapter 5 shows how to handle sorting, the fifth experiment described in chapter 6 shows how to use linked lists, and the sixth experiment described in chapter 7 shows how to code and encode data in the Huffman code. These are useful experiments that teach the use of data structures. Experiments 8 and 9 show how to emulate a computer and write an assembler for one. These tie together the software techniques introduced in earlier experiments. Experiment 10 shows how to execute floating point addition. It is a useful experiment for those students who have to handle and process numeric data.

Chapter 11 begins the hardware interfacing experiments. A memory system designed there relates to ideas discussed in chapter 3 of *Single- and Multiple-Chip Microcomputer Interfacing*. A traffic light controller in chapter 12 of this manual relates to ideas discussed in chapters 4 and 5 of the textbook and teaches parallel output and synchronization of output signals with the outside world. The IC tester described in

chapter 13 is an application of ideas discussed at the end of chapter 4 of the textbook and shows the use of parallel I/O. Chapter 14 of the manual shows a logic analyzer, which exploits many of the I/O devices of the 6811. It relates to ideas discussed in chapters 4 and 7 of the textbook. Chapters 15 and 16 of the manual show a bar code reader and a magnetic card code reader, which relate to ideas discussed in chapter 5 of the textbook and feature interrupt handling. Chapter 17 of the manual shows a keyboard and display, which relate to ideas discussed in chapter 6 of the textbook and feature practical input/output for small computer systems. Chapter 18 uses the analog to digital converter, discussed in chapter 6, to implement a voltmeter, and chapter 19 uses it to build a thermometer. Chapter 20 of the manual shows an alarm clock, which relates to ideas discussed in chapter 7 of the textbook and features timing and music generation. Chapter 21 of the manual shows a local network, which relates to communication system ideas discussed in chapter 8 of the textbook. Chapter 22 of the manual shows a floppy disk controller, which relates to ideas discussed in chapter 9 of the textbook and features practical storage systems for small computer systems. Chapter 23 shows how to implement a project on the MC68HC11A2 chip, which has 2K bytes of EEPROM to contain longer programs than the MC68HC11A8, which has 512 bytes of EEPROM.

Each chapter begins with a bit of background information and theories relevant to the concepts presented in the experiments. In presenting these, we have tried to limit the discussion to bare essentials; just detailed enough to give students a feeling of confidence about the experiment to be carried out. In these presentations, you may find that the discussions lack depth and technical terms, as we have placed heavy emphasis on this manual being perceived as "a friendly guide" to students, and not a technical report. Although designed to be used with the text *Single- and Multiple-Chip Microcomputer Interfacing*, this lab manual could be used with any similar text that explains the concepts used in the experiments. We also tried to avoid a cookbook style of presenting experiments, where the experiments are finished if students do only what is told. We feel that students will learn more by actually designing their own solutions, in addition to conducting the experiments to verify them.

A student is not expected to complete all experiments in one semester, but they are presented here for further work and stimulation. The large variety of experiments should enable instructors to select suitable experiments for their class.

For the experiments in this book, the M68HC11EVB board or the M68HC11EVM board can be obtained from Motorola (See appendix B for ordering information). The M68HC11EVB is currently priced quite reasonably for university students. Assembler programs, AS11 for the IBM PC and DEBUG11 for the Macintosh, can be downloaded from Motorola's freeware line (See appendix B for information). Other parts used in the experiments are listed in appendix A. Most can be obtained from Motorola, but others may be found in surplus parts houses, or may be available in your inventory.

1 Introduction to the Buffalo Monitor

The Buffalo monitor is a program that resides in 8K bytes of ROM in MC68HC11A8. Since this 8K bytes of ROM is not user-programmable, Motorola has developed the Buffalo monitor and burned it into the ROM in the MC68HC11A8s to aid program development. Not all 6811s have this program. Customers can submit their own programs and Motorola will burn them into the 8K bytes of ROM in the MC68HC11A8s that they will buy, in place of the Buffalo monitor. The M68HC11EVB board also has the Buffalo monitor in a EPROM to be used with almost any 6811 chip. Although the details presented here pertain to the version 2.6, other versions of the Buffalo should behave quite similarly, if not identically.

1.1 MC68HC11 Modes

The MC68HC11 has four operating modes, of which two are of interest to us. They are single-chip and expanded multiplexed mode. Buffalo is designed to work in either mode, and is aware of the mode in which the MC68HC11 is operating after a reset. The behavior of Buffalo is essentially the same in either mode, with some minor differences that are described in this section.

Buffalo resides between addresses $E000 and $FFFF. It expects the internal RAM to be at $0000 to $00FF and the internal registers at $1000 to $103F. These addresses should not be changed. The entire 512 bytes of EEPROM is available for programming at $B600 to $B7FF. However, Buffalo uses a portion of the internal RAM for variables, look-up tables, and user stack, leaving only the first 54 bytes (from locations 0 to $35) for the user. Buffalo assumes a memory cycle rate of 2MHz for the MC68HC11 (which requires an 8 MHz crystal).

1.1.1 Single-Chip Mode

In the single-chip mode, the terminal interface is available through the Serial Communication Interface (SCI) port with 8 data, no parity, and one stop bits at 9600 baud rate. The baud rate of the SCI port can be changed by changing the contents of address $102B with the MM (Memory Modify) command to the following:

$30 = 9600	$31 = 4800	$32 = 2400
$33 = 1200	$34 = 600	$35 = 300

There is usually no host connection in single-chip mode, and consequently the HOST command does not work. The hardware diagram for single-chip mode operation is included at the end of this chapter.

1.1.2 The Expanded Multiplexed Mode

The only difference in the operating features between single-chip and expanded multiplexed modes is the loss of ports B and C to the multiplexed address/data bus in expanded multiplexed mode. Depending on its version, a particular Buffalo monitor expects additional devices such as RAM, ACIA, DUART, and flip flops to be interfaced to the MC68HC11A8. In this section, the configuration of the M68HC11EVB Evaluation Board (EVB) is described.

The EVB board emulates the single-chip mode of the MC68HC11A8 while actually running the resident MC68HC11A8 MPU in expanded multiplexed mode. It is designed primarily for developing application programs using the features of the MC68HC11A8 in single-chip mode, although expanded multiplexed mode operation is possible. Buffalo in the MC68HC11A8 internal ROM is disabled. Instead, an EPROM containing Buffalo is provided at the locations $E000 to $FFFF. This configuration enables a developer to upgrade the Buffalo monitor or totally replace Buffalo with an application program. For aiding the program development, 8K bytes of RAM is provided at the locations $C000 to $DFFF. The internal RAM, registers, and the EEPROM appear as in single-chip mode.

A terminal interface (HOST) is provided by the SCI port, as in single-chip mode. An additional serial interface (TERM) is provided through an Asynchronous Communication Interface Adaptor (ACIA). Upon coming out of a reset, Buffalo automatically determines which connection to use for the subsequent dialog interface by sending a message to both and selecting the first one that responds with a carriage return. The TERM port can be set to one of 300, 600, 1200, 2400, 4800, and 9600 (default) baud rates with a jumper (J5). Although the baud rate of the HOST port is not hardware selectable, it can be changed in software. The default baud rate (upon reset) is 9600.

1.2 Using Buffalo for Program Development

When an MC68HC11A8 is powered up, or reset, and Buffalo is invoked, the following message is displayed on the terminal:

BUFFALO X.X (ext) - Bit User Fast Friendly Aid to Logical Operation

where X.X is the version number of the particular Buffalo program. The (ext) indicates an external EPROM version as in an EVB; (int) indicates an internal ROM version as in an MC68HC11A8 chip. Type a **carriage return**, and Buffalo should respond with an angle bracket, >. The angle bracket is the prompt issued by Buffalo, indicating that it is ready to receive a command. The available functions are shown below to give the feel of the Buffalo monitor:

MEMORY MODIFY	BLOCK FILL	TRACE	BULK ERASE
TRANSPARENT MODE	MEMORY MOVE	HELP	DOWNLOAD
MEMORY DISPLAY	ASSEMBLE	GO	CONTINUE
REGISTER MODIFY	BREAKPOINT	VERIFY	CALL SUB

1.2.1 Loading a Program into Memory

Suppose you want to assemble the following program from address $000C and run it on an MC68HC11A8:

```
            LDAA    #32
LOOP        DECA
            BNE     LOOP
            SWI
```

In order to run this program, it must first be assembled and loaded into the memory. The ASM command will do all that. Type **ASM 000C** , followed by a carriage return, to go into the assembly mode. Normally, only enough characters of a command to distinguish it from other commands are required for Buffalo to identify the command. So you could type **A 000C**, and that would be sufficient. Now, Buffalo should respond by displaying something like this:

```
>asm 000c           {This is the line you just typed.}
000C  STOP $FFEF    {Buffalo responds with a line that looks somewhat like this.}
>                   {Buffalo waits for  a line of assembly mnemonics.}
```

Type in the first instruction, end it with a carriage return, and you should see the following:

```
> ldaa #20          {This is what you should have typed.}
      86 20         {Buffalo displays the code for the instruction.}
000E  STOP $FF,X    {Buffalo tries to disassemble the next locations but fails.}
>                   {Buffalo waits for the next instruction.}
```

The hexadecimal numbers $86 and $20 are the object code for the instruction LDAA #$20. Buffalo automatically interprets every number to be in hexadecimal notation, so instead of 32 (in decimal), 20 (in hexadecimal) should be used. This is different from most assemblers, in which all numbers are assumed to be in decimal notation unless indicated otherwise by some special character. The second instruction in our little program has a label, but the one-line assembler does not understand labels, hence labels are not allowed. Instead, you should make note that the address of the label LOOP is $000E and type in just the instruction **DECA**. The third instruction has a branch to a label. For this, you should type **BNE 000E**, in which case the following is displayed:

```
> bne 000e          {This is what you type.}
      26 FD         {Buffalo assembles the BNE $000E instruction.}
0011 LDAA $2421 {Buffalo tries to disassemble the next locations and succeeds.}
> swi
      3F            {Buffalo assembles the SWI instruction.}
0012 BHS $0033 {Again, Buffalo succeeds in disassembling.}
>
```

In branch instructions, the address of a label should be given as the operand. To exit from the assembly mode, type **Control A**. At this point, if the command **A 000C** is given for the second time, Buffalo will respond with

```
000C LDAA #$20
>
```

which is exactly the instruction that was assembled into that and the next location. Before accepting a new instruction to assemble, Buffalo will try to disassemble the opcodes at that and the following locations. That is, it will try to convert the hex numbers into instruction mnemonics. If the numbers happen to be those of valid instruction opcodes, as in this case, its mnemonic is displayed. If not, the instruction STOP is displayed. If you type a carriage return without giving a new instruction, the next locations are disassembled, and Buffalo is ready to assemble and load an instruction into those locations. This is a convenient way of checking to see whether the program is correctly assembled and loaded into the memory.

The ASM command works only in the RAM area, both in internal and external. To program the internal EEPROM in the single-chip mode, first load the program into RAM locations and then move the block of codes from the RAM to the EEPROM with the **MOVE** command. The command **MOVE 0002 00A3 B600** will move the block of memory from locations $0002 to $00A3 to locations starting at $B600. The Memory Modify command can also be used to modify the EEPROM, RAM, or the internal registers, but only one byte at a time. For example, to modify the content of location $B600, type **MM B600**, ending with a carriage return. Buffalo will respond by showing the content of the location $B600, and is ready to receive a byte of data to be put in this location. You may type the data (the data is optional; if given, it is put into the location; if not, nothing is done to the location) followed by a carriage return to quit the command, a slash to examine the same location, a backspace or an up arrow to open up the previous location, a linefeed to open up the next location, or a space to open up the next location without displaying its address.

1.2.2 Executing a Program

Now that it is loaded in memory, we are ready to execute the program. The **GO 000C** command will start executing the program from the address 000C until an SWI instruction is encountered. With more complicated programs, in which case there are likely to be bugs, executing the program without first debugging it is not advisable. It only takes one bug to go into an infinite loop or, worse yet, wipe out the program that you just assembled by overwriting itself. Instead, breakpoints should be set at critical places in the program, such as the beginning or ending of loops, branch points, or any other places that you know what the "bug-free" program should do. The command **BR 000F** will set a breakpoint to the address $000F. That breakpoint can be removed with the **BR -000F** command. Breakpoints can only be set to locations in RAM and never on EEPROM or self-branching instructions (L BRA L). The SoftWare Interrupt

instruction (SWI) is used by the BREAK command, so a breakpoint cannot be set to that instruction either. When a SWI instruction is encountered, the MC68HC11A8 saves the registers PC,X,Y,A,B,CCR, and S used by the user program, and starts executing the SWI handler routine. Through this routine, Buffalo takes over the control of the MC68HC11A8 from the user program and puts you back into the command mode. There can be up to four breakpoints set at any one time.

After stopping the program before a suspected bug, instructions can be "traced" to monitor the program closely. **T 10** will trace, that is, execute one instruction and display the contents of the registers, for 16 ($10) instructions. **T** will trace one instruction. A **carriage return** by itself will repeat the previous command, which is useful in tracing one instruction at a time. Be sure that the registers, especially the PC, are set to the desired values before the trace command is given. The Proceed (**P**) command will continue the program from the current user program counter until the next breakpoint or SWI instruction is encountered.

If you want to debug a block of code that is in the middle of the program or is difficult to get at because of difficult-to-satisfy conditions, the Register Modify (**RM**) command can be used to change any of the registers, including the program counter. The command **RM A** will display the content of all the registers and open up the Accumulator A for modification. When given without an argument, all the registers are displayed and the PC is opened up. After typing in the new value for a register (this is an option, as with the **MM** command), you may type a carriage return to quit or a space to open up the next register. When using this command, be careful not to change the condition code register and stack pointer to some unknown value. Buffalo initially sets the S,X, and I bits of CCR and assigns the stack pointer to the top of the available memory spaces in internal RAM ($4A in Buffalo 2.6). Also any subroutine, both in your program and in Buffalo, can be called at any point in the program with the **CALL** command. For instance, **C B702** will invoke and execute a subroutine whose starting address is $B702. Such a subroutine must terminate with an RTS instruction.

The command Memory Display (**MD**) can be used to view the contents of <u>any</u> memory blocks, in a multiple of 16 bytes at a time. For instance **D 0002 0032** will display the contents of the locations $0000 to $003F. **MD 0002** will display the contents of locations from $0000 and the next eight blocks of 16 bytes. If you are ever in need of help, type **H** (for Help) or **?**, in which case Buffalo displays the command summary.

1.2.3 Downloading into the EVB

Loading a program into memory in the EVB board is somewhat easy. Instead of using the one line assembler, the program can be assembled in a host computer such as a VAX, and the object codes can be downloaded into the memory into the EVB board. To do that, first go into the transparent mode using the **TM** command. This mode enables you to communicate directly to a host machine (the VAX), as if the terminal that you are working with is connected directly to a host (which by the way is connected to port B of the DUART). Login to the host machine, and do what ever is required to set up for download. The Control A character is used to exit from this mode.

In the host machine, you should have already assembled the program and the object code in Motorola "S record" format. Your program should be assembled to use the RAM locations, namely $C000 to $DFFF and the first 50 bytes of the internal RAM. The command **LOAD <cmd>** is used to start the downloading. Here, the <cmd> is the command that you give to the host to display the content of the object code file. Usually, it is something like CAT, LIST, TYPE, or SHOW. For example, if the host is a Unix machine and the object file name is demo.out, the command **LOAD cat demo.out** will start the download process. Buffalo does not echo the content of the object file, but a message is printed when downloading is done. Then issue **VERIFY cat demo.out** to verify the download data. The VERIFY command works the same as the LOAD command, but instead of storing, it compares what is in the memory with the data from the host.

Downloading can be done through the terminal interface as well. If you are using a terminal emulator program on a personal computer such as Macintosh, IBM PC, or TRS-80, the program can be assembled with an MC68HC11A8 cross assembler into S record format and downloaded with a file transfer function. The command **LOAD T** is used. If a Macintosh is accessible, you can use Debug11, a freeware program written by Dr. G.Jack Lipovski, to simplify the tasks of editing, assembling, and downloading.

1.3 Buffalo Commands

1.3.1 Syntax

In describing the syntax of a command, optional arguments are inside a pair of square brackets [ ] and hexadecimal arguments are inside a pair of angled brackets < >.

ASM [<adrs>] Starts the single line assembler for the MC68HC11A8 at <adrs>, defaults to the starting location, 0. Use Control A to exit.

BF <adrs1> <adrs2> <data> Fill a block of memory from <adrs1> to <adrs2> with value <data>.

BR [-][<adrs>] . . . Specify <adrs> to set up to four breakpoints. Use the - option alone to delete all breakpoints, or specify -<adrs> to delete the breakpoint at location <adrs>.

BULK Bulk erase the EEPROM.

BULKALL Bulk erase the EEPROM and the CONFIG register.

CALL [<adrs>] Executes the subroutine at <adrs> and returns to Buffalo. Defaults to the current value of the program counter.

G [<adrs>] Sets the program counter to <adrs> and starts executing the instructions until SWI instruction is encountered. Defaults to the current value of the program counter.

HELP or ? Displays the command summary of Buffalo.

LOAD <host display command> Downloads an object code file via HOST port. <host display command> is issued to the host.

LOAD T Downloads an object code file via TERM port.

MD [<adrs1> [<adrs2>]] Displays a block of memory on a 16 byte boundary. The <adrs1> and <adrs2> are used as the range, and if not specified, defaults to the last opened location.

MM [<adrs>] Opens up the memory location <adrs> for modification. Defaults to the last opened location.

MOVE <adrs1> <adrs2> [<dest>] Moves a block of memory from <adrs1> to <adrs2> to locations starting from <dest>. The destination defaults to <adrs1>+1.

P Same as G, but ignores the breakpoint on the first instruction.

RM [P,X,Y,A,B,C,S] Displays the contents of all registers, and opens up the specified register for modification. Defaults to the program counter.

TRACE [n] Single steps through the next <n> instructions. The program counter should be set to the first instruction to be traced. Defaults to one instruction.

TM Enters transparent mode, so that you can talk directly to a host computer. Use Control A to exit.

VERIFY <host display command> Verifies downloaded data via HOST port.

VERIFY T Verifies downloaded data via TERM port.

1.3.2 Special Characters

Control A	returns to Buffalo from transparent mode and assembly mode
Control H	backspace
Control B	sends break to host in transparent mode
Control W	suspends the execution of a program, restarted by any key
Control X	abort
DEL	also abort
<cr>	repeats the previous command
?	help

1.4 A Peek into the Buffalo ROM

Rather than writing many utility routines such as those for input and output, you can use subroutines that are in the Buffalo ROM. These are listed below. Also, the interrupt handler jump table is handled by Buffalo. This table is described next. Finally, the reset vector is described.

1.4.1 Input and Output Routines

Of the numerous I/O and utility subroutines in Buffalo, only a few that seem to be useful are described here. For the descriptions of other routines, refer to the EVB User's Manual.

OUTSTRG **$FFC7**

Output string of ASCII characters pointed by the X register until the end-of-transmission ($04) character is met. A carriage return and a linefeed is output before the character string. X register is changed. The output process can be suspended with the Control W key and is resumed by any key.

OUTA **$FFB8**

Output the ASCII character in accumulator A. No register is changed.

OUT2BSP **$FFC1**

Convert two consecutive binary bytes pointed to by register X to four ASCII characters, in hexadecimal representation, and output them, followed by a space. Register X is changed.

INCHAR **$FFCD**

Wait until a character is received. Return in accumulator A the ASCII character received from the I/O port, and echo the character. Other registers are not changed.

INPUT **$FFAC**

Read the I/O port once and return, in accumulator A, the character if there, or 0 if not. This subroutine does not wait for a character to be received. Other registers are not changed.

INIT **$FFA9**

Initialize the I/O device indicated by the IODEV flag at location $AA. Set IODEV to 0 for SCI, 1 for ACIA.

UPCASE **$FFA0**

Convert the ASCII character in accumulator A to uppercase.

Normally, if these routines are used after Buffalo is invoked, you need not change anything, because Buffalo initializes them. However, if Buffalo is bypassed (how this is done is described in the last section), your program should initialize the necessary I/O devices using the INIT routine. The SCI is always initialized at 9600 baud rate, but can be changed after the initialization by modifying the BAUD register at $102B.

1.4.2 A Jump Table for the Interrupt Vectors

The original interrupt vector locations for the MC68HC11A8 are between $FFC0 and $FFFF. These are the locations where the starting addresses of the various interrupt handlers are stored. For instance, the address of the SWI interrupt handler routine should be put into the locations $FFF6 to $FFF7, so that when an SWI interrupt occurs, the CPU will know exactly where to go to service the interrupt. But these locations are in ROM space ($E000 to $FFFF), which means that you, as a programmer, cannot write to them.

One solution is to burn in the vector locations in ROM with the addresses of internal RAM locations, so that when an interrupt occurs, the program counter is loaded with an address in RAM from the interrupt vector and starts executing from the location in RAM. At these locations, you can put JUMP instructions to jump to your interrupt handler routines. This is exactly how Buffalo provides you with access to the interrupt vectors. The addresses that are in the interrupt vectors in ROM are listed below. As an example, if an IRQ handler starts from location $B700, the opcodes for the instruction **JUMP $B700** should be put into locations $00EE to $00F0. That is, the locations should have the values $7E, $B7, and $00. When using Buffalo to debug your program, don't use the Output Compare 5 module; it is used by Buffalo for the TRACE function.

SCI	$00C4
SPI	$00C7
Pulse Accumulator Input Edge	$00CA
Pulse Accumulator Overflow	$00CD
Timer Overflow	$00D0
Timer Output Compare 5	$00D3
Timer Output Compare 4	$00D6
Timer Output Compare 3	$00D9
Timer Output Compare 2	$00DC
Timer Output Compare 1	$00DF
Timer Input Capture 3	$00E2
Timer Input Capture 2	$00E5
Timer Input Capture 1	$00E8
real-time Interrupt	$00EB
IRQ	$00EE
XIRQ	$00F1
Software Interrupt	$00F4
Illegal Opcode	$00F7
COP	$00FA
Clock Monitor	$00FD

1.4.3 The Reset Vector

When the MC68HC11A8 running Buffalo is reset, two things can happen, depending on its hardware configuration. If the port E0 (that is port E, bit 0) is tied to ground, the Buffalo monitor will come up. If it is tied to VDD, execution immediately proceeds to address $B600 in EEPROM. If you have a program stored in EEPROM, that will be executed without really going into the Buffalo monitor. Of course, the program must start at address $B600.

The reset vector is not really changed at all. The following is the actual listing of a portion of the Buffalo monitor program, which will explain how this is done:

```
RSTHND   LDX #PORTE
         BRCLR 0,X $01 SKIP      see if bit 0 is clear
         JMP $B600               jump to EEPROM
SKIP     . . .                   beginning of Buffalo
```

The reset vector is set to the address of RSTHND, and depending on the signal on port E bit 0, the program at $B600 can be executed. It is important to set the stack pointer in such programs, using an instruction like LDS #$FF, because the stack pointer comes up in an unpredictable state after reset and the Buffalo monitor does not get a chance to initialize it before it causes the program at $B600 to be executed.

One last word of caution: When the 6811 stops running as power is turned off, it runs wild. While computers without EEPROM also do this, those computers do not experience any problems. The 6811, with EEPROM, may erase parts of EEPROM when running wild as power is turned off. To prevent this, you should always initialize the interrupt vector for the ILLEGAL INSTRUCTION, to execute the routine below:

```
ILLINS  CLRA
        TAP
        STOP
        BRA ILLINS
```

Also, the COP monitor may try to restart your machine when it is STOPPED. You may have to defeat the COP monitor, or cause it to run the same routine as the previously given illegal instruction handler.

Finally, a chip called a low-voltage inhibit chip (LVI) has been developed to help prevent program runaway when power is turned off. This three-terminal chip senses the voltage between two of its pins (sense, connected to the power supply 5 volt line, and ground) and shorts the third pin to ground as long as the sense pin is below 4.5 volts. When power is turned off, it stops the processor from dammaging its EPROM by forcing it into a reset condition. By the way, when power is applied, it also prevents the processor from starting until the power supply is high enough for reliable operation. SEIKO made the first LVI and Motorola has developed one, the MC34064. The SEIKO part uses much less current and is preferred in low power systems. We will show the Motorola part in the next section.

1.5 A Complete Three-chip Computer

Figure 1.1 shows a complete three-chip computer. The front cover of this manual shows a picture of this computer (except that a capacitor was used rather than the MC34064 LVI in the reset circuit). With the Buffalo monitor in ROM, this is a complete microcomputer. Chapter 2 shows an experiment that gets this system working with an MC68HC11A8 chip. It can be used for all the experiments in this book except the last two (chapters 21 and 22) for the "standard" part. The EVB board is needed for the "extra credit" and "optional" parts. However, chapter 23 shows how to use the MC68HC11A2, which has 2K bytes of EEPROM, and is able to run all experiments, with the MC68HC11A2 in place of the MC68HC11A8 in this diagram.

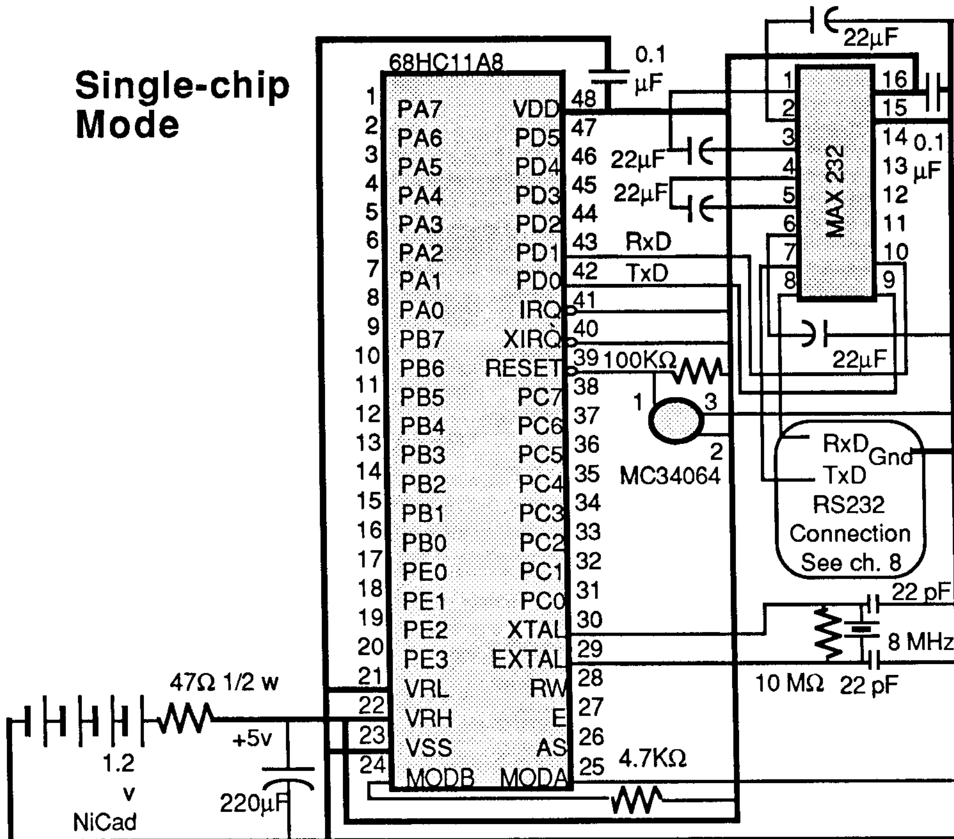


Figure 1.1. Hardware diagram for the single-chip mode

2 The Buffalo Monitor

2.1 Goals

1. To build a two-chip microcomputer system based on the MC68HC11A8
2. To get acquainted with the Buffalo monitor

2.2 Introduction

The Motorola MC68HC11A8 is an 8-bit microcomputer that has 256 bytes of RAM, 512 bytes of EEPROM, 8K bytes of ROM, timers, serial and parallel interface, and analog-to-digital converters in one chip. With only a minimum amount of additional hardware, a very useful microcomputer system can be built. In the 8K bytes of internal ROM resides the Buffalo monitor, which stands for Bit Users Fast Friendly Aid to Logical Operation. This program, written in 6811 assembly language, aids users in programming the MC68HC11A8. It is somewhat amazing: here is a chip that has just about everything that a microcomputer needs and a program to program itself.

We recommend that you review chapters 1 and 2 of *Single- and Multiple-Chip Microcomputer Interfacing* and chapter 1 of this manual, "Introduction to the Buffalo Monitor".

2.3 Description

Port D is a general purpose I/O port that can also be used as a serial communication interface. Bit 0 is used to receive characters from a terminal through a RS232C connection. Likewise, bit 1 is used to transmit characters. This RS232C connection is provided to the MC68HC11A8 by another "whiz" chip, the Maxim's MAX232C (discussed in chapter 8 of the textbook).

In addition, the MC68HC11A8 needs to have a crystal oscillator to get timing marks from. A crystal oscillator, when given electric energy, vibrates at a fixed frequency, like quartz in a watch. An 8 MHz crystal is needed to give the clock frequency of 2 MHz for the MC68HC11A8. Also, capacitors are needed to protect the IC chips from sudden voltage spikes, which could damage the chips. Capacitors are also needed to filter the signals, which sometimes are noisy, between IC chips. Pull-up resistors are needed to tie some of the input and open-collector output pins to a logic high.

2.4 Preparation

The following parts are needed:

a. Motorola 68HC11A8
b. MAXIM MAX232C
c. MC34064 LVI chip
d. 8 MHz crystal
e. two 30 (or 22) pF, two 0.1 μF capacitors
f. four 22 μF polarized capacitors
g. one each, 4.7K 100KΩ and 10M Ω resistors
h. a reset switch
i. for a battery operated system, 220 μF capacitor, 4.7K Ω (1/2 w) resistor, and four 1.2 volt NiCad batteries

2.5 Procedure

1. Build the circuit shown at the end of chapter 1. Make the connections sturdy, for this circuit will be used in upcoming experiments. If you do not know how the breadboard is connected, use an ohmmeter or a continuity checker to find out.
2. Before connecting the circuit to a power supply, check for a short between the power and ground.
3. Connect the RS232 connector to a terminal, and connect the power. When you see Buffalo's greeting message, type a carriage return.
4. Assemble the following program

```
        ORG     $000B
        LDAA    $E003
        LDAB    $E00C
        LDX     $E049
L       NEGA
        ABA
        DECB
        NEGA
        ABX
        STAA    $0003
        CBA
        BLE BLS L
        CLRA
        SWI
```

5. Using the Buffalo monitor, answer the following questions:

a. What are the values of accumulators A and B after the instruction ABA is executed for the 1st, 2nd, and 3rd times ?
b. Trace the program from the instruction DECB to CBA during its 3rd loop, and give the values of all registers.
c. How many times is the loop repeated ?

d. What is the value stored in location $0003 after the program is finished ?
e. Move the program to EEPROM locations, starting at $B600, and give the contents of locations $B600 to $B01F in dump format.
f. Can a program in EEPROM be traced ? Explain your answer.

2.6 Hints and Suggestions

When a breakpoint is set to an instruction, the execution of the program stops just before the instruction. To find out the number of times a loop is repeated, set a breakpoint to either the beginning or the end of the loop, and count the number of times the instruction is repeated. Or if there is a loop counter inside the loop, the difference between the initial and final value of the counter is the loop count.

When using the GO or the PROCEED command, make sure that the program counter points to an instruction inside the program. Otherwise, the program goes into never-never land, and the system will have to be reset.

3 The Greatest Common Divisor

3.1 Goals

1. To understand how the characters are transmitted between a terminal and a computer in the ASCII character set
2. To understand how numeric input/output operations are done
3. To implement several subroutine parameter passing techniques
4. To introduce the recursive programming technique

3.2 Introduction

In this experiment we will write programs to find the greatest common divisor of two integers using various algorithms. We will write the programs so that they will read and write numbers from and to the terminal. The numeric input/output operations are different and more complicated than the text I/O operations because they are built on top of the character I/O operations. We will learn how this is done.

We recommend that you review sections 1-2, 2-1, 2.2, and 2-3 of *Single- and Multiple-Chip Microcomputer Interfacing*. For further information, consult chapter 1 of *Algorithms*, by Robert Sedgewick, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1984.

3.3 Description

3.3.1 The Numeric Input/Output Operation

Inside a terminal, there is a microprocessor that scans the keyboard periodically to see whether a key has been pressed. When a key is pressed, the microprocessor knows exactly which one it is and sends out the information indicating that a particular key has been pressed on the keyboard to another computer. But in order for the other computer to understand which key, both the terminal and the computer have to agree upon the meaning of the bit stream each sends and receives. This is why we have the ASCII character set. The term ASCII is an abbreviation for American Standard Code for Information Interchange, a system for representing alphanumeric data using seven binary digits. For example, the character *A* is represented as 1000001, and the character *9* is represented as 0111001 in ASCII. This is how the terminals and computers understand each other; whenever one receives a binary number 1000001, it knows that this represents the character *A*. There are other standards, such as EBCDIC, that use different representations for the alphanumeric characters, but the ASCII code is prevalent in the United States.

Suppose that you want to input the number 29 into a program. If you press the keys 2 and 9, which is 29, what is sent to the computer is the binary number 0110010 ($32) and 0111001 ($39). These are the numbers 2 and 9 represented in ASCII for transmission. These two numbers must be converted to binary to be used in the computer. First the numbers must be converted to decimal representation. This is easy: subtract $30 from each number, and you have 2 and 9 in decimal. Now the numbers must be converted to binary representation. This is also easy: multiply the number 2 by 10 (since 2 has a weight of 10 in decimal representation) and add the number 9 to it. The result is the number 29, which is 00011101 in binary.

To display the number 29 on screen, the reverse must be done. The binary number 00011101 is first converted into decimal digits 2 (00000010) and 9 (00001001). Each decimal number is then converted into ASCII $32 (00110010) and $39 (00111001). Then the ASCII numbers are sent to the terminal, one at a time. The terminal knows what to do with the ASCII numbers.

3.3.2 The Brute-Force Method to find The GCD

The brute-force method to find the greatest common divisor of two numbers is to test all the integers from the smaller of the two down to 1, until a number that divides both numbers is found.

3.3.3 Euclid's Algorithm

Euclid's algorithm for finding the greatest common divisor is shown below. This algorithm is based on the discovery that the greatest common divisor between two numbers, say u and v, is the same as the greatest common divisor between v and $u - v$, if u is greater than v. So to find the gcd of u and v, find the gdc of v and $u - v$, and repeat this process until it is found. Here, u is always the number that is the greater of the two. However, when the smaller number, v, is repeatedly subtracted from the larger number, u, until the result is smaller than v, the result is the same as the remainder of u **mod** v. In other words, the gdc of u and v is the gcd of v and u **mod** v, ... until u **mod** v leaves no remainder. Then v is the gcd. For example:

gcd(102,15) = gcd(15, 102 mod 15) = gcd(15, 12)
= gcd(12, 15 mod 12) = gcd(12, 3)
= gcd(3, 12 mod 3) = 3, since 12 mod 3 leaves no remainder

Euclid's algorithm is shown below in C.

```
int gcd(u,v)
int u,v;
{ if (v == 0)  return(u);
  else  gcd(v, u % v);
}
```

3.3.4 The Iterative Method

The recursive solution to finding gcd is somewhat tricky and is certainly inefficient. For instance, the return address and the variables *u* and *v* have to be saved on stack when the recursive call to itself is made. In many instances, tail recursions can be removed easily. Instead of having it recursively call itself, branch back to the beginning of the routine with new, smaller values of *u* and *v*. The converted algorithm is shown below in C.

```
int gcd(u,v)
int u,v;
{ int t;
  while ( v != 0 )
   { t = u % v;   u = v;   v = t; }
  return(u);
}
```

3.4 Procedure

3.4.1 Standard Part

Write a program that calculates the greatest common divisor of two numbers. The specifications are:

1. The program must be position-independent and modular.
2. The two numbers are to be input from keyboard, and greatest common divisor of two is to be displayed on the terminal screen, all in decimal notation. The input/output format is flexible.
3. The "brute-force" algorithm must be used to calculate the gcd.
4. The functions to input, calculate gcd, and output must be implemented as subroutines.
5. The variables *u* and *v* must be passed on the stack.
6. The value of the gcd may be passed through a register.
7. The two numbers are less than or equal to 65535, and are error free.
8. Document all routines with algorithms expressed in a high-level language such as Pascal, C, or a pseudo-language.

