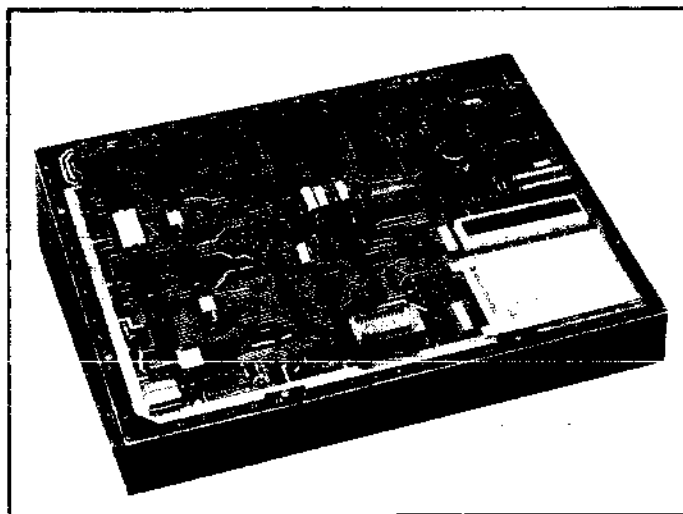


MAC-8



1A MICROPROCESSOR TRAINING AID MAC TUTOR REFERENCE MANUAL



Bell Laboratories

NOTICE


Not for use or disclosure outside the Bell
System except under written agreement.

*Prepared and published for the
Microprocessor Systems Development Department
by the
Technical Documentation Department
Bell Laboratories*

Printed in U.S.A.

CONTENTS	PAGE NUMBER	ISSUE NUMBER AND DATE											
		4-1-79	7-1-79										
PAGE INDEX	A1	1	2										
	A2	1	2										
	A3	1	2										
	A4	1	2										
	A5	1	2										
	A6	Blank											
TITLE PAGE, FRONT	-	1	2										
TITLE PAGE, REAR	-	1	-										
FOREWORD, FRONT	-	-	2										
FOREWORD, REAR	-	Blank											
CONTENTS	i	1	2										
	ii	1	2										
CHAPTER 1 TITLE PAGE													
FRONT	-	1	-										
REAR	-	Blank											

THE CONTENT OF THIS MATERIAL IS PROPRIETARY AND CONSTITUTES A TRADE SECRET. IT IS FURNISHED PURSUANT TO WRITTEN AGREEMENTS OR INSTRUCTIONS LIMITING THE EXTENT OF DISCLOSURE. ITS FURTHER DISCLOSURE IN ANY FORM WITHOUT THE WRITTEN PERMISSION OF ITS OWNER, BELL LABORATORIES, INCORPORATED, IS PROHIBITED.

PAGE INDEX NOTES	SUPPORTING INFORMATION	
	CATEGORY	NUMBER
<ol style="list-style-type: none"> WHEN CHANGES ARE MADE IN THIS DOCUMENT, ONLY THOSE PAGES AFFECTED WILL BE REISSUED. THIS PAGE INDEX WILL BE REISSUED AND BROUGHT UP TO DATE EACH TIME ANY PAGE OF THE DOCUMENT IS REISSUED, OR A NEW PAGE IS ADDED. THE ISSUE NUMBER ASSIGNED TO A CHANGED OR NEW PAGE WILL BE THE SAME ISSUE NUMBER AS THAT OF THE PAGE INDEX. PAGES THAT ARE NOT CHANGED WILL RETAIN THEIR EXISTING ISSUE NUMBER. THE LAST ISSUE NUMBER OF THE PAGE INDEX IS RECOGNIZED AS THE LATEST ISSUE NUMBER OF THE DOCUMENT AS A WHOLE. 		
 Bell Laboratories	MAC TUTOR	AT&T Co.
	REFERENCE MANUAL	Provisional PA-800515-A1

CONTENTS	PAGE NUMBER	ISSUE NUMBER											
		1	2										
1. SYSTEM OVERVIEW	1-1	1	2										
	1-2	1	2										
	1-3	1	2										
	1-4	Blank											
	1-5	1	2										
	1-6	Blank											
	1-7	1	2										
	1-8	Blank											
	1-9	1	2										
	1-10	Blank											
CHAPTER 2 TITLE PAGE FRONT REVERSE	-	1	-										
	-	Blank											
2. MAC-TUTOR HARDWARE	2-1	1	2										
	2-2	1	2										
	2-3	1	2										
	2-4	-	2										
	2-5	1	2										
	2-6	Blank											
	2-7	1	2										
	2-8	Blank											
	2-9	1	-										
	2-10	Blank											
	2-11	1	2										
	2-12	1	2										
	2-13	1	2										
	2-14	1	2										
	2-15	1	2										
	2-16	1	2										
	2-17	1	2										
	2-18	1	2										
	2-19	1	2										
	2-20	Blank											

CONTENTS	PAGE NUMBER	ISSUE NUMBER											
		1	2										
CHAPTER 3 TITLE PAGE													
FRONT	-	1											
REVERSE	-	Blank											
3. MAC-8 ARCHITECTURE	3-1	1	2										
	3-2	1	2										
	3-3	1	2										
	3-4	1	2										
	3-5	1	2										
	3-6	1	2										
CHAPTER 4 TITLE PAGE													
FRONT	-	1	-										
REVERSE	-	Blank											
4. MAC-TUTOR SOFTWARE	4-1	1	2										
	4-2	1	2										
	4-3	1	2										
	4-4	1	2										
	4-5	1	2										
	4-6	1	2										
	4-7	1	2										
	4-8	1	2										
	4-9	1	2										
	4-10	1	2										
	4-11	1	2										
	4-12	1	2										
	4-13	1	2										
	4-14	1	2										
	4-15	1	2										
	4-16	1	2										
CHAPTER 5 TITLE PAGE													
FRONT	-	1	-										
REVERSE	-	Blank											

CONTENTS	PAGE NUMBER	ISSUE NUMBER											
		1	2										
5. SOFTWARE	5-1	12	2										
	5-2	1	2										
	5-3	1	2										
	5-4	1	2										
	5-5	1	2										
	5-6	1	2										
APPENDIX TITLE PAGE													
FRONT	-	-	2										
REAR	-	Blank											
APPENDIX, RESIDENT EXECUTIVE SOFTWARE	A-1	-	2										
	A-2	-	2										
	A-3	-	2										
	A-4	-	2										
	A-5	-	2										
	A-6	-	2										
	A-7	-	2										
	A-8	-	2										
	A-9	-	2										
	A-10	-	2										
	A-11	-	2										
	A-12	-	2										
	A-13	-	2										
	A-14	-	2										
	A-15	-	2										
	A-16	-	2										
	A-17	-	2										
	A-18	-	2										
	A-19	-	2										
	A-20	-	2										
	A-21	-	2										
	A-22	-	2										
	A-23	-	2										
	A-24	-	2										
	A-25	-	2										



MAC-8

ISSUE
2

PA-800515
-A4

FOREWORD

MAC-Tutor has been coded by the Western Electric Company as the No. 1A Microprocessor Training Aid (component code 103180717), but will be called MAC-Tutor throughout this manual.

The following manuals are shipped with the No. 1A Microprocessor Training Aid:

PA-800515 MAC-TUTOR REFERENCE MANUAL
PA-800516 MAC-TUTOR SELF-TRAINING MANUAL
PA-800517 MAC-8 HEXADECIMAL CODING CHART

For questions or comments concerning MAC-Tutor usage, repairs, documentation, and/or to be placed on distribution for future documentation updates, dial the MAC-Phone on CORNET 233, extension MAC8 (6228).

MAC TUTOR REFERENCE MANUAL

CONTENTS

1. SYSTEM OVERVIEW	1-1
1.1 Introduction	1-1
1.2 System Features	1-1
2. MAC TUTOR HARDWARE	2-1
2.1 Functional Description	2-1
2.2 Electrical Characteristics	2-1
2.2.1 MAC-8 Microprocessor and Reset Circuitry (See Figure 2-2.)	2-2
2.2.2 ROM and RAM (See Figure 2-3.)	2-2
2.2.3 I/O (See Figure 2-4.)	2-7
2.2.4 Keypad (See Figure 2-5.)	2-11
2.2.5 PROM Programmer (See Figure 2-6.)	2-12
2.2.6 TTY Terminal and Data Set Interface (See Figure 2-7.)	2-12
2.2.7 Cassette Tape Interface (See Figure 2-8.)	2-15
2.2.8 Power Supply Circuitry (See Figure 2-9.)	2-16
2.2.9 Timing	2-17
3. MAC-8 ARCHITECTURE	3-1
3.1 General Registers	3-1
3.2 Register Pointer	3-1
3.3 Pushdown Stack	3-2
3.4 Addressing Modes	3-3
3.5 Conditions	3-4
3.6 Interrupts	3-4
3.7 Traps	3-5
3.8 Reset	3-5
4. MAC TUTOR SOFTWARE	4-1
4.1 Functional Description	4-1
4.2 Operation	4-1
4.2.1 Keypad/Display	4-1
4.2.2 Keypad Button Control	4-1
4.2.3 TTY Control	4-7
4.2.4 System Utilities	4-11
4.3 Programming	4-11

4.4 Available Programs	4-12
4.4.1 Move Memory - *022F	4-12
4.4.2 Write a PROM - *0541	4-12
4.4.3 Verify a PROM - *057B	4-13
4.4.4 Dump to Audio Tape - *06C6	4-13
4.4.5 Read from Audio Tape - *05EE	4-14
4.5 Testing and Diagnosing	4-16
5. GLOSSARY	5-1

APPENDIX

Resident Executive Program

Chapter 1

SYSTEM OVERVIEW

1. SYSTEM OVERVIEW

1.1 Introduction

The MAC Tutor is a low cost, self-contained, microprocessor-based system developed to familiarize users with microprocessor basics and, in particular, with MAC-8 microprocessor operation.

1.2 System Features

The MAC Tutor contains an on-board keypad and an 8-digit display whereby MAC-8 programs can be entered, executed, and debugged. In addition, the necessary interface is available for various peripheral equipment, such as a teletypewriter (TTY) terminal, a time-sharing computer, or a cassette tape recorder. Figure 1-1 shows the MAC Tutor sections.

MAC Tutor features include:

- MAC-8 microprocessor
- 2K bytes of random-access memory (RAM)
- 2K read-only memory (ROM) executive program to control hardware features
- Sockets for three 1K-byte programmable read-only memories (PROMs)
- Eight 7-segment light-emitting diode (LED) displays
- 28-button, calculator-type keypad
- PROM programming socket capable of creating and verifying Intel 2708-type, 1K-byte PROMs
- Audio cassette interface for storing and retrieving data at a rate of 166 baud
- 32 input/output (I/O) lines with a socket to add another 24 lines
- RS232C interface for TTY-compatible terminals capable of running at rates from 0 to 2400 baud
- Data set interface with software-controlled data direction switch
- Address and data buses available on 16-pin connectors for addition of memory or peripherals
- Ability to single-step program instructions
- On-board power supply (110-volt ac, 60-Hz input required)

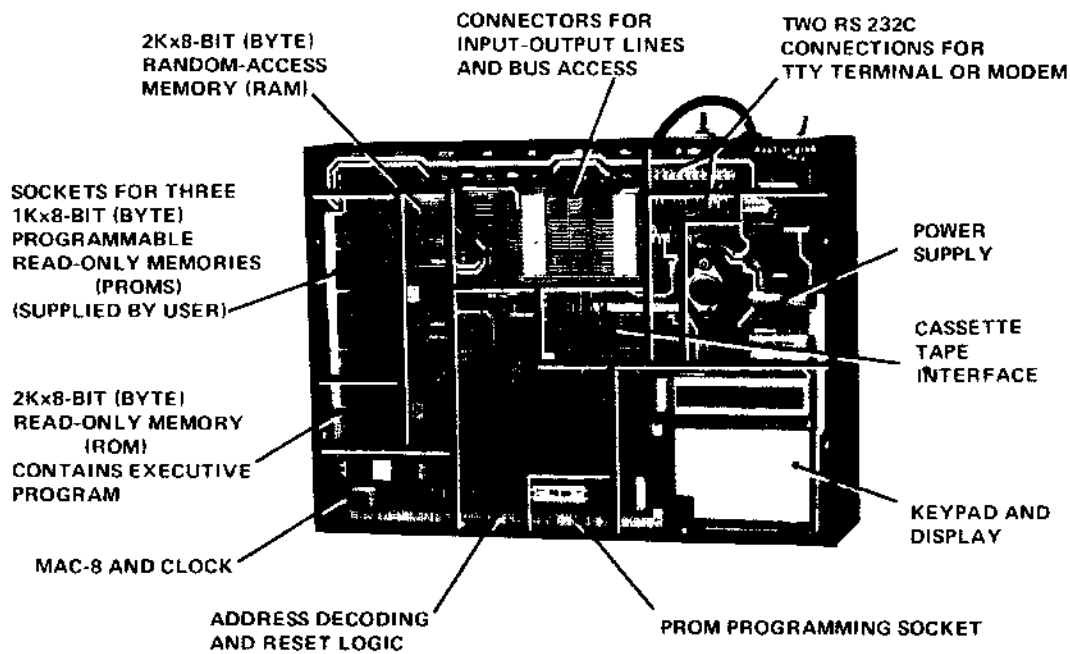


Figure 1-1. MAC Tutor Sections

Note: The material in this manual pertains to ISSUE 4 models (prototype version). ISSUE 4 schematic diagrams are shown in Figures 1-2 and 1-3. ISSUE 3 schematic diagrams, Figures 1-4 and 1-5, are included for reference only.

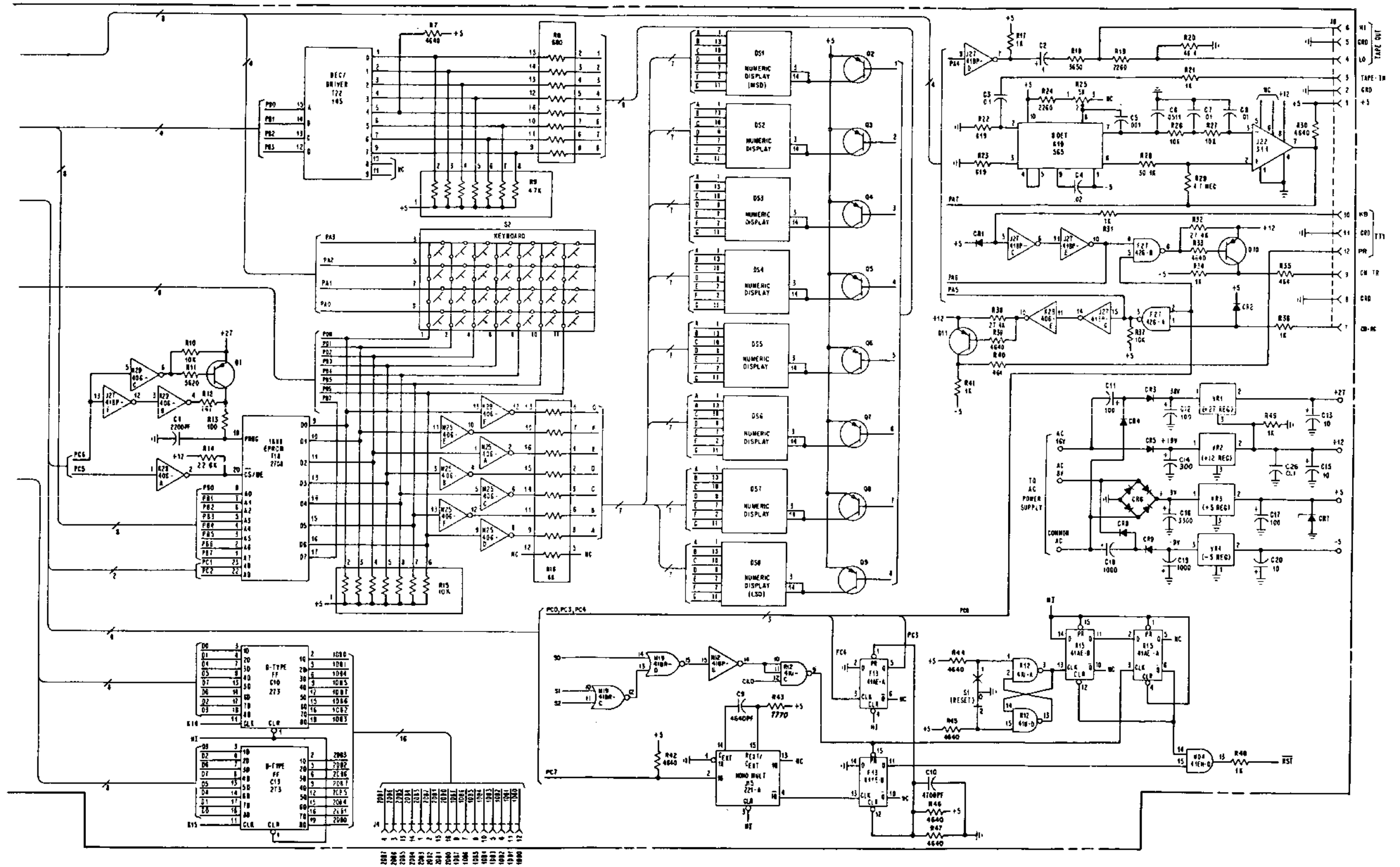


Figure 1-3. MAC Tutor Schematic Diagram
Issue 4, Sheet 2

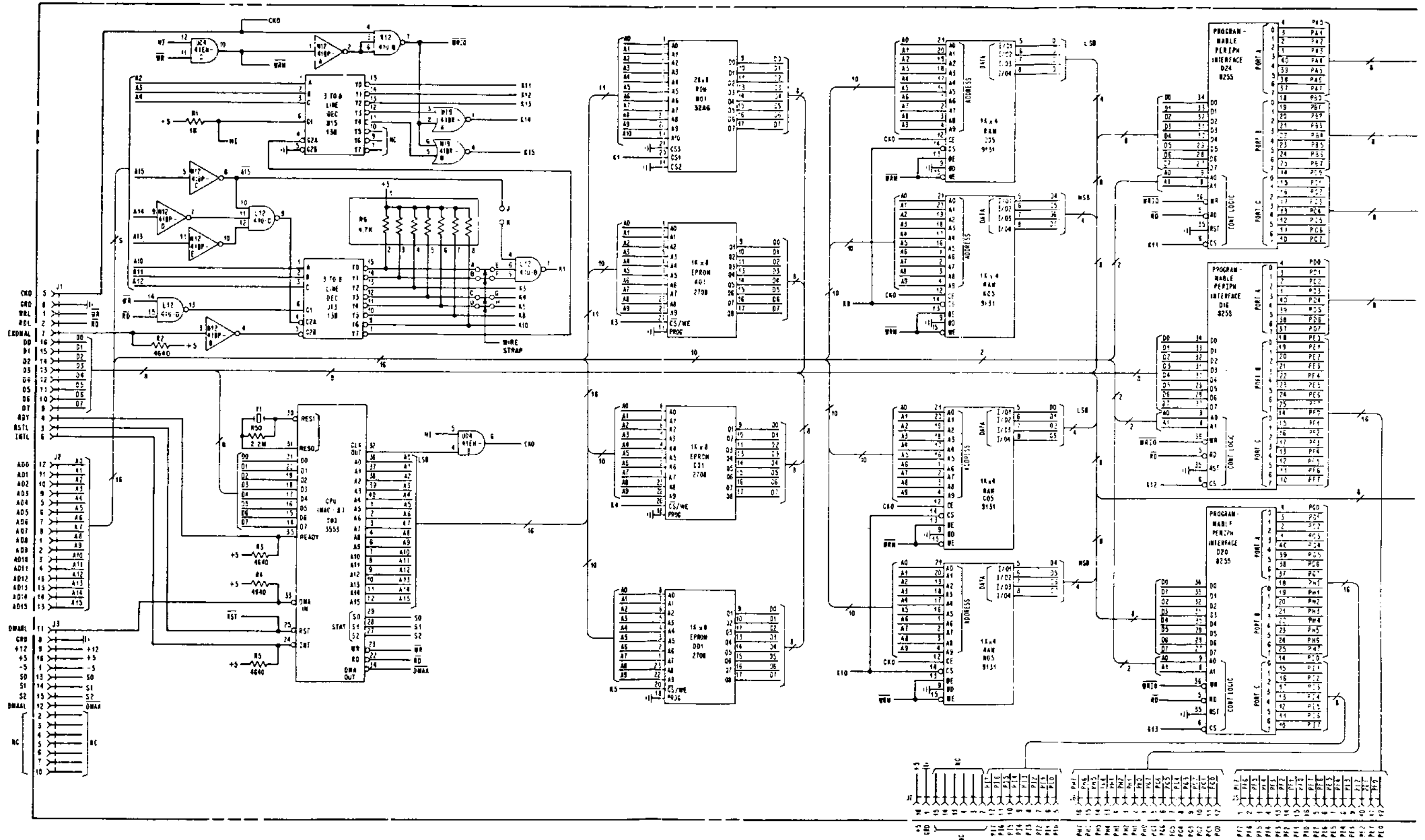


Figure 1-4. MAC Tutor Schematic Diagram
 Issue 3, Sheet 1

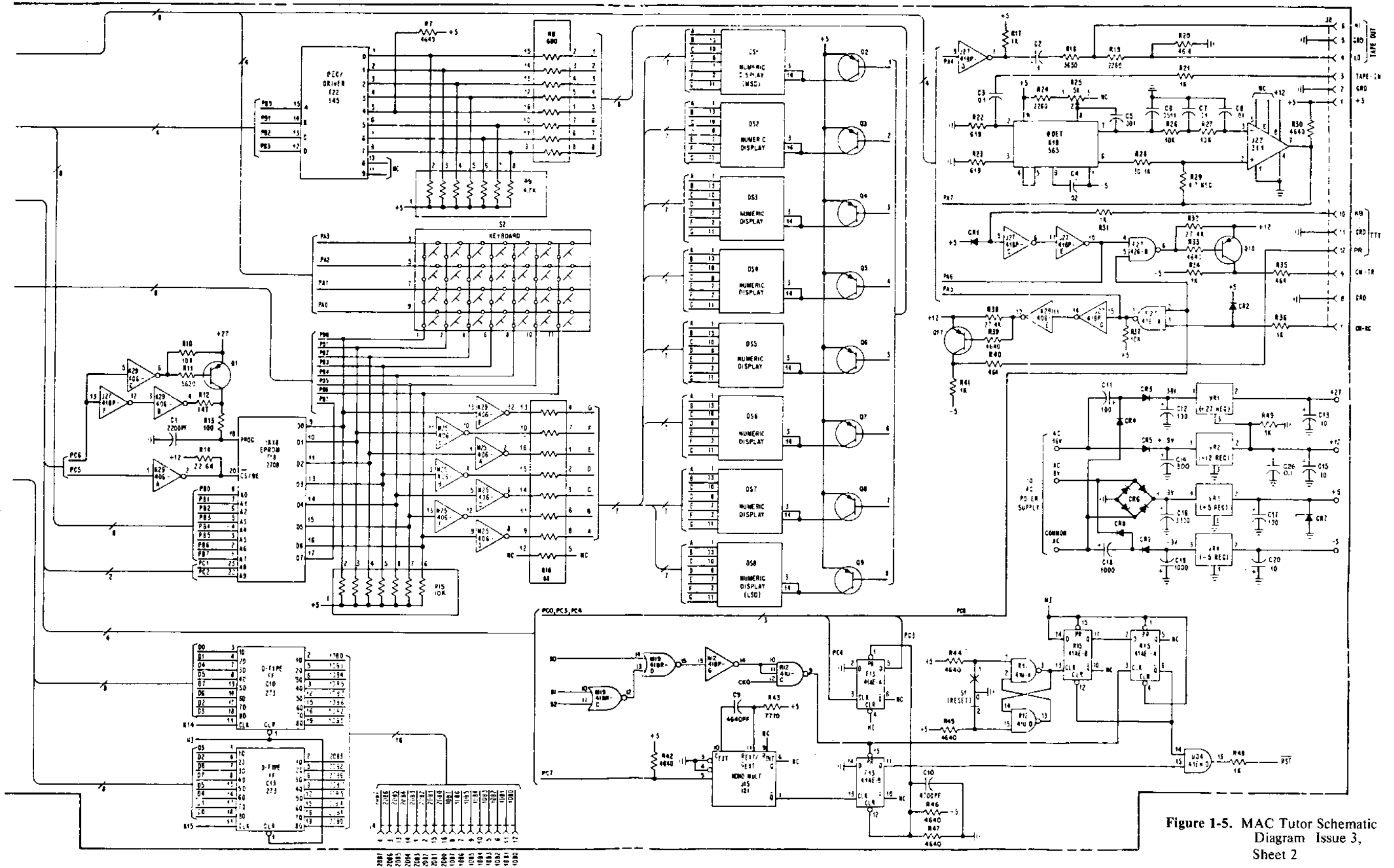


Figure 1-5. MAC Tutor Schematic Diagram Issue 3, Sheet 2

Chapter 2

MAC TUTOR HARDWARE

2. MAC TUTOR HARDWARE

2.1 Functional Description

The MAC Tutor contains a MAC-8 microprocessor and the associated control circuitry to perform the computing and controlling functions for the entire MAC Tutor. Figure 2-1 is a block diagram of the MAC Tutor hardware.