3.4.2 Optional Part

Write a program that calculates the gcd of two numbers using Euclid's recursive algorithm. Registers may be used to pass the variables *u*, *v*, and the result in the recursive subroutine.

3.4.3 Extra Credit

Write a program that calculates the gcd of two numbers using an iterative algorithm.

3.5 Hints and Suggestions

3.5.1 Structured, Modular Programming

The following structure is suggested for the main program:

```
main()
{ unsigned u,v,    /* allocate variables on stack */
            d,     /* accumulator D */
           gcdb();
  input(&u,&v);  /* input u and v, convert to binary, return through the stack */
  d = gcdb(u,v);  /* calculate the gcd, return it through accumulator D */
  out5(d);        /* convert the gcd into decimal, display it on screen */
}
```

In this way, the subroutines input and out5 can be used in all three parts of the experiment without any modification.

3.5.2 Binary-to-Decimal Number Conversion

One way to perform the binary-to-decimal number conversion is to count the number of times a decimal weight can fit in the number (i.e., count how many 1000s can fit in 9999). This is repeated until the number of 1s is counted. This process should start with the decimal weight whose magnitude is one order less than the largest number possible. For instance, to convert the binary number 104 into decimal, you want to know that there is one 100s, no 10s and four 1s. If you get something like ten 10s, you are in trouble because the number 10 is not an allowable digit in decimal notation. In decimal notation, only the digits 0 to 9 are allowed.

If integer **div** and **mod** operations are available, as with the 68HC11s **idiv** instruction, the conversion is simple. If the number n is less than or equal to 999, then (n div 100) will give the number of 100s and ((n mod 100) div 10) will give the number of 10s. If these operations are not available, repeated subtraction operation can be used instead, as shown below in C:

```
for (hundred = 0; number >= 0; hundred++)  number -= 100;
number += 100;   /* restore after subtracting once too many. */
```

4 A Random Number Generator

4.1 Goals

1. To understand how pseudorandom numbers can be generated in computer
2. To implement parameter passing after the call techniques
3. To implement multiple-precision arithmetic routines

4.2 Introduction

The term *random number* implies unpredictability or a lack of pattern in a sequence of numbers. In mathematics, this term is more clearly defined as an element in a set of numbers whose probability of occurrence is equal. The equal probability of occurrence does not mean that every number in the set must occur once every so often. It is more likely that, with uniform random numbers, some may occur more than once, and others may not occur at all. In addition, the sequence of occurrence must also be random. A number sequence of a million integers is not particularly random if they occur in increasing or decreasing order.

The random numbers are used in cryptography, where a message is scrambled so that it is unrecognizable to persons other than the intended recipients. The random numbers are also used to simulate some aspects of the real world, where a set of events occur randomly. They are also used in situations where one or more arbitrary numbers are required, such as in a lottery or dormitory room assignments. Usually, a random number can be used instead of an arbitrary number, but not vice versa.

It is not possible for a finite state machine, such as a computer, to generate truly random numbers; the sequence of numbers it generates eventually cycles, no matter how large it is. However, if the cycle is very large compared to the required length of the random number sequence, then the computer-generated pseudorandom numbers can still be considered "random."

We recommend that you review sections 2-3 and 2-4 of *Single- and Multiple-Chip Microcomputer Interfacing.* For a detailed study of random number generators, consult section 3.6 of *The Art of Computer Programming,* by Donald Knuth, vol. 2, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1984. For an in-depth presentation of the parallel binary multiplication and division techniques, consult unit 21 of *Fundamentals of Logic Design,* 2nd ed., by Charles H. Roth, Jr., West Publishing Company, New York, 1979.

4.3 Description

Random number sequences are used extensively in many applications, especially in simulations. And as such, many systems provide uniform Random Number Generators (RNG), which can be used to generate random numbers of any distribution. A uniform RGN generates a random sequence of numbers in the range of [0,1]. Using various transformations, this random sequence is converted to the desired distribution. Because of its frequent use, the efficiency, as well as the randomness, is a critical performance feature of random number generators. Almost all uniform RNGs use integer, and not floating-point, operations because they are fast. The generated integer is converted to a number in the [0,1] range by dividing it with the largest possible integer that the uniform RNG can produce, which in many cases is the largest possible integer that the host machine can represent.

There are three proven methods of generating a random number sequence; linear congruential, additive congruential, and subtractive. In this experiment, we will deal only with the more popular linear congruential and subtractive methods.

4.3.1 The Linear Congruential Method

The general equation of the linear congruential method is

$$A_i = (A_{i-1} * B + C) \text{ MOD } M$$

where B, C, and M are some constants and A_{i-1} is the previously generated random number. An arbitrary number, A_0, known as the seed, is used to start the sequence, and depending on the seed, different number sequences are generated. The number generated is always less than M.

For obvious reasons, the value of M should be as large as possible. This may be the largest unsigned integer value that a computer can represent, which usually is some power of 2. In this way, the MOD operation can be done simply by ignoring the overflows or by truncating. Choosing the constant B is neither so obvious nor so simple. Many statistical studies indicate that it should not be too large or too small. It should be a number that is one order of magnitude less than M, and it should end in a sequence of digits ...x21, where x is an even number. The choice of the constant C does not seem to have much effect on the quality of the random number sequence generated, and in many cases the value of 1 is used. The value of the first random number also seems to have little effect on the validity of the sequence. The period of the linear congruential RNG is at most M.

The complexity of this algorithm is one word of storage and three arithmetic operations. If the word length is long enough, say longer than 32 bits, this algorithm is fast. However, smaller microprocessors, such as the MC68HC11A8, must use multiple-precision arithmetic routines to provide a sufficiently long period. Sometimes the generated random number sequence is shuffled to increase the period. This technique is the basis of the subtractive method described in the next section.

4.3.2 The Subtractive Method

The subtractive method uses more than one previously generated random numbers to generate the new number. Although there are others, the most commonly used equation is

$$A_n = (A_{n-55} - A_{n-24}) \text{ MOD } M$$

The intermediate result is in the range of (-M,M), which eliminates the problem of overflow. The above equation can be implemented as follows:

```
REPEAT
        An = An-55 - An-24
        IF An < 0 THEN An = An + M
UNTIL An ≠ 0
```

In many applications using random numbers, the value of 0 may cause some problems, such as violating the assumptions. In general, it is a good practice to avoid generating 0 as a random number, as indicated in the algorithm.

Unlike the linear congruential method, very little has been proven about the randomness properties of this method. However, many empirical studies indicate that it is reliable. Note that this method is very efficient; it only requires two integer additions and 55 words of storage. However, the first 55 numbers must be established before a random number is returned. Knuth suggests an algorithm much like the Fibonacci sequence to initialize the RNG. The suggested initialization algorithm is shown in section 4.5. The period of the RNG using this method is longer than 2^{55}, and $M = 10^9$ is sufficiently large for most applications. Note that $2^{29} < 10^9 < 2^{30}$.

4.3.3 Parallel Binary Multiplication

When the hardware required to do the direct multiplication is not available, the multiplication can still be carried out using a series of additions. One way to simulate this is to add the multiplicand the number of times that is equal to the multiplier. However, the execution time varies linearly with the multiplier. Fortunately, this method can be modified for binary numbers so that the execution time is constant. First, an example of an unsigned binary multiplication of 7 by 5 is as follows:

```
     111
     101
  ------
     111          partial sum = 000111
    000           partial sum = 000111
   111            partial sum = 100011
  ------
  100011
```

Notice that the partial sum is generated least significant bit first, and that the summands are shifted left one bit to keep their positions. Also note that there will be at most three additions, corresponding to the number of bits in the multiplier. Using these facts, the binary multiplication can be modified, whose algorithm is described below in a C-styled register-transfer-level language:

```
pr[5 to 0]                 6-bit wide product register
multiplier[2 to 0], multiplicand[2 to 0]     3-bit wide multiplicand, 3-bit wide multiplier
carry[0]                   1-bit carry out from a 3-bit wide adder
pr[5 to 3] = 0;            clear partial sum (partial sum is to be kept in pr)
pr[2 to 0] = multiplier;       a single shift right will shift both partial sum and multiplier
for (i = 3; i > 0; i--)    do this loop three times
{ if (pr[0] == 1)          if least significant bit of multiplier is 1, add multiplicand
  carry:pr[5 to 3] = pr[5 to 3] + multiplicand;
  pr[5 to 0] = carry:pr[5 to 1];         shift right both partial sum and multiplier one bit
}                          final sum is in product register
```

4.3.4 Parallel Binary Division

Similarly, binary division can be carried out using a series of subtractions and shifts. A binary division of a 6-bit dividend by a 3-bit divisor could leave up to a 6-bit quotient and a 3-bit remainder. However, we will say that it will leave a 3-bit quotient, and that any quotient larger than 3-bits will be an overflow. The following algorithm describes the parallel binary division, shown again in a C-styled register transfer level language:

```
dr[5 to 0]                 6-bit wide dividend register
divisor[2 to 0]            3-bit wide divisor register
carry[0]                   1-bit carry register of an adder/subtractor
if (dr[5 to 3] > divisor)  check for overflow of quotient
  overflow;
else for (i = 3; i > 0; i--)   do this loop three times
 { carry:dr[5 to 1] = dr;  shift left dividend register one bit
   if ( carry:dr[5 to 3] > divisor)  if divisor can be subtracted from dividend, do so
    {dr[5 to 3] = carry:dr[5 to 3] - divisor;
     dr[0] = 1;            set the quotient bit
     }
   else dr[0] = 0;         clear the quotient bit
 }                         quotient is in dr[2 to 0] and remainder is in dr[5 to 3]
```

4.4 Procedure

In this experiment, we give a standard part and an optional part, but not an extra credit part. The optional part is not dificult. We encourage you to try it.

4.4.1 Standard Part

Write a program that will read in the value of the seed, in decimal, and display, also in decimal, one newly generated random number after each carriage return, or quit after receiving the character *Q* instead of a carriage return.

The random number generator routine should be called as a subroutine, using the parameter passing after the call technique to pass the constants B and C. Any registers may be used for other parameters. The calling sequence should include the following lines of instructions:

```
BSR     RANDOM
FDB     . . .
FDB     . . .
```

Notice that, in this way, the values of the constants can be modified to fine tune the generator after the program is assembled. The value of the previously generated random number may be passed in, and out, through a register. Choose the value of M to be 65536, so that the MOD operation can be simple. However, assume that the values of B and C are 16-bit unsigned numbers, so a multiplication routine that multiplies two 16-bit unsigned numbers and returns 32-bit result should be used.

4.4.2 Optional Part

Modify the random number generator so that it returns a number in the range of [0 to M-1], where M is not a power of 2. The program should input the decimal values of the seed before displaying the values of the generated random numbers, one after each carriage return it receives. Assume M to be a number less than 65536, so that a division would have to be used instead of a truncation. The constant M should be passed in after the call, just like the other constants B and C. Under certain conditions, the overflow does not occur in the parallel division algorithm described above. Explain these conditions, and determine whether or not an overflow would occur in your program.

4.5 Hints and Suggestions

4.5.1 Multiple-Precision Arithmetic Routines

An easy way to implement the multiply routine is to pass the numbers and the result through the registers. For instance, the registers X and D can hold the input arguments, and on return, they can be concatenated to hold the 32-bit result. The multiply routine would need the following resources: a loop counter to count 16, a 32-bit product register, and an adder. The loop counter would have to be allocated on stack. The product can be formed by concatenating the D accumulator with a 16-bit variable allocated on the stack.

The divide subroutine should be able to handle the 32-bit dividend and 16-bit divisor and return a 16-bit quotient and a 16-bit remainder. Try passing the dividend through the concatenated register X:D, the divisor through Y, the quotient through X, and the remainder through D. Similar resources are needed: a loop counter, a 32-bit dividend register, and a subtractor. The dividend register can be formed by concatenating the D accumulator with a 16-bit variable allocated on stack. Notice that this routine performs both the MOD and DIV operations.

4.5.2 Subtractive RNG

The subtractive RNG and the initialization routines are shown below in C.

```
long int seed = 31415987;      /* import double-precision seed */
long table[55];          /* table of previous 55 random numbers */

int init_random() /* initialization routine */
{   long int temp1, temp2;  int i,j;
    table[54] = seed;   temp1 = seed;   temp2 = 1;        /* initialize variables */
    for (i=0; i<54; i++)        /* establish the first 55 random numbers */
    {       j = 21 * i % 55;  /* spread the indices of the table */
            table[j] = temp2;
            temp2 = temp1 - temp2;   /* generate the new random number */
            if (temp2 < 0) temp2 += 1000000000;        /* if negative, add M  */
            temp1 = table[j];
    }
    for (i=0; i<220; i++) random();      /* warm up RNG with 220 random nums */
}

long int random() /* subtractive RNG */
{   static int index = 0;          /* pointer to the next random number */
    long int temp;  int i;
    do
            if ( index > 54 )  /* if all random numbers in table are used */
            {       for ( i = 0; i ≤ 23; i++ )   /* generate next 24 random nums */
                    {       temp = table[i] - table[i+31]; /* use subtract. meth. */
                            table[i] = ( temp > 0 ) ? temp : temp + 1000000000;
                    }
                    for ( ; i ≤ 54; i++)          /* generate next 31 random nums */
                    {       temp = table[i] - table[i-24]; /* use subtract. meth. */
                            table[i] = ( temp > 0 ) ? temp : temp + 1000000000;
                    }
                    index = 0;          /* reset the pointer to next random number */
            }
    while ( table[index++] == 0 );       /* until the random number is not zero */
    return ( table[index-1] );
}
```

5 Internal Sorting

5.1 Goals

1. To analyze the trade-off between the efficiency and implementation difficulty of two common sorting methods
2. To a use top-down, structured approach to writing software
3. To be proficient in debugging software

5.2 Introduction

If the size of an array is small enough so that all the elements can be fit into memory, the method used to sort the array is "internal," as opposed to "external." If the amount of data to be sorted is so large that all the data cannot be stored in memory, an external sorting technique must be used. An external sorting technique requires the use of a external temporary storage buffer to store the partially sorted data while the remaining data is being sorted in memory. External sorting techniques require a merge phase. Associated with the internal sorting methods are two important performance parameters: running time and the memory usage. Most internal sorting methods require time proportional to N^2 or N logN to sort N elements. Some methods sorts elements in place, using no extra memory other than a small number of variables. Others use a linked list to represent the elements in the form of a binary tree, using N extra words for the pointers. Still others require an extra copy of the array to do the sorting. In this experiment, two internal sorting methods are studied: bubble sort and quicksort. Incidentally, it is proven that no algorithm can sort an array of N elements in less than $O(\log_2 N)$ time.

For further information, consult chapters 8 and 9 of *Algorithms* by Robert Sedgewick, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1984, or chapter 8 of *Data Structures and Algorithms,* by A. Aho, J. Hopcroft, and J. Ullman, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1983.

5.3 Description

5.3.1 Bubble Sort

The strategy in bubble sort is to compare two adjacent elements, swapping them if necessary, until all the elements are sorted. Lower values rise like bubbles. The sorting is finished when there is not a need for a swap in a scan through all the elements in the array. This method requires the running time proportional to N^2, but the exchange is done in place and requires no additional memory. The algorithm is as follows in C:

```
extern element_type a[];        /* array of element_type */
bubblesort()                              /* sort in ascending order using bubble sort */
{   int j;  element_type temp; /* index and a temporary variable */
    do                                        /* repeat until done */
    {       temp = a[0];              /* sort from top to bottom */
            for ( j = 0; j < N - 1; j++ )        /* for all N elements */
            if ( a[j] > a[j+1] ) /* if element in top is greater than one in bottom */
            { temp = a[j];  a[j] = a[j+1];  a[j+1] = temp; } /* then swap */
    } while ( temp = a[0] );    /* done when A[0] remains intact */
}
```

5.3.2 Quicksort

The basic algorithm for quicksort was first introduced by C. A. R. Hoare in 1960. It is a general-purpose, in-place sorting algorithm that requires time proportional to N logN. However, the drawback of the algorithm is that it is recursive in nature, which makes it difficult to implement in assembly language. The sorting strategy is to partition the array into two parts and sort each part independently. The subdivided parts are sorted again by partitioning each into two parts, which are then sorted independently. This is recursion. The recursive algorithm is as follows in C:

```
quicksort(l,  r)
int l, r;                                 /* left and right range pointers */
{   int i;                                /* partition pointer */
    if ( r > l )                          /* repeat while right pointer > left pointer */
    {       i = partition(l, r); /* partition the unsorted array */
            quicksort(l, i-1);  /* sort left partition */
            quicksort(i+1, r); /* sort right partition */
    }               /* if left and right pointers cross each other, done */
}
```

The elements are exchanged in the partitioning function so that

1. all elements left of i, (i.e., from a[l] to a[i-1]) are smaller than or equal to a[i];
2. all elements right of i, (i.e., from a[i+1] to a[r]) are larger than or equal to a[i]; and
3. a[i] is in its final sorted place.

The partitioning algorithm is also shown below in C:

```
partition(l,r)
int l, r;                         /* pointers to left and right ends of the array */
{   int v, t, i, j;
    v = a[r];  i = l - 1;  j = r;   /* v is pivot element */
    do
    { do i++ while ( a[i] < v );/* scan from left until an element > pivot is found */
```

```
        do j--   while ( a[j] > v ); /* scan from right until an element < pivot is found */
        t = a[i];  a[i] = a[j];  a[j] = t;         /* swap */
      } while (i < j);     /* repeat until end pointers cross each other */
      a[j] = a[i];  a[i] = a[r];  a[r] = t;        /* undo the last step */
      return(i);                       /* return the position of the partition */
  }
```

When the **do . . . while** loop is terminated, the left scan pointer, i, and the right scan pointer, j, are already crossed, and the elements are wrongly exchanged. So the last three assignment statements reexchange a[i] and a[j] and put the partitioning element a[r] in its final place, at i.

5.4 Procedure

5.4.1 Standard Part

Write a subroutine that will sort N elements of one-byte, 2's complement numbers, using the bubble sort algorithm. The subroutine is to be called with the following calling sequence:

```
LDX     #ARRAY   point reg. X to the first element of the array ARRAY.
LDAB    #N       number of elements in the array ARRAY.  N < 256.
BSR     BSORT    bubble sort
```

5.4.2 Optional Part

Write a recursive quicksort subroutine to sort the array ARRAY. To handle the recursion, an EVB may have to be used for this part.

5.5 Hints and Suggestions

The following is suggested in writing the recursive quicksort subroutine:

1. Pass the variables l and r through the accumulators and the pointer to the array in register X to the quicksort and partition subroutines. Return the variable i through an accumulator from the partition subroutine.
2. The first instruction in the recursive subroutine should be the test of the exit condition.
3. Save the variables l, r, and i on stack before calling the partitioning subroutine or the recursive quicksort subroutine itself.

4. Allocate the space for all the variables on stack at the beginning of the partitioning subroutine, and use the index addressing to use them. Be sure to balance the stack before returning from the subroutine.

Be sure to allocate a large stack area for the recursive calls. Pay particular attention to the following points while debugging:

1. Use a signed conditional branch after comparing array elements; and unsigned after comparing array indices.
2. The partitioning subroutine should be debugged before the recursive quicksort subroutine.
3. Before attempting to debug the partitioning subroutine, you should have a full understanding of how this algorithm works.
4. You need to put at least three breakpoints on the partitioning subroutine: one after each **do ... while** loop. Since actual sorting is done in this routine, make sure that the elements are in the correct place before exiting the subroutine. Also make sure that variable i is correct.
5. The quicksort subroutine should be "bug free" if the partitioning and recursive call to itself is made with correct values of l, r, and i.

6 Linked Lists

6.1 Goals

1. To understand how an abstract data structure, a linked list, is actually implemented in memory
2. To implement operations to work on the linked lists

6.2 Introduction

Linked lists have many uses in abstract data structures, especially in situations where a large amount of data movement is expected. In these situations, instead of moving the data itself, the pointers to the data are moved to substantially reduce the amount of data movement involved. Picture a text editor that uses a very large character array to keep the text in order. And then picture the work needed to insert one character in the beginning of the text; the entire block of the text would have to be moved down by one character in the array to keep it in order. If linked lists were used, the newly inserted character would be placed in an unordered character array, and a pointer to that character would be inserted in a place to keep the list of the pointers to the characters (actually, to blocks of characters) ordered.

Linked lists are also used in situations where dynamic memory allocation is required. This happens when the exact amount of the memory space required is not known before run time. Of course, more than enough memory space can always be allocated before run time, but this is considered inefficient use of resources and is not always desirable.

Linked lists do have some disadvantages. Each pointer used is an overhead, and in some situations, the amount of overhead may be too great to warrant the use of linked lists. Use of linked lists also leaves fragmented memory space. Refer back to the example of a text editor using linked lists. If a word is deleted, the space occupied by the word in the unordered character buffer is now available. But the space may be too small to fit most of the newly entered text. If there are many of these fragmented spaces, the program may "run out of" memory, although there is available memory scattered throughout the buffer. If this happens, an operation that collects all the available memory into one big chunk, as well as adjusting the pointers so that they point to the correct places, is required. This is sometimes called compaction, and it is pure overhead.

6.3 Description

We will implement a program that manages a priority queue in a linked list. It is to be able to execute the following commands on the queue:

I data priority
This command should prompt the program to insert data according to its priority in the queue. If this operation cannot be performed because there is not enough available memory, the program should indicate so.

D data
This command should prompt the program to delete data from the queue. If there is more than one data item, the one with the higher priority should be deleted. If there is no data, the program should indicate this.

P
This command should prompt the program to print the elements of the queue in the decreasing order of their priority. The elements and their addresses should be printed.

C
This command should prompt the program to compact the buffer and display the elements and their addresses.

The priority queue is to be implemented with a linked list and is to be kept sorted at all times by decreasing order of priority. A node in the queue is to be of the following structure:

```
struct node {                /* structure of node type consists of ... */
  int priority;              /* priority is of integer type */
  datatype element;          /* element is of user-defined datatype */
  struct node *next;         /* next is a pointer to node type */
}
```

6.4 Procedure

Implement the program described above. The priority is to be a number between 00 and 99, the element is to be a character, and "next" is, obviously, a two-byte pointer to the next node. A block of 40 bytes in ram is to be set aside for the buffer, from which four consecutive free locations are assigned to a node when an element is inserted in the queue. Use the "first-fit" method to search for the available free space for a node; that is the first available free space large enough for a node should be used. In this case, this is trivial since the size of a node is always the same. When a compaction is called for, the free spaces may be collected at either end.

Only one global variable should be used to point to the first node in the list. Any number may be used to represent the NULL pointer, but it should not conflict with the addresses in the buffer. You may use 0 to represent NULL if the buffer starts at a higher address. The element is guaranteed to be something other than 0, so that 0 can be used to represent "free" space in the buffer.

Try to use a uniform method to return error conditions from the subroutines. A

common method is to set the carry bit to indicate an error condition, and have accumulator A or B indicate the cause upon return from the subroutines. In this way, the error conditions can be checked by checking the carry bit upon returning from a subroutine. Assume that the input is error free.

6.5 Hints and Suggestions

The program will need, among others, the following routines:

1. The main routine to input a command, identify it, and call the proper subroutine to carry out the desired function.
2. A routine to input the priority and convert it to a binary value.
3. A routine to insert an element in the priority queue. This requires a search for a free block and a search for the insertion point in the list.
4. A routine to delete an element. This also requires a search for the element to be deleted.
5. A routine to print the elements in the priority queue.
6. A routine to compact the buffer. This involves a search for free blocks embedded in the buffer occupied by the queue and moving them out of the buffer.

As with any database management system, the function used most often is the search operation. So it is imperative that the search routine be as efficient as possible. However, it is also desirable to have the search routine be somewhat general so that it can be used in many functions. For instance, it is possible to write a search routine that can search for either an element or a priority.

Instead of having separate lists for "free" and "occupied" blocks, the element field in the blocks can be set to null (anything that can never be an element) to indicate that they are "free." Then the search for a free block is simple; just look for a block with its element field set to null. The delete operation is also simple; just search for the element and change it to null.

To compact the fragmented memory space in the buffer, all the occupied blocks should be moved to the top end, and all the free blocks to the bottom end of the buffer. With fixed-size blocks, this process is simple; swap the free blocks with the occupied blocks so that they are collected at opposite ends. The free block closest to the top end should be swapped with the occupied block closest to the bottom end. This process is repeated until all the occupied blocks are above the free blocks. Since the occupied blocks are part of a linked list, the pointers must also be changed so that the list is not modified in any way.

7 Huffmann Code

7.1 Goals

1. To introduce the Huffmann code, which can be used in data compression and cryptography,
2. To understand how "right" data structures make programming easier.

7.2 Introduction

Data compression plays an important part in digital communication, especially today, when the need for, and new applications of, electronic data transfer are increasing at an astounding rate. The idea of data compression is not new to the world of computers or digital communications; Morse code, abbreviations, and shorthand writing have been in use since the turn of the century. Then what is data compression?

Data compression is the reduction of the amount of information in a spatial or temporal medium, mainly for the purposes of transmitting and storing. With a spatial medium, the total volume of data can be reduced using source coding techniques such as Huffmann and run-length coding that use very efficient representation of the data. For instance, in the ASCII character set, each character is represented by a 7-bit code. However, more frequently used characters can be represented with a shorter code, such as in Huffmann coding.

Data compression techniques may be classified into two general categories: entropy reduction and redundancy reduction. Entropy is defined as the information content, or average information in some literatures. With an entropy reduction operation, the information content of data is reduced, and the loss is irreversible. An example of an entropy reduction is seen in the analog-to-digital conversion operation. With a limited resolution, the original analog signal cannot be accurately recovered from the converted digital data. With a redundancy-reduction technique, the redundant information, rather than the information content, is reduced or eliminated. Many forms of data often contain redundancy. Consider a serial transmission of a color image. If the image is composed of a few large regions of the same color, the transmission will contain relatively few transitions between the pixel colors. In this situation, instead of sending the color codes of each pixel, only the color code of the pixel in a transition and the count of the pixels with the same color can be sent. No information is lost, but the amount of data transmitted may be reduced.

One redundancy reduction technique is Huffmann coding. In 1952, D. A. Huffmann developed an encoding procedure that produces the shortest average word length based on probabilities alone. This code makes use of a coding tree, without which decoding is impractical. This feature makes it applicable in situations where both data compression and security are desirable, but not imperative.

Review section 2-1.3 of *Single- and Multiple-Chip Microcomputer Interfacing.*

For further information on Huffmann coding techniques, consult section 3.5 of *Data Compression Techniques and Applications,* by Thomas Lynch, Lifetime Learning Publications, Belmont, California, 1985.

7.3 Description

7.3.1 The Huffmann Coding Technique

The basic idea of the Huffmann coding technique of character strings is to assign the shorter codes to more frequent letters to reduce the average word length. For instance, if we wanted to encode the string THE UNIVERSITY OF TEXAS AT AUSTIN, we would assign the shortest code to the letter T, the next shortest to E, I, S, A, N, and so on. We will ignore the space between words for the moment. If we used the binary numbers for the codes, we would assign 0 to the letter T, 10 to E, 110 to I, 1110 to S, 11110 to A, 111110 to N, and so forth. Note that since the letters E, I, S, and A all occur three times in the string, different code assignments to them would not make any difference.

In real applications, we do not have the prior knowledge of the source. If we try to encode a Huffmann code for strings other than THE UNIVERSITY OF TEXAS AT AUSTIN with the same code assignments, we would probably not obtain the shortest average word length possible. This is true for all schemes of data compression, not just for Huffmann code. However, if the probabilities of occurrence of the letters in the source are known, Huffmann code will give the minimum average word length. The probability distribution is not too difficult to find out; in English language, the letters E, T, and R occur more frequently than others, and vowels occur more frequently than consonants.

7.3.2 The Modified Huffmann Coding Technique

One problem with the Huffmann coding technique is that the code length gets quite long for letters with low probability distribution. For example, if the letters of the alphabet were to be encoded in Huffmann code, some letters would require more than 20 bits. The modified Huffmann coding technique does away with this problem by grouping all the low-probability letters in one category and using a special code (for the group) and unique codes (for each letter in the group) to represent them. For instance, the code 111111111 could signify that the following 7 bits are not Huffmann code and that it is to be taken as an ASCII character. In addition, the modified Huffmann coding technique can encode a larger character set.

7.3.3 Decodability

One property of Huffmann code is instantaneous decodability; that is, the original code is known as soon as the decoder looks at the encoded string. There is no calculation

involved in decoding, only the state transitions. Another property is that without the coding tree, it is impractical to decode. In the worst case, it would take N! tries to guess the coding tree for the data with N different elements.

7.4 Procedure

7.4.1 Standard Part

Write an encoder and decoder program using straight Huffmann coding technique. The source is to be the set of letters in the alphabet (only the capital letters), and the probability distribution is to be that of the lettering sequence in the alphabet; A has the highest probability, next B, C, and so on. The program should input the ASCII string from the keyboard, encode it into Huffmann code, and then decode and display the decoded text. The string is terminated by a carriage return.

The source text string should not be stored, only the encoded string is to be stored. Assume that the length of the encoded bit string will not be longer than 30 bytes, and that the input is error free.

7.4.2 Optional Part

Write the same program described in the standard part, using modified Huffmann coding technique. The letters with the highest eight probabilities are to be encoded in Huffmann code, and others are to be encoded in ASCII. Use the probability distribution found in the string THE UNIVERSITY OF TEXAS AT AUSTIN. However, the program should have no knowledge of the probability distributions. In fact, the program should be "tunable" with different probability distribution. This time, designate $04 as the end of transmission (EOT) character so that all characters in the ASCII character set can be encoded. The input stream is terminated with the EOT (type CTRL-D) character.

7.5 Hints and Suggestions

You would need a counter to count to eight (bits/byte), as well as an accumulator to store the encoded bit strings. Instead of using two, one accumulator can be used to count to eight, as well as to store the encoded bits. The strategy is to initialize an accumulator with the value 1. To count to eight, just shift this accumulator left eight, at which time the carry bit will be set. The carry bit will be clear at other times. The bits to the right of this 1 in the accumulator are free to store the encoded bit string.

To make the encoder/decoder tunable, the probability-specific information must be in a look-up table. Since the number of bits that are 1 in Huffmann codes increases linearly with decreases in probabilities, this information can be implicitly embedded in the look-up table. Each entry in the look-up table would contain only one character,

which is to be encoded. If the entries are ordered in decreasing probability, their positions would indicate the number of 1s that are required to encode this character. For instance, the character in the first entry should be coded as 0, the second as 10, and so on.

In order to indicate the end of an encoded string, a special code must be used. Since the end of the string occurs only once in the data, the longest code should be used for this purpose.

Godfried Toussaint and Rajjan Shinghal reported in the paper "Cluster Analysis of English Text", (IEEE, 1978) that the eight most frequently used characters in English texts are blank (17.3%), E (10.3%), T (7.7%), A (6.5%), I (6.3%), O (6.3%), N (5.9%), and S (5.5%).

8 A 6811 CPU Emulator

8.1 Goals

1. To gain a better understanding of how CPUs in microprocessors operate
2. To develop a simple 6811 CPU emulator

8.2 Introduction

One of the more prevalent uses of computers today is in simulation. Computers are used to simulate all kinds of problems and solutions, from something as simple as the motion of a bouncing ball to something as complicated as global weather forecasting or the human thought process. The reason is simple; simulation is generally nearly as good as, and sometimes even better than, the "real thing" that is being modeled. In many cases, simulation yields a better understanding of, or reveals in greater depth, the nature and characteristics of a problem. In some cases, simulation provides a cost-effective substitute. Other times, there is no other solution but simulation.

The events in the world can be broadly divided into two categories: continuous and discrete. Continuous events are those that occur, or change, continuously. These are events like fluid motions in streams, chemical reactions that takes place in a fraction of a second, or temperature gradients in a frying pan that is being heated. On the other hand, discrete events are those that change states in discrete steps of time. These are events like the formation of an automobile in an assembly line, the propagation of a carry in a serial adder, or the flow of traffic in rush hour. Actually, the discrete events are a subset of the continuous events in the sense that when discrete events occur or change states fast enough, they appear to be continuous.

In discrete-event simulation, the simulation process is driven by the changes in events, such as finite change of time or input variables. The accuracy of the simulation depends heavily on the granularity of the allowed changes. For instance, it would be more accurate to simulate the assembly process of an automobile by the addition of one part rather than ten parts. Some events occur so frequently and change so much that it is impossible, or at least impractical, to model them as discrete events. With these events, continuous event simulation is used. Here, the simulation process is driven by a desire to attain an equilibrium state, and thus the process is repeated until either an acceptable state is reached or time runs out. Of course, even the continuous-event simulations in a computer must be carried out discreetly. After all, a computer is a finite machine.

8.3 Description

8.3.1 Why Simulate a Computer?

Simulating a computer itself is inherently easy. A computer is a finite state machine. It is also predictable; given the same instructions, it will do the same thing every time.

Now one might ask, why is it necessary to simulate a computer? Well, suppose that you have a copy of a program in machine-code form, and you absolutely need to run the program but do not have the target machine. Either you would have to obtain the target machine, or you could write a program that simulates the target machine. This program would take the machine codes, perform the functions that an actual target machine would with these codes, and produce the same result. These kind of programs are referred to as *emulators*.

Interpreters are another kind of programs that simulate either a real or an abstract machine. Many of today's compilers produce instead of machine codes, some intermediate codes. These codes are similar to machine codes but are intended to be run on an abstract machine. They can be processed further to produce machine codes, or can be run on an abstract machine by an interpreter. This way, compilers for different languages can produce intermediate codes that can be linked together.

8.3.2 How Do You Simulate a Computer?

The most appropriate model of a computer for simulation is the register-transfer-level (RTL) description. At this level, the data movements, both the opcode and operands, are described in terms of transfers between various registers. It would not be sensible to model a computer at the gate level, nor is it possible to model it at a higher level, for the purpose of writing an emulator. In the next few paragraphs, we will develop a simulation model for the 6811 in a RTL description.

First, we need to model the machine state of the 6811. There is a program counter (PC), accumulators A and B, index register X, and a conditional code register (CC). We will ignore other registers in this experiment. In addition, it has three hidden registers: an instruction register (IR), a destination register (DEST), and a source register (SOURCE). These are hidden from programmers because these cannot be accessed directly. In CC, there are three bits: carry (C), zero (Z), and negative (N). We will also ignore other conditional code bits.

Second, we need to model the processes involved in executing instructions. The Central Processing Unit (CPU) continuously repeats the cycle of fetching, decoding, and executing an instruction, unless the instruction is halt. The fetch process is as follows:

$$\begin{aligned} IR &\leftarrow M[PC] \\ PC &\leftarrow PC + 1 \end{aligned}$$

The first line indicates that the opcode is fetched from the memory location whose address value is contained in the program counter. The program counter is then incremented.

The next phase is more complex. In the decoding phase, the fetched opcode is decoded to obtain the following information: the class of operation, the source(s) of operand(s), and the destination of the result of the operation. Like many other 8-bit microprocessors, the 6811 uses one and a half operand instructions, meaning that one operand is from a register and the other is from memory. Thus the general format of the operations we will emulate is:

$$\text{DEST} \leftarrow \{\ \text{DEST op}\ \}\ \text{SOURCE}$$

where { DEST op } indicates a null operation, such as in the move class of operations. In other classes, the "op" would indicate the operation needed. Branch instructions can be viewed as a special case of the arithmetic class of instructions involving the program counter. There are many special instructions using the inherent addressing mode, such as increment and push, but these are just special cases of the format shown above, provided for an efficient execution of frequently used operations.