The instructions to be executed by the MAC-8 are contained in the ROM and RAM. The ROM can be mask programmed at the factory or field programmed by inserting a blank PROM into the PROM programmer. The RAM can be read or written directly with the microprocessor.

The 2K-byte ROM (mask programmed) contains an executive program that includes the routines required to drive the display, read the keypad, and communicate with a terminal.

The 1K-byte RAM is used for MAC-8 registers, stack memory, and a user's program. Because this memory is volatile, it must be recorded into a PROM or cassette tape for retention.

Three sockets are provided for 2708-type PROMs, each having a capacity of 1K bytes. These PROMs can be programmed with the on-board programmer, using the separate 24-pin socket. Programs are erased by exposing the PROMs to ultraviolet light.

Users enter and debug their MAC-8 programs by interfacing with the 28-button keypad and eight 7-segment LED displays. Commands to the executive program are issued through the keypad and acknowledged through the display.

Sixty-four I/O lines, with a socket to add another 24 lines, are provided. Thirty-two I/O lines are used for internal operation and the remaining lines terminate at the 16-pin periphery sockets. Sixteen of these lines are transistor-transistor logic (TTL) outputs with an 8-mA current drive. The others that can be programmed as I/O lines are also TTL compatible, but have a 1.6-mA current drive (4 LSTTL Loads).

The computer/TTY data switch allows a remote computer or TTY terminal to communicate with the MAC Tutor.

A commercial quality cassette tape recorder can be used to store and retrieve files by connecting the microphone input and earphone output to the MAC Tutor.

A conventional 110-volt input connects to the on-board power supply, which generates the required voltage levels of ± 5 , +12, and +27 volts dc.

2.2 Electrical Characteristics

The electrical sections of the MAC Tutor are: the MAC-8 and reset circuitry, ROM and RAM, I/O, keypad and display, PROM programmer, TTY terminal and data set interface, cassette tape interface, power supply circuitry, and timing.

2.2.1 MAC-8 Microprocessor and Reset Circuitry (See Figure 2-2.)

Conventionally, the reset input to a CPU resets the program counter to zero and a program begins to execute. However, the MAC-8 CPU also handles the reset input as a nonmaskable interrupt. That is, the status of the CPU is saved before resetting. As a result, the MAC Tutor uses the reset input for a power-on reset, single stepping, and nonmaskable interrupt. The reset button then allows the user to stop the execution of a program and monitor the location and status at that point. This unique feature requires the reset circuit to clock-in on only one reset request.

When the reset button is depressed, the first operation code (opcode) fetch generates the reset and the succeeding opcode fetch disables the reset.

A power-on detect flip-flop (F13-A) serves to distinguish between the reset function and the nonmaskable interrupt. Multivibrator J15 applies the reset signal 25 μ s after a low to high signal transition to provide the single-stepping capability.

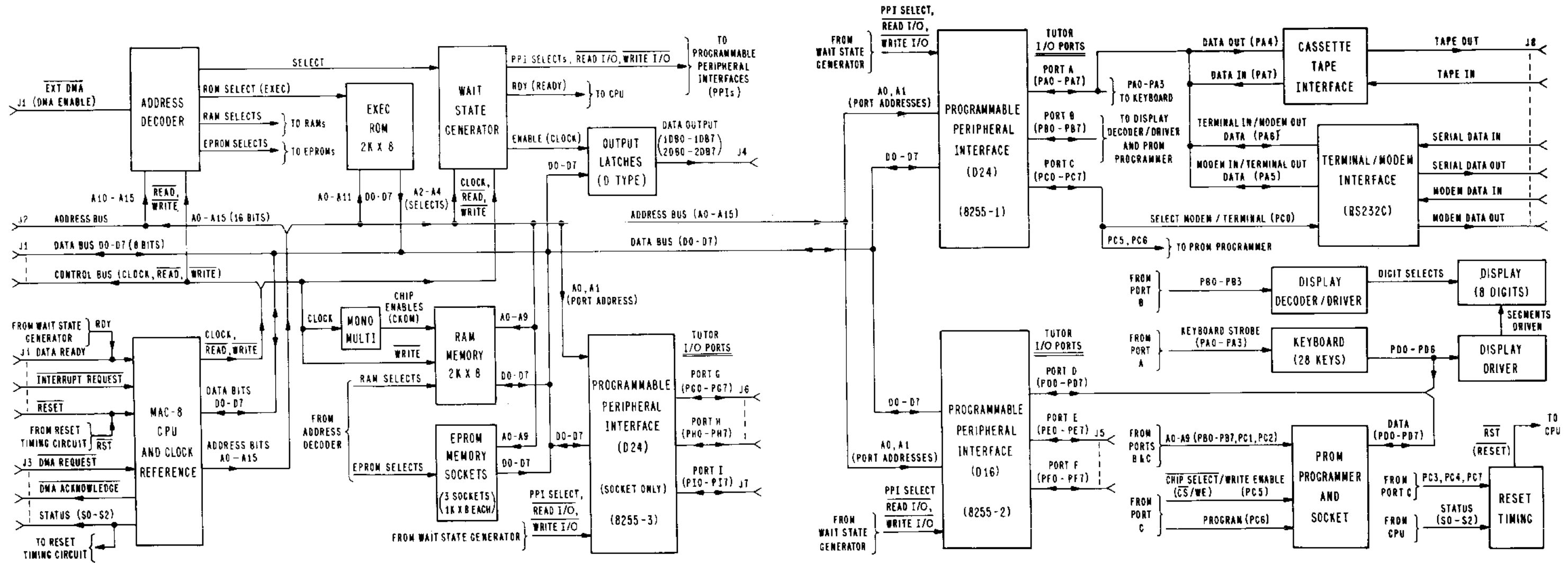
The basic controlling signals for remote access or system expansion are available at connectors J1, J2, and J3. These signals include the address and data buses as well as the +12, +5, and -5 volt dc buses. The 1-kilohm resistor (R48) allows the reset pin to be externally driven.

2.2.2 ROM and RAM (See Figure 2-3.)

The MAC Tutor circuitry is capable of driving one 2K by 8 ROM, three 1K by 8 PROMs, and four 1K by 4 RAMs. The chip-select lines are decoded from the address space through a 3- to 8-line decoder (J13 138). Table 2-1 lists the address assignments provided through these decoders.

The AMD 9131 clocked static RAMs do not need refreshing, but require a clock transition to latch in the address and chip-select signals. The required clock pulse edge is generated by a monostable multivibrator (J15 221-B, one-half of the 74221). The multivibrator is triggered by the falling edge of the clock-out pulse (CKO). Then, after a 400-ms delay, a positive going clock pulse (CKOM) is generated. For 2-MHz operation, a faster clock pulse edge is required.

PA-800515
Issue 2, July 1979



NOTE
Pin information for connectors J1 through J8 is shown in Table 2-2. MAC Tutor Pinouts.

Figure 2-1. Functional Block Diagram of MAC-Tutor

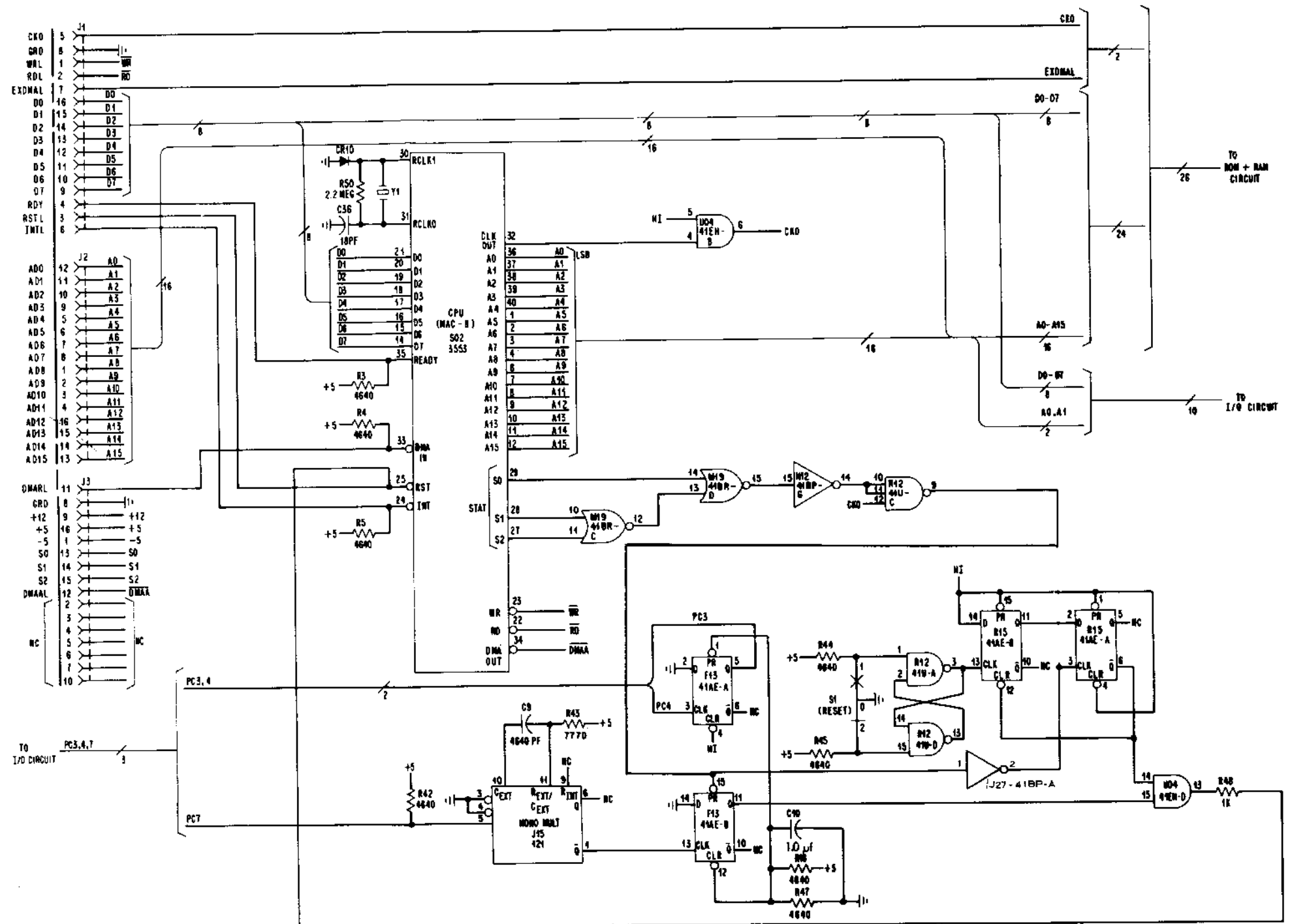


Figure 2-2. MAC-8 Microprocessor and Reset
Circuitry Schematic Diagram

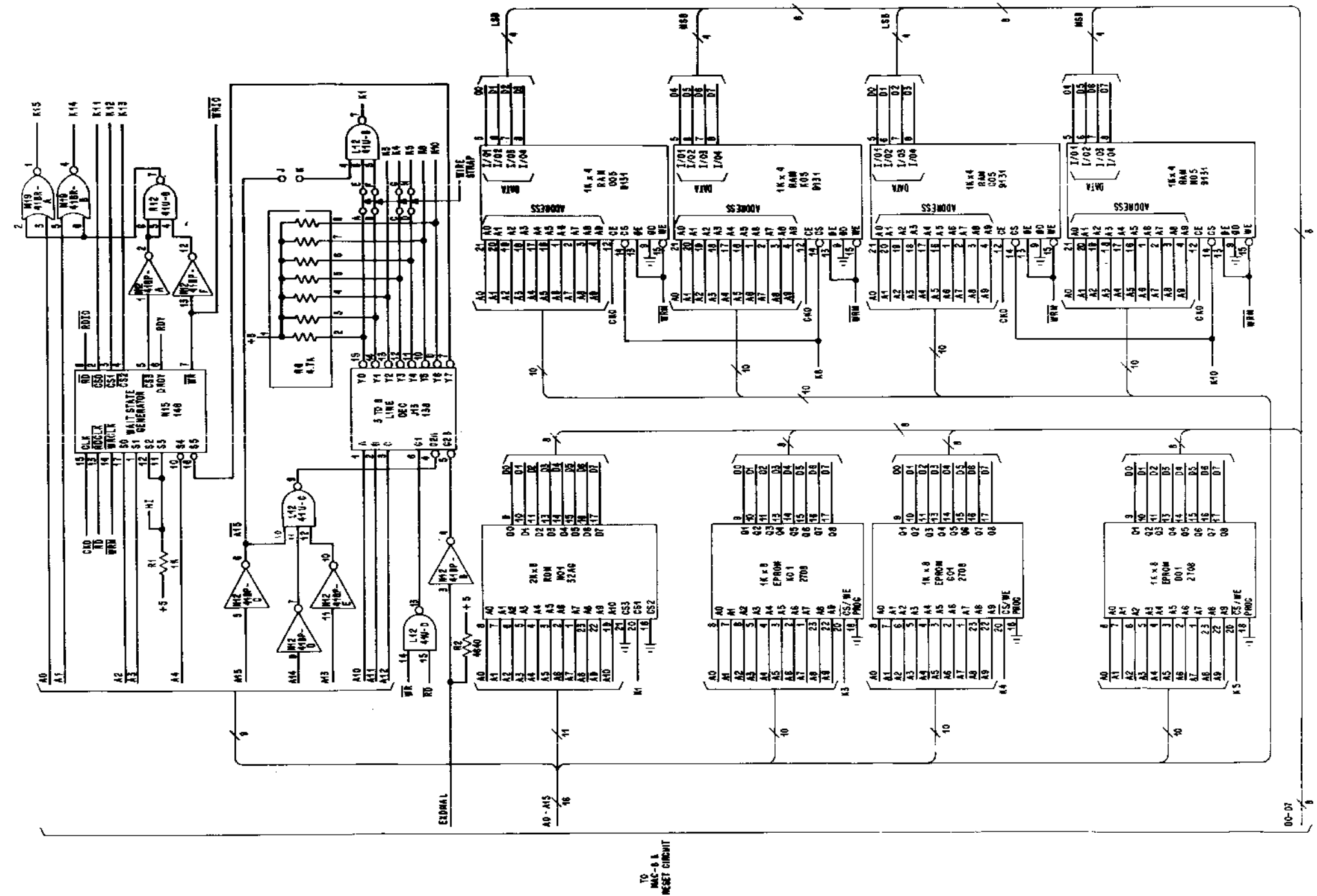


Figure 2-3. ROM and RAM Schematic Diagram

TABLE 2-1. ADDRESS ASSIGNMENTS/MEMORY MAP

Device	Physical Location											Hex Addresses				
		A15	A14	A13	A12	A11	A10	A4	A3	A2	A1		A0			
32AG ROM } 32AAF ROM }	N01	0	0	0	0	0	X									0000-07FF
2708 PROM	K01	0	0	0	0	1	0									0800-0BFF
2708 PROM	G01	0	0	0	0	1	1									0C00-0FFF
2708 PROM	D01	0	0	0	1	0	0									1000-13FF
9131 RAM	D05-K05	0	0	0	1	0	1									1400-17FF
9131 RAM	G05-N05	0	0	0	1	1	0									1800-1BFF
8255 I/O	D24	0	0	0	1	1	1	0	0	0						1F00-1F03
8255 I/O	D16	0	0	0	1	1	1	0	0	1						1F04-1F07
8255 I/O	D20	0	0	0	1	1	1	0	1	0						1F08-1F0B
74LS273 I/O	C10	0	0	0	1	1	1	0	1	1	0	1				1F0D
74LS273 I/O	C13	0	0	0	1	1	1	0	1	1	1	0				1F0E

- Table Notes:
1. X designates either logical 1 or 0. Blank areas indicate future expansion.
 2. Unit comes equipped with one of the two listed ROMs.

Four wire straps connecting points A through D to E through H provide memory assignment flexibility. By interchanging points A and B with C and D, the address of executive ROM is interchanged with that of PROM 2 and PROM 3. This allows the user's PROM to have immediate control under a power-on or reset condition.

A wire strap between points J and K allows an interrupt to the MAC-8 to cause control of the program to transfer to the first location in PROM 1.

The memory configuration can be expanded or replaced by connecting external address signals to the two 16-pin dual in-line package (DIP) connectors, J1 and J2, located at the periphery. The entire memory can be deactivated by keeping EXDMAL (J1, pin 7) low.

2.2.3 I/O (See Figure 2-4.)

The MAC Tutor circuitry is capable of driving three Intel 8255 programmable peripheral interface (PPI) integrated circuits and two 74LS273 octal latches.

Each 8255 PPI has three I/O ports (eight lines per port) that can be programmed as either inputs or outputs. In the output configuration, they can only drive one medium-power TTL load. However, the two output ports provided by the 74LS273 latches have a drive fanout of 10 Low Power Schottky TTL (LSTTL) loads.

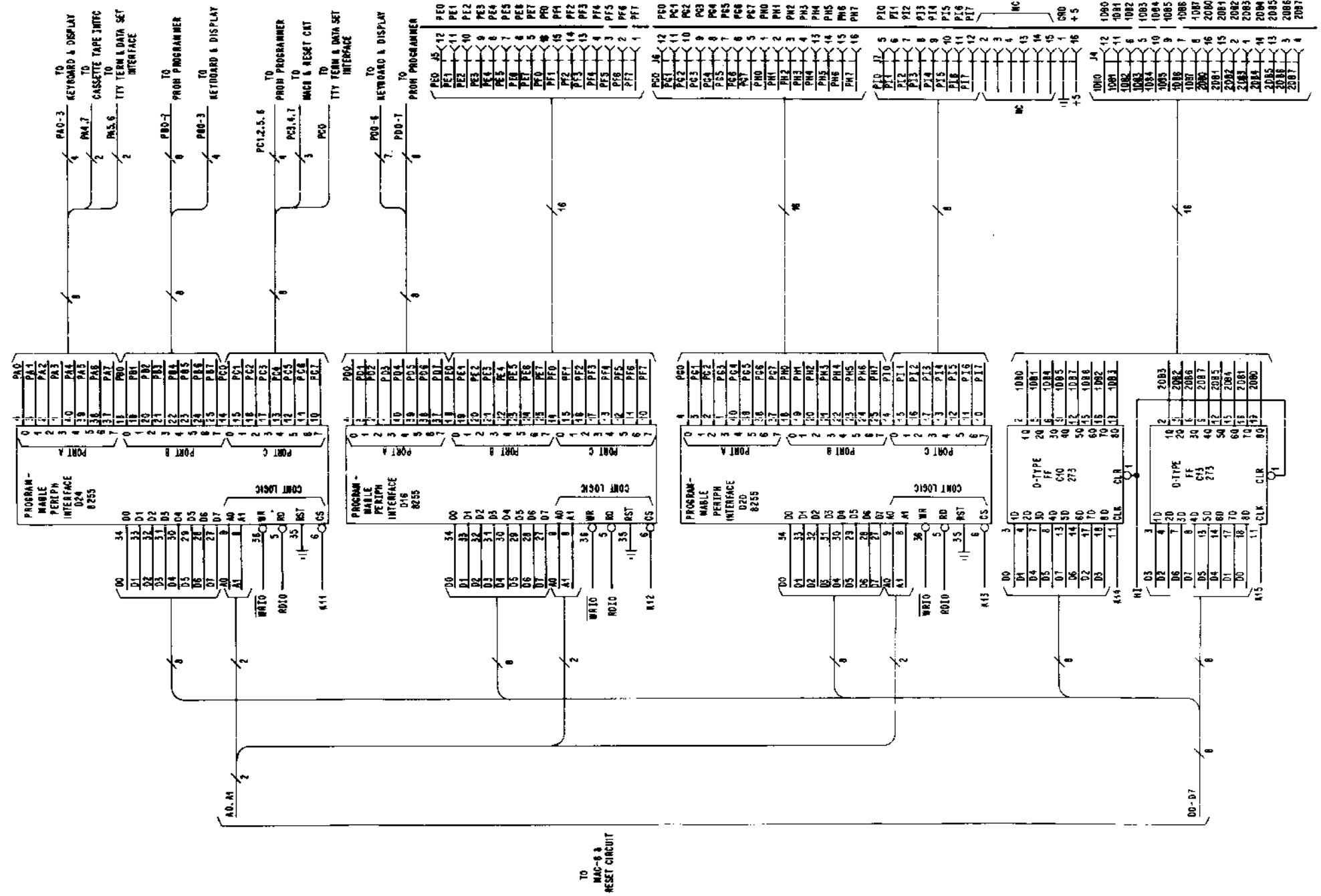


Figure 2-4. I/O Port Schematic Diagram

Four of the I/O ports are used mainly to drive the keypad display, and PROM. Additionally, two 8255 I/O ports, or five 8255 I/O ports when fully equipped, are available at connectors J5 through J7. The remaining two 74LS273 ports are available at connector J4.

A wait state generator integrated circuit (WE-146D) provides the required decoding and timing for the I/O devices. Refer to Figure 2-3 for circuit details.

Table 2-2 contains a listing of all the I/O pinouts. (This information is also included on Figures 1-2 and 1-3.)

TABLE 2-2. MAC Tutor Pinouts

Pin Number/Connector								Connector J8	
J1	J2	J3	J4	J5	J6	J7	Pin	Designation	Pin No.
A00	A04	A08	A12	A16	A20	A24	No.		
WRL	AD8	-5	2DB3	PF7	PH0	GRD	1	+5 VOLTS	1
RDL	AD9	NC	2DB2	PF6	PH1	NC	2	GRD	2
RSTL	AD10	NC	2DB6	PF5	PH2	NC	3	TAPE-IN	3
RDY	AD11	NC	2DB7	PF4	PH3	NC	4	TAPE-OUT-LO	4
CKO	AD4	NC	1DB3	PE7	PG7	PI0	5	GRD	5
INTL	AD5	NC	1DB2	PE6	PG6	PI1	6	TAPE-OUT-HI	6
EXDMAL	AD6	NC	1DB6	PE5	PG5	PI2	7	CM-RC	7
GRD	AD7	GRD	1DB7	PE4	PG4	PI3	8	GRD	8
D7	AD3	+12	1DB5	PE3	PG3	PI4	9	CM-TR	9
D6	AD2	NC	1DB4	PE2	PG2	PI5	10	TTY-KB	10
D5	AD1	DMARL	1DB1	PE1	PG1	PI6	11	GRD	11
D4	AD0	DMAAL	1DB0	PE0	PG0	PI7	12	TTY-PR	12
D3	AD15	S0	2DB5	PF3	PH4	NC	13		
D2	AD14	S1	2DB4	PF2	PH5	NC	14		
D1	AD13	S2	2DB1	PF1	PH6	NC	15		
D0	AD12	+5	2DB0	PF0	PH7	+5	16		

2.2.4 Keypad (See Figure 2-5.)

The keypad includes a 4 by 7 array of switches that is read with a strobing algorithm. Each row is strobed with a logical 0 signal and the state of the seven columns is read. Since the column outputs are converted to logic highs by a set of resistors (R15), a keypad depression in a particular column will cause a logical 0 reading at that input line. Strobing is repeated for the four rows so the MAC-8 can determine the state of the keypad.

The display contains eight 7-segment LED displays where digits are multiplexed in time and driven by common segment drivers. The same lines (PD0 through PD6) that are used to read

the keypad also drive the segments. Output lines PB0 through PB3 are decoded to select the appropriate digit.

2.2.5 PROM Programmer (See Figure 2-6.)

The programming procedure for 2708 PROMs requires the following:

- Initiate write enable by applying 12 volts to \overline{CS}/WE pin.
- Sequence the address space of the 2708 PROM and apply data to be programmed for each address.
- When the address and data are valid, apply a 27-volt pulse of 1-ms duration to PROGRAM pin throughout the address sequence.
- Repeat address sequence 100 times.

A mix of software and hardware is used to implement the preceding procedure. High-level timing and control are done in software. The hardware has the 12-volt driver for the write enable signal and the 27-volt driver for the program pulse. This program pulse is generated by the resistance-capacitance (RC) circuit (R12, R13, C1) to produce a 1- μ s rise and fall time level.

The PROM address and data lines are driven directly from the I/O ports so the MAC-8 can sequence through the address and data, and control the high-voltage drivers. After device programming, the MAC-8 is able to read the PROM if a low-level signal is coupled to the \overline{CS} pin. This allows the PROM to be verified prior to programming for an erased condition (all 1s) and after programming for programmed contents.

2.2.6 TTY Terminal and Data Set Interface (See Figure 2-7.)

When a TTY terminal is connected to the MAC Tutor, all operations provided from the on-board keypad/display can be controlled from the TTY terminal. The interface to the TTY terminal is through a serial I/O line under direct control of the MAC-8. The MAC Tutor adapts to the baud rate of the terminal (up to 300 baud automatically and manually to 2400). Data can also be accepted from a remote computer through a telephone line when a modem is connected. A built-in, software-controlled data switch allows one of two configurations to be selected. In one configuration, the TTY terminal is fully connected to the modem with the MAC Tutor in the listening mode. In the other configuration, the TTY terminal is connected to the MAC Tutor and the modem is switched out. Both configurations are selected from the TTY terminal. Table 2-3 lists the TTY terminal and data set interface connections.