There are six types of addressing modes in an 6811: immediate, direct, extended, indexed, inherent, and relative. All instructions have at least one of these addressing modes. Inherent and relative modes are mutually exclusive with the other modes and with themselves. Motorola uses the concept of effective address (EA) to refer to the address of the operand or the destination in memory. For instance, in load instructions, EA becomes the source address, whereas in store instructions, EA becomes the destination address. The EA calculation is listed below for various addressing modes:

immediate	EA	←	PC
direct	EA	←	0:M[PC]
extended	EA	←	M[PC]:M[PC+1]
indexed	EA	←	X reg. + M[PC]
inherent	no EA calculation		
relative	EA	←	PC

A close study of the opcode map reveals that the instructions are grouped together in some fashion. This is not surprising, for the design of the instruction set is quite systematic. Each bit in opcodes carries a special meaning. For instance, a group of 3 bits can be used to encode the six different addressing modes shown above. Naturally, decoding should be systematic. As we make few observations on the opcode design of the 6811, we ask you to verify these and convince yourselves.

1. Bits 7, 5, and 4 indicate addressing modes.
2. Bit 2 differentiates between the arithmetic and move classes.
3. For the move class, bit 0 indicates the use of EA as the source or destination.
4. For the move class, bit 3 differentiates between 8- and 16-bit registers, bit 6 differentiates between the A and B accumulators, bit 1 differentiates between the D accumulator and the X register.
5. There are some opcodes whose encoding patterns do not fit as well as others; these are probably some special or exceptional cases.

Now that the instruction is decoded, the proper operations must be carried out. These include not only the proper movement of operands and result but also the correct setting of the condition code bits as well. Zero and negative conditions are easy to test, but the carry (or borrow) condition is not. In unsigned addition, a carry must be generated if DEST' < DEST, that is, if two unsigned numbers are added and the result is less than either of the two. In unsigned subtraction, a borrow must be generated if DEST' > DEST. That is, if an unsigned number A is subtracted from an unsigned

number B and the result is greater than the original number B. Special care must be paid to proper setting and clearing of conditional code bits because some instructions affect certain conditions whereas others do not.

8.4 Procedure

8.4.1 A Simplified 6811 Model

Write a 6811 emulator that recognizes the machine codes for the following instructions:

LDAA/LDAB	all modes, but index on Y
STAA/STAB	same
LDX	same
STX	same
ADDA/ADDB	same
SUBA/SUBB	same
DECA/DECB	
INX	
BNE/BCS/BMI/BRA	
SWI	

Our model of the 6811 does not have the Y or the S registers. Consequently, the SWI instruction does not save the registers. It simply halts the CPU. You may assume that the input is error free. The program should not simply transfer the condition codes set by your program to the CC register of the emulator machine state. In fact, the C, Z, and N bits are defined as follows in the CC register:

CBIT	EQU	$80	bit 7
ZBIT	EQU	$40	bit 6
NBIT	EQU	$20	bit 5

8.4.2 Emulator Validation

The emulator must be tested to verify that it behaves exactly the same as a real 6811 CPU for each instruction. This process is known as validation. The emulator must be validated before it can be used to execute a 6811 program. The list below shows the necessary test conditions for validation. However, the tests may not be sufficient, depending on the implementation of the emulator, for validation. # denotes immediate, > denotes extended, < denotes the direct addressing mode, M[3] denotes the content of the memory location 3, (X) denotes the content of the X register, and : denotes concatenation.

	INST	A	B	X	C	Z	N	PC	COMMENT
1.	LDAA #1	1	-	-	-	0	0	+2	
2.	LDAA #0	0	-	-	-	1	0	+2	
3.	LDAA #4FF	$FF	-	-	-	0	1	+2	
4	LDAA >1	M[1]	-	-	-	?	?	+3	
5.	LDAA <1	M[1]	-	-	-	?	?	+2	
6.	LDAA 1,X	M[(X)+1]	-	-	-	?	?	+2	
7 to 12.			Repeat 1 to 6 with B accumulator.						
13.	LDX #1	-	-	1	-	0	0	+3	
14.	LDX #0	-	-	0	-	1	0	+3	
15.	LDX #$FFFF	-	-	$FFFF	-	0	1	+3	
16.	LDX >1	-	-	M[1]	-	?	?	+3	
17.	LDX <1	-	-	M[1]	-	?	?	+2	
18.	LDX 1,X	-	-	M[(X)+1]	-	?	?	+2	
19.	STAA >1	-	-	-	-	?	?	+3	M[1] = (A)
20.	STAA <1	-	-	-	-	?	?	+2	M[1] = (A)
21.	STAA 1,X	-	-	-	-	?	?	+2	M[(X)+1] = (A)
22 - 24.			Repeat 19 to 21 with B accumulator.						
25.	STX >1	-	-	-	-	?	?	+3	M[1] = (X)
26.	STX <1	-	-	-	-	?	?	+2	M[1] = (X)
27.	STX 1,X	-	-	-	-	?	?	+2	M[(X)+1:(X)+2] = (X)
Assume that (A) = 1, (B) = 1, (X) = 1 from here on.									
28.	ADDA #0	1	-	-	0	0	0	+2	
29.	ADDA #$FF	0	-	-	1	1	0	+2	
30	ADDA #$7F	$80	-	-	0	0	1	+2	
31.	ADDA >1	M[1]+1	-	-	?	?	?	+3	
32.	ADDA <1	M[1]+1	-	-	?	?	?	+2	
33.	ADDA 1,X	M[(X)+1]+1		-	?	?	?	+2	
34.	SUBA #0	1	-	-	0	0	0	+2	
35.	SUBA #1	0	-	-	0	1	0	+2	
36.	SUBA #$FF	2	-	-	1	0	0	+2	a borrow is generated
37.	SUBA #$81	$80	-	-	1	0	1	+2	a borrow is generated
38.	DECA	0	-	-	-	1	0	+1	
39.	DECA	$FF	-	-	-	0	1	+1	a borrow is generated
40 to 51.			Repeat 28 to 39 with B accumulator.						
52.	INX	-	-	(X)+1	-	?	-	+1	
53.	BNE r	-	-	-	-	-	-	+r	branch taken (Z = 0)
54.	BNE r	-	-	-	-	-	-	+2	branch not taken (Z=1)
55.	BCS r	-	-	-	-	-	-	+r	branch taken (C=1)
56.	BCS r	-	-	-	-	-	-	+2	branch not taken (C=0)
57.	BMI r	-	-	-	-	-	-	+r	branch taken (N = 1)
58.	BMI r	-	-	-	-	-	-	+2	branch not taken (N=0)
59.	BRA r	-	-	-	-	-	-	+r	branch always taken
60.	SWI	-	-	-	-	-	-	+1	stop, no stacking

8.5 Hints and Suggestions

Use the assembler in Buffalo to assemble the test cases. Be sure that the program counter is initialized correctly.

A suggested structure for the emulator is shown below. The notations are that of C language with some changes. The bit position of a variable may be indicated with a bracket (i.e. c[0] as bit 0 of the variable c). The function parameters are call-by-reference and not call-by-value.

```
int     pc;                 /* 16-bit program counter */
short   ar;                 /* 8-bit A accumulator */
short   br;                 /* 8-bit B accumulator */
int     xr;                 /* 16-bit X register */
short   cc;                 /* 8-bit conditional code register */
short   ir;                 /* 8-bit instruction register */
int     source;             /* source address, may be memory or register */
int     dest;               /* destination address, may be memory or register */
short   memory[memsize];    /* memory */

do                          /* repeat */
{   ir = fetch(pc);         /* fetch the opcode */
    if (ir[7] == 1)         /* if bit 7 of opcode is 1 */
    {   source = adrsmode(ir, pc); /* determine the addr. mode and source reg. */
        if (ir[2] == 0)     /* if bit 2 of opcode is 0, then arithmetic class */
        {   dest = getreg8(ir); /* determine the 8-bit destination register */
            arithmetic(source, dest, cc); /* execute a arith. operation */
        } else              /* handle move class */
        {   dest = getreg16(ir);    /* determine 8 or 16-bit destination register */
            if (ir[0] != 0) /* if STORE operation, */
                swap(source, dest);     /* then swap source and dest. register */
            move(source, dest, cc);     /* execute a move class operation */
        }
    }
                            /* now handle inherent and relative addressing modes */
    else switch (ir) {
        case 0x26 : bneop(pc, cc);      /* if opcode = $26, BNE opcode */
        case 0x2B : bmiop(pc, cc);      /* if opcode = $2B, BMI opcode */
        case 0x25 : bcsop(pc, cc);      /* if opcode = $25, BCS opcode */
        case 0x20 : braop(pc);    /* if opcode = $20, BRA opcode */
        case 0x5A : decaop(ar,cc);/* if opcode = $5A, DECA opcode */
        case 0x4A : decbop(br,cc);      /* if opcode = $4A, DECB opcode */
        case 0x08 : dexop(xr,cc); /* if opcode = $08, DEX opcode */
        }

} while (ir != swiop);      /* until SWI instruction */
```

```
short     fetch(pc)              /* fetch opcode, increment pc */
{    return ( memory[pc++] ) }

int       adrsmode(ir,pc)   /* set source address  by determining addressing mode */
{    switch (ir[5:4]) {        /* if bits 5 and 4 of opcodes are */
     case 11 :        source = fetch(pc) << 8 + fetch(pc) /* 11 extended */
     case 10 :        source = xr + fetch(pc); /* 10 indexed*/
     case 01 :        source = fetch(pc);       /* 01 then direct mode */
     case 00 :        source = pc;  /* 00 then immediate mode */
                      if (ir[3] == 0 || ir[2] == 0) /* if bit 3 or 2 of opcode is 0, */
                           pc += 1; /* then 8-bit register is in use */
                      else pc += 2; /* else 16-bit register is in use */

int       arithmetic(source, dest, cc) /* ADDA and SUBA instructions */
{    if (ir[3:0] == 1011)     /* if bits 3, 1, and 0 of opcodes are set */
          reg[dest] += memory[source];   /* then reg. -> dest used with ADD op. */
     else
          reg[dest] -= memory[source];    /* else SUB opcode */
     update_carry(dest, cc); /* update carry bit in cc */
     update_zero(dest, cc);  /* update zero bit in cc */
     update_neg(dest, cc);   /* update negative bit in cc */
}

int       bcsop(pc, cc)        /* BCS instruction */
{    if (cc[cbit] == 1)        /* if carry is set, */
          pc += fetch(pc);    /* then take the branch */
     else pc++;                /* else fall through */
}

int       move(source, dest, cc) /* LOAD and STORE instructions */
{    [dest] = [source];        /* move data pointed by dest to source */
     if (ir[3] == 1)           /* if bit 3 of opcode is set, */
          [dest+1] = [source+1]; /* then move next byte as well */
     update_zero(dest, cc);  /* update the zero bit in cc */
     update_neg(dest, cc);   /* update the negative bit in cc */
}
```

Other routines are similar.

9 A 6811 Assembler

9.1 Goals

1. To understand how assemblers work
2. To develop a primitive, one-pass assembler with forward reference capability

9.2 Introduction

When computers were first being developed, the programs were written in machine codes, that is, the 1s and 0s. Imagine writing a program, even as small as a hundred instructions long; it would be a slow, laborious process. Programmers had to keep track of all the numbers (mostly in octal and hexadecimal) that represented all the machine operations (not to mention the address and offset calculations for branch operations), and the elaborate comments to make the program "maintainable." In short, machine-language programming is awkward.

To enable programmers to code in a way that resembled their own thought process and not that of machine operations, many "high-level" languages were developed. With high-level languages, programmers can take a problem, model it, develop algorithms, and instruct a computer to solve the problem as they would. Most high-level languages give the programmer control over the computer, not the other way around.

The lowest level of languages like high-level languages is symbolic assembly language, where machine operations are modeled in terms of data movements between the memory elements and the arithmetic-logic unit, rather than in terms of gate-level control operations as in machine language. Instead of writing in 1s and 0s, the programmer can write in mnemonics like LOAD and ADD in assembly language. However, a computer cannot execute a program written in assembly language nor in any other high-level languages. The program must first be assembled, or translated, into something that a computer can understand; the machine codes. This is the job of the assembler and compiler.

9.3 Description

9.3.1 What Is an Assembler and What Does It Do ?

An assembler is a program, like any other program that you write, that inputs a program coded in assembly language and outputs an equivalent machine code. The primary purpose of an assembler is to convert symbolic instructions and operands to machine operation codes. For instance, an 6811 assembler would convert the instruction **LDAA #$10** to 10000110 and 00010000. It also keeps track of symbolic names and

substitutes them with their numeric values wherever they appear in the program. The symbolic names can be labels or constants. More sophisticated assemblers also provide a *macro* facility whereby new symbolic instructions can be defined in terms of the symbolic instructions that the assembler understands. In addition, it may provide conditional assembly capability, in which only certain portions of the program will finally be assembled under specific conditions.

Then, what exactly are the functions that an assembler must do? Before we answer this question, let's look at a simple program written in assembly language:

```
TEN     EQU     10
        ORG     54
SUM     RMB     2
        LDAA    #TEN
        LDX     #SUM
ADD3    DECA
        BNE     ADD3
        STAB    1,X
        SWI
```

We assume that you already know what each symbolic instruction and assembler directive does, and we will only explain how an assembler handles each statement. First, you will notice that there are three fields, not counting the comment field, in a line. The fields are separated from each other by one or more spaces. The first field, called label field, is reserved for a label, and only a label should be there. The second field, called opcode field, is reserved for an assembler directive or a symbolic instruction. The third field, called operand field, is reserved for an operand, if required. Every line should have the opcode field.

When a label is encountered in the label field, it is recorded along with its value. The value of a label is different, depending on its uses. In an EQU directive, the label is taken as a constant, and the value of the constant is assigned to the value of the label. In the above example, the value of the label TEN is 10. In all other cases, the value of the "current location" is assigned to the value of the label. The current location can be thought of as the address in memory where the machine code of the instruction being assembled is to be stored. The ORG directive fixes the value of the current location, and other directives move it ahead as code or constants are generated.

When the first character of a line is a blank, the assembler assumes that there is no label declaration. The opcode is read to determine the instruction type and whether an operand is required or not. The operand field indicates the addressing mode. In addition, if a label is found in this field, its value must be substituted. If the opcode is a relative branch instruction, the offset to the label must be calculated. Anything found beyond the operand field is ignored, so comments can be put there.

9.3.2 Forward and Backward References

Consider the program segment shown below:

```
            BRA     L2   forward reference to L2
            . . .
L2          . . .
            . . .
            BRA     L2   backward reference to L2
```

When the first branch instruction is encountered, the label L2 is not yet declared. Since the assembler has no way of knowing the destination address, the operand field (one byte for the BRA opcode) must be left unfilled. The assembler makes a note that it should satisfy this forward reference when the label L2 is later declared. At that time, it should check whether the distance between the label and the reference is short enough to be represented in a one-byte offset. Note that there can be more than one unsatisfied forward references for a label. A label with a backward reference poses no such complication because the destination address is already known.

To solve the problem of forward references, most assemblers are written as "two-pass" types. A two-pass assembler establishes the location of all labels in the first pass, without generating any machine code, and fills in the instruction operands during the second pass, at which time all the values of the labels are known. Two-pass assemblers are slower than one-pass assemblers because they have to read the input twice, at I/O speed. In both types, one of the biggest problems is managing the symbol table for the labels and constants.

9.4 Procedure

9.4.1. Standard Part

Write an assembler that recognizes the following symbolic instructions and directives:

ORG	optional, and if not given, defaults to the address of 0
RMB	
LDAA	immediate, direct, and index on X register addressing modes
ADDA	immediate, direct, and index on X register addressing modes
SUBA	immediate, direct, and index on X register addressing modes
STAA	direct, and index on X register addressing modes
BNE	backward reference only
BRA	backward reference only
END	terminates assembler

You may assume that all numeric values are to be given in hexadecimal representation, and that the index addressing mode only recognizes numeric constants for the offset. Since all numeric values are represented in hexadecimal, the usual $ sign is not needed. Labels are one character long, and are allowed as operands only in branch instructions. Limit the total number of labels a program can have to six. The input is to be read from

the keyboard, and the generated machine codes are to be loaded into appropriate memory locations. The assembler need not check for any errors other than unrecognized instructions/directives and undeclared labels. With erroneous inputs, it should simply sound a bell and ignore the instruction. However, it should not quit unless the END instruction is encountered.

9.4.2 Optional Part

Add the following capabilities to the assembler described in the standard part:

1. Add LDAB, ADDB, STAB, and SUBB instructions.
2. Allow the 16-bit direct addressing mode in all instructions.
3. 8-bit page zero, instead of 16-bit direct, addressing should be used by the assembler whenever possible.

9.4.3 Extra Credit

Add the forward reference capability to the assembler. Limit the number of unsatisfied forward references to six at one time. Note that there could be more than one unsatisfied forward reference to a label. In addition, implement free-format input and more robust error detection and recovery. For instance, allow LDAA ,X as well as LDAA 0,X.

9.5 Hints and Suggestions

9.5.1 Symbol Table Management

The key mechanism of assemblers, and of translators in general, is the symbol table management. In the most efficient implementation of assemblers, some form of binary tree is used to organize the symbols so that the search time is minimal. However, since we are more concerned with the mechanics of assemblers than with efficiency, an array can be used for the symbol table. An entry in the table would need the following information: the symbol identifier and its value. An entry into the table is generated whenever a label is declared.

In addition, there should be another table for the unsatisfied forward references. An entry into this table is generated whenever a branch (or jump or jump to subroutine) is made to a label that is not already declared. It must contain the following information: symbol identifier and the address at which the appropriate machine code is to be put after the reference is satisfied. With a branch instruction, only one byte (the relative offset) is to be stored at this address. For instructions using the 16-bit direct addressing mode, a two-byte address (the destination address) is stored at this address. Note that when a label is detected, the assembler must first check for any unsatisfied references.

Each entry in the symbol table can be implemented with a three byte block; one for the one-character symbol and two for the value. Since the labels are associated only with the addresses, two-byte value is suitable. The symbol field can be set to NULL (0) to indicate that it is free.

9.5.2 Error Detection and Recovery

In all assemblers, robust error detection and recovery is desirable. Most assemblers are designed to detect as many errors as possible in a single assembly. To do so, the assembler must recover from the erroneous inputs as soon as possible. This is simple in assemblers because the end of a line signals the end of an instruction. However, to provide robust error detection, we need a mechanism to report the error at any place in the program. The following mechanism is suggested for error detection and recovery.

```
            STS    STACK  initially, save stack for possible error recovery
REC         LDS    STACK  recover stack after an error
            . . .         do whatever
            JMP    ERROR  jump to error handler routine on error
            . . .
            JMP    REC    jump back to assemble the next instruction
ERROR       PRINT  MSG    print error message
            JMP    REC    restore the stack from any depth of sub. call
```

9.5.3 An Algorithm for the Assembler

```
int  assembler()
{ int           pc;          /* program counter */
  char          label; /* the label read, if given */
  short  opcode;       /* base opcode */
  short  mode; /* addressing mode */
  int           value; /* value of the operand */
  char          buf[30];     /* 30-character long input buffer */
  short  memory[large];      /* program memory */

  struct {                   /* define an entry in a symbol table */
    char        symbol;      /* symbol field */
    int         value; /* value of the symbol */
  } slist[6],                /* symbol table for labels with known values */
    ulist[6];                /* symbol table for labels with unknown values */

  struct {                   /* define the base opcodes */
    char        name[3];     /* three character opcode mnemonic */
    short  code;  /* one byte base opcode value */
```

```
} op_table[] =
    { 'LDA', $86;  'STA', $87;  'ADD', $8B;  'SUB', $80;  'BNE', $26;  'BRA', $20;
'SWI', $3F; 'RMB', 2;  'ORG', 1;  'END', 0;  '',0}  /* indicate end with a null string */

  do                              /* repeat until END is encountered */
  { readln(buf);  /* read a line */
    if ( get_label(buf, &label) == FOUND ) /* if a label is found */
    {      satisfy_ref(label, ulist);    /* satisfy any forward references */
           install(label, slist, pc);    /* install the label with the value = pc */
    }
    opcode = get_opcode(buf, op_table);      /* determine the base opcode from table */
    if (opcode & 0x80)  /* if opcode is $8x, then LDA, STA, ADD, SUB */
           accin(opcode, buf, label, slist);   /* handle instruction using accumulators */
    else if (opcode & 0x20)     /* if opcode is $2x, then branch instructions */
           branch(opcode, buf, slist, ulist);  /* handle branch instructions */
    else                  /* other special instructions */
           switch(opcode) {
           case $3F             :      memory[pc++] = opcode; break; /* SWI opcode */
           case 2        :      pc += get_number(buf); break; /* RMB opcode */
           case 1        :      pc = get_number(buf); break; /* ORG opcode */
           case 0        :      exit;  /* stop the assembler */
           default       :      error('ERROR - unrecognized instruction\n"); break;
           }
  while (TRUE);
}

boolean     get_label(buf, labelptr)
{  *labelptr = readbuf(buf);             /* label is the first character in a line */
   if (*labelptr == BLANK)               /* if first character is a blank, then not a label */
      return FALSE;
   else return TRUE;
}

int   satisfy_ref(label, ulist)           /* backpatch the unsatisfied forward references */
{  while ((ptr = search(label, ulist))    /* if finding the label in the list */
   {  temp = pc - ptr.value - 1;  /* calculate the offset between reference and label */
      if (temp > 127)                     /* forward reference cannot be > 127 bytes */
            error("ERROR - label %s out of range\n", label);
      else memory[ptr.value] = lowbyte(temp);          /* backpatch operand of branch */
   }
}

int   install(label, slist, pc)                     /* install the symbol into the symbol table
*/
{  if ((ptr = search(free, slist))   /* if a free entry if found */
   {  ptr.symbol = label;            /* put the symbol into the symbol table */
```

```
      ptr.value = pc;
   } else error("ERROR - symbol table overflow\n");
}

short        get_opcode(buf, op_table)  /* determine the opcode from the buffer and
opcode table */
{  skipblanks(buf);                              /* skip the blanks in the buffer */
   mnemonic = get_next_three_characters(buf);        /* next three chars are mnemonics */
   return searchop_table(mnemonic, op_table);        /* search opcode and return opcode */
}

int   accin(opcode, buf, label, slist)     /* handle instructions using the accumulator */
{  ch = read_buf(buf);                     /* get the next character i.e. staa or stab */
   if (ch = 'B')          opcode += 0x40;     /* if accumulator B is used, add $40 */
   else if (ch != 'A')                        /* else if the accumulator is not A, error */
      error("ERROR - illegal addressing mode\n");
   value = get_operand(buf,&mode);    /* get the addr. mode and value of operand */
   opcode += mode;                            /* add the mode to the base opcode */
   memory[pc++] = opcode;             /* emit the opcode */
   switch (mode[5:4]) {               /* test bits 5 and 4 of the mode */
   case 11  :       memory[pc:pc+1] = value; /* extended addressing mode */
                    pc += 2; break;
   case     00    :      if (opcode == 0x87) /* if opcode is STORE with imm. mode */
                         error("ERROR - illegal addressing mode\n"); /* fall through */
   case     01    :                              /* direct addressing mode */
   case 10  :       if ( value > 255 )   /* for direct and indexed mode, expect 8-bit  */
                         error("ERROR - operand overflow\n");
                    memory[pc++] = lowbyte(value);
   }
}

int   get_operand(buf, ptrmode)
{  skipblanks(buf);                              /* skip leading blanks */
   ch = readbuf(buf);                            /* get the first non-blank character */
   switch (ch) {
   case '#'  :      *ptrmode = 0;        /* indicate immediate mode */
                    return(get_number(buf));   /* return the hexadecimal number */
   case ','  :      *ptrmode = 0x20;   /* indicate index on X mode */
                    if (readbuf != 'X')    /* make sure that the index register is X */
                         error("ERROR - illegal addressing mode\n");
                    return(0);              /* else offset is 0 */
   default   :      temp = get_number(buf);   /* get a hex number */
                    if (temp < 256)        /* determine direct indexed or extended mode */
                         if (readbuf(buf) == ',')        /* determine direct, indexed mode */
                         {       *ptrmode = 0x20;   /* indicate index mode */
                                 if (readbuf != 'X')    /* make sure the index reg. is X */
```

```
                        error("ERROR - illegal addressing mode\n");
                } else  *ptrmode = 0x10; /* else indicate direct mode */
            else  *ptrmode = 0x30;   /* else indicate extended mode */
  }
}

int  branch(opcode, buf, slist, ulist)
{ memory[pc++] = opcode;            /* emit the opcode */
  skipblanks(buf);
  label = readlabel(buf);           /* get a label from the operand field */
  if (ptr = search(label, slist))  /* if the label is found, it is already declared */
  { temp = ptr.value - pc - 1;  /* calculate the offset to the label */
    if (temp < -128 )    /* a backward reference cannot be more than -128 bytes */
        error("ERROR - label out of range\n");
    memory[pc++] = lowbyte(temp); /* emit the offset in the operand field */
  }
  else             /* else if the label is not found, it is a forward ref. */
    if (ptr = search(free, ulist));     /* if a free entry is found */
    {     ptr.symbol = label; /* install the label */
          ptr.value = pc++;           /* and the address of the reference to be satisfied */
    }     else error("ERROR - symbol table overflow\n");
}
```

10 A Floating-Point Adder

10.1 Goals

1. To understand how floating-point numbers are handled in microcomputers
2. To write a routine that adds two single-precision floating-point numbers

10.2 Introduction

One of the bigger problems with microcomputer systems is the lack of built-in floating-point data types. That is, they lack the hardware mechanism to handle floating-point numbers efficiently. Instead, floating-point operations are handled by a set of special routines, and in more expensive units, they are handled by a resident floating-point coprocessor. The decision to leave out the floating-point capability in most general-purpose microprocessors is based on the performance-vs-cost design trade-off, for the prevalent uses of microprocessors are to handle I/O and to control peripheral devices. Microprocessors are designed more for I/O-intensive environments than for compute-intensive environments. Larger mainframe and supermini computers, whose uses are more compute-oriented, need floating-point capability and usually have it.

As microprocessors became more sophisticated, they became useful in more areas in computing and eventually settled into the area of *personal computing*. Here, many realized the need for handling floating-point numbers, and thus emerged the floating-point coprocessors. The floating-point coprocessors are designed specially to handle floating-point operations only. They do work only when the host processor executes an instruction that specifies the coprocessor, at which time they stop the host processor, occasionally using it to get data, until they complete the coprocessor instruction. At this point, the processor can use the result produced by the coprocessor. The separation of coprocessor and processor to handle floating-point and other operations is still more cost-effective than having one processor to handle all. This separation is also prevalent even in larger computers.

In systems that do not have coprocessors, a floating-point capability can still be provided by a set of utility routines. With these routines, the floating-point operations are slow, as we will see in this experiment. However, this may still be more cost-effective than having a coprocessor if the speed is not a critical factor.

10.3 Description

10.3.1 The Single-Precision IEEE Floating-Point Format

Some years back, the IEEE adopted a format for single-precision floating-point number representation. The format is shown in Figure 10.1.

Figure 10.1. Single-precision IEEE floating-point format

S represents a sign bit of the mantissa. The exponent is biased by 127, so that when **e** is 128, the true value of the exponent is 1 (**e** - 127). The range of the values of the exponent is from -127 (**e** = 0) to 128 (**e** = 255). The mantissa is expressed in sign-magnitude form so that **sig** is an unsigned number. For positive numbers, **S** = 0, and for negative numbers **S** = 1. The **sig** field has a hidden bit, so that the true value of the significand is 1.**sig**. The reason for having such an awkward format for the significand will become clear with an understanding of the "normalization" process. The value of a single-precision floating-point number is given in the following equation:

$$\text{value} = (-1)^{S} \times 2^{(e - 127)} \times (1.\text{sig})$$

The following are a few examples of single-precision floating numbers:

$+1.0 = 1.0 \times 2^0 =$ \$3F 80 00 00
$+3.0 = 1.5 \times 2^1 =$ \$40 40 00 00
$-1.0 = -1.0 \times 2^0 =$ \$BF 80 00 00

10.3.2 Normalized Floating-Point Numbers

The format described above assumes that the numbers are normalized. For a number to be normalized, the value of the significand must be between 1.0 and $2.0 - 2^{24}$. To normalize, the significand is shifted either toward the left (which doubles it) or toward the right (which halves it) until a 1 is seen at the most-significant bit. Then it is shifted left once more to hide the most-significant bit. The exponent must be either decremented or incremented to preserve the value of the number while shifting. This "bit hiding" effectively adds one bit of precision to the single-precision floating-point format without the additional bit.

Normalization is required before any comparison of two numbers can be made. If the numbers are normalized, the simple comparison of exponents will determine the larger number: the one with the larger exponent always has the larger magnitude. Consider two denormalized numbers, N1 = 3.0 (1.5×2^1) and N2 = 2.0 (0.25×2^3). A comparison of exponents indicates that N2 is the larger number when it is really not.

10.3.3 Addition of Floating-Point Numbers

To add two floating-point numbers, the following steps are required:

1. Align exponents
2. Add the significands
3. Normalize the result

The aligning of exponents is equivalent to aligning the decimal point, which is necessary for addition. To align, the significand of the number with the smaller magnitude (indicated by the smaller exponent) is halved, while the exponent is incremented by one, until the two exponents become equal. When the significands of two numbers of same sign are added, the resulting significand could be larger than 2.0. In process of normalizing this number, an exponent overflow could occur. On the other hand, if the significands of two numbers of opposite sign are added, the result could be less than 1.0. In this case, an exponent underflow could occur when the number is normalized.

10.3.4 Special Values

There are three classes of special values possible with floating-point numbers. They are zero, infinity, and denormalized numbers. Figure 10.2 shows the range of numbers that the single-precision floating-point format can represent.

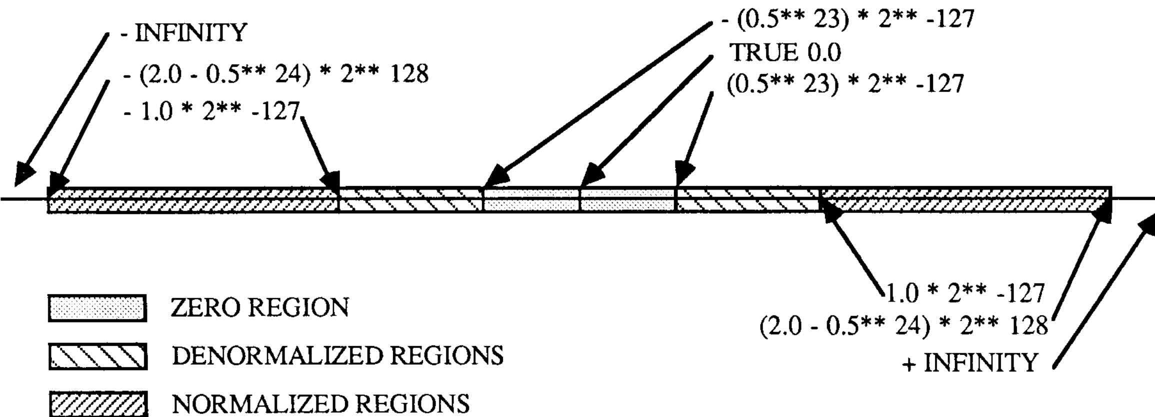


Figure 10.2. Range of single-precision floating-point numbers

Note that there are regions on both sides of the true "0.0" value, which represent the numbers that are just too small in magnitude to represent in single-precision format. These are simply treated as 0.0, which may introduce uncertainty into calculations. The regions outside of the "zero region" represent the numbers whose magnitude is too small to represent them as normalized numbers. These have the biased exponent of 0 (e = 0) and yet, the significands are less than 1.0. These are denormalized numbers. The next outer regions represent the numbers that can be represented by the single-precision

format. The outermost regions represent infinity; these numbers have too large a magnitude to be represented in single-precision format. The format of these special values is shown below:

zero	s = sign	e = 0	sig = 0
infinity	s = sign	e = $FF	sig = 0
denormalized	s = sign	e = 0	sig = non-zero

The resolution of the significand in single-precision format is $(1/2)^{23}$. That is, the changes in magnitude less than $(1/2)^{23}$ cannot be indicated with 23 bit significand. However, the accuracy of the number represented can vary from $(1/2)^{23} \times 2^{-127}$ to $(1/2)^{23} \times 2^{128}$.

10.4 Procedure

10.4.1 Standard Part

Write a subroutine that will add two single-precision floating-point numbers and return the result. Assume that the two numbers do not have any special values in them. However, the result may be any one of the special values. The two numbers are to be passed in as value parameters, which means that their values should not be changed by the subroutine. However, the routine need not be position-independent. The subroutine is to be called by the following calling sequence:

```
BSR   FADD
FCB   N1        address of the first number
FCB   N2        address of the second number
FCB   RESULT    address of the result
```

10.4.2 Optional Part

Add the capability to recognize the special values for any of two inputs and return the appropriate result. You should remember that denormalized numbers do not have that "hidden" bit.

10.4.3 Extra Credit

Write the routine described in optional part in position-independent, re-entrant code. This is much difficult than expected, mainly because of the limited number of index registers.

10.4.4 Validation

The following is a list of 20 test cases for the validation plan of the floating-point adder.

CASE	SIGNS	MAG	N1 (hex)	N2 (hex)	SUM (hex)	COMMENT
1.	+	N1 = 0	00 00 00 00		N2	0 + N2 = N2
2.	+	N2 = 0		00 00 00 00	N1	N1 + 0 = N1
3.	-	N1 = -0	80 00 00 00		N2	(-0) + N2 = N2
4.	-	N2 = -0		80 00 00 00	N1	N1 + (-0) = N1
5.	+	N1 = +∞	7F 80 00 00		7F 80 00 00	∞ + N2 = ∞
6.	+	N2 = +∞		7F 80 00 00	7F 80 00 00	N1 + ∞ = ∞
7.	-	N1 = -∞	FF 80 00 00		FF 80 00 00	-∞ + N2 = -∞
8.	-	N2 = -∞		FF 80 00 00	FF 80 00 00	N1 + (-∞) = -∞
9.	+ +	N1 > N2	40 00 00 00	3F 80 00 00	40 40 00 00	2 + 1 = 3
10.	+ +	N1 < N2	3F 80 00 00	40 00 00 00	40 40 00 00	1 + 2 = 3
11.	+ +	N1 = N2	3F 80 00 00	3F 80 00 00	40 00 00 00	1 + 1 = 2
12.	+ -	N1 > N2	40 00 00 00	BF 80 00 00	3F 80 00 00	2 + (-1) = 1
13.	+ -	N1 < N2	3F 80 00 00	C0 00 00 00	BF 80 00 00	1 + (-2) = -1
14.	+ -	N1 = N2	3F 80 00 00	BF 80 00 00	00(80) 00 00 00	1 + (-1) = ±0
15.	- +	N1 > N2	C0 00 00 00	3F 80 00 00	BF 80 00 00	-2 + 1 = -1
16.	- +	N1 < N2	BF 80 00 00	40 00 00 00	3F 80 00 00	-1 + 2 = 1
17.	- +	N1 = N2	BF 80 00 00	3F 80 00 00	00(80) 00 00 00	-1 + 1 = ±0
18.	- -	N1 > N2	C0 00 00 00	BF 80 00 00	C0 40 00 00	-2 + (-1) = -3
19.	- -	N1 < N2	BF 80 00 00	C0 00 00 00	C0 40 00 00	-1 + (-2) = -3
20.	- -	N1 = N2	BF 80 00 00	BF 80 00 00	C0 00 00 00	-1 + (-1) = -2

10.5 Hints and Suggestions

You will notice that there are many combinations of numbers for addition. An effective strategy to handle these many cases is to categorize them. First, check for zero or infinity conditions, and eliminate those. Note that if one number is zero, the result must be the other number. On the other hand, if one number is infinity, the result is also infinity. Second, swap the numbers so that N1 is always the larger number. This eliminates about half the cases. Third, align the numbers by shifting the significand of the smaller number right. But before this, set the hidden 24th bit if the number is in normalized format (exponent $\neq 0$). If one number is much greater than the other, the second number may become zero during the alignment phase. Set the 24th bit of the larger number also, if it is normalized. Fourth, if the signs are equal, add the numbers. If the signs are opposite, do N1 - N2. The sign of the result is always that of N1, the larger number.