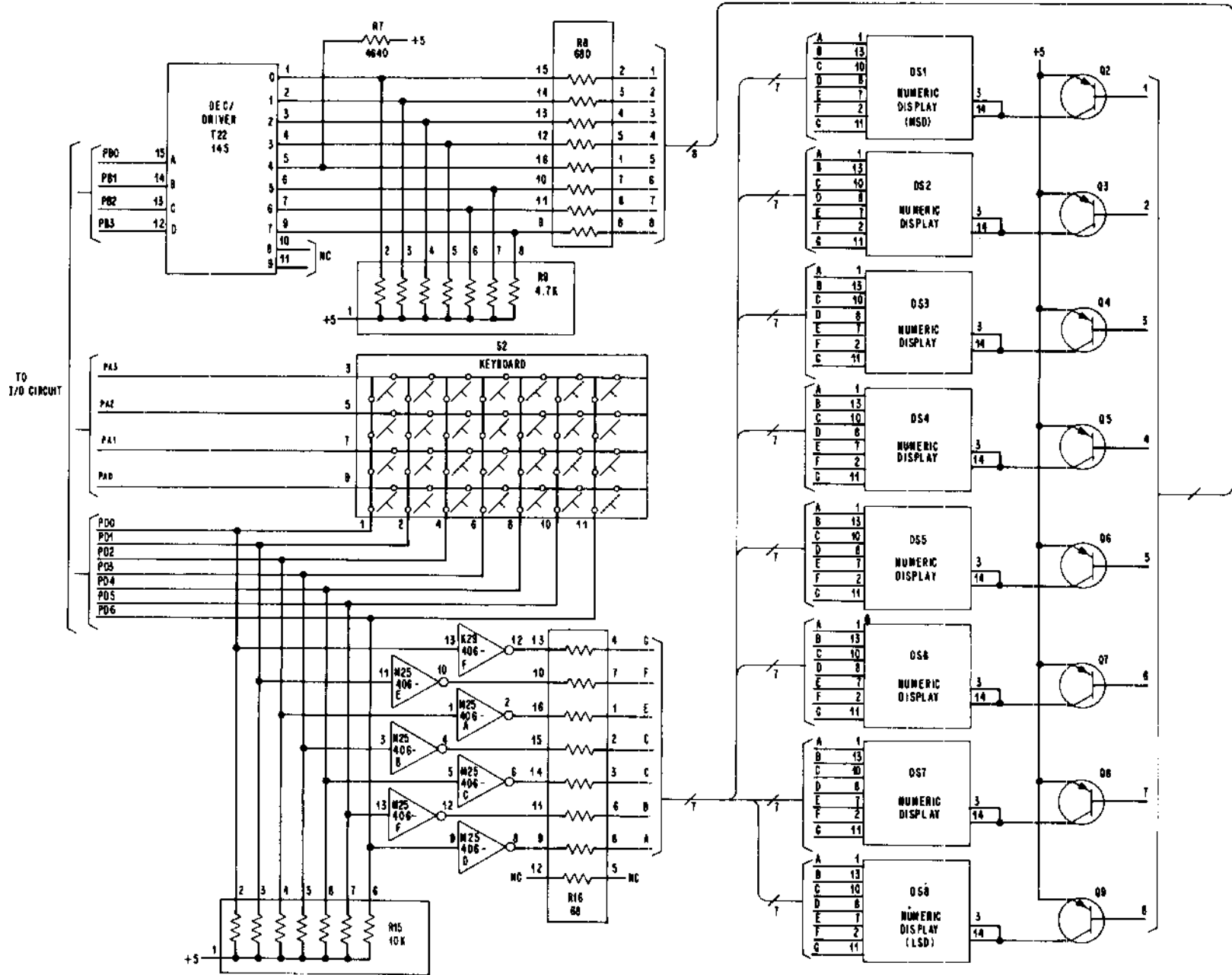


Figure 2-5. Keypad and Display Schematic Diagram

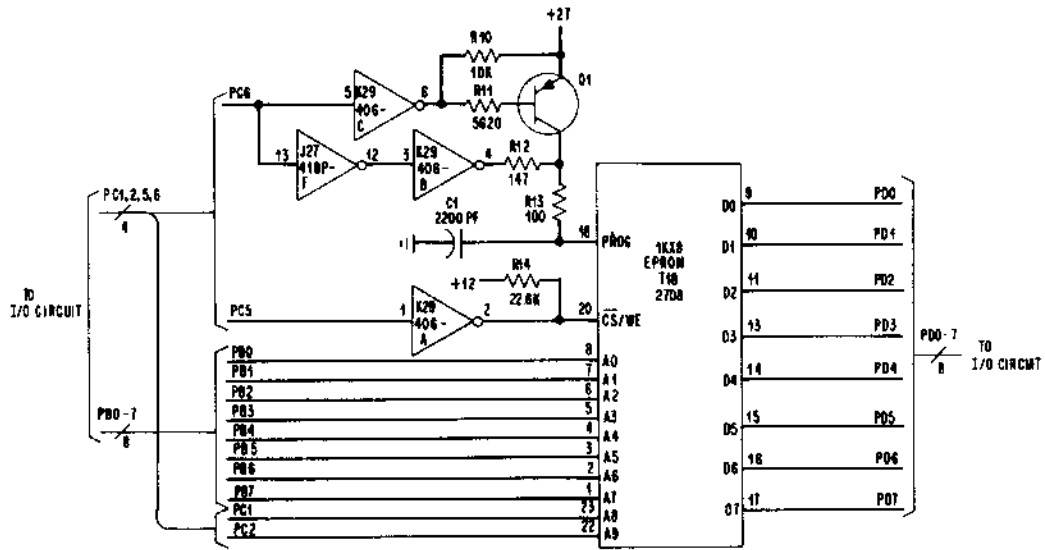


Figure 2-6. PROM Programmer Schematic Diagram

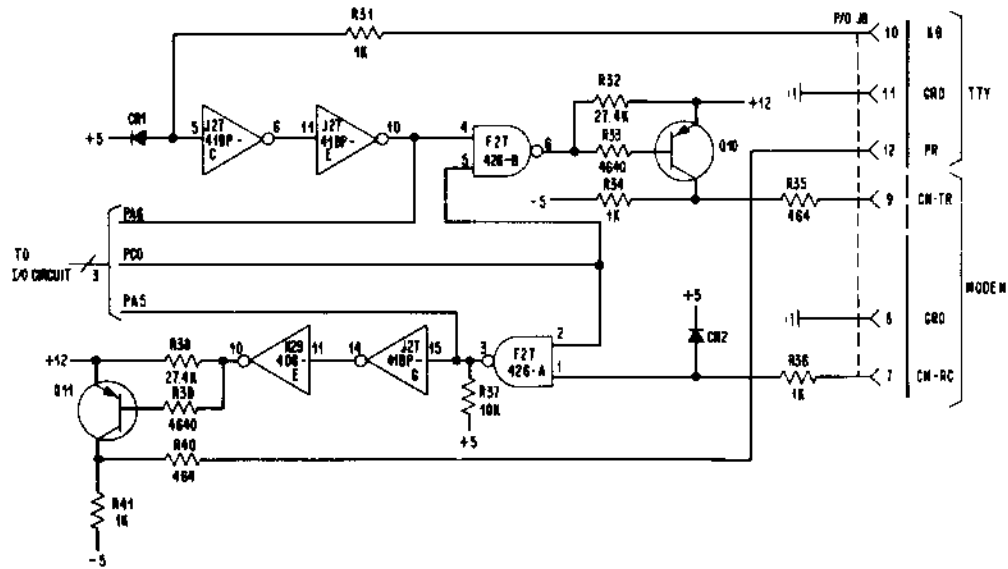


Figure 2-7. TTY Terminal and Data Set Interface Schematic Diagram

TABLE 2-3. TTY TERMINAL AND DATA SET INTERFACE CONNECTIONS

MAC Tutor Connector	TTY Terminal Connector
J8 (RS-232C level compatible)	25-pin interface connector (RS-232C level compatible)
Pin 10 - Terminal/Keypad Pin 12 - Terminal/Printer Pin 11 - Ground	Pin 2 - Terminal/Keypad Pin 3 - Terminal/Printer Pin 7 - Ground
MAC Tutor Connector	Modem Connector
J8 (RS-232C level compatible)	25-pin interface connector (RS-232C level compatible)
Pin 9 - Modem Transmitter Pin 7 - Modem Receiver Pin 8 - Ground	Pin 2 - Transmitter Pin 3 - Receiver Pin 7 - Ground

Table Note: In addition, some of the pins on the TTY terminal connector may be required to be strapped together for proper operation. Typically, pins 4, 5, 6, and 8 should be strapped together.

2.2.7 Cassette Tape Interface (See Figure 2-8.)

A cassette tape recorder microphone input and earphone output can be connected to the MAC Tutor to read and write data.

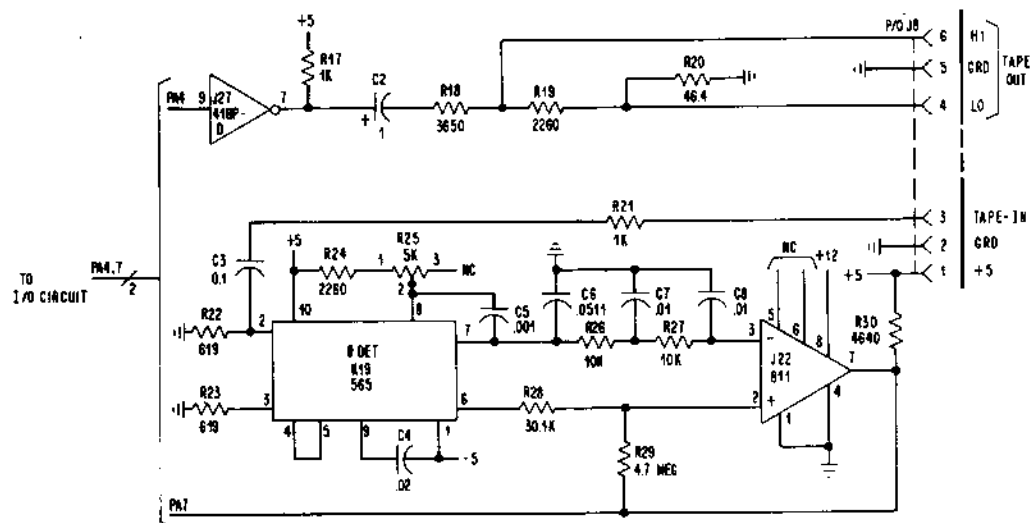


Figure 2-8. Cassette Tape Interface Schematic Diagram

To write data, the MAC Tutor generates a frequency shift keying (FSK) signal that alternates between 2000 and 4000 Hz. When a logical 0 is written on the tape, 2000 Hz appears for two-thirds of the bit time and 4000 Hz for one-third of the bit time. When a logical 1 is written, 2000 Hz appears for one-third of the bit time and 4000 Hz for two-thirds of the bit time.

To read data, an LM565 phase-lock loop integrated circuit (IC) with a free-running frequency of 3000 Hz locks on the input signal. The input voltage to the voltage-controlled oscillator

(VCO), which is available from the LM565 IC, indicates what frequency is being received. This signal is then passed through an RC filter to eliminate the carrier frequencies, while retaining the modulating signal. A comparator converts this low-level signal to a TTL signal for MAC-8 input. The MAC-8 synchronizes to the bit pattern by detecting the negative transition (from 4000 to 2000 Hz) and determines the state of the bit transmitted by the incoming waveform duty cycle.

The operating baud rate is 166 bits per second to ensure low error rates and portability of tape cassettes from one recorder/MAC Tutor to another recorder with a different MAC Tutor. The cassette tape recorder interface connections are listed in Table 2-4.

TABLE 2-4. CASSETTE TAPE RECORDER INTERFACE CONNECTIONS

MAC Tutor Connector	Cassette Tape Recorder Connector
J8, Pin 6 - TAPE-OUT-HI	MICROPHONE JACK
J8, Pin 5 - GROUND	MICROPHONE JACK GROUND
J8, Pin 3 - TAPE IN	EARPHONE JACK
J8, Pin 2 - GROUND	EARPHONE JACK GROUND

Table Note: An additional pin designated TAPE-OUT-LO (pin 4) is provided for cassette tape recorders that require a low-level input to the microphone jack.

2.2.8 Power Supply Circuitry (See Figure 2-9.)

The 117-volt ac line is stepped down by a 16-volt ac center-tapped transformer and four dc voltage outputs are generated, as indicated in Table 2-5.

Two voltage doubler circuits are used to generate the +27 and -5 voltage levels. The +27 voltage doubler circuit operates by charging capacitor C11 through diode CR4 on the negative half-cycle. On the positive half-cycle, CR4 becomes reverse-biased and the conducting path is through C11, CR3, and C12. Therefore, the voltage on C11 gets added to the ac voltage to effectively double the dc output voltage. Regulator VR1 is a 3-terminal, -15 volt regulator that uses the 12-volt supply as a reference. By adding this 15-volt supply to the 12-volt supply, the required 27-volt supply is obtained.

The 5-volt supply uses a full-wave bridge rectifier due to the high current requirement.