Here are some observations on the result of floating-point additions:

1. The result takes the sign of the larger (in magnitude) of the two numbers.
2. If two numbers have the same sign, add the magnitude. To normalize the result, at most one right shift is needed. An infinity may occur if the numbers are large.

3. If two numbers have opposite signs, subtract the smaller number from the larger. To normalize, there could be any number of left shifts. If the numbers have approximately equal magnitude, the result could be too small to be normalized. Then it must be left denormalized.

11 Memory Systems

11.1 Goals

1. To understand the logic and timing relations for interfacing memory systems
2. To write routines to test the memory systems

11.2 Introduction

All computers need, and have, memory systems. Without a memory, the range of functions they can do is very limited. The memory gives them the capability to perform a task, and many variations of it, repeatedly. The memory provides "programmability".

Memory comes in various forms and shapes, but it can be divided largely into two categories: ones that use magnetic media and ones that use semiconductor devices. Magnetic disks, tapes, and bubble memories depend on the permittivity (capability to generate dipoles) of the media to store the binary data, and use electromagnets to re-orient the dipoles, thus modifying the data. These memories retain their data as long as a direct magnetic field is not applied to them. In that sense, they are nonvolatile.

The semiconductor memories use some sort of charge-storing devices, such as a capacitor, or a circuit equivalent to it, to store the data. The "charged" and "discharged" states are used to indicate binary values. There are two types of semiconductor memories: volatile and nonvolatile. The nonvolatile memories are devices such as ROM, PROM, EPROM, and EEPROM; they retain the data even after the power is turned off. A memory cell in ROM (Read Only Memory) has the connection to the power or ground burned into it so that it is always in one state when turned on. A PROM is a device similar to ROM, but it is programmable. In a PROM, all memory cells are configured to one state and have a fuse which can be blown by a strong current to change to the other state. Once a fuse is blown, the cell is no longer programmable. An EPROM is an erasable PROM. In an EPROM, a blown fuse can be restored by exposure to ultra-violet rays. Thus an EPROM has a transparent covering above the memory array, which must be covered to prevent accidental erasure after it is programmed. An EEPROM is an electrically erasable PROM. A strong current is used to blow and restore a fuse. Unlike the EPROM, EEPROM can be made "writable", but the read and write access times are grossly unbalanced.

The volatile type of semiconductor memories are RAMs (Random Access Memories). These memories forget their data when the power is turned off. On the other hand, they have symmetrical read and write access times. There are two types of RAMs: static and dynamic. Static memories retain the data as long as the power is supplied to the memory cells. Dynamic memories, on the other hand, can retain the data only for a short while, about 2 milliseconds. Thus, they need to be "refreshed" at least once every refresh cycle before the stored data is discharged below the threshold level.

The dynamic RAM cell is much simpler, consisting possibly of one transistor and a one capacitor, compared with its static counterpart, which has about six transistors. They are easier to make and thus cheaper, and can have more bits per chip. With current technology, it is possible to produce 256K-bit dynamic RAM at a reasonable cost. A one megabit dynamic RAM is becoming available.

Dynamic RAM is cheaper per bit than static RAM, but it needs refresh circuitry, which puts additional constraints and costs on the design of the memory system. For that reason, dynamic RAM is used in systems in which the cost saved by using it more than makes up for the added cost and complexity of refresh circuitry. This is strongly reflected in the design of dynamic RAM chips; they are organized in by-1 fashion: each chip stores one bit of each memory word for up to a million words. This reduces the number of I/O pins required for data to two, one for input and another for output. The memory array inside the chips is organized in an n-by-n matrix, where n is 256 for 64K-bit dynamic RAM. An address is composed of an n-bit row and an n-bit column of numbers, which are input to the chips over a set of n time multiplexed address pins. Only eight address pins are needed to specify a location in a 64K-by-1 dynamic RAM.

Static RAM, on the other hand, is organized in by-m fashion, where m is normally four or eight. Each chip stores m bits of data in one location, and consequently has m I/O pins for data alone. Because of this, the number of memory locations are relatively small; 8192 for 64K-bit static RAM organized in by-8 fashion. Hence, the address pins are not multiplexed. The advantages of designing with static RAM are, besides having simple interfacing requirements, flexibility and ease of expandability. It only takes one chip to design a 4K-by-8 memory system with static RAM, while eight chips are needed to do the same with dynamic RAM. Furthermore, to expand the memory, additional memory can be placed beyond the existing address space, and the existing memory system does not have to be modified at all. With dynamic RAM, the memory expansion is more involved because of the multiplexing of the addresses.

In this experiment, we will design and test a 2K-by-8 memory system using four of 1K-by-4 static RAM chips. With this experiment, we want to point out that a memory system does not have to be eight bits per word nor 64K or 28K words. In fact, a word can be anywhere from one bit to any desired width, and there can be any number of words in multiples of two, not powers of two. You should realize that it is certainly possible to design a 3K-by-8 memory system with six 1K-by-4 chips, or design a 64K-by-7 with seven 64K-by-1 chips.

We recommend that you review chapter 3 of *Single- and Multiple-Chip Microcomputer Interfacing* and data sheets on any 1K-by-4 static RAM chips.

11.3 Description

The block diagram of the memory system we will design is shown in figure 11.1. Since the memory devices require separate address and data signals, the multiplexed address/data signals from the MC68HC11A8 I/O pins must be demultiplexed. For this purpose, the MC68HC11A8 generates the address strobe signal, AS, indicating when the address is valid on the multiplexed output. The function of the address decoder is to decode the signals on the address bus, and when any of the addresses it recognizes appears, it enables

the memory chips that have the same address in the address space. Note that, since each chip stores only 4 bits of data, two chips (each storing a different four bits of data) must recognize the same addresses (to store eight bits of data).

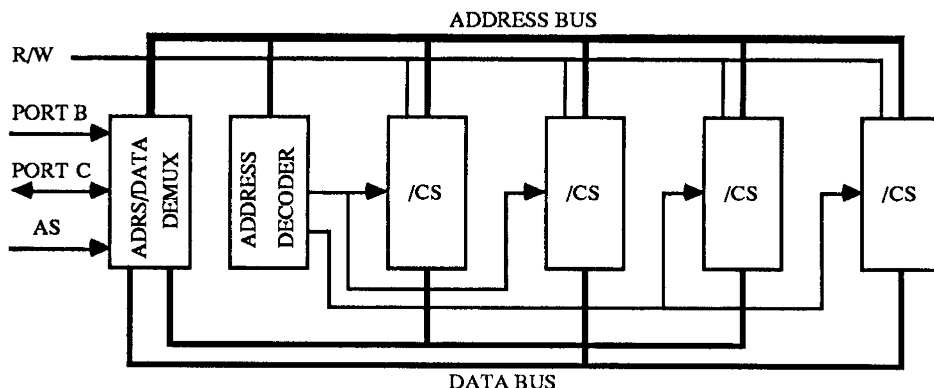


Figure 11.1. Logic diagram of the memory systems

11.3.1 Read Cycle Timing

When the MC68HC11A8 is in the read cycle, it is requesting data from memory. It outputs the address, asserts the R/W signal high, and expects the memory to output data (whose address is on the address bus) on the data bus during a certain time interval. An MC68HC11A8 will read the data bus at the end of this interval whether the data is available or not. So it is the responsibility of the memory device and its decoding circuitry to provide the data at least for this time interval. But before the memory device can output the data, it must first find the data from its storage and amplify the signals to the TTL level. The time it takes to do this is known as *access time*, and it is in the range of 150 to 300 nanoseconds for MOS memories. The precise definition of access time is the time between when the address is valid and when the data is available on the output pins of the memory chip. Since an address is not a dependable clock signal, a separate "chip-select" signal is used to indicate the validity of the address. For dynamic memories, the access time depends on both the row and column address strobe signals. Note that if the MC68HC11A8 expects data sooner than the memory device can output it, the device is too slow. On the contrary, if the memory device outputs the data much sooner than required, this is not a problem because the memory device will hold the data valid as long as the chip-select and read signals are valid.

11.3.2 Write Cycle Timing

On the write cycle, the MC68HC11A8 outputs an address, asserts the R/W signal low, and outputs data for a certain interval. It is the responsibility of the memory system to capture this data while it is available. The decoding circuitry must enable the memory

devices early enough and generate a write signal within this "interval" so that the setup and hold times required by the memory devices are satisfied.

11.3.3 Memory Test

Memory tests are required to identify faults and to verify the proper operation of a memory system. The faults could be in the devices, connections, or improper design. To verify the design, Buffalo can be used to read and write to a few random locations. However, to completely verify the memory system, more elaborate tests are required. The basis of a test is to write a pattern and read to verify for the correct pattern. The patterns must be something that can be easily generated on the fly. Some patterns are:

1. All 0s. Checks stuck-at-1 faults on data connections and devices
2. All 1s. Checks stuck-at-0 faults on data connections and devices
3. Rotating 1s. Checks for proper data connections
4. Rotating 0s. Checks for proper data connections
5. Low byte of address. Checks for proper address connection (low byte)
6. High byte of address. Checks for proper address connection (high byte)

11.4 Procedure

11.4.1 Standard Part

1. Design and build a 2K-by-8 memory system using the MCM2114 (1K-by-4). Use an incompletely specified decoding scheme from address $8000. Show both read and write timing diagrams. Draw a logic diagram, including the pin numbers, for all ICs so that the circuit can be built from it alone.

2. Write memory test programs (six separate programs) to test the memory system that you designed. The program should print the address, the expected data, and the incorrectly read data on screen when a fault is detected. After reporting a fault, the program should wait for a carriage return before continuing the test.

3. Verify that the memory system decoder is working properly by running the following program:

```
LOOP STAA $8000
     BRA  LOOP
```

   The decoder should assert one of two chip-select signals once every seven E clock cycles. Repeat for the other 1K words of memory.

4. Using Buffalo's memory modify command, verify that the memory system is functioning properly.

5. Run the memory test programs and verify the memory system.

11.4.2 Optional Part

Have your TA place a fault in your memory system. Using the memory test programs alone, find the fault. Make note of the symptoms and explain. An easy way to induce a fault is to disconnect one of the data lines from a memory chip so that the line is floating. Do not try to induce a stuck-at fault by connecting the data line to the ground or the supply voltage. This may cause a permanent damage to the MC68HC11A8.

11.4.3 Extra Credit

Suppose that you are designing a custom memory chip to be used with the MC68HC11A8. Analyze the read and write timing requirements of the MC68HC11A8 and draw up the minimum specification for the custom chip. Show the decoder design for the chip.

11.5 Hints and Suggestions

In order to add external memory to an MC68HC11A8, the MC68HC11A8 must be configured to power up in expanded multiplexed mode so that the address and data signals are available on I/O pins.

The MC68HC11A8 uses four areas of address space for the internal memories. They are

\$0000 to \$00ff	for RAM
\$1000 to \$103f	for a register block
\$b600 to \$b7ff	for EEPROM
\$e000 to \$ffff	for ROM.

In addition, Buffalo checks for the sign-on signal from the DUART (at \$D000 to \$D00F) and the Serial Communication Interface (SCI) during the power-up sequence in the expanded multiplexed mode. Although the DUART is not present in the system, the decoder for the memory system must consider these address spaces as occupied because Buffalo uses them.

Since most IC's contain multiple gates, a major design goal, other than satisfying the timing requirements, is to reduce the number of chips required. This may involve rewriting equations into different forms to use the unused portions of existing chips, rather than adding a new chip. For example, a NAND gate can be used as an inverter if one input is tied to a logic high.

11.5.1 Satisfying Read Cycle Timing

During a read cycle, the data setup and hold time for the MC68HC11A8 must be satisfied. The valid data should be available 30 nanoseconds before the fall of the E clock, and remain available 10 nanoseconds after the fall of the E clock. Before the valid data is available on the I/O pins of the memory device, three things must be satisfied:

1. The address must be stable for a certain period of time
2. The /Write Enable signal must be logic high (for read) for a certain period of time
3. The /Chip Select signal must be asserted (low) for a certain period of time

The exact time which the valid data is available on the I/O pins is dependent upon the above three signals. The data will be available only when all three signals are satisfied. The same is true for the holding of the data. As soon as one of the above signals is terminated, the data will disappear from the I/O pins after a certain period, known as the hold time. Like the setup times are different, the hold times may be different for the three signals. The interface circuitry must provide these signals well in advance so that the read data is available 30 nanoseconds before, and 10 nanoseconds after, the fall of the E clock.

11.5.2 To Satisfying Write Cycle Timing

During a write cycle, the data setup and hold time for the memory device must be satisfied. The MC68HC11A8 outputs the valid data at 125 nanoseconds, at the latest, after the rise of the E clock. It holds the data for 30 nanoseconds after the fall of the E clock.

12 A Traffic-Light Controller

12.1 Goals

1. To understand the technique of real-time synchronization
2. To study the usage of array data structures
3. To investigate the features of the Serial Peripheral Interface (SPI)

12.2 Introduction

In many industrial environments, the role of computers is somewhat different from what we perceive it to be. There, the main function of computers is to coordinate numerous tasks and subprocesses to achieve the maximum utilization of resources. The time frame in which the coordination must be satisfied may be in the order of tenths or hundredths of a second. For some, it is even shorter. This is "real-time" synchronization. The functions of the electronic control modules of today's new cars are good examples of real-time synchronization.

In this experiment, we will study the technique of real-time synchronization by implementing a traffic light controller. We suggest that you review sections 4-3 and 4-5 of *Single- and Multiple-Chip Microcomputer Interfacing* and section 6, Serial Peripheral Interface, of the *MC68HC11A8 HCMOS Single-Chip Microcomputer (ADI 1207).*

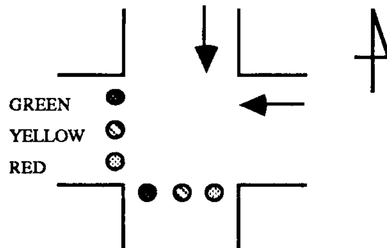


Figure 12.1. Traffic light arrangement

12.3 Description

12.3.1 The Traffic Light

Picture a fictitious intersection of two one-way streets, as shown in Figure 12.1. There are two traffic lights; one facing the southbound traffic, and the other facing the westbound traffic. The light "on" sequence of the traffic light for the normal operation is the same as the ones being used in the real world. That is the cycle of red, green, and

yellow. The duration of light "on" time for each sequence is not known at the time of design because the traffic condition of the intersection is not known. Late at night, when the traffic is light, the lights will be flashing; red for one direction and yellow for the other direction. The traffic light should not create a hazardous condition by turning both lights to green, or one to green while the other is still in yellow.

12.3.2 The Control-Sequence Interpreter

The control sequencer should be implemented with a sequence descriptor and an interpreter. In this way, the control sequence can be changed at any time to better accommodate the traffic conditions. The sequence descriptor is an entry in a table containing the light pattern and the duration. The control sequence is the increasing (or decreasing) order of the index of the table. In order to have a variable-length table, a null entry (pattern of 0 and duration of 0 seconds) can be designated as the end of the table, for this will never be a part of the sequence. For the experiment, the following sequence descriptor is suggested.

Table 12.1 Traffic light sequence descriptor

PATTERN		DURATION
SOUTH-BOUND	WEST-BOUND	
RED	GREEN	15
RED	YELLOW	3
GREEN	RED	10
YELLOW	RED	2
NULL	NULL	0

12.3.3 The LED Interface

The traffic lights are to be implemented with colored LEDs. Since the MC68HC11A8 lacks the current sourcing or sinking capability to drive the LEDs, it should not be connected directly to the LEDs. Instead, the following interface is suggested:

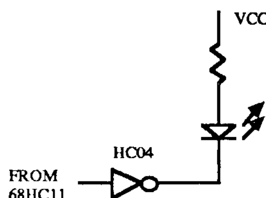


Figure 12.2. Control of an LED

Assuming that the forward current is 10 mA and the forward voltage is 2.2 V, the value of the current-limiting resistor is calculated using the following equation:

$$R = (V_{CC} - 2.2 - V_{OL})\ V / 10\ mA = 230\ ohms$$

The value of V_{OL} (of HC04) increases with the increase of sink current. For the sink current of 10 mA, it is about 0.5 volts.

12.4 Procedure

12.4.1 Standard Part

Write the control sequence interpreter. The interpreter should have a delay subroutine to waste exactly 1 second. Use the parallel I/O port B of the MC68HC11A8 to control the LEDs. The MC68HC11A8 must be in single-chip mode.

12.4.2 Optional Part

Use an output register at address $8000 to control the LEDs. The MC68HC11A8 must be in expanded multiplexed mode. Incompletely specified decoding may be used for the register.

12.4.3 Extra Credit

Use the Serial Peripheral Interface (SPI) to control the LEDs and low-current relays in place of the 74HC04 (figure 12.2). An 8-bit Serial-In, Parallel-Out Shift register (74HC595) will be needed to capture the bit stream from the MC68HC11A8. Also, modify the control sequence table so that the green light will flash twice at 1 Hz before changing to yellow.

12.5 Hints and Suggestions

12.5.1 Using the SPI and the 74HC595

The 74HC595 has two registers, shift and storage, each with separate clocks for shifting and storing data. These two clocks can be tied together to simplify the hardware. However, if this is done, the shift register state will be one clock ahead of the storage register, which means that the first bit sent will be available on the output after the

second clock. Since SPI sends the most significant bit first, and if the lower 6 bits are used for the light pattern, the last bit sent to the 74HC595 will still be in the shift register (and not available in storage register). What is needed is one more clock pulse from the SPI. To compensate for an additional clock, shift the light pattern once to left before writing it to the Serial Peripheral Data I/O Register (SPDR). The light pattern will be available on the output of the 74HC595, from Q_A to Q_F after 8 clocks.

The programming ritual for the SPI is presented below for a better understanding of it.

1. Configure the control register (SPCR)
2. Configure the data direction register for port D (DDRD)
3. Read the status register to clear any flag that is set (SPSR)
4. Write a pattern to the data I/O register (SPDR)
5. Repeat steps 3 and 4 as needed

Note that the MISO pin is not used and that the SS pin must be tied with a 4.7 KΩ pull-up resistor to +5 volts if port D bit 5 is an input.

12.5.2 Driving Relays and Inductive Loads

If an electromagnetic device like a relay or motor is to be driven, two problems must be solved. First, generally more current is needed to drive the device than is provided by a MC68HC11A8 or even a 74HC04. Study section 6-2,1 of *Single- and Multiple-Chip Microcomputer Interfacing* for some discrete drive devices, and "Interface Integrated Circuits" in the linear device catalogues of different manufacturers for solutions to this problem. The ULN2801 is particularly useful for many applications. Second, these electromagnetic devices create noise spikes when the current through them is abruptly turned off ($v \sim L\ di/dt$) and these noise spikes appear all through the system. These spikes have a serious effect on the MC68HC11A8 and similar microcomputers, causing erratic behavior and possibly erasing EEPROM. The Motorola TCF6000 Peripheral Clamping Array is an integrated circuit that uses excellent techniques that prevent pulses above +5 volts or below ground from entering a subsystem. It should be used on all inputs to the MC68HC11A8, and possibly it's outputs as well, wherever large noise spikes are expected. We built a MC68HC11A2-based relay-controlled sprinkler system that, without the TCF6000, always malfunctioned, but with it, has not yet malfunctionned in over a year.

13 An IC Tester

13.1 Goals

1. To become familiarized with the commonly used IC devices
2. To study the usage of array data structures
3. To investigate the features of the general-purpose Parallel I/O Interface

13.2 Introduction

The testing and verification of integrated circuit (IC) devices is a critical problem. There are many different kinds of testing that must be performed before an IC device can be commercially released. They fall largely into two categories: physical and functional. The physical aspects of the testing involve the measuring of current-driving capability, the rise and fall time of various signals, power dissipation, and others. The functional testing requires verification of each and all valid state transitions, at the least. In addition, the tests must verify that the device enters known (or expected) states on erroneous or unexpected conditions. Today's VLSI devices have more than 100,000 transistors and over 20,000 gates in them. Now, if the device is purely combinational in nature, there can be 20,000 (a very large number) different configurations, or states, of the device. Of course many states are physically not possible, but still there are a very large number of possible machine states. Added to that complexity, the devices are sequential, which only adds to the explosion of machine states. But here, we are concerned more with the logical aspect of simple combinational devices.

In this experiment, we will implement an IC tester to verify the logical functions of most of the 14-pin TTL, or functionally equivalent, devices. For a list of 14-pin combinational and sequential devices, refer to *The TTL Data Book for Design Engineers*, by Texas Instruments, Inc. We recommend that you review chapter 3 and section 4-6 of *Single- and Multiple-Chip Microcomputer Interfacing* and section 4, Serial Peripheral Interface, of *MC68HC11A8 HCMOS Single-Chip Microcomputer (ADI 1207)*.

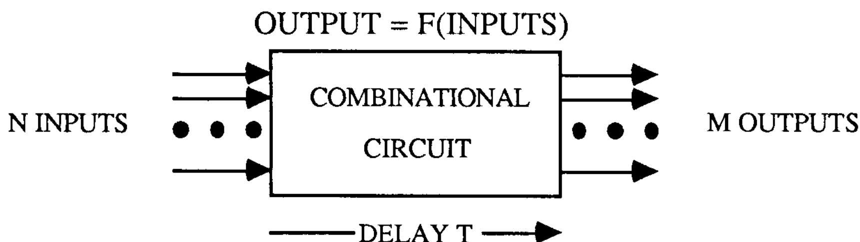


Figure 13.1. Combinational circuit model

13.3 Description

13.3.1 Combinational Circuits

Any combinational circuit can be modeled as a black box, with the inputs and the outputs, as shown in figure 13.1. The function of the black box is described with the output variables as Boolean equations composed of inputs. The behavior of the combinational circuit depends entirely upon the inputs. The circuit will enter a stable state some time after the state changes of the inputs have ceased. This is the maximum delay in the combinational circuit, and is the delay caused by the longest path from the inputs to the outputs. Many times, the longest delay path contains the largest number of "gate levels." If there exist race conditions or hazards in the circuit, induced mainly by the differences in delays in the gates and interconnections, the final state reached with a given input pattern may not always be the same. Under these conditions, the proper testing of the circuit is difficult because of the unpredictability of the delays involved. If properly designed, a circuit should be free of any race or hazard conditions. In this experiment, we will not concern ourselves with such problems, for all the TTL devices we are interested are free of them.

13.3.2 Sequential Circuits

Sequential circuits have the "history" property; that is, the behavior of the circuit depends upon the current state of the circuit and the changes on the inputs. Any sequential circuit can be decomposed into a combinational subnetwork and a memory element, as shown in figure 13.2.

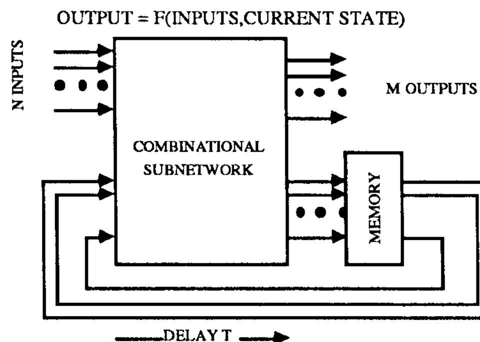


Figure 13.2. Sequential circuit model

When the inputs to the circuit are changed, the circuit may, but not always, change to a new state and produce the corresponding outputs. For Mealy circuits, the outputs depend on both the inputs and the current state. For Moore circuits, the outputs depend only on the current state. The memory portion of the circuit acts as a buffer between the outputs and inputs (or the current state and the next state) of the combinational network

to prevent multiple state transitions for a single input change. It is usually controlled by a clock whose period is longer than the maximum delay through the combinational subnetwork to give enough time for the circuit to reach a final steady state.

13.3.3 Testing Combinational and Sequential Circuits

Shown in figure 13.3 is the block diagram of the IC tester. To test, an input pattern is issued to the device. After allowing the device to reach a steady state, the outputs are read and compared with the expected output values. An error condition is reported if the match fails. Since the tester should be able to test any 14-pin IC device, the tester itself should be free of any device-specific information. The device-specific data can be supplied by a separate device function descriptor to the tester. In a sense, the tester program is an interpreter driven by the function descriptor. The function descriptor should contain the description of the pin type (either input or output) and the expected output pattern for each given input pattern. If a device contains multiple gates, the "inter-gate" effects can be ignored. If the device is sequential in nature, the test pattern sequence should be organized in such a way as to minimize the number of test patterns.

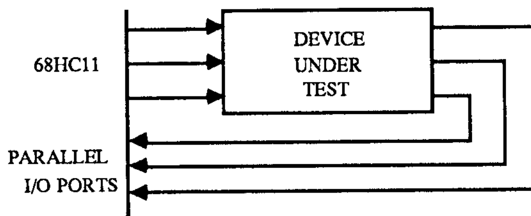


Figure 13.3. Block diagram of an IC tester

13.4 Procedure

1. Write the IC tester program. The program should be able to test and verify the proper functioning of most 14-pin 7400 series IC devices found in The TTL Data Book for Design Engineers. Upon detecting a fault, the program should display the expected test pattern and the detected actual pattern in binary representation for easy comparison. It should, then, wait for a response from the user. This is to give the user time to act upon the fault. The program should quit if *Q* is pressed, otherwise, the testing should continue. Upon completion of the testing, the message "FINISHED" should be printed.

2. Prepare the functional descriptor for the following devices: 7400, 7402, 7404, 7411, and 7474. Select a few of these devices and test. It is likely that you will not find any defective chips.

3. Induce a fault on one of the inputs to the tester by connecting it to either the ground or the VCC. Be sure to disconnect the connection between the parallel I/O pin on the MC68HC11A8 and the faulty pin. Run the test program, and demonstrate to the TA that the fault is properly detected.

14 A Logic Analyzer

14.1 Goals

1. To introduce the concept of indirect I/O
2. To understand the functions of the logic analyzers
3. To build a simple logic analyzer to use with the remaining experiments

14.2 Introduction

The logic analyzer is an indispensable tool for debugging both hardware and software designs. Although their capabilities vary greatly, the basic functions of the logic analyzer is to capture the logic states of the signals in real-time so that the history of the signals may be viewed at anytime, in any sequence. The sampling and storing of the data is usually controlled by four conditions; arm, trigger, qualifier, and delay. First, the sampling function must be armed, or enabled. Once armed, the sampling of data is not started until it is triggered. The trigger function is based on the state transitions of the selected signals, be they address, data, control, or whatever. Once triggered, the further state changes on these triggering signals have no effect on the sampling of the data. Not all the data being sampled is stored in the memory. The storing of the data is controlled by the qualifier. The qualifier is normally a binary signal whose state determines whether the data being sampled is to be stored or not. The state of the qualifier can be set to either or both the logic high or low. If the qualifier is set to logic low and if the signal connected to the qualifier has the logic-low state when sampled, the data is stored. The qualifier can be set to both (usually denoted by don't care) to allow storing of all the data being sampled. The data on the channels are sampled on either the rising or the falling edge of the external clock. Sometimes, a high-speed internal clock is used instead. Furthermore, the storing of data can be set to begin after a specified time.

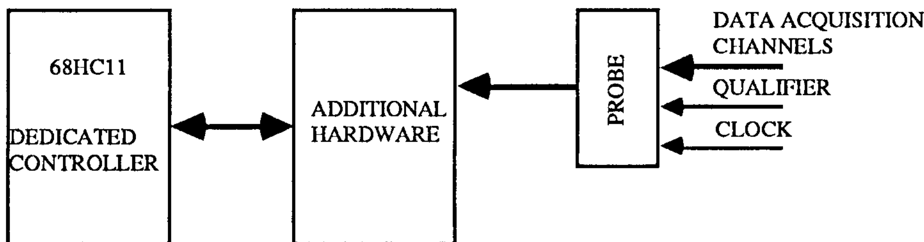


Figure 14.1. Simplified block diagram of the logic analyzer

Logic analyzers also enable the user to view the stored data in a random sequence, although the data is stored in time-sequential order. Many display the data in a waveform mode for easier understanding of the signal history. The pattern matching capability is also commonly provided by many. More expensive units provide the disassembling

capability on the stored data, address, and bus control signals. These user-friendly features add more value to the logic analyzers.

In this experiment, we will implement a simple, yet very useful logic analyzer that we can use for the remaining experiments. We recommend that you review chapter 4 of *Single- and Multiple-Chip Microcomputer Interfacing.*

14.3 Description

14.3.1 Features of the Logic Analyzer

The logic analyzer we will implement is to have the following features:

1. eight high-impedance data channels at the maximum sampling rate of 2 MHz
2. stores up to 2K samples
3. arm/disarm function
4. selectable address trigger capability (16 bit)
5. a three state (0, 1, and X) qualifier channel
6. data acquisition always on the falling edge of the external clock
7. A user-friendly interface (This is explained in detail below)

14.3.2 A Functional Description of the Logic Analyzer

The logic analyzer should recognize the following commands from the keyboard:

A Arms the logic analyzer. The message "ARMED" should be displayed before the sampling function is actually armed. The logic analyzer should, then, be disarmed.

T adrs Sets the address trigger. The address is entered in hexadecimal notation (i.e., FE34).

Q value Sets the qualifier state to one of the following; 0, 1, X.

D n Displays the data stored in the nth memory location. The valid range of n is from 0 to 2047. Assume that the number n is in decimal notation, and that it is terminated by a carriage return character. For instance, D 23 will display the data stored at the 24th memory location.

F data Searches the memory for the data. The search begins from the next memory location to the last (2047) memory location. The data is specified in an 8-digit binary notation, and may have don't cares for the digits. For example, F 1101XX01 will search for 11010001, 11010101, 11011001, and 11011101 among the acquired data. If found, the address of the memory location and the data should be displayed. If not found, a message should be displayed indicating this.

+ p Displays the data stored in the pth memory location from the currently displayed data. The number p is in decimal notation and is terminated by a carriage return.

- m Displays the data stored in the mth memory location before the currently displayed data. The number m is in decimal notation and is terminated by a carriage return.

<cr> Displays the data stored in the next memory location.

For all display commands (D, F, +, -, <cr>), the following display format should be used:

ADDRESS DATA

where the ADDRESS is the address of the memory location displayed in hexadecimal notation, and the DATA is the stored data shown in binary notation (1s and 0s). The current memory location should always be pointing to the previously displayed address so that the commands +, -, and <cr> will produce the desired results. Assume that all inputs will be in capital letters.

14.4 Procedure

14.4.1 Standard Part

1. Prepare a detailed hardware and timing diagram for the logic analyzer. You may use simpler hardware-software combinations for the design, as long as the logic analyzer performs as required.
2. Write the software for the logic analyzer to recognize A, T, D, +, -, and <cr> commands.
3. Connect the probe to the address and data bus of an MC68HC11A8 microcomputer running the following program:

```
        ORG 1
        CLR $0
LOOP    INC $0
        BRA LOOP
```

Set the logic analyzer to trigger on the address of 1. Note that the MC68HC11A8 being monitored must power up in the expanded multiplexed mode, and the address/data bus must be demultiplexed to provide the stable address from the middle of the E cycle. View the captured data and explain the cycle-by-cycle operation of the MC68HC11A8 for the first 24 cycles.

14.4.2 Optional Part

Upgrade the logic analyzer to recognize the commands Q and F. Understand that these commands require support hardware for their functions. Repeat step 2 of the standard part. This time, sample only the data being written to location 0.

14.4.3 Extra Credit Part

Expand the sampling channel width to 24 bits. Of the 24, the first 16 channels may be used for capturing the addresses. Modify the **F** command to search for the given address, as well as the data. The format of the new **F** command is as follows:

F adrs data

where **adrs** is a 4-digit hexadecimal number and **data** is an 8-digit binary number. Note that don't cares may be embedded in some or all of the **data** specification, and that a don't care (denoted by X) may be given as the address specification. Some examples of this command are:

```
F X 10001X0X
F 34DF XXXXXXX1
F DF34 11001011
F 1234 XXXXXXXX
```

Also modify the display format of the D, +, -, <cr>, as well as F, commands to the following:

```
adrs1 adrs2 data
```

where **adrs1** is the address of the store memory in the logic analyzer, **adrs2** is the value of the first 16 channels, and **data** is the value of the last 8 channels. **adrs1** and **adrs2** are in hexadecimal notation, and **data** is in binary notation.

14.5 Hints and Suggestions

14.5.1 A Block Diagram of the Logic Analyzer

A block diagram of a suggested implementation of the logic analyzer is shown in figure 14.2.

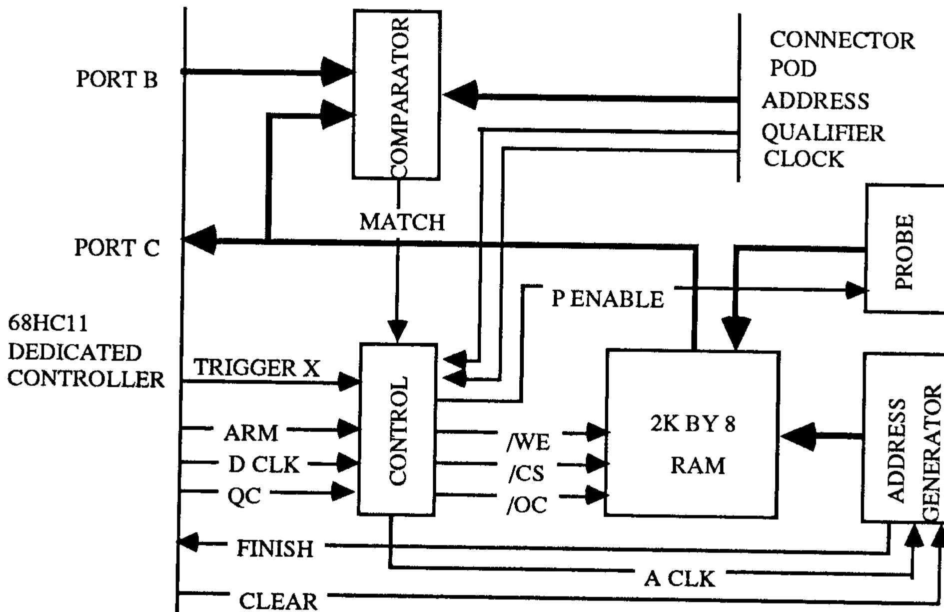


Figure 14.2. Block diagram of the logic analyzer

The logic analyzer operates in one of three states: unarmed, armed (but not triggered), and triggered. In the unarmed state, the logic analyzer is attentive to the commands from the keyboard. It may display the acquired data or set the trigger address, qualifier state, and external clock polarity. In the armed state, the sampling function is enabled and is waiting for the trigger from the address comparator. Once triggered, the logic analyzer enters the triggered state, in which the channels are sampled and the data is stored in memory. Of course, the qualifier condition must be satisfied in order for the data to be stored in the memory. The Store Clock (S_CLK) is fed to the address generator to provide the sequential addresses from 0 to 2047 for the memory. There is no internal clock for sampling. However, the E clock from the dedicated controller may be used as the internal sampling clock to acquire data from slow systems. The data is collected until all 2048 samples are stored in the memory, at which time, the logic analyzer returns to the unarmed state.

The critical portion of the logic analyzer is the write cycle of the memory. The address and the control (/CE and /WE) signals must be properly asserted to capture the data on the falling edge of the external clock. This suggests that the /WE signal is a derivative of the external clock. A rough sketch of the required timing relations for the memory system is shown in figure 14.3.

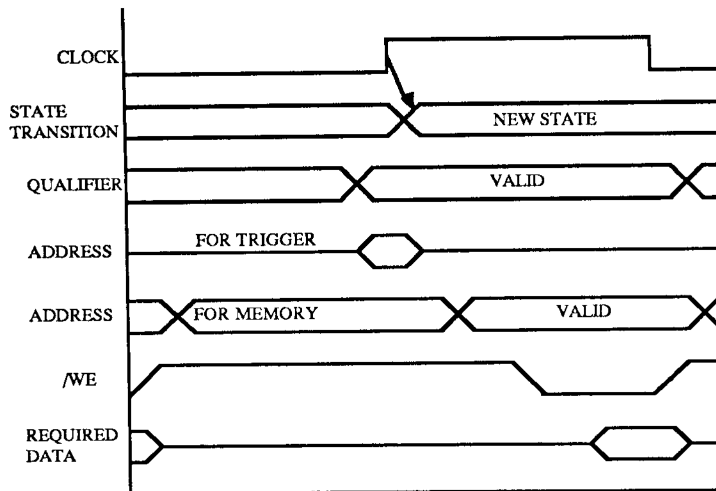


Figure 14.3. Timing diagram of the write cycle

14.5.2 A State Diagram of the Logic Analyzer

A suggested Mealy state machine for the logic analyzer is shown in figure 14.4.

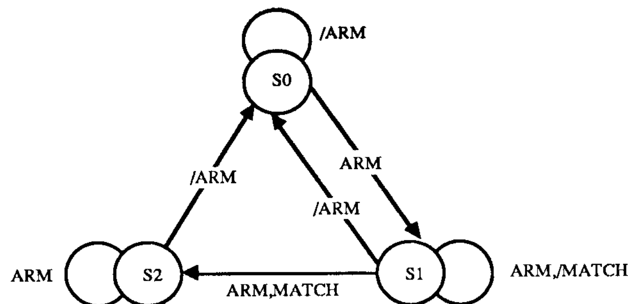


Figure 14.4. State diagram of the logic analyzer

The state transition should be made to occur in the middle of the external clock so that the /WE signal can be generated early. The memory control signals can be generated with the following equations:

/WE	=	(S2 * QUALIFIER_MATCH * CLOCK)'
/CS	=	(FINISH')' = FINISH
P_ENABLE	=	ARM or ARM'
/OC	=	(S0 * D_CLK)'

14.5.3. Qualifier Setting

To specify the QUALIFIER state, a four-state variable QC is used. The QC may be defined as follows:

QC	=	0,	sets the qualifier to logic 1
QC	=	1,	sets the qualifier to logic 0
QC	=	2,3	sets the qualifier to don't care

From this the QUALIFIER_MATCH signal can be expressed with the following equation:

$$\text{QUALIFIER_MATCH} = (\text{QUALIFIER} \oplus QC_0) + QC_1$$

14.5.4 The Address Generator

The 74HC4040, a 12-bit asynchronous counter, is an ideal choice for the address generator. Of the 12, the lower 11 bits are used as the address signal to the memory. The highest bit of the counter, Q_L, acts as the "FINISH" signal as its state changes from low to high. This signal is monitored by the MC68HC11A8 to detect the end of the sampling session, upon which the ARM signal is negated to force the state transition from "triggered" to "unarmed."

The address generator requires two clocks; one for generating the addresses to store the data, and the other for generating the addresses to read the data. In order to generate arbitrary addresses, a software controlled D_CLK is used. The signal D_CLK is toggled the number of times as required to generate the desired address. The CLEAR signal is use to reset the counter to a known state. Note that the 74HC4040 is cleared on the high level of the CLEAR pulse. The following equation may be used as the clock signal to the address generator:

S2 * QUALIFIER_MATCH * CLOCK + (S0 * D_CLK)'

The maximum ripple delay through the chain in the 74HC4040 is about 264 nanoseconds (11 stages x 24 nanoseconds delay per stage). If this ripple delay is too long to satisfy the address setup time (of about 20 nanoseconds), the falling edge of the /WE signal must be delayed to provide the needed setup time. The rising edge must still occur on the falling edge of CLOCK.

14.5.5 The High Impedance Probe

A bus transceiver, such as the 74HC244 can be used as the probe.

14.5.6 The Address Trigger

Comparators, such as 74HC85s, can be used as the address trigger. The comparators can be connected to ports B and C. However, since port C is also used as input, a register is needed to latch the data from port C. The register latch control can be generated by the STRB signal, as shown in the logic diagram in figure 14.5.

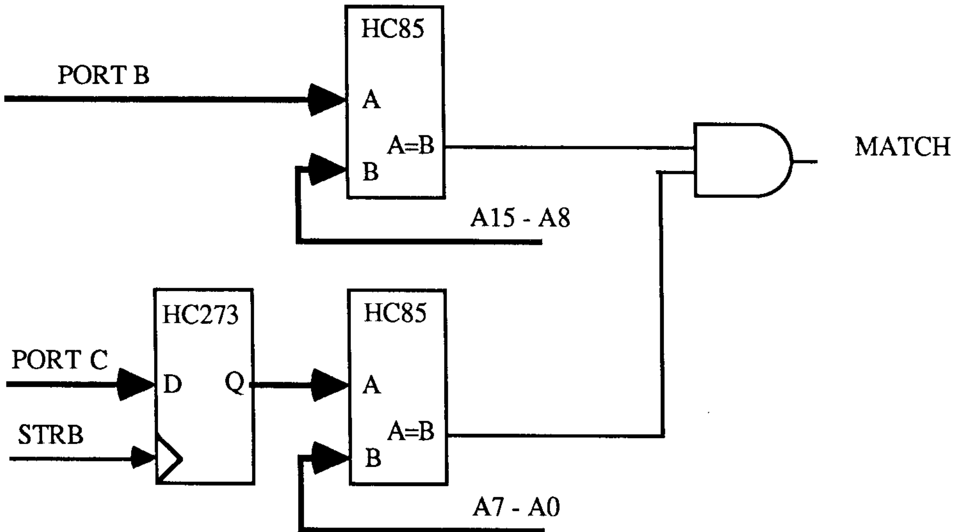


Figure 14.5. Logic diagram of the address trigger

15 A Bar-Code Reader

15.1 Goals

1. To interface an interrupt-driven device
2. To write software to interpret bar codes

15.2 Introduction

The bar-code reader is a very efficient and convenient input device for systems in which it is necessary to identify items quickly. Such systems are widely used in automated factories, warehouses, and even in retail stores. The bar-code system can be used in all situations dealing with packaged goods. In fact, it is not unreasonable to expect that, in the near future, all cash registers and similar equipment dealing with finished products will be equipped with a bar-code reader.

The bar-code system that we are most familiar with is the price scanner found in many grocery stores. There, instead of keying in the price of an item, the bar-code reader scans and identifies the encoded item number. The price of the item is then obtained from the central data base containing the pricing information for every item found in the store. Currently, almost every product has a Universal Product Code (UPC) on its package. There are numerous advantages to using the bar-code system as opposed to not using it, including easier and more accurate price adjustments as well as accurate, strict inventory control.

In this experiment, we will implement a simple bar-code system using a digital bar-code wand and the industrial 2-of-5 code. Although the implemented bar-code system will be trivial compared to the commercial bar-code systems, the key concepts and the techniques used should be the same.

We advise you to review sections 5-2 and 7-1 of *Single- and Multiple-Chip Microcomputer Interfacing*. We also recommend that you consult Hewlett Packard, "Application Note 1013: Elements of a Bar-Code System" for further information.

15.3 Description

15.3.1 The Industrial 2-of-5 Code

The industrial 2-of-5 code is the simplest of the width-modulated industrial bar codes. Its characteristics are:

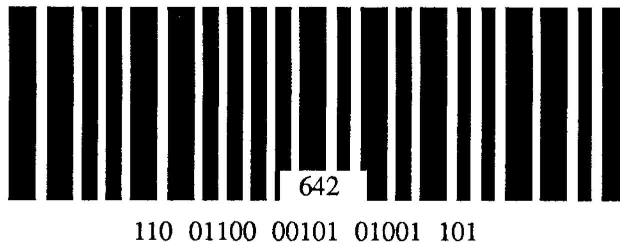
1. Numeric character sets only; 0 to 9
2. Uses 2-of-5 codes to represent the number; there are two 1s and three 0s in the code. Even parity used

3. Binary encoding; wide bars represent 1s and narrow bars represent 0s. A wide bar is typically two to three times wider than a narrow bar
4. Use of a start and a stop character
5. Use of intercharacter spaces; a space whose width is the same as that of a narrow bar separates two bars. Bar is black and space is white
6. Optional message checksum character

Table 15.1. 2-of-5 code

CHARACTER	LSB 1	 2	 4	MSB 7	PARITY P
0	0	0	1	1	0
1	1	0	0	0	1
2	0	1	0	0	1
3	1	1	0	0	0
4	0	0	1	0	1
5	1	0	1	0	0
6	0	1	1	0	0
7	0	0	0	1	1
8	1	0	0	1	0
9	0	1	0	1	0

The message structure of the industrial 2-of-5 code is shown in figure 15.1.



110 01100 00101 01001 101

Figure 15.1. Number 642 in 2-of-5 format

The message is preceded by the start code (110) and followed by the stop code (101). The least significant bit of a character precedes the more significant bits. Note that a wide bar represents 1 and a narrow bar represents 0. Each bar is separated by a space whose width is same as that of a narrow bar. In the above example, the checksum is not present. At both ends of the message (before the start character and after the stop character) are margins.

15.3.2 Hardware Description

The required hardware description of the bar-code system is shown in figure 15.2. When the wand is over the reflective surface (white spaces and margins), a logic 0 is output on the VO pin. When it is over the nonreflective surface (bar), a logic 1 is output on the VO pin. When the bar is held in midair, a logic 1 is output on the VO pin because the optical sensor in the wand does not receive the reflected light.

The parallel port strobe A function of the MC68HC11A8 is used to detect the transitions between the reflective and nonreflective areas. For instance, the falling edges which occur on the space to bar transitions, and the rising edges, which occur on the bar to space transitions, can be sensed using strobe A with suitable control bits in the PIOC register. The time between these transitions can be measured by the main program, which increments a number every 10 μsec. or so. This time can be examined when a strobe A edge interrupt occurs, to get the size of the bars in the code. The Schmitt-trigger inverter (74HC14) is used to de-glitch the signal output from the wand.

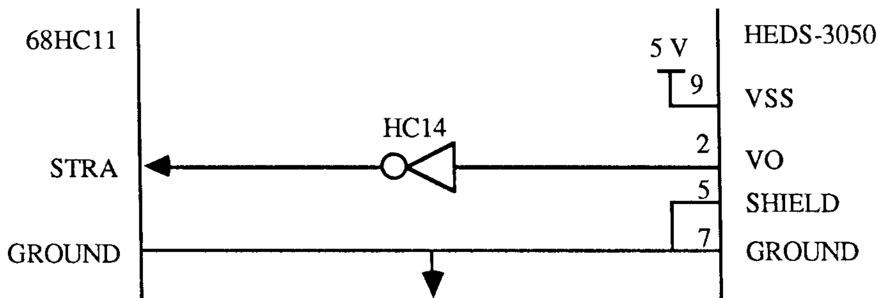


Figure 15.2. Bar-code reader to the MC68HC11A8 interface

15.3.3 Software Design Considerations

There are several design issues in implementing a robust bar-code system, one of which stems from the fact that the width of the bars seen by the system varies with the scan velocity. A bar scanned at slow speed will appear wider than when scanned at a faster speed. The inconsistency of scan velocity is seen at two levels. At the lower level, the scan velocity changes (it generally increases) as the wand is swept across a bar-code. This is because the sweeping motion starts from zero initial velocity. This makes the bars appear wider in the beginning than near the end of the scan motion. At worst, the narrow bars near the beginning will appear to be wider than the wide bars near the end of the scan motion.

At the higher level, the scan velocity varies from sweep to sweep, which suggests that a design using a predetermined bar width will be of limited use. It will impose a narrow range for the scan velocity. What is needed is a "self-tuning" system in which the change in scan velocity is reflected in the assumed width of the bars. The following scheme is suggested:

1. Count the width of the bars and spaces in the start character, or stop character in reverse scan, to calculate the width of the start character.
2. From the calculated width of the start character, calculate the width of the narrow and wide bars. Since there are two wide bars, a narrow bar, and two spaces in the start character, the total width is divided by 7 to produce the width of a narrow bar (if the narrow to wide bar ratio is 1:2).
3. Calculate the threshold width to be the average of the narrow and the wide bars. The threshold width can be used to decode the bars into binary states 0 or 1.
4. Continuously update the width of the wide and narrow bars, and subsequently the threshold width, as more bars are detected and decoded. In updating the bar width, use the average of the previously calculated width and the new bar width. This places more weight on the width of the newly detected bar in order to adapt quickly to the changing scan velocity.

Another design issue is the treatment of the margins. Since the wand held in midair is interpreted as a bar (nonreflective surface), the midair-to-margin transition (for getting ready to scan) will cause a premature interrupt. This problem can be fixed easily if only the forward scanning of the bar-code is allowed. However, if both forward and reverse scanning of the bar code is allowed, the appropriate handling of the midair-to-margin transition, and vice versa, is crucial. There are now two situations. In one situation, the wand is held in midair, placed in the margin, and then swept across the message. The second situation occurs when the wand is swept across the message in the reverse direction of the previous sweep, without ever leaving the margin. In the first situation, a very wide bar is detected, and in the second, a very wide space is detected in the beginning of the scan. One solution to this problem is to place an upper limit on the width of the bar and the space. And when that limit is violated, simply reset the system to detect a bar that is not too wide.

The solution to the problem described in the previous paragraph can also be used to detect the end (or beginning if scanning in the reverse direction) of a variable-length message. Since there is not a fixed number of bars in a variable-length message, the only way to detect the end of the message is to detect a margin.

15.4 Procedure

15.4.1 Standard Part

Implement the bar-code system, assuming a fixed message length of three characters and forward scanning only. Also assume that the narrow-to-wide bar width ratio is 1:2. Verify the system on the bar codes shown in section 15.5.

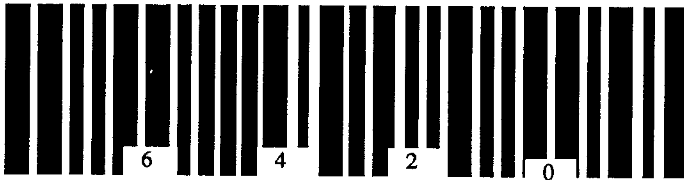
15.4.2 Optional Part

Allow both forward and reverse scanning. The wand may or may not leave the paper between a continuous forward-and-backward scan.

15.4.3 Extra Credit Part

Assume a variable-length message, in addition to allowing scanning in both directions. For practical purposes, the longest bar-code may be assumed to be 10 characters. Use the counter/timer system in the 6811 to measure pulse width (use Input Capture 1 and 2).

15.5 Test Patterns



15.6 Hints and Suggestions

15.6.1 The Algorithm

The algorithms for the functions in the extra-credit part are shown below:

```
main()
{ int       detec_start_stop(),        /* detects start/stop code, sets flag accordingly
*/
  int       getchar();        /* function returning a 2-of-5 code */
  char      reverse();        /* function to reverse the bit ordering when reversed */
```

```
  init();                          /* set up the interrupt vectors, enable interrupts */
  while TRUE            /* loop forever */
  {    dark = false;        /* initially, wand is over a space */
       if (detec_start_or_stop(&rev))         /* if start or stop character is detected, */
       {          for (i = 0 ; ! getchar(a) ; i++)          /* repeat reading a char until error */
                            code[i] = a;
                  if (rev & a == start_code)  /* if reverse scan */
                            for (i--; i ; i--)      /* process the codes in LIFO fashion */
                            {          a = reverse(code[i]); /* reverse bit pattern in code */
                                       conv_print(a);  }  /* conv. 2of5 code to num., print */
                  else if (! rev & a == stop_code)       /* if forward scan */
                                       for (j = 0; j < i; j++)        /* proc. in FIFO fashion */
                                                 conv_print(code[j] /* conv 2of5 code */
       }                               /* end of if */
  }                                    /* end of while */
}                                      /* end of main */


boolean           detec_start_or_stop(flag)   /* detects start/stop char. */
{  char            *flag;
   if (count_space(n))       /* wait for a bar (skips over the margin) */
        error(1);            /* if space too wide, error -- goto the while TRUE loop */
   if (count_bar(n))         /* count the width of the first bar */
        error(2);            /* if bar too wide, error -- goto the while TRUE loop */
   bar1 = n;
   if (count_space(n))       /* count the width of the first space */
        error(3);
   space1 = n;
   if (count_bar(n))         /* count the width of the second bar */
        error(4);
   bar2 = n;
   if (count_space(n))       /* count the width of the second space */
        error(5);
   space2 = n;
   if (count_bar(n))         /* count the width of the third bar */
        error(6);
   bar3 = n;

   nbar = (bar1 + space1 + bar2 + space2 + bar3) / 7;
   wbar = nbar << 2;      /* calculate the width of a wide bar */

   p = bar1;                 /* point p to bar1. */
/* Assume space1, bar2, space2, bar3 are in consecutive locations */

   for (a = i = 0; i < 5; i++)        /* repeat 5 times */
```

```
    a = ( a | get_level(p++) ) << 1;        /* convert each bar into a binary state */
  if (a == start_code)    *flag = false;    /* if star code is detected, forward scan */
  else if (a ==stop_code)*flag = true;      /* if stop code is detected, reverse scan */
  else error(7);          /* else error -- reset and goto the while TRUE loop */
  return (0);             /* exit, no error */
}

boolean        count_space(number)        /* counts the width of a space */
{ int          *number = 0;
  while (! dark)          /* while space */
  if ((*number)++ > limit)    return(1);  /* if counter overflow, error -- too wide */
  return(0);              /* no error */
}

boolean        count_bar(number)          /* counts the width of a bar */
{ int          *number = 0;
  while (dark)            /* while bar */
    if ( (*number)++ > limit )   return(1);        /* if counter ov., error -- too wide */
  return(0);              /* no error */
}


int     get_level(number)          /* converts a bar width into a binary state */
{ int          *number;
  return ( (*number > ((wbar + nbar) >> 1)) ? 1 : 0 );
}

int     getchar(a)                 /* counts 5 bars, convert their widths to a binary state
*/
{ char         *a = 0;
  for ( i = 0; i < 5; i++)/* repeat 5 times */
  {    if ( count_space(n) )    return (1);   /* error, if space too wide */
       if ( count_bar(n) )      return (1);   /* error, if bar too wide */
       *a = ( *a || get_level(n) ) << 1;
  }
  return (0);             /* return with no error */
}
```

The functions reverse() and print() are simple routines, hence their algorithms are not given here. The interrupt handler routines for IC1 and IC2 are shown below:

```
ic1hnd()                           /* interrupt on falling edge */
{ clear IC1F in TFLG1 register;  dark = true;   }

ic2hnd()                           /* interrupt on rising edge */
{ clear IC2F in TFLG1 register;  dark = false;   }
```

15.6.2 Measuring the Width of Bars and Spaces

Roughly, the width of bars, both wide or narrow, will be translated into a time period in the order of tens of milliseconds. In fact, you can, and should, measure the width of a bar with the logic analyzer. To tune the bar-code system to give the widest operating range of the scan velocity, you should conduct a sensitivity study to determine the best amount of delay between the counts. If the count is incremented too fast, the lower limit of the scan velocity will be high. On the other hand, if the count is incremented too slowly, the system will be unstable at low scan speeds. This is because there is not much difference in counts between the narrow and wide bars, so that some bars may be interpreted wrongly. As an extreme example, a system that counts 2 for a narrow bar and 4 for a wide bar is very unstable compared to one that counts 2000 and 4000.

15.6.3 Debugging Interrupt-Driven Software

The first thing to check for if the program exhibits strange behavior is the interrupts. Since an interrupt handler is difficult to debug in real-time, a good way to see if it is working is to embed "print" statements in the codes so that you can actually see the flow of the program. For debugging purposes, it would be a good idea to check for every error condition and print messages (simple codes or numbers will be sufficient). This is a common debugging technique without the use of a debugger.

For uniform and efficient handling of error conditions, a goto instruction can be used at all levels of routines. For example, the first thing that a program does is save the stack pointer on a global variable (not on a stack). On erroneous conditions, the program simply jumps to the error-handler routine, which issues the appropriate messages and resets the system by restoring the stack pointer. This is an elegant way to take care of the error conditions which have occurred in subroutines nested several levels below.

16 A Magnetic Card Code Reader

16.1 Goals

1. To interface an interrupt-driven device,
2. To write software to interpret magnetic card codes.

16.2 Introduction

The magnetic card code reader is a very efficient and convenient input device for credit cards and security cards. Each user has his or her own card, and the card has a magnetic strip on it. The card is pulled through a card reader, and the magnetic strip is read in the same manner as an audio magnetic tape is read. Such systems are widely used in retail stores, warehouses, and secure research labs. The magnetic card code system can be used in situations dealing with money or access. In fact, it is not unreasonable to expect that in the near future, many security systems and money transaction systems will be equipped with a magnetic card code reader.

The magnetic card code system that we are most familiar with is the credit-card scanner found in many grocery stores. There, instead of keying in the user's account number from the the credit-card, the magnetic card code reader scans the credit-card and identifies the encoded item number. There are many advantages to using the magnetic card code system: ease of use, more accurate accurate accounting and stricter security.

In this experiment, we will implement a simple magnetic card code system using a digital magnetic card code reader. Although the implemented magnetic card code system will be trivial compared with commercial magnetic card code systems, the key concepts and the techniques used should be the same.

We advise you to review section 5-2 and scan section 9-1.1 of *Single- and Multiple-Chip Microcomputer Interfacing.*

16.3 Description

16.3.1 The Credit Card Code

The credit card code is the simplest of the commercial magnetic card codes. You need a magnetic card reader such as the American Magnetics MagStripe™ Card Reader Model 40S5DA, or equivalent, to read credit cards. Figure 16.3 shows the hardware connections, which identify the clock and data wires from the reader. Figure 16.1 shows the timing of the data and clock signals. The clock is normally high. When a card is moved through the reader, the clock pulses low for 50 microseconds each time a bit of data is to be read, and data should be read on the rising edge of the clock pulse. A high signal is a 1, and a low is 0.

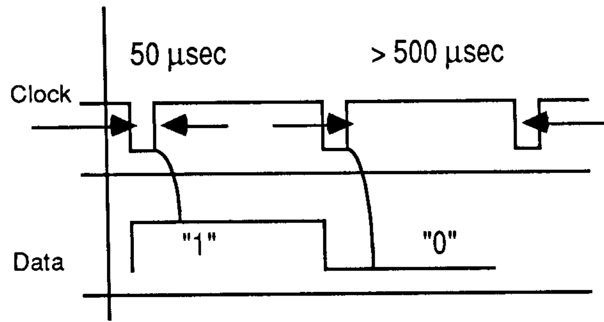


Figure 16.1. American Magnetics MagStripe™ Card Reader signals

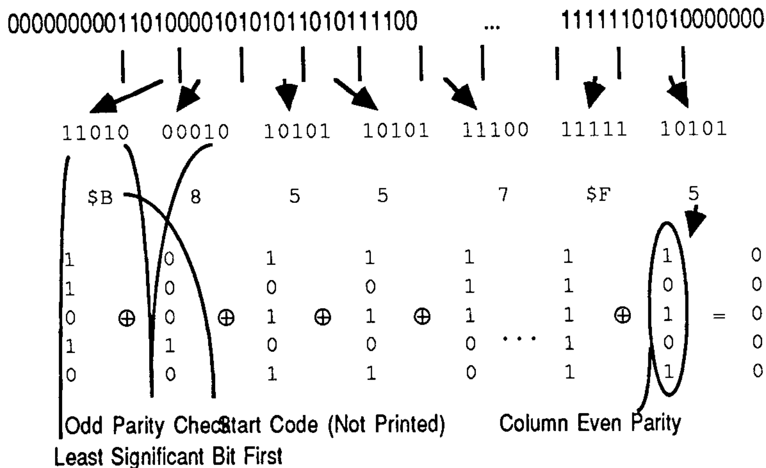


Figure 16.2. Credit card code

The message structure of the credit card code is shown in Figure 16.2. If a card is moved through the reader, you should see a pattern of 0s and 1s as shown below:

00000000000000000000110100001010101101011010111100001000110100001000010
1010110000100111110011100011010001000100011010000111111010100000000000

This pattern corresponds to the numeric code (which is also embossed on the credit card, except for some of the last few digits) as shown below:

855574608519776846OF5

Its characteristics are

1. Numeric character sets only; 0 to 9 and special marking characters ($B, $D, $F)
2. 5-bit BCD codes to represent the number, least significant bit first; the last bit is an odd parity on the 5-bit pattern

3. Use of a start ($B) and a stop ($F) character and an optional character ($D) about three quarters through the pattern
4. Message parity character after the stop character ($F). The least significant bit of this character is the exclusive-or of the least significant bits of all the previous characters, including the $B, $D and $F characters, so this parity is even, and the second least significant bit of this character is the exclusive-OR of the second least significant bits of all the previous characters, including the $B, $D and $F characters

16.3.2 Hardware Description

The required hardware description of the magnetic card code system is shown in figure 16.3. The reader has four wires in a cable. The black wire is +5 v and the black-with-white stripe wire is ground. The red wire is the clock (watch this!) and the white wire is data. The parallel port strobe A function of the MC68HC11A8 is used to detect the clock transitions, and should cause an interrupt on the rising edge. A port C data input such as bit 0 will be used to read the data bit at the time that that edge occurs.

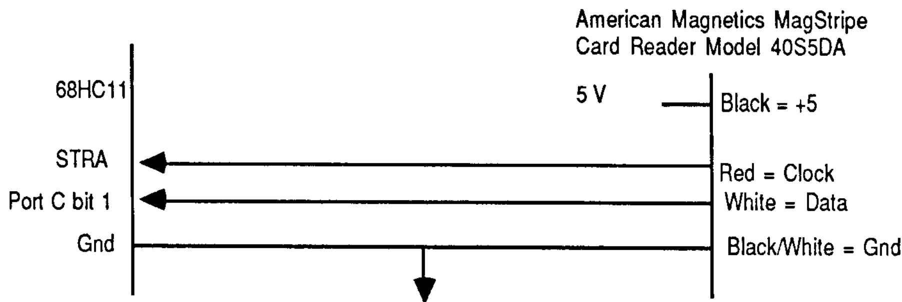


Figure 16.3. Magnetic card code reader to the MC68HC11A8 interface

16.3.3 Software Design Considerations

There are several design issues in implementing a robust magnetic card code system, one of which stems from the fact that the timing of the bits seen by the system varies with the scan velocity of the card through the reader. A magnetic card scanned at slow speed will input data bits slower than when scanned at a faster speed. The scan velocity varies from sweep to sweep, which suggests that a design using a predetermined time to sample data will be of limited use. It will impose a narrow range for the scan velocity. What is needed is a "self-tuning" system in which changes in scan velocity are accounted for. The following scheme is suggested

1. Use interrupts to pick up a bit each time the clock signal rises.
2. Shift bits into a memory word. When the pattern $B (%11010) appears, set up a counter to count bits modulo 5.

3. After each 5 bits arrive, check the parity bit for odd parity, and if all is well, store the 4-bit BCD code in a buffer and exclusive-or the 5-bit pattern into a memory word, otherwise record an error.
4. After the stop pattern $F (%11111) arrives, pick up exactly one more word, and then compute the even parity across the bit positions verified by the message parity character.
5. If any errors occur, put out a message "Bad Read", otherwise convert the data stored in the buffer to ASCII and print it out, followed by a carriage return.

16.4 Procedure

16.4.1 Standard Part

Implement the magnetic card code reader, which displays the digits on the terminal connected to the 6811. Do not verify the parity checks. Read some credit cards (preferably expired cards).

16.4.2 Optional Part

Verify the parity of the encoded number.

16.4.3 Extra Credit Part

Implement a credit system with a timer and at least two cards. After reading a card, the user will enter an amount (in cents only) on the terminal connected to the 6811. This amount will be added to the total for that card, which is kept in memory. Periodically (every 2 minutes, say) a list of account values will be printed on the terminal. Interest of 12.5% will be added on the outstanding amount of each account at this time. The user pays his or her bill by entering a negative amount.

16.5 Hints and Suggestions

See also section 15.5.2, Debugging Interrupt-Driven Software, for suggestions for this experiment.

16.5.1 Getting the Codes

You may wish to temporarily write a program to display the bits as they are read from the card reader input. This program is useful for verifying that the reader is functioning.

17 The Keyboard and LED Display

17.1 Goals

1. To introduce the techniques of keyboard interfacing
2. To introduce the techniques of controlling 7-segment LED displays

17.2 Introduction

Today, the most prevalent input human-interface in all computer systems is the keyboard. Although much research is being conducted to find "better" ways to communicate with computers, it is unlikely that a new input technique will emerge to replace keyboards in the near future. Although it is true that there already exist systems with speech interface mechanisms, their lack of robustness, limited vocabulary and flexibility, and high cost of current speech recognition techniques restrict their uses. Rather than being replacements for keyboard input to computers, speech and other techniques will serve as complements to keyboards in providing a more user-friendly interface. Track balls, light pens, digitizer tablets, and mice are other such devices. It is only fitting that we study keyboard interfacing techniques in this experiment.

In many microcomputer applications, a full-blown display device is not necessary because it is sufficient to output only a small amount of data at one time. Calculators and various measurement devices, such as thermometers, timers, and voltmeters, fall into this category. For these devices, a panel of Light Emitting Diodes (LEDs) is all that is needed. LEDs are small, rugged, easy to control, and most of all, cheap. For these reasons, they are also used in process-control devices as status indicators. In this experiment, we present the techniques to control 7-segment LED displays. As a whole, this experiment may be viewed as building a dumb terminal with a LED, rather than a CRT, display.

We recommend that you review section 6-4 of *Single- and Multiple-Chip Microcomputer Interfacing.*

17.3 Description

17.3.1 Matrix-Keyboard Sensing

On a matrix keyboard, each key is a momentary-contact switch positioned at an intersection of a row and a column of sensing wires. When a key is pressed (hence closing the switch), contact is made between the two wires. So to find the key being pressed is to find the row and the column that are shorted to each other. This is done by

scanning each row and column. Actually, each column sensing wire is tied to a logic high through a pull-up resistor. Then one row is set to a logic low, and the rest set to a high, and each column is tested to see if one line in it is a logic low. Since the wire-or connection is used at the intersections, only the key(s) whose row is set to a logic low will produce low on its column sensing wire. The keyboard scanning can be done continuously or only once when the pressed key generates an interrupt.

To illustrate the key sensing and mapping techniques without regard to a specific keyboard or keyboard pattern, a virtual keyboard is presented first. We will denote the first key being scanned for a closure as key 1, the second key as key 2, and so on. A lookup table can be used to map the virtual keys to the actual keyboard patterns with ease. In this way, one key-sensing program can be used to decode different keyboards with very little modification. In fact, by changing the contents of the keyboard map, a user-definable "soft key" feature can be implemented. The diagram of an eight by eight virtual keyboard is shown in figure 17.1.

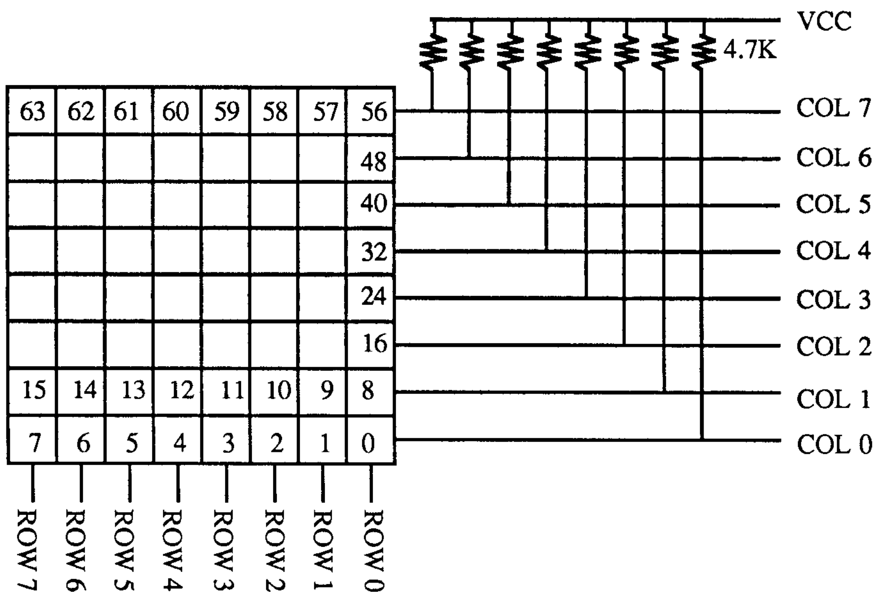


Figure 17.1. A 64-key virtual keyboard

A key scanning technique is expressed as a function as follows:

```
scan ( )                /* row and column number */
{  for ( col = 0 ; col  < 8 ; col++ )       /* for each column */
      for ( row = 0 ; row < 8 ; row++)  /* for each row */
         if ( key_matrix ( row, col ) == low )       /* if a virtual key is pressed, */
      return ( key_map ( row, col ) );      /* return the real character from key map */
   return ( null );       /* no key is pressed */
}
```

This routine scans the keyboard until either a key closure is detected or it fails to detect one. This technique works well for detecting a single key, but in many instances, it is necessary to detect two (or more) keys simultaneously being pressed such as with the shift- and control-key patterns. To detect such keys, the SHIFT, CAPS LOCK and CONTROL keys are tested separately from other keys. The other keys are scanned as discussed above.

17.3.2 The Auto-Repeat Feature

Auto-repeat is a convenient feature that enables the user to enter more than one pressing of a key by holding a key pressed for a longer period. This feature can be implemented by checking to see, once a key is pressed, if the same key continues to be pressed for, say, a half second. A shorter delay, say 0.1 second, can be used before reporting the next auto-key closure after the last key closure was reported.

17.3.3 Key Debouncing

Each keyboard has different key bouncing characteristics. For most keyboards, the key bounce is in the order of milliseconds. A logic analyzer can be used to determine the duration of the key bounce. Ten milliseconds is a conservative estimate for key bounce.

17.3.4 Interrupt-Driven Keyboard Sensing

To free the microprocessor from continuously scanning the keyboard, an interrupt-driven key sensing technique can be used, for an average person cannot type more than a few keystrokes per second in sporadic bursts. The column sensing wires can be connected to a NAND gate to generate an interrupt (logic high state) whenever a key(s) is being pressed. The keyboard is scanned only once after an interrupt is generated. However, in order to detect the shift- and control-key patterns, the SHIFT and CONTROL keys should not be allowed to generate an interrupt. Otherwise, the second key in the shift- and control-key pattern will not trigger an interrupt, and cannot be detected. In the interrupt-driven approach, the key debouncing can be done more practically in hardware than in software. It makes little sense to use the "wait-and-see" technique in the interrupt handler routine.

17.3.5 7-Segment LED Display Control

The most common 7-segment common-cathode LED display comes in a 14-pin DIP, whose logic and pin-out diagram is shown in figure 17.2

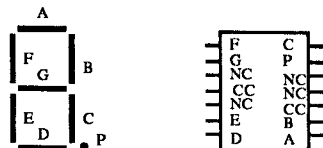


Figure 17.2. A common-cathode 7-Segment LED

The pins labeled A to G denote the segments A to G, P denotes the decimal point, NC denotes no connection, and CC denotes the common cathode. When there is a voltage difference of about 1.7 volts across the anode (+) and the cathode (-) terminals of a LED segment, the segment is turned on. The current requirement is on the order of tens of milliamperes, and more current will brighten the segment. Because of the high current requirement, a current driver is normally required to control the anodes. The cathodes of the segments are connected together (hence the term common cathode), so the sum of the currents flowing through the segments that are turned on flows through this pin. On some packages, more than one pin is used for the common cathode. Usually an NPN transistor or a peripheral driver is used to control the cathode (on/off control) because of its high current sink capability.

To control more than one 7-segment LED display, a multiplexing scheme is used. A control line is connected to the anodes of one segment in each of the LEDs being multiplexed. The cathode of each device is controlled separately to turn on one at a time. Since most human eyes do not detect changes faster than 24 Hz, each segment only needs to be turned on 24 times a second to give the appearance of being turned on continuously.