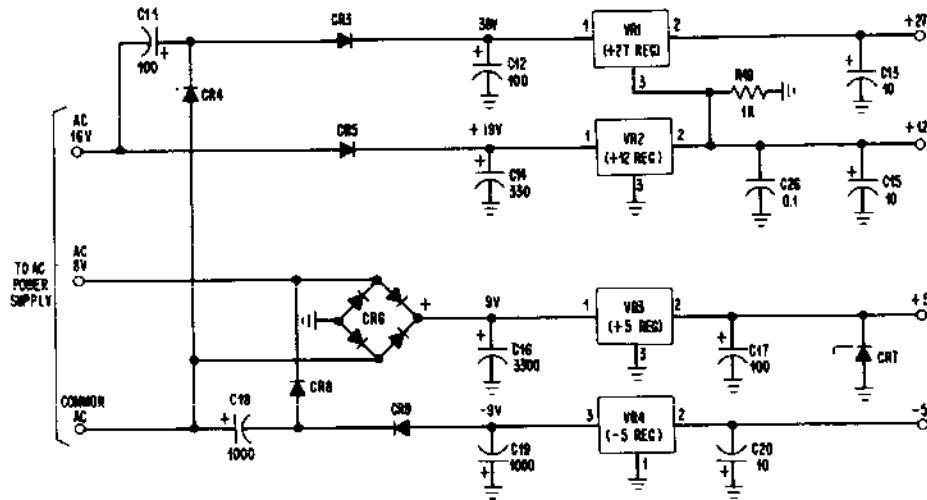


Figure 2-9. Power Supply Schematic Diagram

TABLE 2-5. POWER SUPPLY VOLTAGE AND CURRENT RATINGS

Voltage	Current Rating
+5Vdc	1.5A
-5Vdc	120 mA
+12Vdc	250 mA
+27Vdc	20 mA

Table Note: The 5-volt supply has 350 mA of spare current available at J8, pin 1 to drive the external logic.

2.2.9 Timing

Several factors are involved in the execution time (as defined in terms of microprocessor clock cycles) of an instruction. In one clock cycle a byte can be read from memory, a byte can be written into memory, or some internal function can be accomplished. To minimize requirements on the memory response time, there is a pipeline processor internal to the MAC-8 that imposes a lower bound on the total execution time of any instruction. A simple no outpulsing (NOP) instruction requires four cycles for completion and most instructions are multibyte to ensure that minimum time is used effectively. Timing detail diagrams include the following:

- Fast Memory Accessing, Figure 2-10
- Slow Memory Accessing, Figure 2-11
- A Wait State Generator, Figure 2-12.

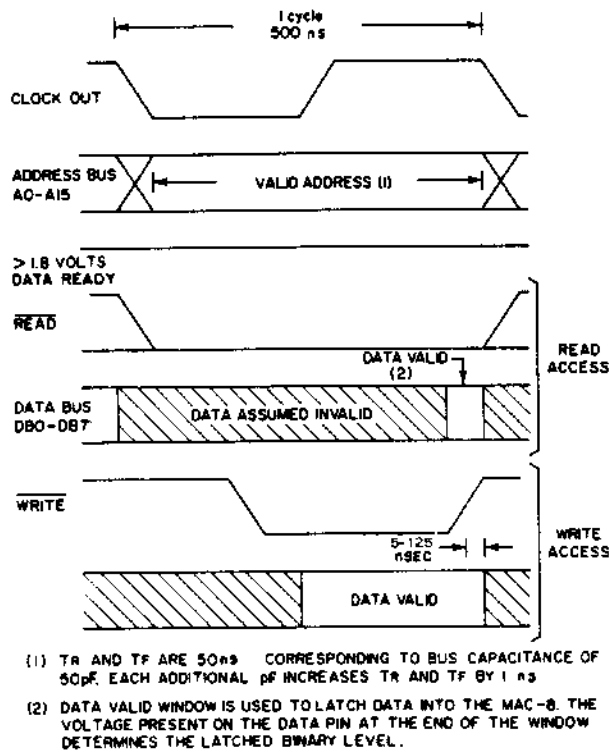


Figure 2-10. Fast Memory Accessing

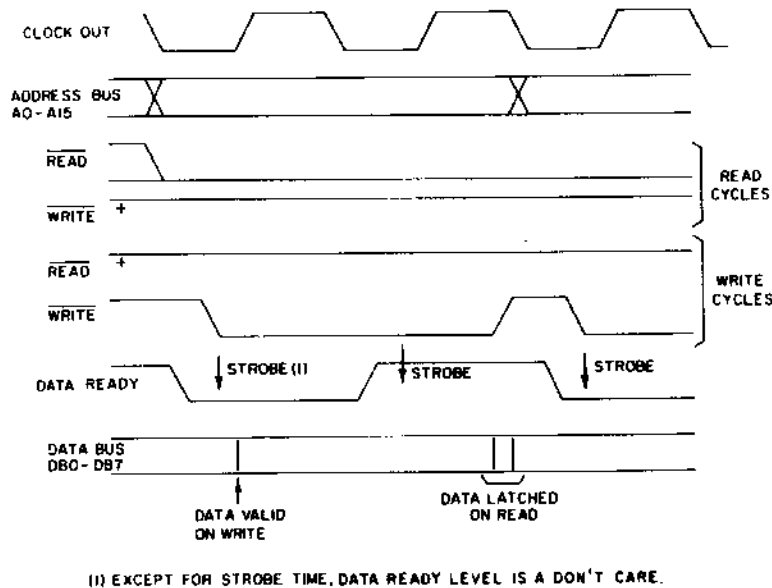
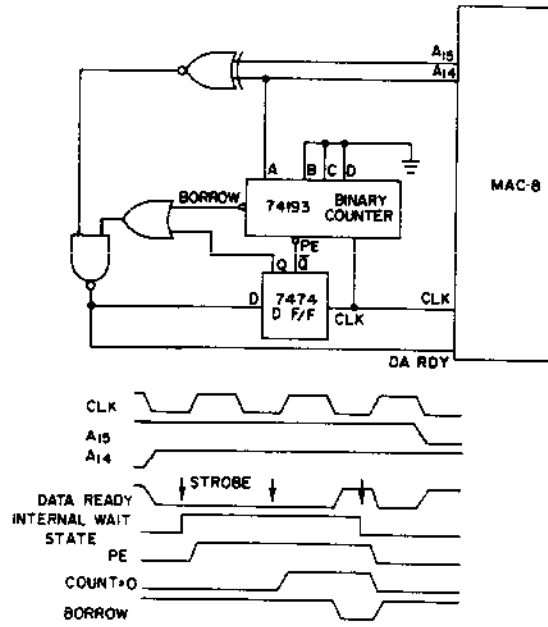


Figure 2-11. Slow Memory Accessing



* FOR A SINGLE WAIT STATE GENERATOR, THE COUNTER CAN BE ELIMINATED WITH Q BEING CONNECTED AS AN INPUT OF THE 2 INPUT NAND GATE.

Figure 2-12. A Wait State Generator

Chapter 3

MAC-8 ARCHITECTURE

3. MAC-8 ARCHITECTURE

The MAC-8 is a byte-oriented, general purpose microprocessor in which the instruction repertoire emphasizes Boolean logical and integer arithmetic operations on 8-bit quantities. These instructions are supplemented by 16-bit operations chosen to facilitate address arithmetic.

Because the MAC-8 is a 2-address microprocessor, typical instructions for dyadic operations such as addition specify only two operands, the augend and addend. By convention, one of the operands is also the destination of the result. To distinguish the operands, one is called the source and the other is the destination, even though both are operand sources for dyadic operations. For monadic operations such as incrementing, there is only one operand, called the destination, which is also the source.

A set of memory-addressing modes is available for accessing up to the maximum of 65,536 bytes of storage. These modes, together with a set of identical general purpose registers, are used to form a highly symmetrical set of operand combinations for the instructions. The same memory-addressing modes are used to specify the destinations of control transfer instructions.

A pushdown stack is used as the subroutine call/return mechanism and allows dynamic storage management. Interrupts allow the processor to respond to unusual events in periphery.

3.1 General Registers

There are 16 general registers available to the MAC-8 at any given time that can be accessed in two different ways:

- As a 16-bit base register (b register) used primarily to hold memory addresses.
- As a low-order, 8-bit accumulator (a register) for arithmetic and logical operations.

When the register is used as an a register, only the low-order byte participates. Some operations, such as addition, can be performed with either the 8-bit or 16-bit register set. Certain operations, such as negation, can be performed only with an a register.

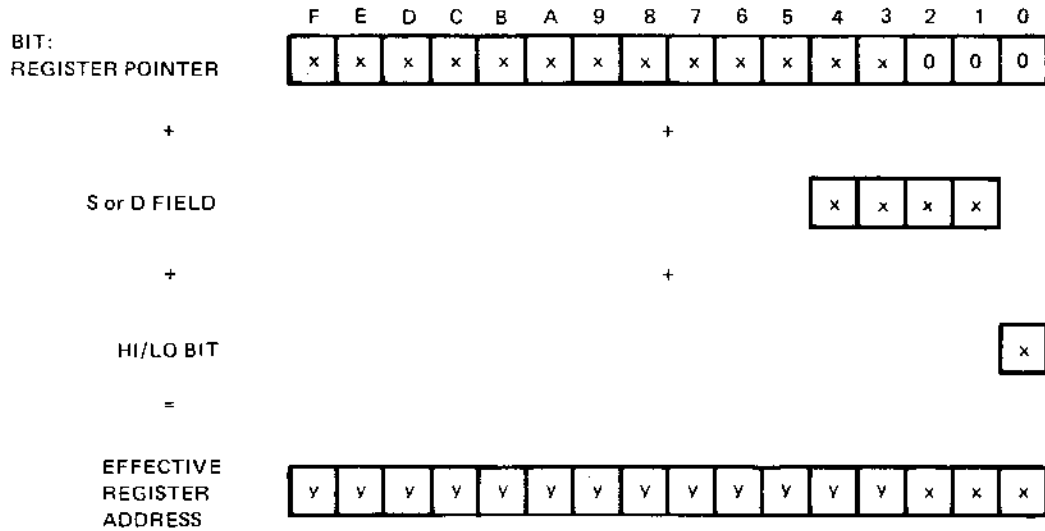
3.2 Register Pointer

The MAC-8 general purpose registers, unlike those of most computers, are not special hardware registers located in the microprocessor. A 32-byte section of regular memory is used as the register set. The first two bytes of this section are register 0, the next two are register 1, etc. The starting address of this section (which must be in writable memory) is contained in a 16-bit, on-chip register called the register pointer (rp). By changing the address in the rp, under program control, the user can locate the general registers anywhere in the memory space. The rp can be thought of as pointing to a movable 32-byte window in the memory space (a window through which the MAC-8 "sees" the register set).

The three low-order bits of the rp are always zero. For each instruction that accesses a general register, the complete effective address of the register is computed from the current value in the rp and the source or destination qualifier field of the instruction. Also included is a bit supplied by the MAC-8 designated as the HI/LO bit. The HI/LO bit determines whether the high-

or low-order byte of the 16-bit register is being addressed. The formation of the effective address is shown in Table 3-1. Notice that the three quantities are aligned as shown and added, each being treated as an unsigned integer.

TABLE 3-1. EFFECTIVE REGISTER ADDRESS



The bump and debump instructions can be used to add or subtract a 1 to bit 3 or 4 of the rp. The effect is to move the general register window up or down in memory by 8 or 16 bytes, respectively, corresponding to a change of four or eight 16-bit registers. The effect is to introduce a new set of registers that partially overlaps the previous set. This makes it possible to save and restore the contents of the register set without actually moving any data.

3.3 Pushdown Stack

The stack pointer (sp) can be used to implement a last-in, first-out queue or "pushdown stack." The sp points to the top of the stack (the last item pushed on or the next item to be popped off). Since only the top item and those under it are valid, items above the top of the stack should not be used. An item is pushed onto the stack by decrementing the sp by 1 or 2, depending on the length of the item, and storing the item at the new address. Conversely, an item is popped off the stack by incrementing the sp by 1 or 2, depending on the length of the item. The item may or may not be moved somewhere before the sp is incremented.

In purely software terms, it does not matter whether pushing something onto the stack increments or decrements the sp, as long as pushes and pops are complementary. In the MAC-8, a push decrements and a pop increments, i.e., the stack grows downward in memory because this arrangement often facilitates systemwide memory allocation. In any case, the term "top of the stack" always refers to the *logical* top of the stack, whether or not this represents the highest absolute address.

The most common use for the pushdown stack is in calling subroutines. Since the dynamic nature of nested subroutine calls corresponds exactly to the action of a stack, a call is a push and a return is a pop. The MAC-8 uses the stack to save and restore the program counter (pc) when subroutines are called and when interrupts are accepted. In the latter case, the condition

register (cr) is also saved on the stack. The depth of nesting of subroutines, plus interrupts, is limited only by the amount of memory allocated to the stack. In addition to these automatic uses of the stack, the executing program can use explicit push and pop instructions to place subroutine parameters and temporary variables on the stack. This use is facilitated by several special addressing modes that allow easy access to items at or near the top of the stack.