17.4 Procedure

17.4.1 Standard Part

Implement a keyboard decoder using the continuous scan and wait-and-see debounce techniques. Connect port B to the rows and port C to the columns. The CAPS LOCK, SHIFT, and auto-repeat features are desired. Also implement a 4-digit LED display to display decimal numbers entered from the keyboard. Use the Serial Peripheral Interface function to control the MC14499. Use a 2N2222 or any NPN transistor with collector current rating greater than 300 milliamperes. The logic diagram is shown in figure 17.3.

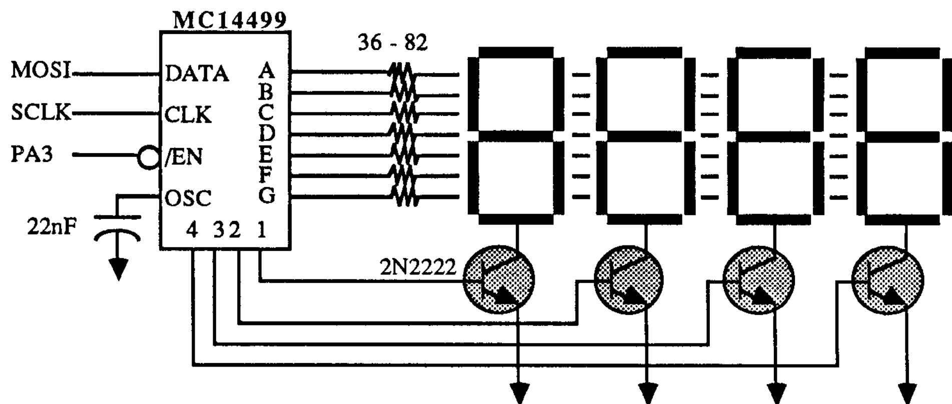


Figure 17.3. Logic diagram of the MC14499 control of a 4-Digit LED display

Configure CPOL=1 and CPHA=0 in the SPI and use a shift clock frequency of the memory clock frequency divided by 16 or 32. The data from MOSI is latched to the shift register in the MC14499 on the falling edges of SCLK, the serial clock, while /EN = low. The data in the shift register is transferred to the output latch on the rising edge of /EN. MC14499 requires 16 clocks to receive 16 bits, each 4 bits representing the binary value to be displayed. For example, if the pattern $2359 is sent, most significant bit first, then 2359 will be displayed from left to right on the 4-digit displays. Initially, $ffff must be sent to the MC14499 to blank the displays. Note that SPI sends eight bits at a time. Since the display must be updated after each new digit is entered, it is easier to maintain a four digit buffer and send the previous three digits before sending the new digit.

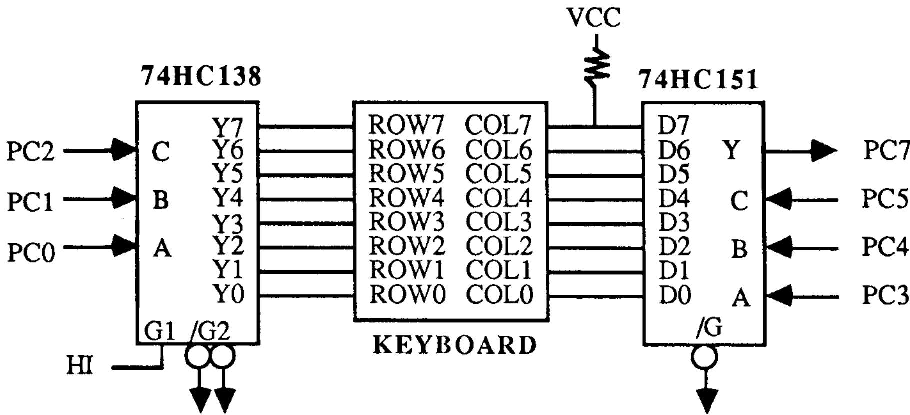


Figure 17.4. Multiplexer-decoder logic for keyboard scanning

17.4.2 Optional Part

Modify the standard part to use the interrupt driven-keyboard scan technique. The column sensors can be connected to a NAND gate (74HC30) to generate a rising edge when a key is pressed. Use an Input Capture Module such as IC1 to detect this rising edge and generate an interrupt request. To simplify the task of detecting two-key sequence characters such as SHIFT and CONTROL keys, connect the SHIFT and CONTROL keys to a separate Input Capture Module. An input capture module can be configured to generate an interrupt on both the rising and the falling edges to detect when they are pressed and when they are released. Boolean flags can be used to indicate their status. A similar step is needed to handle the CAPS LOCK key. A hardware or a software debouncing technique may be used. However, if a software debouncing is used, the interrupt should not be cleared until the debouncing is done.

17.4.3 Extra Credit Part

Use only port C with a multiplexer and decoder to interface the keyboard. A suggested logic diagram is shown in figure 17.4.

Note that each column sensing wire must be tied high with a pull-up resistor. A 74HC152 may be used instead of a 74HC151. Notice that if a decimal value 21 is output through PC5 to PC0, the status of the twenty second key in the virtual keyboard is seen at PC7. PC6 may be configured as a grounded input to simplify the software.

Use the real-time interrupt feature of the MC68HC11A8 to control the 4-digit 7-segment LED displays. Use an MC75491 to control (supply current to) the anode of a segment and a device in the MC75492 or a 2N2222 transistor to control the common cathode. A current limiting resistor should be placed between the Vcc and the collector input of the MC75491 to supply about 20 milliamperes to a LED segment. The value of this resistor is calculated as follows:

$$R = (V_{SS} - V_{CE(491\ ON)} - V_{OL(492)} - V_F) \div I_F$$
$$= (5.0 - 0.8 - 0.5 - 1.7) \div 0.02 \qquad = 100 \text{ ohms}$$

The current limiting resistor may be adjusted to control the brightness of the LEDs. Note that the MC75492 (or its substitute) must be capable of sinking at least 140 (7 x 20) milliamperes when all segments in a LED is turned on. The emitter of the MC75491 is connected to the anode of the LEDs. A suggested logic diagram is shown in figure 17.6.

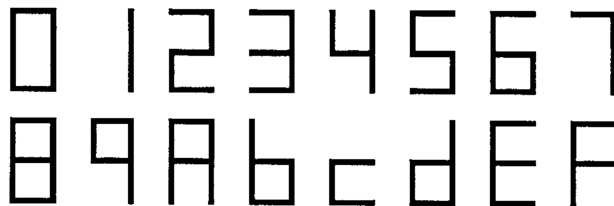


Figure 17.5. Design of hexadecimal digits on 7-Segment LED display

Modify the display routine to display hexadecimal numbers, whose design is shown in figure 17.5.

A further extra-credit experiment is to use the Liquid Crystal Display (LCD) in place of the LED. LCDs are particularly suited to low power applications because they are essentially capacitors rather than diodes, and they interfere with the transmission of light rather than generating it. Use the MC145000 Serial Input Multiplexed LCD Driver and a 6-digit 7-segment LCD multiplexed display with 4 backplanes, such as the General Electric LX69D3F09KG described in section 6-4.3 of *Single and Multiple-Chip Microcomputer Interfacing*. However, we have used the MC145000 with LCD displays from broken calculators, which had 8 digits and 3 backplanes, with success.

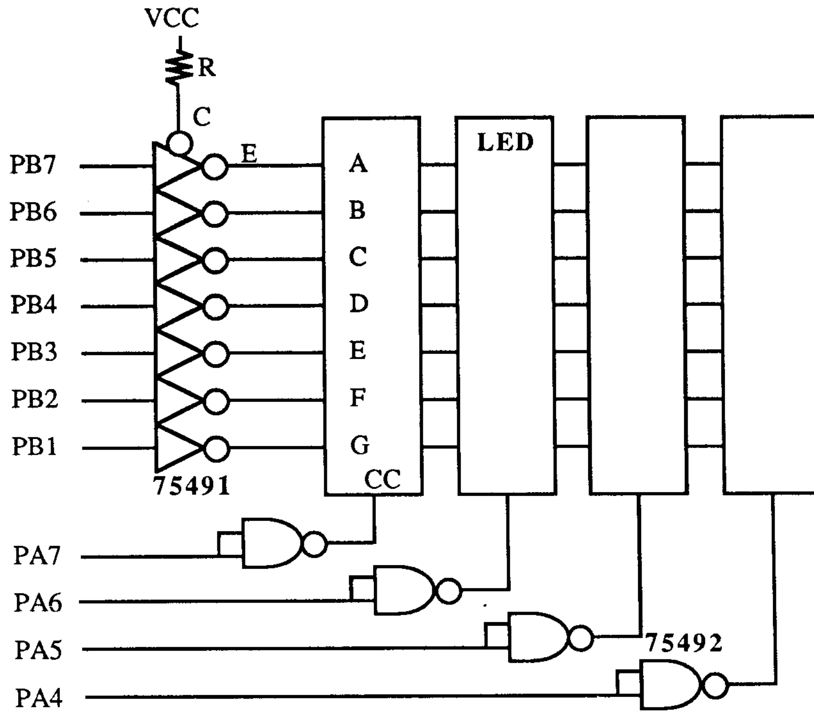


Figure 17.6. Hardware for software multiplexing of a 4-Digit LED display

18 A DC and RMS Digital Voltmeter

18.1 Goals

1. To implement a digital voltmeter using the MC68HC11A8 analog-to-digital Converter
2. To introduce the numerical concepts necessary to perform fixed-point arithmetic using integer arithmetic instructions
3. To demonstrate an effective algorithm for computing the square root of an integer value using only integer subtract operations.

18.2 Introduction

This is a simple experiment designed to introduce the MC68HC11A8 analog-to-digital Converter System. Since an analog-to-digital converter translates from an analog voltage to a digital value, a logical first experiment is the design of a digital voltmeter.

18.3 Description

This experiment builds off material covered in Section 6-5.2 of *Single and Multiple-Chip Microcomputer Interfacing.*

18.3.1 Analog-to-Digital Conversion Fundamentals

The first step in designing an analog measurement system is choosing an appropriate transducer. The transducer's job is to convert a physical signal such as pressure, temperature, or light intensity into an electrically measurable quantity, typically a resistance or voltage change. Once an electrical parameter can be varied, it is possible, through a number of means to obtain a voltage signal that varies over a certain range. From this varying voltage, one can produce a digitized sample using an analog-to-digital converter ADC.

The most common type of analog-to-digital converter available today is based on a technique known as successive approximation. One common aspect of most successive approximation analog-to-digital converters is that they must contain a digital-to-analog converter as an integral part. The successive approximation technique (which is used in the MC68HC11A8) relies on a digital-to-analog converter, an analog comparator, and a simple controller to successively test different binary values against the input analog signal. Frequently, another component, called an analog sample and hold, is used to provide a constant input during the analog-to-digital conversion time.

A digital-to-analog converter is commonly constructed in a manner similar to the circuit below (figure 18.1). This circuit is known as an R-2R ladder network, and as shown, it implements a 4 bit binary to analog converter.

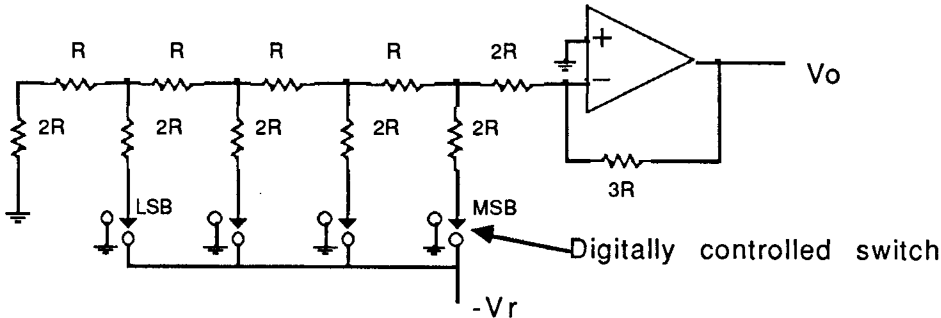


Figure 18.1 A Ladder Digital-to-Analog Converter

Successive approximation is a binary search algorithm. The following fragment of C-like pseudo code illustrates the algorithm.

```
digital AnalogToDigital (actual)  analog actual;
	{
	digital sample;		/* digital version of sample */
	analog guess; /* current value of sample converted to analog */
	sample = 0;
	for (bit = MSB; bit >= LSB; bit--) {
		sample += (1<<i); /* Add weight of current digit */
		guess = DigitalToAnalog (sample);
		if (guess > actual)  /* Analog comparison */
			sample -= (1<<i);  /* Too high, try lower */
	}
	return (sample);
}
```

Two important considerations in selecting an analog-to-digital converter are the converter's accuracy and its conversion time. A fundamental rule of digital signal processing states that the sampling rate must be at least twice the frequency of the signal being measured (the Nyquist Criterion). Numerical techniques also suffer from round off errors. Thus, an ideal analog-to-digital conversion would have infinite precision (number of bits) in order to minimize rounding errors later and very fast conversion time. Clearly, using the successive approximation technique, these are conflicting goals. Typical ADC accuracy is 8-16 bits, and conversion times range from a few tens of nanoseconds to hundreds of microseconds. The analog-to-digital converter in the MC68HC11A8, with a 2 MHz memory clock rate, can compute an 8-bit analog-to-digital conversion in 16 microseconds.

18.3.2 Voltage Sensing Fundamentals

The circuit shown below (figure 18.2) implements a fairly typical level shifting function. When properly adjusted, input voltages on V_{in} between -2.5 and +2.5 volts will result in output voltages between 0 and 5 volts on V_{out}. Such level shifting is frequently necessary prior to analog-to-digital conversion because ratiometric analog to digital converters like the one in the MC68HC11 cannot handle negative input voltages. The Zener diode serves to protect the MC68HC11 from overvoltage caused by carelessness.

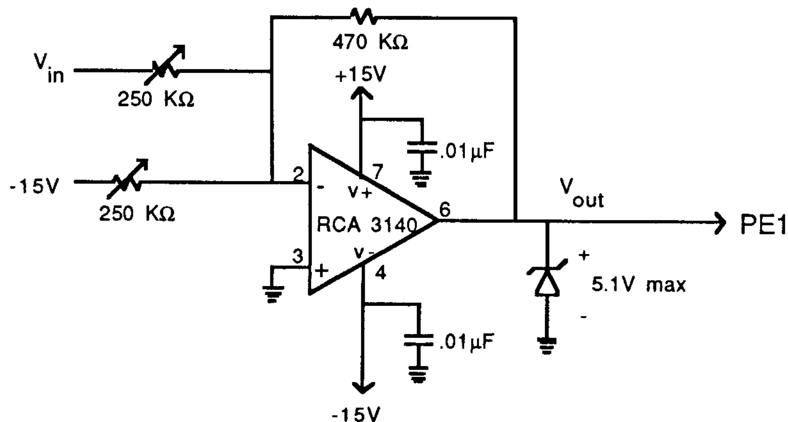


Figure 18.2 A Level Shifter

This circuit can be used as the basis for a digital DC voltmeter over the range -2.5 to +2.5 volts. Given the 8-bit analog-to-digital converter of the MC68HC11, this will provide a value with a maximum error of approximately 0.02V.

18.3.3 AC Voltage Sensing

The most useful voltage measurement of an AC signal is the Root Mean Square (RMS) value. However, many analog voltmeters actually measure the peak voltage and scale the result by a factor of √2. This is only an approximation of the RMS voltage, and it isn't even a good approximation unless the input is sinusoidal. With the analog-to-digital conversion hardware of the MC68HC11, we can construct a true RMS voltmeter using the circuit above and a little bit of software.

The RMS value of a signal is defined by the relationship below:

$$V_{RMS} = \sqrt{\frac{1}{T}\int_0^T V(t)^2\,dt} \approx \sqrt{\frac{1}{N}\sum_{i=0}^{N} V_i^2}$$

It is important when computing the RMS voltage that the samples be equally spaced over the period. And obviously, the more samples that can be collected during the period the better the result will be. For our applications, 32 samples will be adequate. Because each conversion takes 32 E clock cycles, the highest theoretical frequency we will be able to sample is just under 2 KHz However, in order to achieve this rate, it will be necessary to perform the RMS conversion in stages.

1. Determine the period of the waveform by performing A/D conversions to find the time between zero crossings
2. Use one of the timer output comparison functions to generate the periodic interrupts that you will need to accurately time your analog-to-digital conversions
3. As each interrupt occurs, reset the timer output compare function, read the A/D result, and start another A/D conversion. Store your 32 samples in a buffer
4. After collecting 32 samples, perform the RMS calculation

Using the level shifting circuit above will cause the values returned by the analog-to-digital converter to be shifted by 128 from their normal values. Thus, before squaring each sample, you must first correct this bias and generate the absolute value for the unsigned multiply instruction. The code sequence below demonstrates the proper technique.

```
        LDAA    0,X     Get A/D Result from buffer
        BMI     L1      If positive, go to L1
        NEGA            Get absolute value
L1      ANDA    #$7F    Remove sign bit = 1
        TAB             Get ready to square
        MUL             Multiply unsigned numbers
```

After executing that code sequence, the squared sample is in accumulator D and occupies 15 bits (128*128 = 16384 = $4000). Thus, 32 squared samples sum to a result that might need 20 bits to represent requiring that you perform the accumulation using extended precision arithmetic. You will need the ADC instruction for the high byte.

18.3.4 Computing Square Roots Using Integer Arithmetic

Once you have determined the sum of the squares of your samples, you must take the square root. The MC68HC11 has an integer (and fractional) divide instruction, but square root is not provided. Fortunately, there exists an algorithm for determining an approximate square root that, while requiring a fairly long time, is easily programmed using only integer arithmetic instructions. It is based on the following observation:

$$(n+1)^2 = n^2 + 2n + 1$$

which can be recursively applied to yield:

$$n^2 = \Sigma\, 2i + 1,\ i = 0 \text{ to } n-1$$

This yields the algorithm:

```
int square_root (n)
long n;
{
      int         root = 0;
      int         odd = 1;
      while (n >= odd)  {
                  n -= odd;
                  odd += 2;
                  root++;
      }
      return  (root);
}
```

18.4 Procedure

18.4.1 Standard Part

Construct the level shifter circuit of figure 18.2 and trim it to properly shift an input voltages between -2.5 and +2.5 V to 0 to 5 V. After verifying that the circuit operates properly, connect it to on of the A/D inputs of the MC68HC11. If you are using an EVB board for this experiment, be aware that the Buffalo monitor uses Port E bit 0 at reset to decide whether to execute out of ROM or EEPROM; you will want to use another input.

Build the LED display in Experiment 17. Write software to implement a DC voltmeter using your sampling and display hardware. You should display the voltage to two decimal places, and use the decimal point of the one's digit to represent negative voltages. Compare the accuracy of your digital voltmeter to a commercial voltmeter. Display continuously, but provide a time delay that avoids annoying flickering of the least significant digit.

18.4.2 Optional Part

Construct the circuit as above, but implement an AC RMS voltmeter using the algorithms described above. Test your results using small AC signals between 40 and 400 Hz Your results should compare well with a digital voltmeter for sinusoidal inputs, but depending on the digital voltmeter, you may get differences for square and sine waves. However, if you perform the integration yourself, your results should compare quite well. (For a square wave, $V_{RMS} = V_{PEAK}$, while for a triangle wave, $V_{RMS} = V_{PEAK}/2$).

18.4.3 Extra Credit

Construct several copies of the level shifter circuit and adjust them to provide various levels of sensitivity. For instance, have one channel that adjusts ±0.25 V to the 0 to 5V range, another that measures ±25 V, and the original one that measures ±2.5V. Make sure you have 5.1V Zener diodes on the outputs of these circuits! Rewrite your programs from the standard and optional parts to construct an auto-ranging voltmeter which checks the highest range first and selects successively smaller ranges until it finds the appropriate range. Use a switch connected to a pin of port C to select DC and RMS operation.

18.5 Hints and Suggestions

You will need to pay careful attention to the scaling of the values while you are calculating the RMS voltage for the AC meter. Remember that your sum is potentially a 20 bit number. Thus, its square-root is potentially a 10 bit number, which means that you will need to perform a lot of multiple-precision arithmetic.

Avoid the temptation to divide the square root by 32 immediately, because you will throw away very nearly half of your accuracy when you do. Instead, think of the result of your square root as representing the number of 1/32s of a volt. To properly determine the hundredths digit without losing accuracy, first multiply by 100 and then divide by 32. This will need to be done in multiple-precision as well since the intermediate result may require 17 bits. This number can then be converted to decimal and displayed.

19 A Thermometer

19.1 Goals

1. To implement a simple temperature sensing system using the MC68HC11 analog-to-digital conversion subsystem
2. To explore the the use of piecewise linear and polynomial spline models as techniques for correcting nonlinearities of a physical system

19.2 Introduction

In many applications of single-chip microcomputers as system controllers, it is important to measure some real-world external stimulus and to respond according to this measurement. For instance, in an automotive fuel injection system it might be important to determine such quantities as engine vacuum level, throttle position, engine speed, etc. and from these quantities adjust the fuel to air ratio and ignition timing to improve performance, fuel economy, and emissions.

Some of these quantities such as throttle position and engine speed can be obtained by purely digital means such as grey-code encoders and pulse counters. However, quantities like fuel pressure, vacuum level, and many others are inherently analog signals. Before a microcomputer can use these signals to perform any internal operations, they must somehow be converted into digital information. One of the strongest features of the MC68HC11 single chip microcomputer is it outstanding on-chip analog-to-digital conversion system.

19.3 Description

This experiment expands on material in section 6-5.2 of *Single and Multiple-Chip Microcomputer Interfacing.*

19.3.1 Thermistors

When the analog signal that needs to be measured is temperature, a cost-effective transducer is often the thermistor. There are a number of chemical substances which exhibit a change in their electrical resistance over a temperature range. Figure 19.1 shows a typical relationship between thermistor resistance and temperature.

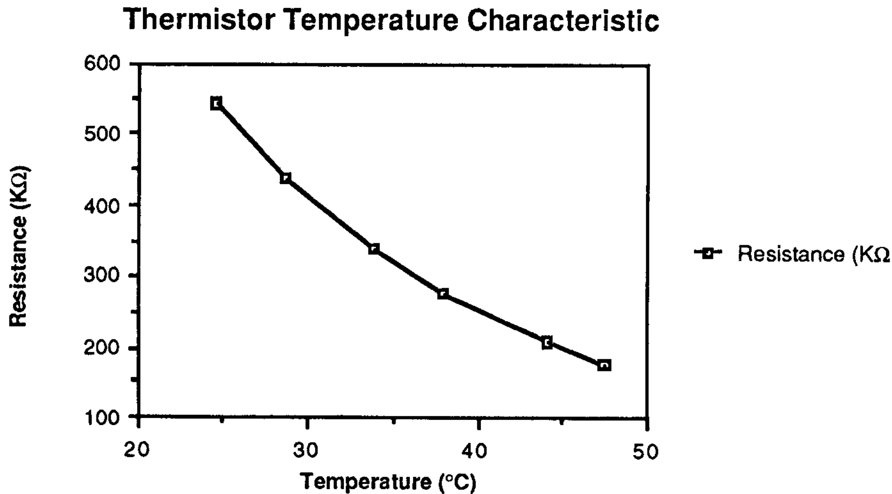


Figure 19.1 Temperature-voltage nonlinearity of a thermistor

This thermistor exhibits the most usual type of behavior: the resistance decreases with increasing temperature in an inverse logarithmic fashion over a fairly wide range of temperatures. This is known as a negative temperature coefficient. Many thermistors have this logarithmic behavior, and this can cause some problems with the accuracy of the analog-to-digital conversion, since at higher temperatures there is a smaller change in resistance per change of temperature. An ideal transducer is usually one that is linear, however, in many instances, software techniques can be used to counter the transducer nonlinearity.

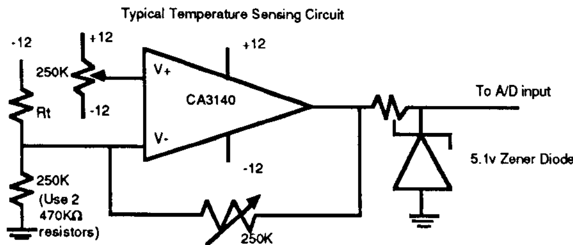


Figure 19.2 A Temperature Sensing Circuit

19.3.2 Temperature Sensing

A typical temperature sensing circuit is shown in figure 19.2. There are many ways that a thermistor can be used to control an analog circuit. The design was chosen for its simplicity. The potentiometer on the non-inverting input of the op-amp is used to adjust the zero point of the circuit. The potentiometer in the feedback path adjusts the gain of the circuit, and thus its sensitivity. These two adjustments are not mutually independent, and proper adjustment will require a divide and conquer approach of first adjusting the zero point and then the gain and back and forth.

19.4 Procedure

The particular thermistor we used in this circuit had exactly the characteristic shown in figure 19.1. If you cannot obtain a similar thermistor, adjust the other component values proportionately. Try to keep the current which flows through the thermistor to a minimum since any resistive heating will affect the accuracy of your temperature measurements.

19.4.1 Standard Part

Build the simple temperature sensing circuit above and adjust it so that it reads approximately 0 to .5 volts (positive) in freezing water and about 4.5 to 5 volts at some known higher temperature, perhaps 98.6°F, human body temperature. You will find this adjustment much easier to accomplish if you use good quality 10 turn potentiometers. You will need to protect the thermistor with plastic film or ideally with thermal epoxy resin before dunking it in the ice bath. Record the two voltages and temperatures.

Connect this circuit to one of the analog-to-digital inputs of the MC68HC11. If you are performing the experiment using an EVB board, you will not be able to use PE0, since that pin is used by the Buffalo ROM to select either EEPROM or ROM boot-up. Tie the V_{RL} pin to ground and the V_{RH} pin to +5. Build a three segment LED display using the SPI interface and the MC14499 driver as outlined in experiment 17. Program the 6811 to act as a digital thermometer assuming a linear thermistor circuit response. You should use the A/D converter's SCAN=0 single channel mode and average the four readings that you obtain to filter out any electrical noise.

19.4.2 Optional Part

With a thermometer and a water bath, obtain a few more temperature/voltage pairs for the temperature sensing circuit perhaps at every 10° between 40° and 120°F. Store these points in a table and rewrite your digital thermometer to model the system with piece-wise linear approximation. Compute each approximation as you need it, do not precompute them and store them in a table. Compare the performance of your digital thermometer to actual over the temperature range. Every five seconds (use one of the timer output compare interrupts), switch display modes from Fahrenheit to Celsius like a bank thermometer. Also, keep track of the decimal point properly so that Celsius temperatures are displayed as **dd.d** and Fahrenheit temperatures are displayed as **dd.d** when below 100 °F and **ddd** when above 100 °F (where **d** is a decimal digit).

19.4.3 Extra Credit Part

Using the temperature-voltage information gathered for the optional part, rewrite the digital thermometer software so that a polynomial spline model is used instead. As before, you must do the computation on the fly, not by table lookup. Provide a comparison of linear versus piecewise linear versus cubic spline model accuracy.

19.4.4 Semester Project

Expand this project into a full featured programmable thermostat for a home heating/cooling system. This will require combining most of the techniques explored in this lab with experiment 17, the keyboard-display and experiment 20, the alarm clock. Provide alternating time of day and temperature display, as well as the ability to program several different temperatures throughout the day, perhaps with weekday/weekend scheduling. You might also try to put the MC68HC11A2 chip as discussed in chapter 23.

20 An Alarm Clock

20.1 Goals

1. To introduce the techniques of using timer/counter modules to build an alarm clock
2. To introduce the techniques of sound synthesis to generate melodies

20.2 Introduction

There probably is at least one digital alarm clock in every household that we know. It is a simple device that we have come to depend on as our social interactions increase. Imagine how inconvenient life in modern society would be without alarm clocks.

The heart of the digital clock is the timer/counter module. It is used to generate periodic events to update the time and sometimes sound the alarm. In many digital clocks, the events are set to occur every minute. In others, they are set to occur more frequently, in which case a counter is used to count up to a minute. Every time the time is advanced to the next minute, the alarm setting is compared with the current time. If they match, an alarm is sounded.

Of course, every alarm clock has means to set the time and alarm. In addition, it may have the option of rudely waking everyone in the room with a blasting buzz or, more gently, with a pleasant melody or music from radio. It may also have a snooze button, which when pressed, momentarily turns off the alarm. In this experiment, we will build a digital alarm clock with all the features mentioned above.

A couple of years ago, musical greeting cards were introduced, and they were an instant success. What made them possible was the use of miniature piezo transducers that have a bandwidth of 1 to 3 KHz. Piezo transducers require little current, in the order of tens of milliamperes, to drive them, which make them ideal for use with CMOS devices. For generating melodies in this experiment, we suggest that you use a piezo transducer available from Radio Shack (Cat. 273-073) for under a dollar.

We recommend that you review section 7-2 of *Single- and Multiple-Chip Microcomputer Interfacing.*

20.3 Description

A very simplified theory of sound is presented below for generating melodies. Those interested in music synthesis are referred to *Musical Applications of Microprocessors,* by Hal Chamberlin.

Sounds are the effect of mechanically induced vibrations in the air. Steady, unchanging sounds can be described by a number of parameters (frequency, amplitude, and harmonic content) that are also steady. Changing sounds can be similarly described

by these same parameters changing with time. For the following presentation, assume a pure sine wave, the simplest possible pitched sound.

20.3.1 Frequency and Pitch

If a sine wave is repeated every millisecond, its period is 1 millisecond, and its frequency is 1 kHz. The frequency parameter determines the perceived pitch of the tone. Whereas frequency is a physical parameter of the sound waveform, pitch is a subjective measure that exists only in the mind of the listener. When frequency is increased, the pitch is also perceived to be increased, provided that the frequency is within the audible range. For the human ear, this is about 20 to 20 kHz. However, the relation between pitch and frequency is not linear. For example, an increase of frequency from 100 to 200 Hz would be perceived as a doubling of pitch, but the increase from 10.0 to 10.1 kHz is nearly imperceptible. The listening tests have shown that the relation between the frequency and pitch is somewhat exponential.

Table 20.1 Pitch scale

NOTE	FREQ (Hz)	INDEX COUNT (@2 MHz)	
A4	440		
A5	880		
D#5	1244	1	1608
E5	1318	2	1517
F5	1397	3	1432
F#5	1480	4	1351
G5	1568	5	1276
G#5	1661	6	1204
A6	1760	7	1136
A#6	1865	8	1072
B6	1976	9	1012
C6	2093	10	956
C#6	2217	11	902
D6	2349	12	851
D#6	2489	13	803
E6	2637	14	758
F6	2794	15	716
F#6	2960	16	676
G6	3136	17	638
G#6	3322	18	602

Musical pitch is measured relatively rather than absolutely. For example, if tone B is twice as high as tone A, the frequency of B is one *octave* higher than that of A, and the perceived pitch is twice as high. The octave is the fundamental unit of measure of pitch. Another unit is *half step* , which is one twelfth of an octave, or a frequency ratio of 1.05946. A half step is also the difference in pitch between any two adjacent keys on

a conventionally tuned (equal temperament tuning) piano. Since the pitch units are relative, a basis point is needed to define an absolute pitch scale. One such basis is the international pitch standard, which defines the note A4 (usually referred to as A above middle C) as being 440.0 Hz. A portion of the pitch scale is shown in table 20.1.

20.3.2 Amplitude

The amplitude parameter is related to the height of the sound wave. In the air, it is related to the degree of the change in air pressure. The human ear is capable of responding to a very wide range of sound amplitude. The amount of sound power at 2 kHz that can be listened to without undue discomfort is about a trillion (10^{12}) times greater than the power in a barely audible sound. For convenience in working with such a wide range of power, the *bel* scale of sound intensity was developed. Like musical pitch units, the bel scale is also relative. The bel unit refers to a ratio of 10 between the power of two sounds. Thus, sound B contains 1.0 bel more power than sound A if it is 10 times as powerful. Since power increases with the square of voltage, a 10:1 ratio of voltage is equivalent to a 100:1 ratio in power, or 2 bel. In actual audio work, the unit decibel (abbreviated as dB) is more commonly used.

20.3.3 Harmonic Content

In the 17th century, the French mathematician Joseph Fourier proved mathematically that any waveform is actually a mixture of sine waves of different frequencies, amplitudes, and phases. Furthermore, he showed that if the waveform repeats steadily, the frequencies of the component sine waves are restricted to being integer multiples of the repetition frequency (the fundamental frequency) of the waveform. These component sine waves are called *harmonics* , or overtones. The harmonic content of a wave, which determines its shape, influences the overall quality of a tone, known as *timbre*.

The significance of Fourier's theorem can be realized by noting that all the acoustically important aspects of the shape of a waveform can be specified with a comparatively small number of parameters. For instance, a 1 kHz waveform, no matter how complicated, can be specified by 20 amplitudes and 20 phase angles corresponding to the 20 audible harmonics, because the upper limit of the audio range is around 20 KHz. However, the human ear is somewhat insensitive to tones of moderate frequency and amplitude, according to most audio specialists. As a result, phase can usually be ignored when synthesizing sound waveforms. However, recent advances in loudspeaker design have demonstrated that maintaining phase information maintains the sense of depth in stereo sound reproduction.

20.3.4 Duration

To make music out of sounds, another parameter is required, and that is the duration of a pitch. This is also a relative quantity. A few of the duration symbols are shown in figure 20.1.

Figure 20.1. Pitch duration symbols

If a musical composition is specified with 120 = one quarter note, then the speed of the music should be such that there are 120 quarter notes in one minute, or a quarter note should last one half of a second.

20.4 Procedure

20.4.1 Standard Part

A block diagram of the digital alarm clock to be built is shown in figure 20.2.

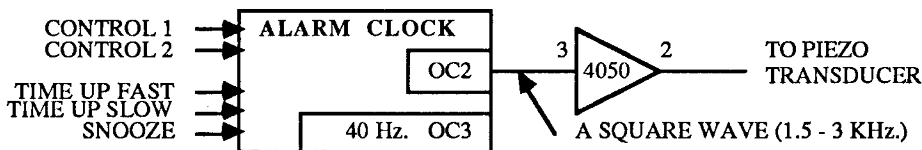


Figure 20.2. Block diagram of the digital alarm clock

The control two signals, set the alarm clock to one of four operating modes: set-time, set-alarm, alarm-on, and alarm-off. To set the time (or the alarm), the clock is set to the set-time (or set-alarm) mode, and the TIME UP FAST or the TIME UP SLOW switch is used to advance the time. If the clock is set to the alarm-on mode, the alarm should go off at the right time until it is turned off (alarm-off mode) or the SNOOZE button is pressed. To sound an alarm, generate a continuous square wave of 2 KHz to drive the piezo transducer using the output compare 2 module in interrupt-driven mode. For the experiment, set the time to advance 1 minute/second with the TIME UP SLOW button and 10 minutes/second with the TIME UP FAST button. Also set the snooze alarm to go off every 5 seconds. To display the time on the screen, backspace characters may be used to erase and overwrite the old time. A 24-hour format may be used, as in HOUR:MIN:SEC. Alternatively, the 4-digit 7-segment LED displays from the previous experiment may be used to display the time (HOUR:MIN or MIN:SEC only).

Note that when the operating mode is changed to set-alarm from the other modes, the current setting of the alarm should be displayed. In all other modes, the current time should be displayed. The signals input to the clock may be tied to a pull-up resistor to

simulate a normally open (logic high) switch. For instance, to simulate the pressing of the snooze button, the SNOOZE signal can be momentarily shorted to the ground. At other times, the signal is left unconnected to float to a logic high. Be sure to debounce all the switches, either in hardware or in software.

The 4050 is a CMOS driver, so you should handle it with some care. CMOS devices are easily damaged by static charges, so try to eliminate static build-up in your hand by touching a grounded material before picking up the device. **Also note that the Vdd is not pin 16 and that Vss is not pin 8.** The Vdd (+5 volt) is pin 1, and Vss (ground) is pin 8.

20.4.2 Optional Part

Instead of sounding a buzzer for the alarm, play the melody shown in section 20.6. The same melody is represented as an array of notes denoting duration and pitch in table 20.2 in section 20.6. Because of the rather limited dynamic range of the piezo transducers, the range of scales that can be played is also limited to about an octave.

To play a note, two timers are needed to control the pitch and the duration. One timer is used to generate the continuous square wave to drive the piezo transducer at the right pitch (frequency), and the other is used to time out at the end of the duration. For instance, to play a quarter note of pitch C6, a 2093 Hz square wave must be generated for 0.5 seconds. In this melody, the smallest increment of the duration is one sixteenth of a note (at 120 quarter notes/minute), which is 0.125 seconds long. To control the duration, use the Pulse Accumulator to count 40 Hz square waves generated by the Output Compare Module 3, as shown in figure 20.3. The Pulse Accumulator is set to count up on the rising (or falling) edges so that a count of 5 is equivalent to the duration of a sixteenth note. Use the Pulse Accumulator overflow interrupt to time out the end of a note.

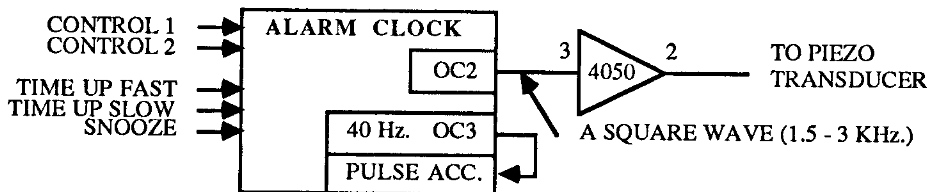


Figure 20.3. Block diagram of the alarm clock for the optional part

20.5 Hints and Suggestions

20.5.1 A State Diagram of the Alarm Clock

Two input signals are used to control the state of the alarm clock. To eliminate the race conditions introduced by the switches in changing between 00 and 11 states, the state

transitions shown in the state diagram in figure 20.4 are suggested. The state diagram shows only the valid state transitions. The unrecognized switch settings, such as 10 in the TIME SET state, are ignored. Of course, it can be designed to change the state from TIME SET to ALARM ON directly, although it is not necessary.

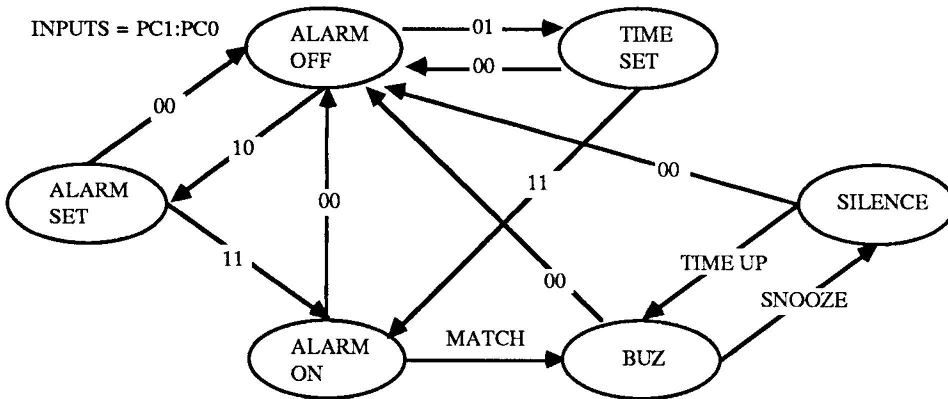


Figure 20.4. State diagram of the alarm clock

The main routine of the alarm clock should do nothing but sample the switch settings and make the state transitions. A global variable, MODE, may be used to indicate the state. To debounce the switches, the switch should be read , say, every 3 ms. until there is no change in the switch settings. The BUZ and SILENCE states are internal to the alarm clock only. This may be implemented as parts of the ALARM ON state using two flags; SOUND_ON and DELAY. The SOUND_ON flag is set to true when the alarm goes off, and is turned off as ALARM OFF state is entered. The DELAY is set to the snooze delay when the SNOOZE is pressed and is counted down in the SILENCE state. The buzz or melody is to be generated when SOUND_ON is true and DELAY is 0. DELAY is also reset as the ALARM OFF state is entered.

20.5.2 Interrupt Handlers

Since the counter-timer modules cannot be programmed to generate an interrupt every second, a software counter will have to be used. For instance, the Output Compare 3 can be set to generate an interrupt every 12.5 milliseconds. In the OC3 interrupt handler, a counter can be used to count 80 to derive at a second. For the optional part, the output pin can be configured to toggle to generate the required 40 Hz square wave.

The pitch of a note can be represented using either the index value or the actual count value of the timer. Using the index values, as shown in table 20.1, any pitch in the melody can be represented with 4 bits (for a melody whose range is about one octave). The index 0 can be used to denote a silent pitch. Similarly, the duration of a note can be represented with 4 bits; 0 for a sixteenth, 1 for two sixteenths, and so on up to 15 for a whole note. This way, a note can be represented in one byte.

The interrupt handlers are shown below.

```
oc2hnd()                                        /* generate pitch of a note */
{ if ( TFLG1 & OC2F )          /* if OC2F bit of TFLG1 reg. is set */
  {    TFLG1 &= OC2F;          /* clear OC2F bit to clear interrupt */
       TOC2 += PITCH;          /* add PITCH count so that next interrupt can occur */
       if ( DELAY > 0 )/* if in SILENCE state, */
               TCTL1  &= 0x3f;          /* disable the sound */
       else    TCTL1  |= 0xc40;         /* else enable the output to generate sound */
  }                                     /* else spurious interrupt, ignore it */
}


oc3hnd()                                        /* generate 40 Hz square wave */
{ if ( TFLG1 & OC3F )          /* if OC3F bit is set, service interrupt */
  {    TOC3 += 25000; /* next interrupt 12.5 ms. later */
       TFLG1 &= OC3F;          /* clear OC3F bit to clear interrupt */
  }                                     /* else ignore spurious interrupt */
}


paovfh()                                        /* interrupt every 1/8 second */
{ if ( TFLG2 & PAOVF )         /* service interrupt if PAOVF flag of TFLG2 is set */
  {    TFLG2 &= PAOVF;         /* clear interrupt */
       PACNT = -5;      /* next interrupt 5 input pulses later */
       if ( SOUND_ON && (DELAY == 0) )        /* need to generate sound? */
               if ( --DURATION == 0 )   /* count down duration of this note */
                       get_next_note( DURATION, PITCH); /* duration and pitch  */
       if (--COUNT8 == 0)        /* one second elapsed ? */
       {       COUNT8 = 8;     /* reset the software counter for one second */
               if ( DELAY > 0 )/* decrement snooze delay, if not zero */
                       DELAY--;
               increment( TIME );        /* increment one second of timer */
               if ( ALARM_ON && !SOUND_ON )        /* if ON hasn't gone off */
                       if equal( TIME, ALARM )/* and setting matches time */
                       {       SOUND_ON = true;        /* BUZ */
                               get_first_note( DURATION, PITCH);
                               enable oc2 interrupt;       /* enable sound */
                       }
       }               /* else one second has not been passed yet */
  }            /* ignore spurious interrupt */
}
```

20.6 Melody

Any melody that does not have a range greater than an octave can be used instead of this one. This is a very familiar tune, but the name of the melody is withheld to prod your interest.

Table 20.2 An array of notes denoting (duration,pitch)

(1/4,G#5) (3/8,G#5) (1/8,F5) (1/4,F5) (1/4,G#5) (3/8,G#5) (1/8,D#5) (1/4,D#5) (1/4,F5)
(1/4,F#5) (1/4,G#5) (1/4,A#6) (1/4,C6) (3/4,G#5)
(1/4,G#5) (3/8,G#5) (1/8,F5) (1/4,F5) (1/4,G#5) (3/8,G#5) (1/8,D#5) (1/4,D#5) (1/4,D#6)
(1/4,D6) (1/4,D#6) (1/4,F6) (1/4,A#6) (3/4,D#6)
(1/4,G#5) (3/8,F6) (1/8,F6) (1/4,D#6) (1/4,C#6) (3/8,C#6) (1/8,C6) (1/4,C6) (1/4,C#6)
(1/4,D#6) (1/4,C6) (1/4,A#6) (1/4,G#5) (3/4,C#6)
(1/4,C#6) (3/8,C#6) (1/8,A#6) (1/4,A#6) (1/4,C#6) (3/8,C#6) (1/8,G#5) (1/4,G#5)
(1/4,G#5)
(1/4,A#6) (1/4,C#6) (1/4,G#5) (1/4,D#6) (3/4,C#6)

21 Local Networks

21.1 Goals

1. To introduce the digital data communication techniques
2. To introduce Ethernet LAN protocol
3. To introduce High-Level Data Link Control (HDLC) protocol

21.2 Introduction

As the cost of computer hardware declined and the complexity and functionality of microprocessors increased, there emerged a variety of intelligent workstations and single-function systems at more affordable prices. Intelligent file servers, high-resolution graphics printers, computer-aided design stations, and many other office automation equipment are examples. In order to share data between these systems, and to better utilize the expensive resources, many have found the need of a network to connect these. Hence there emerged local networks.

Stalling (see reference below) defines a local network as "a communication network that provides interconnection of a variety of data communicating devices within a small area," and he emphasizes that the local network is a communication network, not a computer network. That is, the goal of the local networks is to provide an inexpensive means to transfer data between the connected devices, and not to increase the performance or the reliability of the connected devices. The local networks are most commonly used in one or more closely located buildings, such as in office buildings, factories, college campuses, or in military installations. Their typical characteristics are high data rate, short operating distances, and low error rates. These characteristics essentially determine the nature of the protocols used. For instance, in local networks a simple error detection and retransmission scheme is more desirable than a more expensive error correction scheme because of the low transmission error rate.

In early eighties, the International Standards Organization (ISO) developed the *Open Systems Interconnection* (OSI) model for connecting "open" systems for distributed applications processing. In this model, a seven-layer hierarchy of protocols are defined. The local network protocols cover the lowest two levels: the physical and link layer. The functions in the link layer are further divided into two sublayers in IEEE 802 standard. They are Logical Link Control (LLC) and Medium Access Control (MAC). Note that ISO's OSI is a model whereas IEEE 802 is a standard. In this experiment, we will study two protocols; HDLC, an OSI link layer protocol and the precursor of the IEEE 802 LLC standard, and Ethernet, one of the first commercially available LANs.

We recommend that you review sections 8-3 and 8-4 of *Single- and Multiple-Chip Microcomputer Interfacing*. For further reading on local networks, we recommend *Local Networks; An Introduction,* by William Stalling, Macmillan Publishing Company, New York, 1984.

21.3 Description

21.3.1 High-Level Data Link Control

HDLC is a bit-oriented protocol that treats a block of data as a sequence of bits. The meaning or the interpretation of the data (bit stream) is not defined in this protocol. It uses synchronous transmission, meaning that all stations connected to the network use a common clock. On the network is a *primary* station that oversees the operation of the network. The other stations are termed *secondary.* In this unbalanced configuration, the primary station sends a command to a secondary station, to which the addressed secondary station responds. The secondary stations cannot initiate a dialogue. In a typical operation, the primary station polls the secondary stations, giving each station a chance to communicate with the primary station. If a secondary station has nothing to send, the next station is polled. Otherwise, the station is allowed to transmit as long as it needs. In a balanced configuration, each station functions as both primary and secondary (combined). However, the connection is point-to-point and there can only be two combined stations on the network.

There is a limit to the maximum length of a message, which is set by the desired reliability of the transmission. Since the transmission errors are handled with retransmissions, the longer messages may degrade the network performance with higher than expected error rates. For that reason, a message longer than the allowed length is divided into *frames.* A special bit pattern is defined to indicate the beginning and the end of a frame. The frame format is shown in figure 21.1.

Figure 21.1. Frame format

The flag, address, and control fields are 8 bits, and the address and control fields are extensible. The address field indicates the address of the secondary device that is either to receive the frame or is sending the frame to the primary station. The control field indicates the function the receiver station should perform upon receiving the frame. The data field is optional depending on the nature of the transmission. The length is also variable. The Frame Check Sequence (FCS) field is either a 16- or 32-bit Cyclic Redundancy Check (CRC). Note that in this protocol, there is no definition of how a bit is to be physically transmitted, nor of how the bit stream is to be interpreted. Those are defined in other layers.

The flag is defined to be 01111110. Since a frame may contain an arbitrary bit sequence, there is no guarantee that this flag pattern will not appear in a frame. To avoid this problem, a technique known as *bit stuffing* is used. The transmitter will always insert an extra 0 after the 5th consecutive 1s in the frame, with the exception of the flag fields. The receiver must be aware of this, and should remove the extra 0s in the bit stream.

21.3.2 The Ethernet Protocol

In local networks, all stations connected must share the network's transmission capacity, and there must exist some means of controlling the access to the network to establish a fairness of sharing. There are two ways to achieve this; centralized and distributed control. The HDLC protocol is centralized control. Its advantage over a distributed control protocol is that there is no contention over the network, the network interface hardware is simpler, and the network performance may be better because of the tighter control. However, the principal disadvantage is that the controller (the primary station) is the bottleneck and the single point of failure.

The Ethernet protocol is distributed control in that each station must compete for the access of the network. Its medium-access control protocol is called CSMA/CD, short for Carrier Sense Multiple Access with Collision Detection. CSMA/CD is a protocol in which each station listens on the channel (CS), and when the channel becomes idle, the stations who need to transmit may attempt to do so (MA). Meanwhile, if the transmitting station detects a collision, which occurs when more than one station simultaneously starts transmitting, it stops and retries at a later time. The rescheduling of the failed transmission uses what is known as a "truncated binary exponential back-off" algorithm. The algorithm calls for a random delay from an increasingly larger interval [0, 2^n - 1] as the number of retries (n) increases until, after 16 tries, the transmission attempt is aborted. For 11 to 15 attempts, the delay interval is truncated and remains at [0, 1023]. The unit of time for the retransmission delay is 512 bit times.

21.4 Procedure

21.4.1 Standard Part

The standard part of the experiment involves implementing the CSMA/CD medium access protocol, as used in Ethernet. To simplify the underlying hardware, we will use a wire-OR bus as the channel. This will greatly simplify the task of detecting channel idle and collisions. The logic diagram of the network interface is shown in figure 21.2.

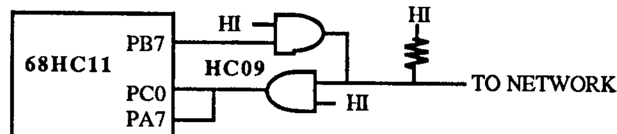


Figure 21.2. Network interface logic

Bit 7 of port B will be used to transmit the encoded data, and bit 0 of port C will be used to listen into the channel. The Manchester encoding is to be done in software by sending the complement of the data bit (the clock bit) before sending the data bit. For example, to send the data 1011, 01100101 must be sent, as shown in figure 21.3. In figure 21.3, C denotes the clock.

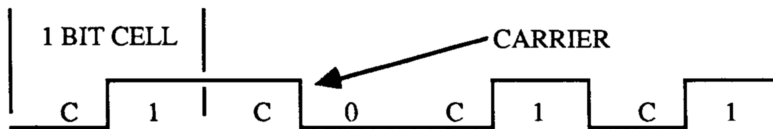


Figure 21.3. Manchester encoding of data 1011

The channel is deemed idle when the carrier is not present. In this scheme, the carrier would be the transition in the middle of a bit cell. For the wire-OR channel the idle, or quiescent, state would be a logic high. This simplifies the collision-detection task. Unless two or more stations are transmitting an identical bit pattern, a collision would force the channel to a logic-low state.

The stations on the network may communicate with each other through a packet. A short frame format is defined, as shown in figure 21.4.

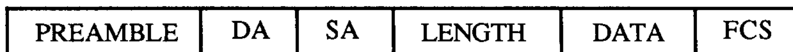


Figure 21.4. Short frame format

The preamble is an 8 bit pattern 10101011; the destination address (DA) and the source address (SA) are 8 bits long; and the length is also 8 bits long, indicating the number of bytes in the data field. Correspondingly, the maximum data length is 255 bytes, but for experimental purposes, limit the message length to 80 bytes. The FCS covers only the DATA fields. For the FCS, use an 8-bit Longitudinal Redundancy Check (LRC). LRC is formed by exclusive-OR of the LRC register with each 8-bits of the data.

The data rate is to be 200 bits/second Note that the actual transmission rate is 400 bits/second because of the encoded clock. Since the stations must be listening to the channel at all times, the receiver should be operating at all times. Use an output compare module in interrupt-driven mode for the receiver. It should sample the channel at the transmission rate, separate data from the clock, and generate an LRC so that error detection can be performed. When a frame is correctly received, it should save the source address and the data, and sound a bell to indicate that a message has been received. Actually, there is a better way to implement the receiver. See section 21.5 for a more detailed discussion.

We will pretend that this network interface unit is being used by a mail utility program whose sole function is to send and receive mail messages. The utility program recognizes two commands; R (read) and S (send). On an R command, the utility program should display the received message and its source. On an S command, it should input the destination address and a one-line message from the terminal, and transmit the message. If the transmission was successful, it should indicate so. Otherwise, it should indicate the failure and retry it until it succeeds. For the back-off algorithm, use a simple random number in the range of [1, 16] for the retransmission delays. Use 100 bit time as the unit of retransmission delay. To jam the channel to force a collision, use 10 bit times. Also, assume that there are at most 10 stations connected in the network.

21.4.2 Optional Part

Use a 16-bit CRC whose generating function $G(X) = X^{16} + X^{15} + X^2 + 1$ for the FCS. This function can be implemented in software using shift and exclusive-OR operations. A pictorial diagram of the function G(X) is shown in figure 21.5.

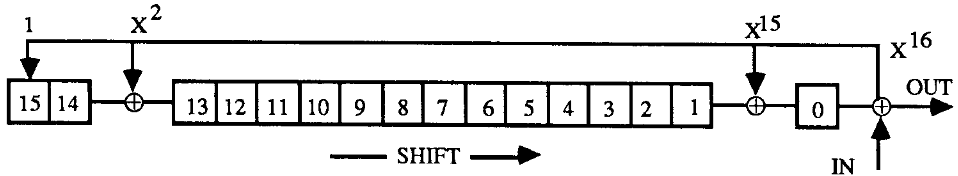


Figure 21.5. CRC-16 generating function

Initially, the 16-bit register is cleared. As each bit is ready to be sent, the bit is exclusive-ORed with the least significant bit of the CRC register. This bit is transmitted. Meanwhile, this bit is fed to three taps in the register. It is shifted into bit 15, is exclusive-ORed with bit 14 and shifted into bit 13, and is exclusive-ORed with bit 1 and shifted into bit 0. This process is repeated for each bit. After all data bits are sent, the content of the CRC register is transmitted, without adding further to the CRC. The receiver should do the same operation.

21.4.3 Extra Credit

This experiment involves using a Serial Peripheral Interface module to implement a local network based on the HDLC protocol for the unbalanced configuration shown in figure 21.6.

The communication is limited to the primary station and the addressed secondary station. The primary station also controls the clock. The secondary stations may not initiate a transmission unless addressed by the primary station in a polling sequence.

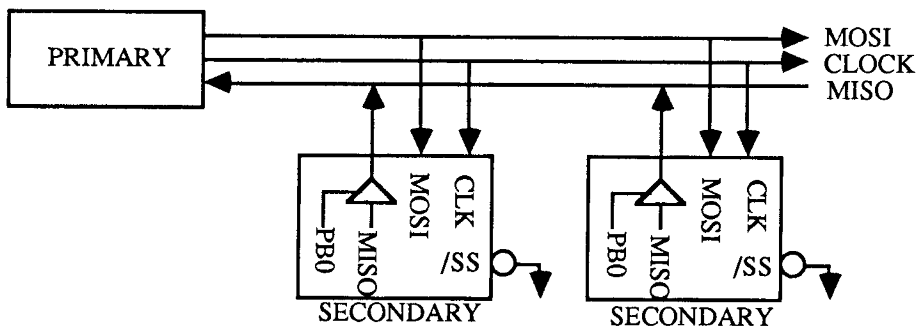


Figure 21.6. Unbalanced configuration

We will use the frame format defined in figure 21.1, without the FCS field. There are five control functions (CF) defined for this experiment, as shown below.

1. indicates that this is a polling packet, and that the addressed station should enable its transmitter. Other stations should disable their transmitter
2. indicates data ready
3. indicates data not ready
4. indicates that the selected secondary station should transmit the data
5. indicates that the packet contains data

We will use a simplified HDLC protocol for the communication. If a primary station needs to send a message to a secondary station, it uses CF = 5. Other times, it polls the secondary stations using CF = 1. If the polled station sends back CF = 2, the primary station should send CF = 4 so that the secondary can transmit its data with CF = 5. If the polled secondary responds with CF = 3, the next logical secondary is polled. Note that only the packets with CF = 5 contain a data field.

The SPI in the primary station should be configured as the master. The others should be configured as slaves. Since the slave SPI's cannot transmit unless clocked by the master, the primary station should supply the clocks until a complete packet is received from a secondary. To do so, the master must send a dummy bit stream. You may use any bit patterns, but we suggest all 0s.

Although a full-duplex transmission is possible, implement a half-duplex communication system for its simplicity. In addition, treat the primary station as just another secondary. In other words, check to see if the primary station has a message to send to a secondary in the polling sequence. Note that this message is different from the packets involved in the polling. The polling is done in the link layer, while the message to be sent comes from higher layers. Implement the mail utility program described in the standard part as the higher-layer program. The data rate is not specified, but it should be as high as possible.

21.5 Hints and Suggestions

The Ethernet server can be implemented in a simple and efficient way if the server can be decomposed into three independent functional units that coordinate with each other through signals. The functional units are a channel monitor, a transmitter, and a receiver. The channel monitor constantly listens to the channel and sets the flag that indicates the status of the channel. The channel status can be IDLE, IN-USE, and SCRAMBLE. The channel enters the SCRAMBLE status when the transmitting stations detect a collision, stop transmission of data, and start transmitting a scramble message so that all stations will know that a collision has occurred. The channel monitor can be thought of as a separate processor that has exclusive control over the channel status indicator.

For the following discussion, refer to figure 21.7. When the Ethernet server is powered on, the channel monitor assumes that the channel is idle. If it detects the carrier present on the channel, it sets the channel status to IN-USE and enables the receiver. As long as the carrier is present, the channel status remains in this state. In this state, two things can happen. When the message is correctly transmitted, the channel will become idle. Or a collision can occur. When a collision occurs, the channel monitor kills the receiver. If the station was transmitting, the transmission is aborted, and the transmitter is set to send the scramble signal. The channel status remains in the SCRAMBLE state until the scramble signal is transmitted.

The transmitter is enabled only when the mail routine has a message to transmit. The mail routine must check the channel status and wait until the channel is idle before the transmitter is enabled. The transmitter disables itself when the entire message is transmitted.

The receiver, after enabled, must be synchronized with the incoming data stream. There are many ways to achieve the synchronization. However, for this experiment, a simple technique will be used. Instead of sampling the bit stream many times, a cell time, one sample from the middle of a cell will be used. (One bit cell can be thought of as two cells; a clock and a data cell). The trick is to enable the receiver so that its first sample will be about half of a cell before the first carrier signal. This is explained in more detail in the algorithms.

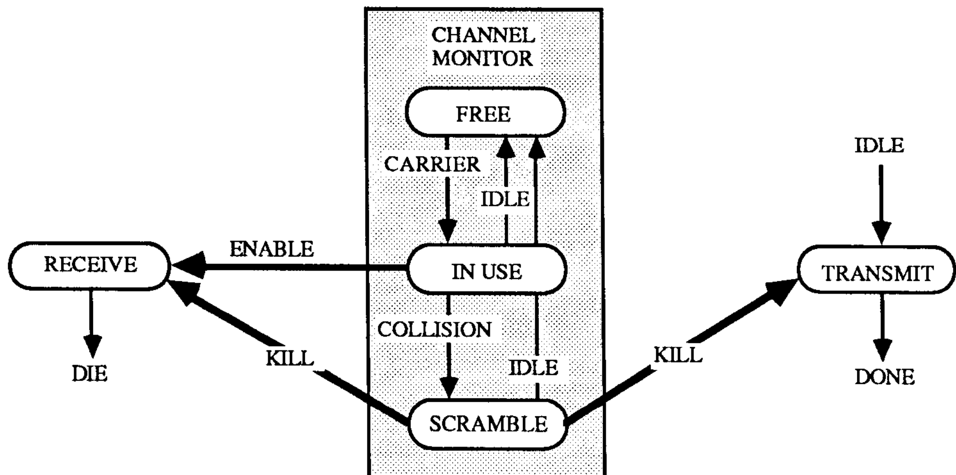


Figure 21.7. The three functional units of an Ethernet server

21.5.1 The Channel Monitor

The pulse accumulator module is used to implement the channel monitor. The pulse accumulator is set to operate in gated time accumulation mode (PAMOD = 1). The pulse accumulator count is set to -80 so that it takes

80 x 64 E clocks @2 MHz = 2.56 ms

which is slightly longer than one bit time (2.5 ms) to generate a pulse accumulator overflow. Two interrupts are used: pulse accumulator overflow and pulse accumulator input edge interrupt. If the pulse accumulator input edge interrupt occurs, then a carrier has been detected. So, reset the pulse accumulator counter to -80 and look for a carrier in the next cell. On the other hand, if a pulse accumulator overflow interrupt occurs, a carrier has not been detected. This means that the channel is in collision or is idle. Since the transmitter interrupt is competing with other interrupts, the carriers may not be placed exactly 1 bit time apart. To handle the discrepancies, a wider cell window may be used to detect the carriers.

21.5.2 Algorithms

```
int         channel_status;   /* channel status indicator */

/* variables used by transmitter handler */
bits        trans_reg[16];    /* 16-bit register holding Manchester coded data */
int         cnt16;                     /* count 16 bits in transmitter handler */
int         trans_cnt;                 /* counts total number of bytes transferred */
char        *trans_ptr;                /* buffer pointer */
char        trans_buffer[85]  /* buffer */
int         trans_checksum;   /* checksum accumulator */

/* variables used in receiver */
bits        rec_reg[16];               /* 16-bit register holding rec'd bit stream */
int         rec_cnt;                   /* counts number of bytes of data received */
char        *rec_ptr;                  /* buffer pointer */
char        rec_buffer[85]    /* buffer */
int         rec_checksum;     /* checksum accumulator */

main()
{ init();                              /* initialize global variables, CLI */
  start_channel_monitor();
  while (TRUE)                /* endless loop */
  {   c = getchar();          /* read a command */
      if (c == 'R')     display(rec_buffer);        /* display the rec'd message */
      else if (c == 'S')
      {       collect(trans_buffer);      /* read the dest. add. and a line of message */
              transmit(trans_buffer);     /* transmit the message */
      } else outch(bell);     /* else echo bell */
  }
}
```

collect() and display() are trivial, and hence not shown.

```
transmit(buffer)
char            *buffer; /* pointer to buffer containing the message to be sent */
{ int           retry = 0;                    /* retry of transmission counter */
  boolean       done = FALSE;  /* transmission is not finished */
  do
  {    while ( channel_status != FREE ) ; /* wait until channel is free */
       trans_ptr = trans_buffer;    /* trans_cnt must also be initialized, too */
       cnt16 = 1;                          /* 1st time in int. handler, trans_cnt = 0 */
       start_transmitter();
       while ( channel_status != IN_USE ) ;          /* wait until trans. has begun */
       while ( (channel_status != FREE) &&
                 (channel_status != SCRAMBLE));          /* wait done */
       if (channel_status == FREE)
       {         done = TRUE;
                 printf("SUCCESSFUL TRANSMISSION\n");  }
       else
       {         scramble();                  /* send a scramble message */
                 printf("RETRY %d\n", retry++);
                 backoff(random());  }        /* wait random delay */
  } while (!done && (retry < 12))            /* retry at most 12 times */
}

/* Channel Monitor uses PAOVF and PAII handlers */
paii_handler()  /* If this interrupt handler is invoked, a carrier must have been detected */
{  pactl = pactl ⊕ 0x10;                    /* flip the PEDGE bit to detect next carrier */
   pacnt = -80;                              /* reset the pulse accumulator counter */
   tflg2 = paif;                             /* clear PAIF and PAOVF bits to clear int */
   if ( channel_status == FREE )  /* if channel status was free, make it IN USE */
   {    channel_status = IN_USE;
        rec_ptr = rec_buffer;          /* point to receiver buffer */
        enable_receiver();             /* schedule the first rec'd. int. at 1/2 cell time later */
   }
}

paovf_handler()                              /* If this handler is invoked, a carrier is
absent in one-cell window */
{  if ( channel_status == FREE )            /* if channel is idle */
        if ( channel == LOW )                /* and if channel is low */
                 channel_status = IN_USE; /* then assume that the channel is in use */
        else       ;                                  /* else channel is idle and FREE */
   else
   if ( channel_status == IN_USE )          /* else if channel status is IN_USE */
        if ( channel == LOW )       /* and if channel is LOW */
        {        channel_status = SCRAMBLE;      /* a collision has occurred */
                 disable_transmitter();       /* disable transmitter interrupt */
```

```
                disable_receiver();   }        /* disable receiver interrupt */
        else                                            /* else channel is high (IDLE) */
                channel_status = FREE;  /* the channel is free again */
    else                                                /* else status is SCRAMBLE */
    if ( channel == HIGH )          /* and if channel is high */
        channel_status = FREE;     /* scrambling is finished and the channel is released */
    else ;                                              /* else status is SCRAMBLE*/
    pacnt = -80;                            /* set up next cell window */
    tflg2 = paovf;                          /* clear the interrupt */
}

transmitter_handler()                       /* transmitter uses OC2 module interrupt */
{  tflg1 = oc2f;                            /* clear interrupt */
    toc2 += 2,000,000 / 400;        /* schedule next interrupt one cell time later */
    if ( --cnt16 > 0 )                      /* if one byte has not been sent completely */
    {    trans_reg <= 1;    return;  }      /* put next bit on the channel */
    else                                                /* else cnt16 is 0 */
        if ( --trans_cnt > 0 )      /* if there are more data to be sent */
        {           cnt16 = 16;             /* take next 16 clocks to send one byte */
                trans_reg = manchester_encode(*trans_ptr++);  }
        else                                            /* else data is transmitted */
                disable_transmitter();
}
```

receiver_handler() is similar to transmitter_handler. The channel monitor must schedule the first receiver one half a cell time after the first carrier so that the subsequent samples will be taken around the middle of the bits.

22 A Floppy Disk Drive Controller

22.1 Goals

1. To implement a simple disk drive controller to read a sector
2. To introduce a hierarchical file concept

22.2 Introduction

A floppy disk system is now quite inexpensive and easy to implement with small computers. The advantages of disk storage are obvious. This experiment illustrates the basic techniques used in reading a floppy disk.

22.3 Description

This experiment expands on the material in section 9-1 of *Single- and Multiple-Chip Microcomputer Interfacing*. The hardware and software in that section are to be implemented and tested, and some extensions of them are suggested for optional and extra credit work.

22.3.1 Magnetic Recording

A floppy disk is made of mylar as the base material and a magnetic coating on both sides for double sided disks (a hard disk uses aluminum in place of mylar). The coating consists of fine magnetic dipoles aligned along the direction of the head movement, as shown in figure 22.1.

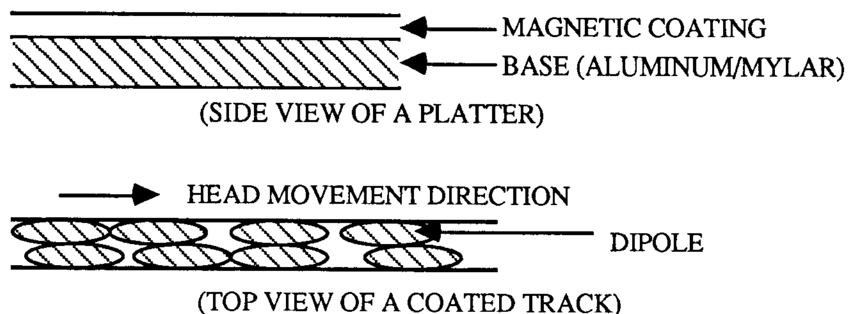


Figure 22.1. Magnetic coating with grains of floppy disk

Each dipole can be oriented in two ways: the north pole along the direction of the head movement or in the opposite direction. As the head moves along a track, it will experience numerous changes of the dipole orientations, with each change inducing an electric current on the read coil of the head. These current pulses are transformed into digital pulses by the amplifier/shaping circuit. If you will, imagine that each such pulse represents one bit of a logic high. Also, there is a limit to how closely these pulses can be placed together because of the interactions between the oppositely charged regions which tend to cancel each other out if placed too close to one another.

22.3.2 Encoding

Because magnetic recording relies on magnetic flux changes, encoding is required to guarantee periodic flux changes. Consider a long stream of 0s (or 1s). There will be one flux change, as shown in figure 22.2.

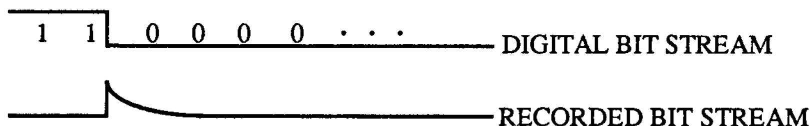


Figure 22.2. Magnetic flux change a on recorded bit stream

Now, how many 0s do you suppose there are in the bit stream? Unless the motor speed of every disk drive is precisely matched, a data written in one drive may not be readable by another drive. Without an encoding scheme, an arbitrary bit stream cannot be recorded and read reliably. An inexpensive encoding scheme is Frequency Modulation (FM) or Manchester Coding. With FM encoding, a clock pulse is inserted between every data bit. As you can see, FM encoding uses only 50% of the bit capacity. A more sophisticated encoding scheme uses Modified Frequency Modulation (MFM). In MFM, a clock is inserted only between two or more consecutive 0s in a bit stream. This way, the storage capacity is 100% utilized. This is commonly known as double-density recording. There are other encoding schemes too. For instance, Macintosh uses what is known as Group Code Recording (GCR). In GCR, every 4 bits are replaced by a 5-bit pattern such that no more than two adjacent 0s occur. The mapping of 5-bit patterns to the 4-bit patterns is arbitrary. This scheme is only 80% efficient. More expensive drives use Run Length Limited (RLL) encoding. This scheme uses a binary tree to generate the encoded data. As an example, a (2,7) RLL guarantees that there will be at least 2 and at most 7 consecutive 0s between any two 1s. With this scheme, up to 150% storage efficiency is possible.

22.3.3 The Disk Format

The major difference between disk and tape storage systems is that random access is possible with disk system while it is not with the other. To provide random access capability, the disk is divided into tracks, which in turn are divided into sectors. The

number of tracks on a disk depends on the accuracy of the head positioning mechanism and the material of the disk itself. For instance, the 5-1/4-inch floppy disks expand significantly when a drive warms up, so that accurate positioning is not practical beyond about 80 tracks/inch. For more sturdy 3-1/2-inch floppy disks, 135 tracks/inch is possible with more accurate "voice-coil" positioning rather than stepper motor mechanisms. The number of sectors/track depends on the format of the track that a system designer has decided upon. For instance, IBM 3740 format uses 26 sectors of 128 bytes each. In the OS-9 operating system used with MC6809 and MC68000 microcomputers, a track may be divided into 16 sectors of 256 bytes each. When a disk is said to be formatted, it means that some control information is written to the disk such that the disk is logically divided into numerous sectors. The most commonly used double-density floppy disk format is shown in figure 22.3. When a disk is formatted, all information but the DATA and EDC/ECC field is written.

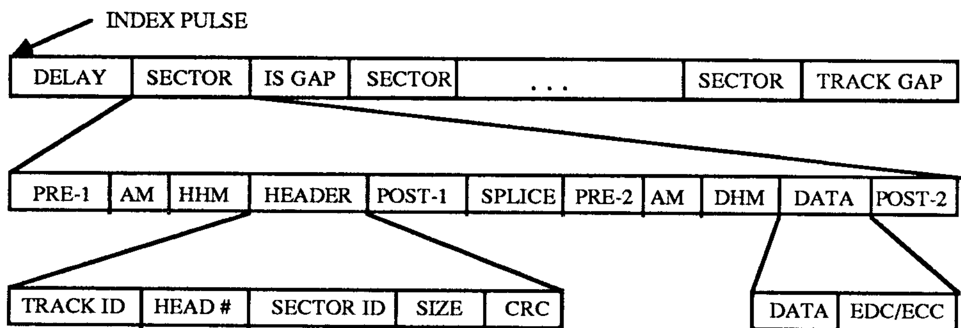


Figure 22.3. Double-density floppy disk format

All floppy disk drives now use *soft-sectored* disks; the ones with an index hole. The term *soft-sectored* refers to the fact that the disk itself carries no information concerning sectors. The sector information is written with a formatting program. A hard-sectored disk would have a hole for each sector. When this hole is detected, the drive generates the index pulse to indicate the beginning of a track. The purposes of each field in the format are described below.