3.4 Addressing Modes

The addressing modes of an instruction are the different ways in which the effective addresses of the operands of the instruction are formed. Some instructions do not address memory and therefore have no modes.

Generation of a memory address usually involves one of the b registers. The b register (0 through 15) is specified in a 4-bit field of the instruction, called the s field for the source and d field for the destination. There are eight modes, with each mode representing a way of determining a source operand address and a destination operand address. To extend the MAC-8 addressing capability, s and d fields of register 15 often have special interpretations. In addition, mode 4 (memory-to-memory mode) is presently implemented only for 8-bit operations.

In summary, the three factors that determine how an operand address is calculated are as follows:

- The mode number (0 through 7).
- Whether the operand is the source or the destination.
- Whether or not the specified register is 15.

Refer to Table 3-2 for a list of addressing modes.

TABLE 3-2. ADDRESSING MODES

Addressing Mode	Source		Destination	
	s!=15	s==15	d!=15	d==15
0	Rs	*pc	Rd	R15
1	Rs	*pc	*Bd	**pc
2	Rs	*pc	*(Bd+n)	(SP+n) [*pc+n]
3	Rs	*pc	*Bd++	*B15++
4	*(Bs+n1)	*(sp+n1)	*Bd+n2)	*(sp+n2)
5	*Bs	**pc	Rd	R15
6	*(Bs+n)	*(sp+n)	Rd	R15
7	*Bs++	*B15++	Rd	R15

Table Key:

- B - The contents of a 16-bit base register
- d - The destination operand qualifier (d field)
- n, n1, n2 - An 8-bit signed displacement
- pc - The contents of the program counter
- R - A 16-bit b register for 16-bit operations or an 8-bit a register for 8-bit operations
- s - The source operand qualifier (s field)
- sp - The contents of the stack pointer
- ++ - Indicates a post increment of the b register
- [] - Special interpretation for transfer instructions

An external device requests an interrupt by setting the MAC-8 interrupt request pin. If the enable condition in the MAC-8 is 0, it will ignore the request because it is in a masked condition. If interrupts are enabled and a request is received, the following sequence occurs at the completion of the instruction being executed:

- The *cr* is pushed into the stack.
- The *pc*, which contains the address of the instruction that would have been next executed, is pushed onto the stack.
- The enable condition is set to 0.
- The MAC-8 performs a normal read operation, addressing location X(FFFF). In most applications, this address will not represent regular memory, but will serve as an interrupt acknowledgment to the interrupting device. The data byte read by the MAC-8 is supplied by the device and is used in the next step.
- The data byte read is right-adjusted with leading zeros placed in the *pc*. The next instruction is then taken from that location.

The value placed on the data bus by the interrupting device is effectively a pointer to an instruction in the first 256 bytes of memory. This should be the first instruction of the routine to process that particular type of interrupt. Depending on the application, there can be one or many interrupt handling routines.

It is the responsibility of the interrupt handler to save other registers (if necessary) before processing the interrupt. At completion of the routine, saved registers are restored and a return from interrupt instruction is executed, causing resumption of the program that was executing when the interrupt was accepted. Except for possible changes made by the interrupt handler, the state of the microprocessor will be identical to that before the interrupt was accepted.

3.7 Traps

A trap occurs when the MAC-8 controller has no valid transition defined for the present state and present inputs. This situation can develop when the MAC-8 attempts to execute an invalid opcode, when electrical transients disrupt the controller, or when a fault develops in the controller. However, not all transients and faults will cause a trap. Also, traps cannot be masked.

When a trap condition is recognized, the sequence occurs as follows:

- The *cr* is pushed onto the stack.
- The *pc*, which points two bytes beyond an invalid opcode byte, is pushed onto the stack.
- The enable condition is set to 0.
- The *pc* is set to X(0008) and the next instruction is taken from that location.

Location X(0008) should contain the first instruction of a routine to handle traps. The address of the interrupted instruction (which may have an invalid opcode) can be calculated from the saved *pc*.

3.8 Reset

An external device resets the MAC-8 by setting the reset pin. When this signal (which cannot be masked) is applied, the sequence occurs as follows:

- The *cr* is pushed onto the stack.
- The *pc*, which contains the address of the instruction that would have been the next one executed, is pushed onto the stack.

- The enable condition is set to 0.
- The MAC-8 performs what appears to be a normal read operation, addressing location X(FFFF), but the data byte read is ignored. The operation serves only to acknowledge the reset.
- The pc is set to X(0000) and the next instruction is taken from that location.

Location X(0000) should contain the first instruction of the routine to handle resets. If a reset occurs immediately after power-up, the values of the sp and rp are unpredictable.

Since the dedicated memory locations are associated with interrupts (traps and resets overlap), it is possible to simulate traps and resets by appropriate interrupt signals, as well as by direct jumps or calls from other routines.

Chapter 4

MAC TUTOR SOFTWARE

4. MAC TUTOR SOFTWARE

4.1 Functional Description

A resident executive program is supplied (see Appendix) to allow the user to access the hardware components. The primary purpose of this executive program is to enable the user to store programs in memory and then execute them. In addition, the executive program provides the following:

- Supplies the necessary interface routines to store information permanently on cassette tapes or PROMs.
- Allows program debugging with single-stepping, breakpoints, or nonmaskable interrupts.
- Allows communication between a TTY terminal and a time-sharing computer.

The executive program is divided into three major sections:

- *Keypad and Display* — Commands and directives are given with the keypad and the results appear on the LED displays.
- *TTY* — All of the capabilities of the executive keypad are available through a TTY terminal and communication with a time-sharing computer is possible at the same time.
- *Utilities* — Programs are available for such functions as writing PROMs, verifying PROMs, and writing/reading magnetic cassette tape information.

4.2 Operation

4.2.1 Keypad/Display

The keypad consists of a standard calculator-type button-pad with four rows, each containing seven keys. Each key is marked with two labels, one in blue and the other in yellow. The blue labels are presently in use and the yellow labels are intended for future system expansion requirements.

The eight 7-segment LED displays are used mainly to display memory addresses and contents of memory locations. The standard arrangement uses the left four digits for memory address and the next two digits show the contents of that memory address plus one. The right two digits show the contents of that memory location. For example, the number 18001234 indicates that memory location 1800 contains hexadecimal number 34 and location 1801 contains 12. The left four digits are the address, the next two digits are the high contents, and the last two digits are the low contents.

4.2.2 Keypad Button Control

There are 16 keypad buttons labeled 0 through F that represent hexadecimal digits 0 through F. A is the decimal number 10, B is the decimal number 11, and so on through F, which is the decimal number 15. These keys are used in conjunction with the other function keys to specify exactly what will be done.

Initialize – init

The purpose of the **init** button is to reinitialize memory to recover from some abnormal condition. When this button is pressed, operations are performed as follows:

- The executive registers are set to the last 32 bytes of RAM, locations 1BE0 through 1BFF.
- The user program registers are assigned to the preceding 32 bytes of RAM, locations 1BC0 through 1BDF. These are the registers that are examined with the **/a** and **/b** buttons.
- User register **b11** is set to the address of the I/O page, location 1F00. This is done so that a user program can call subroutines in the executive program without setting this register beforehand.
- User register **b12** is set to the constant FF02. This enables a user program to easily use the executive subroutines to display numbers on the LEDs.
- The stack is set to just below the user registers. The stack will then grow down toward lower addresses.
- The return address into the executive program is pushed onto the stack. This is to enable a user program to make a normal return to the executive program on termination.
- A zero byte, representing an empty user condition register, is pushed onto the stack.
- The default value of the program counter (1800 is the first location of RAM) is pushed onto the stack.
- The address of the user registers is pushed onto the stack and becomes the user register pointer.

TTY – @

The button with the Bell System logo allows a TTY terminal keyboard to enter commands and directives.

Memory Address – *

This button is used to specify a memory address. After the ***** button is pressed and as each succeeding button is pressed, the memory address is shifted one place to the left (the last button pressed becomes the rightmost digit). For example, if the current memory address is 19AB and we wish to look at location 03FD, refer to Table 4-1.

Register Pointer – /d

When the **/d** button is pressed, the display address is set to that location in memory which contains the register pointer and the right four display digits will indicate the register pointer value.

This enables the following:

- Manual change of register pointer. By pressing the **=** button and changing the two memory locations containing the register pointer, operating registers can be set to any memory position.
- Since the register pointer is stored on the stack, the address field will now indicate where the bottom of the stack is located. This makes it possible to examine what the program has pushed onto the stack.

TABLE 4-1. MEMORY ADDRESS EXAMPLE STEPS

Key Pressed	Display Reading
	19AB1234*
.	1800ABCD*
3	8003FFFF
F	003F5498
D	03FDAFED

Table Notes:

1. Although memory addresses consist of four digits, the immediacy of the executive program required only three digits to be entered. Whatever memory address is displayed, whether by chance or design, the digits to the right will display the contents of those two addresses.
2. It is a good idea to specify all four digits of a memory address, otherwise leftover digits from the previous address could produce unexpected results.
3. The content of nonexistent memory, in this example 8003, is *always* FF.

Display a Register — /a

The /a button allows examination of the contents of the sixteen 8-bit registers that have been assigned for use. After this button is pressed, the display changes to indicate an a register and not memory. The left two digits of the address and the digits indicating the high contents are blanked out. The right two digits of the address change to the letter a, indicating that the display is showing an a register, followed by a digit representing the particular register displayed. Register a10 is displayed as AA, a11 is AB, and on through a15, which is AF. By default, the register a0 is displayed when /a is pressed. The low contents then show what is contained in the register indicated by the address. Since an a register contains eight bits, only the two digits of low contents are required (that is why the high contents display is blanked out). Once the /a is pressed, the displayed register can be specified as follows:

- Pressing any of the digit buttons from 0 to F will cause that register (0 to 15) to be displayed.
- The + button will cause the next higher numbered register to be displayed. If the register displayed is 15, AF in the address digits, the + button will cause register a0 to appear.
- The - button will cause the next lower numbered register to be displayed. If the register displayed is a0, the - button will cause register a15 to appear.

For example, to assume that registers a9, a8, a15, and a1 are to be displayed in that order, refer to Table 4-2.

TABLE 4-2. REGISTER DISPLAY EXAMPLE STEPS

Key Pressed	Display Reading	
	1800	ACED
/a	A0	10
9	A9	34
-	A8	AA
F	AF	E0
+	A0	98
+	A1	BA

Table Notes:

1. The /a button causes the display format to change. This allows determination of whether the display refers to memory or registers.
2. The last digit pressed determines which register will be displayed.
3. Digit, +, and - buttons can be mixed at will to specify which register to display.

Display b Register – /b

The /b button allows examination of the contents of the sixteen 16-bit registers that have been assigned for use. When this button is pressed, the display is changed to a format indicating that b registers are being shown. The left two digits of the address are blanked out and the right two digits change to the letter b, followed by a digit that indicates the register being displayed. The right four digits of the display are then used to show the 16-bit contents of the b register being examined. The /b button operates in the same fashion as the /a button.

Display Next Location – +

The + button allows examination of successive locations in memory. When this button is pressed, the current memory address is incremented by one and the contents of the new memory locations are displayed.

For example, it is possible to step through memory looking at successive locations, one after another. Refer to Table 4-3.

TABLE 4-3. MEMORY LOCATION EXAMPLE STEPS

Key Pressed	Display Reading
	18002211
+	18013322
+	18024433

Table Note: The standard display has the low contents showing whatever is in the memory location pointed to by the address, and the high contents showing whatever is in the following location. This explains why every time + is pressed, whatever was showing in high contents is now displayed in low contents.

Display Previous Location – -

The - button performs a function similar to the + button, but in the opposite direction. Every time the - button is pressed, the address is decremented by one, which makes it possible to go backward in memory and look at different locations one at a time.

Change Contents — =

The = button makes it possible to change memory. Normally, any digit button depressed causes a change in the address. However, after the = button is pressed, any digit button pressed causes a change to the low contents. The low contents are shifted left by one digit, losing the leftmost digit, and the button pressed becomes the rightmost digit. Also, after every digit is pressed, the new value of the low contents is stored into the location pointed to by the address. For example, to assume that the numbers 1, 2, and 3 are stored into the locations 1900, 1902, and 1A00, refer to Table 4-4.