IS GAP	An intersector gap is used to compensate for the motor speed variations, and is in the order of 20 to 30 bytes.
TRACK GAP	The track gap is the remaining space after desired number of sectors are carved out.
PRE-1,2	A preamble is used to allow the data separator logic to be synchronized with the incoming bit stream during read operation. It is about 10 to 20 bytes long.
AM	An address mark is used to signal the pending arrival of header mark. Three bytes of $F5 or $A1.
HHM	A header header mark indicates that this is a header field. One byte of $FE.

TRACK #	One byte track identification.
HEAD #	One byte head (or side) identification.
SECTOR ID	One byte sector identification.
SIZE	One byte sector size indicator.
CRC	16-bit CRC sum covering the previous four bytes.
POST-1,2	A postamble is used to give the drive controller some time to act on the data found in the header or data field. Usually the same size as the preamble.
SPLICE	The head electronic is changed from the read to the write mode in this area so that if a glitch is generated, the data will not be changed. About 10 bytes of garbage usually fill this area.
DHM	A data header mark indicates that this is a data field. One byte of $FB.
DATA	Contains 128, 256, 512, or 1024 bytes of data.
EDC/ECC	Error detection/correction code for the data field. Size varies from 2 to 15 bytes.

22.3.4 Logical and Physical Disk Organization in OS-9

OS-9 is an operating system similar to UNIX. The information presented here is specific to OS-9. However, the concepts should apply to many other systems.

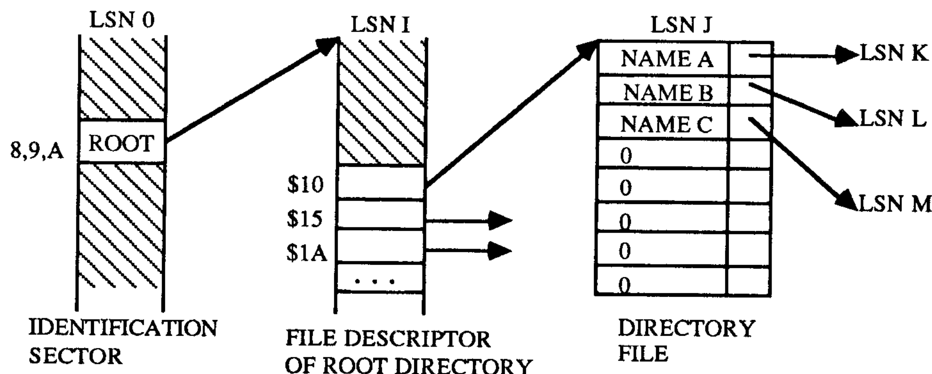


Figure 22.4. File organization in OS-9

The logical sector zero (LSN 0) is called the identification sector. It contains all the information concerning the logical and physical organization of a particular disk. Some examples are total number of sectors, number of sectors per track, disk density, number of sides, disk name, date of creation, owner, location of the bootstrap file, and location of the root directory. LSN 1 is used as the allocation map of the disk. Bits in it indicate which sectors are free and which are not.

OS-9 supports hierarchical file system. A hierarchical file system can be thought of as a tree structured (usually not a binary tree) file system in which each nonleaf node represents a directory and each leaf node represents a file. The root is the topmost

directory. In OS-9, every file has a descriptor, which occupies an entire sector. The file descriptor contains information such as file attributes (Directory, Sharable, [Public] Executable, [Public] Readable, [Public] Writable), date last modified, file size, and pointers to the sectors if the file is stored in logically segmented sectors. These pointers are called a segment list. A segment list is composed of up to 48 of 5-byte entries. Each entry consists of 3-byte LSN that specifies the beginning of the next segment and 2-byte segment size (in sectors). The end of the segment list is indicated with a 0 entry.

Note that directories are treated as files. In fact, the only difference between a file descriptor and a directory descriptor is the attribute (D flag). Each directory file is composed of an integral number of 32-byte entries. The first 29 bytes contain file (or subdirectory) names, and the last three bytes contain the LSN of the file's descriptor sector. The last character of the file name has the most-significant-bit set. Unused entries have the first byte of the name string field set to 0. The file organization is shown in figure 22.4.

The logical sector number must be translated into the physical sector number (PSN). This translation scheme could be quite complex. It could be as simple as direct mapping, in which LSN 0 is mapped to PSN 0. In designing the mapping scheme, the overriding concern is to minimize the disk read/write head movements in the seek operation. The secondary concern is to minimize the latency by the use of an interleaved sector organization. The mapping scheme is built into the format routine of the disk drive device drivers so that it is hidden from the users.

22.4 Procedure

22.4.1 Standard Part

Write a subroutine to read a sector whose LSN is passed into the X register using the design shown in figure 9-6a of *Single- and Multiple-Chip Microcomputer Interfacing* . To verify the experiment, read LSN 0. With OS-9, the **dump** command can be used to see the content of the entire disk in binary form. Compare the content of the read buffer against the dump of the disk. Include the error checking features on seek and read operations. You may use the INIT routine given in section 9-1.3 of *Single- and Multiple-Chip Microcomputer Interfacing* to initialize the floppy controller.

22.4.2 Optional Part

Repeat the standard part using the indirect I/O control shown in figure 9-6b of *Single- and Multiple-Chip Microcomputer Interfacing.*

22.4.3 Extra Credit

Write a program to list the content of the root directory, one per line. Use the indirect I/O design.

22.6 Hints and Suggestions

This experiment is not as difficult as it may seem. The INBUF routine shown in section 9-1.3 of *Single- and Multiple-Chip Microcomputer Interfacing* is the core of this experiment. For the extra credit part, use two read buffers.

There are three kinds of errors possible in a seek operation. They are device-not-ready, seek (mismatch of track I.D.), and crc errors. In addition, there are three more kinds of errors possible during a sector read operation. They are record-not-found, crc-error-on-data-field, and data-overrun errors. Refer to the data sheet of the floppy controller device for more details.

23 Using the MC68HC11A2 Chip

23.1 Goals

The MC68HC11A2 is similar to the MC68HC11A8, except that it has 2K bytes of EEPROM rather than 512 bytes of EEPROM, and it does not have any ROM so it has no built-in Buffalo monitor. It is particularly suited to stand-alone projects because 2K bytes of program memory is sufficient for a reasonable project. The main other differences between the MC68HC11A2 and the MC68HC11A8 are the location of the EEPROM and the contents of the CONFIG register ($103f). The high nibble of the CONFIG register indicates where the EEPROM is located in expanded multiplexed mode. In single-chip mode, EEPROM is from $f800 to $ffff. If the high nibble of the CONFIG register is $f, then EEPROM is from $f800 to $ffff in expanded multiplexed mode too. The CONFIG register is set to $ff whenever the chip is bulk erased.

This chip can be used for stand-alone projects such as those used in senior year design courses. Experiment 19 and some problems at the ends of the chapters of *Single- and Multiple-Chip Microcomputer Interfacing* are suitable for such problems, and have solutions in the instructor's manual. An example is a frequency meter with LED readout.

In order to use the MC68HC11A2, a means to program the EEPROM is needed. You can use the M68HC11EVM board to program them, if you have one, because it has a mechanism for programming these chips. If you only have an M68HC11EVB board, the following procedure can be used to program these chips. The program listed in section 23.5 is loaded into the M68HC11EVB board, which extends the Buffalo monitor to enable it to program a MC68HC11A2, which is in a separate development board.

23.2 Procedure

1. Copy and assemble the "download" program in section 23.5.

2. Connect the SCI ports between the MC68HC11A2 and the EVB board and configure the 'A2 for "bootstrap" mode (see hardware notes, section 23.3).

3. Load the download program to the EVB board RAM using the Buffalo monitor <load t> command. 9600 baud is preferable here if you can use it, but you may have to change the baud rate in a moment.

4. Start the download program with the monitor command <call c000>. The program will display a reminder to reset the 'A2 and give a menu of selections. You must reset the 'A2 (while it's configured in boot mode) before selecting the first command. This reset can also be done before starting the download program.

5. Reset the MC68HC11A2 by momentarily grounding its reset pin if not already done. (A debounced switch is preferred, but we have not yet had problems just shorting the MC68HC11A2 reset pin to ground).

6. Select the "Download" option (number 0) and use DEBUG11 or your communications package to download the program. (Use Bulk Erase (option 3) only to erase the configuration register if necessary). If using DEBUG11, use "slow download" rather than download. If using any other system, change your send baud rate to 300 and move the jumper on J5 of the EVB board to the 300 baud position (the other end of the header) if you had been communicating with the EVB at 9600 baud. If you are switching back and forth, make sure that the baud rate settings agree each time you change it.

7. Send the desired "S Record" file over the serial port to the EVB board. On completion of this transfer, your program should be loaded into the 'A2 EEPROM.

8. Either reconfigure, or use the SCI feature to start your program. To reconfigure, tie the reset pin of the 'A2 to ground and reconfigure MODB for single chip mode. Release the reset pin. Alternatively, to use the SCI feature, tie SCI input to output (port D pin 0 to port D pin 1, with a pull-up, and reset the 'A2).Your application should start up on the 'A2 chip if you programmed the reset vector in your source code to jump to the beginning of your code.

It is difficult to determine what is wrong with a complete system that doesn't work because you do not have a Buffalo monitor in it. You should have a program that already works correctly in the M68HC11EVB board. You should then have a functional MC68HC11A2 chip. To check the chip, you should try to first download and execute a simple program such as this:

```
        ORG $F800
        LDS $#FF
LOOP    LDAA #$FF
        STAA $1004
        BSR WAIT
        CLR $1004
        BSR WAIT
        BRA LOOP
WAIT    LDX #0
W1      DEX
        BNE W1
        RTS
```

This simple program flips the signals on all port B pins at a slow rate that can be easily observed on a logic probe. Inputs can be tested by tying them to outputs and outputting signals and testing the inputs. You might test all I/O pins that will be used in your project in similar manner. Be sure that your MC68HC11A2 is functional before loading your project program.

23.3 Hardware Notes

See figure 1.1 in section 1.4.3 for a typical logic diagram for an MC68HC11A2. Note that IRQ, XIRQ, and RESET are pulled up to +5 volts through resistors. Tie MODA to ground as it is low for both bootstrap and single chip modes. Tie MODB to +5 volts through a resistor. MODB high selects single chip mode, and MODB low selects bootstrap mode. Both MODB and RESET can be switched between low and high states by shorting the pins to ground or letting the resistors pull the pins high. (ie, leave the resistors in the circuit and just short the 'A2 pins to ground when you want a low level on them).

Note that the crystal oscillator circuit requires two 22 pf capacitors connected to ground and a 10 Megohm resistor paralleled across the crystal. The oscillator circuitry inside the 'A2 is merely a 2-input NAND gate, with an output on the XTAL pin and inputs from the EXTAL pin and a flip-flop controlled by the STOP function. It and the external parts make up a Pierce oscillator. This circuit is pretty sensitive to additional capacitance, so if you touch any of the leads while your application is running it will probably malfunction. The oscillator also stops when the STOP instruction is executed.

Before applying power to the MC68HC11A2, check +5 volt and ground lines thoroughly, and check for short circuits. Check the oscillator with an oscilloscope. Then check the reset pin on the MC68HC11A2, to see that the reset signal rises.

The EVB and the 'A2 communicate via the SCI systems in both chips. Connect the EVB transmit pin (PD1) to the 'A2 receive pin (PD0) and vice versa, and connect the ground lines of the MC68HC11A2 and the M68HC11EVB board.

If you are using any analog circuitry in your application, do not use the same ground bus for both the analog and digital circuitry. Instead provide separate ground busses and connect them independently to the ground terminal of your power supply. This will prevent the noisy digital ground from corrupting your analog circuit.

Switching power supplies can produce noise and/or surges that erase EEPROM because many switchers require a minimum current to be suitably regulated and the current used by the 'A11 is too small for reliable switcher operation. Batteries or linear regulators are generally more suitable.

Although MC68HC11A2 can be used in a dual in-line package, and this package is especially useful for experiments done on a proto-board, the quad package is more compact and is suitable for projects that feature miniaturization. Figure 23.1 shows a layout of a circuit board that can use the MC68HC11A2 quad package. The bottom three holes of the left row of holes is for an MC34064 undervoltage sensing circuit mounted flat side out (pin 1 on top), and the parallel pads above and to the right of it are for three 4.7 KΩ pull-up reisitors (Murata Erie RX3910G472GTA or equivalent). An 8 MHz crystal is mounted in the holes above and to the left of the MC68HC11A2, and a 10 MΩ resistor is put in parallel with it (Murata Erie RX3910G106GTA or equivalent). Two 18 pF capacitors are put from its leads to ground (Murata Erie GR39N750180M25VPB or equivalent). A 10 μF tantalum capacitor should be put in the holes at the top left of the board. The positive lead faces the middle of the board. Holes to the right of that are for connecting MODA and MODB to +5 or ground. Other pads are provided for convenient connections to any pin on the chip.

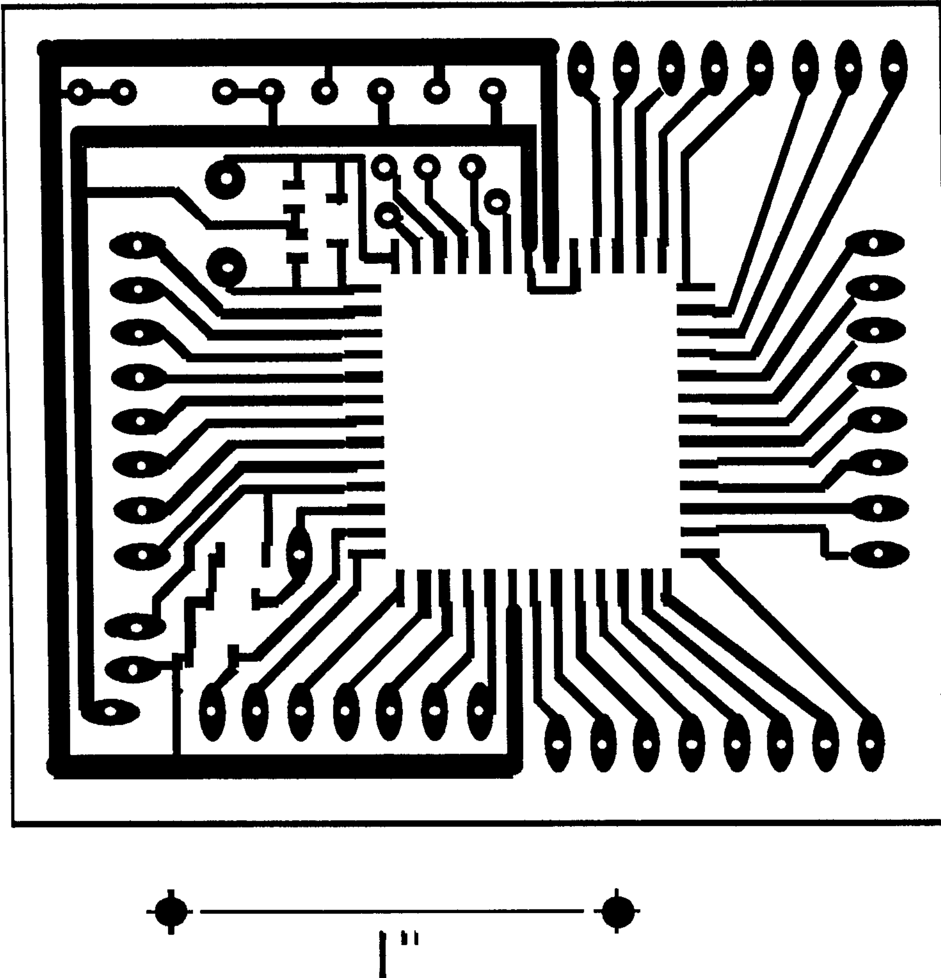


Figure 23.1 A Layout for an MC68HC11A2 quad surface mount chip

23.4 Software Notes

When you switch from the EVB developmental version of your code to the 'A2 code, be sure to change the starting address from $C000 to $F800, which is the start of the 'A2 EEPROM. Interrupt vectors must be changed from the JMP instructions in low RAM used with the EVB to two-byte addresses in high memory for the 'A2. You must specify the reset vector to jump to the beginning of your code to get the 'A2 to start running your application on power up. You must specify the illegal opcode trap vector to jump to the handler as shown in section 1.4.3. The very first instruction in your reset handler, before any subroutines are called, should be a LDS #$FF or equivalent to prepare the stack to handle subroutine return addresses.

23.5 Program Listing

```
*** A2LOADER ***
* 12/29/87 REV 1.1 GREG RASKIN
* 12/9/87 Rev 1.0 TONY FOURCROY
*
***             EQUATES ***
S9FLAG          EQU     $FE
TASK            EQU     $C000           org address
PORTA           EQU     $1000
PORTC           EQU     $1003
PORTB           EQU     $1004
DDRC            EQU     $1007
PORTD           EQU     $1008
DDRD            EQU     $1009
PACTL           EQU     $1026
SPCR            EQU     $1028
BAUD            EQU     $102B
SCCR1           EQU     $102C
SCCR2           EQU     $102D
SCSR            EQU     $102E
SCDR            EQU     $102F
PPROG           EQU     $103B
CONFIG          EQU     $103F
ACIA            EQU     $9800           acia address
EOT             EQU     $04             end of text
OUT1BSP         EQU     $FFBE
OUT2BSP         EQU     $FFC1
OUTCRLF         EQU     $FFC4           Buffalo Routines
OUTSTRG         EQU     $FFC7
INCHAR          EQU     $FFCD
*****************************************
*** "DOWNLOAD TO TARGET USING BOOTSTRAP" ***
*** This routine first bootloads a program
*** into target ram which is then executed by
*** the target. This routine then executes the
*** command specified by the entry point of this
*** routine. Commands include:
*** 0) Download S Records
*** 1) Download S Records and Go
*** 2) Go
*** 3) Bulk Erase EEPROM
*** S records to be programmed into EEPROM
*** must be downloaded from the host using the
*** slow download command (delay at least 15 ms
*** between bytes) to allow the target time to
*** program EEPROM.
*****************************************
* Program entry points - Specify target action
                ORG TASK Entry for Download
                JMP MAIN
* Storage Locations
COMMAND         RMB     1
SHFTREG         RMB     2
TMP1            RMB     1
TMP2            RMB     1
TMP3            RMB     1
TMP4            RMB     1
HEADERA         FCC     'Bring target MC68HC811A2 out of reset in boot mode.'
```

```
                FCB     $0D,EOT
HEADERB         FCC     'Enter one of the following commands :'
                FCB     $0D
                FCC     ' 0=Download, 1=Load n Go, 2=Go, 3=Bulk Erase'
                FCB     $0D
                FCC     '> '
                FCB     EOT
MSGJMP1         FCC     'Enter Starting address >'
                FCB     EOT
MSGLOAD         FCC     'Start Slow Download from host. Target EEPROM'
                FCB     $0D
                FCC     'is at $F800 - $FFFF'
                FCB     $0D,EOT
MSG11           FCC     'done'
                FCB     EOT
MSG14           FCC     'rcv error'
                FCB     EOT
MSG12           FCC     'checksum error'
                FCB     EOT

*********************
MAIN            LDX     #HEADERA
                JSR     OUTSTRG
                LDX     #HEADERB
                JSR     OUTSTRG
                JSR     TERMARG
                LDAA    SHFTREG+1
                STAA    COMMAND
*Initialize the sci
                LDAA    #$02
                STAA    PORTD
                STAA    DDRD            drive tx line high
                LDAA    SCSR            clear out sci rcvr
                LDAA    SCDR
                CLR     SCCR2
                LDAA    #$22
                STAA    BAUD
                LDAA    #$0C
                STAA    SCCR2
*Downlaod RAM2EE to target
                LDAA    #$FF
                STAA    SCDR            send control byte
                LDX     #RAM2EE         followed by 256 bytes ...
                CLRB                    ...starting at ram2ee
BLOOP           LDAA    0,X
                JSR     OUTSCI
                INX
                DECB
                BNE     BLOOP           loop 256 times
                LDAA    SCSR            clear idle bit and ...
                LDAB    SCDR            ...sync to target
SYNCLP          LDAA    SCSR
                LDAB    SCDR
                ANDA    #$10
                BEQ     SYNCLP          loop until idle detect
                BRA     GUESS0          skip header the first time
* Execute the specified command
GUESS           LDX     #HEADERB
                JSR     OUTSTRG
                JSR     TERMARG
                LDAA    SHFTREG+1
```

```
                STAA    COMMAND
GUESS0          LDAA    COMMAND
                CMPA    #$00
                BNE     GUESS1
                BSR     DWNLOAD
                BRA     GUESS
GUESS1          CMPA    #$01
                BNE     GUESS2
                BSR     LOADNGO
                RTS                     quit after loadngo
GUESS2          CMPA    #$02
                BNE     GUESS3
                BSR     JMPSTRT
                RTS                     quit after go
GUESS3          CMPA    #$03
                BNE     GUESS           invalid command
                BSR     ERASIT
                BRA     GUESS

*** COMMANDS ***
*Bulk erase command
ERASIT          LDAA    #$03            send bulk erase command
                JMP     OUTIN
*Jump start command
JMPSTRT         LDX     #MSGJMP1
                JSR     OUTSTRG
                JSR     TERMARG         Get starting address
                LDAA    #$02
                JSR     OUTIN
                LDAA    SHFTREG
                JSR     OUTIN
                LDAA    SHFTREG+1
                JMP     OUTIN
*Load and Go command
LOADNGO         BSR     DWNLOAD
                BRA     JMPSTRT

*Download command
DWNLOAD         LDX     #MSGLOAD
                JSR     OUTSTRG
                CLR     TMP3            error flag - no errors yet
LOAD10          JSR     INACIA          read host
                TSTA
                BEQ     LOAD10          jump if no input
                CMPA    #'S'
                BNE     LOAD10          jump if not S
LOAD12          JSR     INACIA          read host
                TSTA
                BEQ     LOAD12          jump if no input
                CMPA    #'9'
                BEQ     LOAD90          jump if S9 record
                CMPA    #'1'
                BNE     LOAD10          jump if not S1
** Read in S1 Record
                LDAA    #$00            inform target of s1 record
                JSR     OUTIN
                CLR     TMP4            clear checksum
                BSR     BYTE            get byte count
                BSR     TOTARG          send count to target
                LDAB    SHFTREG+1
                SUBB    #$2
```

```
        BSR     BYTE        get starting address
        BSR     TOTARG      send staddr to target
        BSR     BYTE
        BSR     TOTARG
        LDX     SHFTREG
        DEX
LOAD20  BSR     BYTE        get next data byte
        BSR     TOTARG      send to target
        INX
        DECB                check byte count
        BEQ     LOAD30      if b=0, go do checksum
        TST     TMP3
        BNE     LOAD10      jump if error flagged
        BRA     LOAD20      finish download
* calculate checksum
LOAD30  TST     TMP3
        BNE     LOAD10      jump if error already
        LDAA    TMP4
        INCA                do checksum
        BEQ     LOAD10       jump if s1 record okay
        LDAA    #$03
        STAA    TMP3         indicate checksum error
        BRA     LOAD10
* if(a = '9') read rest of record;
LOAD90  LDAA    #S9FLAG     allert routine in A2 RAM that there are
        STAA    SHFTREG+1   no more S1 records and the next byte is a
        BSR     TOTARG      command
        BSR     BYTE
        LDAB    SHFTREG+1   b = byte count
LOAD91  BSR     BYTE
        DECB
        BNE     LOAD91      loop until end of record
        LDX     #MSG11      "done"
        LDAA    TMP3
        CMPA    #$01
        BNE     LOAD93      jump not receiver error
        LDX     #MSG14      "rcv error"
        BRA     LOAD94
LOAD93  CMPA    #$03
        BNE     LOAD94      jump not checksum error
        LDX     #MSG12      "checksum error"
LOAD94  JSR     OUTSTRG
LOAD95  RTS

****************************
*** TOTARG - send the byte in shftreg+1 to
** target and wait for an echo
TOTARG  PSHA
        LDAA    SHFTREG+1
        JSR     OUTIN
        PULA
        RTS

**********
* byte() - Read 2 ascii bytes from host and
*convert to one hex byte. Returns byte
*shifted into shftreg and added to tmp4.
**********
BYTE    PSHB
        PSHX
BYTE0   BSR     INACIA      read host (1st byte)
        TSTA
```

```
                BEQ     BYTE0           loop until input
                BSR     HEXBIN
BYTE1           BSR     INACIA          read host (2nd byte)
                TSTA
                BEQ     BYTE1           loop until input
                BSR     HEXBIN
                LDAA    SHFTREG+1
                ADDA    TMP4
                STAA    TMP4            add to checksum
                PULX
                PULB
                RTS
****************
* HEXBIN(a) - Convert the ASCII character in a
* to binary and shift into shftreg. Returns value
* in tmp1 incremented if a is not hex.
****************
HEXBIN          PSHA
                PSHB
                PSHX
UPCASE          CMPA    #'a'
                BLT     UPCASE1         jump if < a
                CMPA    #'z'
                BGT     UPCASE1         jump if > z
                SUBA    #$20            convert
UPCASE1         CMPA    #'0'
                BLT     HEXNOT          jump if a < $30
                CMPA    #'9'
                BLE     HEXNMB          jump if 0 to 9
                CMPA    #'A'
                BLT     HEXNOT          jump if $39> a <$41
                CMPA    #'F'
                BGT     HEXNOT          jump if a > $46
                ADDA    #$9             convert $A to $F
HEXNMB          ANDA    #$0F            convert to binary
                LDX     #SHFTREG
                LDAB    #4
HEXSHFT         ASL     1,X 2           byte shift through
                ROL     0,X             carry bit
                DECB
                BGT     HEXSHFT         shift 4 times
                ORAA    1,X
                STAA    1,X
                BRA     HEXRTS
HEXNOT          INC     TMP1            indicate not hex
HEXRTS          PULX
                PULB
                PULA
                RTS
****************
*** Termarg - Get argument from terminal
****************
TERMARG         CLR     SHFTREG
                CLR     SHFTREG+1
TERM0           JSR     INCHAR
                CLR     TMP1            hex indicator
                BSR     HEXBIN
                TST     TMP1
                BNE     TERM3           quit if not hex
                BRA     TERM0           loop until non hex
TERM3           RTS
```

```
**********
* INACIA - Read from the ACIA, Return a=char or 0.
* Tmp3 is used to flag overrun or framing error.
**********
INACIA          LDX     #ACIA
                LDAA    0,X         read status register
                PSHA
                ANDA    #$30        check ov, fe
                PULA
                BEQ     INACIA1     jump - no error
                LDAA    #$01
                STAA    TMP3        flag reciever error
                BRA     INACIA2     read data to clear status
INACIA1         ANDA    #$01        check rdrf
                BEQ     INACIA3     jump if no data
INACIA2         LDAA    1,X         read data
                ANDA    #$7F        mask parity
INACIA3         RTS

                ORG $D000
****************************************
************* RAM2EE *******************
*** THIS CODE MUST REMAIN RELOCATABLE SO
*** IT CAN BE DOWNLOADED TO $0000
****************************************
*** Download this program to target RAM
*** at address $0000
RAM2EE          BRA     BANANA
COUNT           EQU     $0002
ADDRESS         EQU     $0003
XCOUNT          RMB     1           download byte count
XADDR           RMB     2
BANANA          LDS     #$FF        initialize stack
                CLR     SCCR1       initialize sci
                CLR     SCCR2
                LDAA    #$05
                STAA    SPCR        clear dwom bit
                LDAA    #$22
                STAA    BAUD
                LDAA    #$0C
                STAA    SCCR2       enable sci
                LDAA    #$80        ***debug***
                STAA    PACTL       ***debug***
                LDAA    #$FF        ***debug***
                STAA    PORTA       ***debug***
                LDAA    SCSR        clear out receiver
                LDAA    SCDR
** Start command processing here
RAM2EE1         BSR     INOUT       wait for command
                CMPA    #$00
                BEQ     COMLOAD     load command
                CMPA    #$02
                BEQ     COMGO       go command
                CMPA    #$03
                BEQ     COMBULK     bulk erase command
                BRA     RAM2EE1

**********************************
*** SCI COMMUNICATIONS ROUTINES ***
* Data transferred through register A.
```

```
***
OUTIN     BSR    OUTSCI
INSCI     PSHB
INSCI1    LDAB   SCSR
          ANDB   #$20
          BEQ    INSCI1      wait until rdrf
          LDAA   SCDR
          PULB
          RTS
INOUT     BSR    INSCI       read char and echo back
OUTSCI    PSHB
OUTSCI1   LDAB   SCSR
          BITB   #$80
          BEQ    OUTSCI1     wait until tdre
          STAA   SCDR
          PULB
          RTS
*** COMMAND ROUTINES ***
***
COMBULK   LDX    #CONFIG     bulk erase eeprom
          BSR    EEBULKJ
TOPJMP    BRA    RAM2EE1
***
COMLOAD   BSR    INOUT       load prog sci->eeprom -get # of bytes
LOOPS1    SUBA   #$03
          STAA   COUNT
          BSR    INOUT       get start address
          STAA   ADDRESS
          BSR    INOUT
          STAA   ADDRESS+1
          CLRB
COM1A     LDX    ADDRESS
          ABX
          LDAA   0,X
          CMPA   #$FF
          BEQ    COM1B       jump if erased
          BSR    EEBYTE      else erase
COM1B     BSR    INOUT       get data
          BSR    EEWRIT      store data
          INCB
          CMPB   COUNT
          BLT    COM1A       loop bytes times
          BSR    INOUT       read checksum
          BSR    INOUT       get next
          CMPA   #S9FLAG     is it S9 flag?
          BNE    LOOPS1      no - it is start of new S1 record
          BRA    RAM2EE1
***
COMGO     BSR    INOUT       go execute - read in starting address
          STAA   ADDRESS
          BSR    INOUT
          STAA   ADDRESS+1
          LDX    ADDRESS
          JMP 0,X

EEBULKJ   BRA    EEBULK
*** EEPROM PROGRAMMING ROUTINES ***
EEWRIT    PSHB               program (x) <= A
          LDAB   #$02
          STAB   PPROG
          STAA   0,X
```

```
                LDAB    #$03
                BRA     EEPROG
***
EEBYTE          PSHB                    byte erase (x)
                LDAB    #$16
                STAB    PPROG
                STAB    0,X
                LDAB    #$17
                BRA     EEPROG
***
EEBULK          PSHB                    bulk erase array
                LDAB    #$06
                STAB    PPROG
                STAB    0,X             erase config or not
                LDAB    #$07
EEPROG          BNE     EEPROG1         prevents runaway code ..
                CLRB                    .. from affecting eeprom
EEPROG1         STAB    PPROG
                PULB
***
DLY10MS         PSHX                    10ms delay at E=2MHz
                LDX     #$0D06
DLY10LP         DEX
                BNE     DLY10LP
                PULX
                CLR     PPROG
                RTS
```

Appendix A Parts List

Memory Systems

74HC373, MCM2114 (4), 74HC00, 74HC04, 74HC08

Traffic Light Controller

Two each of red, green, and yellow LEDs, 230 Ω (6), 74HC374, 74HC00, 74HC04, 74HC08, 74HC595

I.C. Tester

14-pin DIP socket (or your protoboard), 74HC00, 74HC02, 74HC04, 74HC11, 74HC74

Logic Analyzer

74HC244 (3), AM9128 or equivalent 2K-by-8 static RAM (3), 74HC73, 74HC21, 74HC04, 74HC32, 74HC85 (4), 74HC273, 74HC86, 74HC4040, 74HC20

Bar Code Reader

74HC14, HP HEDS-3050

Magnetic Card Code Reader

A magnetic card reader (American Magnetics MagStripe™ Card Reader Model 40S5DA suggested)

Keyboard and LED Display

A matrix keyboard, 4,7K Ω (8), 7-segment common cathod LEDs (4), MC14499, 47 Ω (8), 2N2222 (4), 74HC138, 74HC151, 75491 (2), 75492 (2), 100 Ω (8)

DC and RMS Digital Voltmeter

RCA 3140 OP Amp, Zener diode (~ 5.1 v 1/2 watt), 250KΩ potentiometer (2), resistors

Thermometer

RCA 3140 OP Amp, thermister (~ 400KΩ at 30°C), Zener diode (~ 5.1 v 1/2 watt), 250KΩ potentiometer (2), resistors

Digital Alarm Clock

4050, piezo transducer (Radio Shack Cat. no. 273-073)

Local Networks

74HC09, 74HC373 (2)

Floppy Disk Controller

WD1773 or WD1772, 74HC240, 74HC133, 74HC14, 5-1/4" Single-Sided Floppy Disk Drive

For all experiments except chapters 2 and 23:

Part Number	Description	Quantity
74HC00	Quad 2-input NAND Gates, Totem-Pole	1
74HC02	Quad 2-input NOR Gates, Totem-Pole	1
74HC04	Hex Inverters, Totem-Pole	1
74HC08	Quad 2-input AND Gates, Totem-Pole	1
74HC09	Quad 2-input AND Gates, Open-drain	1
74HC11	Triple 3-input AND Gates, Totem-Pole	1
74HC14	Hex Inverters, Schmitt-Trigger, Totem-Pole	1
74HC20	Dual 4-input NAND Gates, Totem-Pole	1
74HC21	Dual 4-input AND Gates, Totem-Pole	1
74HC32	Quad 2-input OR Gates, Totem-Pole	1
74HC73	Dual J-K Flip-Flops with Clear	1
74HC74	Dual D-type Flip-Flops with Preset and Clear	1
74HC85	4-Bit Magnitude Comparators	4
74HC86	Quad 2-input Exclusive-OR Gates, Totem-Pole	1
74HC133	13-input NAND Gates, Totem-Pole	1
74HC138	3-to-8 Line Decoders with Inverting Output	1
74HC151	8-to-1 Line Data Selectors/Multiplexers	1
74HC240	Octal Buffers with 3-State, Inverted Outputs	1
74HC244	Octal Buffers with 3-State Outputs	3
74HC273	Octal D-type Flip-Flops with Clear	1
74HC373	Octal D-type Transparent Latches	2
74HC374	Octal D-type Edge-Triggered Flip-Flops	1
74HC595	8-Bit Shift Registers with 3-State Output Registers	1
74HC4040	Asynchronous 12-Bit Binary Counters	1
74HC4050	Hex Non-Inverting CMOS Buffer	1
MCM2114	1K-by-4 Static RAM	4
MC75491, MC75492	LED drivers	2 ea.
MC14499	7-Segment LED Display Driver with Serial Interface	1
AM9128	2K-by-8 static RAM or equivalent	3
WD1773 or WD1772	5-1/4" Floppy Disk Controller/Formatter	1
LEDs	(two red, two green, two yellow)	6
7-Segment Common Cathode LEDs		4
2N2222	NPN transistor with 300 mA collector current rating	4
HP HEDS-3050	Bar Code Scanner	1
A matrix keyboard		1
470 KΩ, 47KΩ, 4.7K Ω, 47 Ω, 230Ω, 100Ω resistor		8 ea.
250KΩ potentiometer		2
Zener diode	~ 5.1 v 1/2 watt	1
RCA 3140 OP Amp		1
thermister	~ 400KΩ at 30°C	1
Piezo transducer (Radio Shack Cat. no. 273-073)		1
American Magnetics MagStripe™ Card Reader Model ~~40S5DA~~ 405DA, or equivalent		1
5-1/4" Single-Sided Floppy Disk Drive		1