TABLE 4-4. LOW CONTENTS LOCATION EXAMPLE STEPS

Key Pressed	Display Reading
	1 8 0 0 0 0 0 0
*	1 8 0 0 0 0 0 0
1	8 0 0 1 F F F F
9	0 0 1 9 1 3 2 4
0	0 1 9 0 A C E D
0	1 9 0 0 2 2 1 1
=	1 9 0 0 2 2 1 1
=	1 9 0 0 2 2 1 1
0	1 9 0 0 2 2 1 0
1	1 9 0 0 2 2 0 1
+	1 9 0 1 3 4 2 2
+	1 9 0 2 F 8 3 4
0	1 9 0 2 F 8 4 0
2	1 9 0 2 F 8 0 2
*	1 8 0 0 0 0 0 0
1	8 0 0 1 F F F F
A	0 0 1 A F A C E
0	0 1 A 0 D 8 9 A
0	1 A 0 0 E D C 0
=	1 A 0 0 E D C 0
4	1 A 0 0 E D 0 4
0	1 A 0 0 E D 4 0
3	1 A 0 0 E D 0 3

Table Notes:

1. The low contents are the ones affected and only one memory location can be changed at a time.
2. Pressing the = button more than once makes no difference (additional button pressing is ignored).
3. To correct an error, keep pressing buttons until the proper number is obtained.

The a and b registers can be changed in a similar fashion. The only difference to keep in mind is the operation of b registers. Since the b registers contain 16-bit numbers, all of the rightmost *four* digits in the display are affected when a b register is changed, instead of the rightmost *two* digits. For example, to set a8 to 88, a7 to 77, b14 to 00EE, b15 to 00FF, and b0 to 00, refer to Table 4-5.

TABLE 4-5. A AND B REGISTER CHANGE EXAMPLE STEPS

Key Pressed	Display Reading
	1 8 0 0 F E C D
/a	A 0 3 4
8	A 8 1 2
=	A 8 1 2
8	A 8 2 8
8	A 8 8 8
-	A 7 6 7
7	A 7 7 7
/b	B 0 1 2 3 4
E	B E 3 4 0 0
=	B E 3 4 0 0
E	B E 4 0 0 E
E	B E 0 0 E E
+	B F 0 5 5 0
0	B F 5 5 0 0
F	B F 5 0 0 F
F	B F 0 0 F F
+	B 0 1 2 3 4
0	B 0 2 3 4 0
0	B 0 3 4 0 0
0	B 0 4 0 0 0
0	B 0 0 0 0 0

Table Notes:

1. It is only necessary to key in the number of digits to obtain the required number. Two digits were sufficient for b14, whereas all four were necessary for b0.
2. Once the = button is pressed, it does not need to be pressed again if the format of the display remains the same.
3. When the display went from /a format to /b format, the = was necessary to indicate that the b registers were to be changed. However, when going from a8 to a7, the executive program remained in a change register mode.
4. When changing registers or memory, the + and - buttons are used to go to a new register or memory location. After examining a specific register or memory location, the = button can be pressed to make necessary changes.

Program Execution -- go

After a program is placed in memory with the = button, the go button is pressed to start the program running. This is an unconditional start and control will not return to the executive program unless one of three things happens:

- The user's program relinquishes control. (If the program executes a return instruction with no preceding subroutine call, the program will return to the executive program.)
- Illegal instructions will cause the executive program to regain control. To set a breakpoint, just place an illegal instruction (FF is a good choice) where the program breakpoint is desired.
- The reset button will also cause the MAC Tutor executive program to take over.

Single Step -- sst

The sst button operates in the same manner as the go button, but with one difference. Every time the sst button is pressed and immediately released, one instruction from the user's program is executed. The executive program then takes control and displays the address of the next instruction that would have been executed. This allows successive execution of one

instruction at a time from the user's program merely by pressing the **sst** button.

If the **sst** button is pressed and held down, instructions will be executed at a rate of approximately two per second. The address display will contain the address of the next instruction to be executed (used to view the program in operation).

4.2.3 TTY Control

If a TTY-compatible terminal is available, the MAC Tutor has the capability of using this device for the user interface instead of the on-board keypad/display. When the Bell System logo button is pressed, initialization functions are performed as follows:

- If there is no TTY connection or if the TTY is turned off, control will immediately return to the keypad/display portion of the MAC Tutor executive program.
- The executive program pauses, waiting for the user to type in a carriage return (cr). This key is used by the executive program to determine the terminal operating baud rate.
- A header is typed out to indicate what version of the executive program is being used. Currently the header looks like this: MAC Tutor Exec 1.0.
- The executive program displays a 4-digit memory address followed by a space and the 2-digit contents of that memory address. The memory address displayed will be the current value of the program counter, which on initial start-up will be 1800, the first address of RAM.

Operation from the TTY keyboard is the same as from the executive program keypad/display, except for the following differences.

Half Duplex – h

Normally the TTY executive program assumes that the terminal is running in full-duplex mode, therefore the executive program prints out each character as it is typed in. In the half-duplex mode, characters that are typed in will not be printed out. However, every time the **h** key is typed, the executive program switches from either half- or full-duplex to full- or half-duplex operation.

Initialize – i

The **i** key causes the memory to be set up and the header message and location 1800 are displayed.

Terminate TTY – Break Key

Pressing the **break** key, turning off the terminal, or unplugging the terminal will stop TTY operation and return control to the keypad/display.

Memory Address – *

Pressing the ***** key causes the executive program to set up to start displaying memory locations, and the memory address is set to the current value of the program counter.

After typing in the ***** followed by an optional address, a carriage return causes that memory address and its contents to be displayed at the terminal. For example, to examine locations 8003, 1900, and 1800 (in that order), refer to Table 4-6.

TABLE 4-6. MEMORY ADDRESS LOCATION (TTY) EXAMPLE STEPS

User Type Input	Output
	1800 DC
*3'cr'	8003 FF
*1234123111900'cr'	1900 8F
*'cr'	1800 DC

Table Notes:

1. It is required to only type in as few digits as are necessary to generate the proper address. (The digit 3 was sufficient to convert the address 1800 into 8003.)
2. The TTY executive program requires (at most) the last four digits to be typed. If a mistake is made, it can be corrected simply by typing in all the proper digits.
3. An * alone is sufficient to bring back the current value of the program counter.

Register Pointer — r

This key operates in a manner similar to the *. The difference is that the memory address is set to the bottom of the stack, which is where the register pointer is stored. As soon as the r key is typed, the address of the bottom of the stack is displayed, along with the contents of that location, the low byte of the register pointer. Refer to Table 4-7.

TABLE 4-7. REGISTER POINTER (TTY) EXAMPLE STEPS

User Type Input	Output
	1800 DC
r	1BB9 C0
'cr'*	1BBA 1B

Register Display — /

This key sets up the executive program to display the contents of one of the operating registers. After pressing the / key, either the character a or b must be typed, indicating whether examination of the 8-bit a registers or 16-bit b registers is desired. Next, one of the digits 0 through F should be typed to indicate which particular register is to be examined. If more than one digit is typed, the executive program will use the last one to specify which register is desired. After the type is entered and the proper register is selected, a carriage return will cause that register to be displayed. For example, to examine registers a10, b15, a0, and a9, refer to Table 4-8.

TABLE 4-8. REGISTER DISPLAY (TTY) EXAMPLE STEPS

User Type Input	Output
	1800 DC
/aA'cr'	AA 01
/BF'cr'	BF 56D4
/'cr'	A0 FD
/A0123456789'cr'	A9 99

Display Next Location — Carriage Return

In order to examine a successive location, a carriage return (cr) key typed alone on a line will

cause either the next higher memory location or register to be displayed. If the current register number is 15, the cr key will cause register 0 to be displayed. For example, to examine memory locations 1900 through 1903, registers a15 through a3, and registers b15 and b0, refer to Table 4-9.

TABLE 4-9. DISPLAY NEXT LOCATION (TTY) EXAMPLE STEPS

User Type Input	Output
	1800 DC
*1900'cr'	1900 00
'cr'	1901 11
'cr'	1902 22
'cr'	1903 33
/a1'cr'	AF FF
'cr'	A0 00
'cr'	A1 11
'cr'	A2 22
'cr'	A3 33
/BF'cr'	BF 0FFF
'cr'	B0 1200

Display Previous Location — Line Feed

In contrast to the carriage return, the line feed (lf) key causes the next lower memory location or register to be displayed. Otherwise, the lf and cr keys operate in the same manner. For example, to display memory locations 1800 through 17FE, registers a1 through a14, and registers b14 through b1, refer to Table 4-10.

TABLE 4-10. DISPLAY PREVIOUS LOCATION (TTY) EXAMPLE STEPS

User Type Input	Output
	2FCD FF
*1800'cr'	1800 DC
'lf'	17FF FF
'lf'	17FE FF
/A1'cr'	A1 11
'lf'	A0 00
'lf'	AF 11
/BE'cr'	BE 0EEE
'cr'	BF 0FFF
'cr'	B0 0000
'cr'	B1 0111

Change Contents

If an input line consists of nothing but hexadecimal digits followed by either a cr or lf key, the digits are collected into one number. Then when the cr or lf key is typed, the rightmost two digits are stored into the currently displayed memory location or a register, or the rightmost four digits are stored into the currently displayed b register. If fewer digits than necessary to fill up a memory location or register are typed, the leftmost digits are assumed to be zero. For example, to store 59, 00, and 18 into memory locations 1800, 1802, and 1804; FF, 0, and 1 into registers a15 to a1; and 1, 1000, and FACE into registers b8 through b10; refer to Table 4-11.

TABLE 4-11. CHANGE CONTENTS (TTY) EXAMPLE STEPS

User Type Input	Output	
	BADD	FF
*1800'cr'	1800	DC
59 'cr'	1801	11
'cr'	1802	22
0'cr'	1802	33
18*1803'cr'	1803	33
'cr'	1804	44
18'cr'	1805	55
/AF'cr'	AF	FF
'cr'	A0	FF
0'cr'	A1	11
1'cr'	A2	22
/B8'cr'	B8	0888
1'cr'	B9	0999
1000'cr'	BA	0AAA
01234FACE'cr'	BB	0BBB

Table Notes:

1. Something has to be currently displayed before it can be changed.
2. The change does not take effect until the cr or lf key is depressed. This gives the ability to check the input for errors before making a change. If a mistake is made, start typing the desired number from scratch until the proper number is in the rightmost digit position.
3. If it is not desired to change a displayed value, type in a cr or lf key to skip over that location without affecting it.

Program Execution — g

The **g** key signals that the current memory location is the first address of some executable code. When the **cr** key is typed, the executive program starts execution at this address. The current memory location can be specified on the same line as the **g** command, so that the sequence ***1800gcr** would cause the MAC Tutor to start executing the program at location 1800. In order to avoid trying to execute a register address or other strange problems, it is recommended that program execution start with *****, followed by a 4-digit starting address, followed by **g** command, and terminated with **cr**.

Single Step — s

The **s** key operates in the same manner as the **g** key, except that it only executes one instruction.

The **s** key also causes one instruction to be executed without waiting for a **cr** to be typed. After the instruction is completed, the header message is typed out, followed by the address of the next instruction to be executed and the contents of that memory location.

Since the address of the next instruction is the current address, a program can be single-stepped many times by merely using the **s** key. Since the ***** key sets the current memory location to the value of the program counter, it is easy to single-step a program for a while, look at memory locations or registers, then continue single-stepping or executing the program. Refer to Table 4-12 for an example that will single-step a program through two instructions, look at some registers and memory locations, and then restart execution at the third instruction of the program.

TABLE 4-12. SINGLE-STEP (TTY) EXAMPLE

User Type Input	Output
*1900s	2000 FF MAC Tutor Exec 1.0 1902 7F
s	MAC Tutor Exec 1.0 1903 59
/b0'cr'	B0 0000
'lf'	BF FFFF
/AF'cr'	AF FF
*1904'cr'	1904 00
'cr'	1905 18
*g'cr'	

Table Note: Although the last memory location displayed was 1905, the g command caused execution to resume at location 1903. Recall that the * key causes the current memory address to be set to the current value of the program counter, which in this example was left at 1903.

Talk to Host Computer – !

When the ! key is pressed, the TTY executive program connects a modem interface to allow communication with a time-sharing computer. If another ! is typed, the connection between the terminal and modem is broken and the connection is once again made with the MAC Tutor. (This sequence can be repeated as many times as desired.)

Load Hex File – l

If access to a time-sharing computer is available, the TTY executive program has the ability to load programs developed on that computer into the MAC Tutor memory. When the l key is typed, the TTY executive program will load a standard hex. file.

4.2.4 System Utilities

Certain routines have been created that utilize MAC Tutor features. All of these routines are executed as if they were user programs that were loaded into memory. However, because these routines are part of the ROM executive, they are always available and unmodifiable. These routines are invoked by setting certain registers to indicate what is desired, then executed with the keypad go button or terminal g key. Refer to 4.4 for routine details.

4.3 Programming

There are basically three ways to create programs for the MAC-8:

- *Hand Coding.* Pencil and paper are used to create each byte of each instruction in the program.
- *Assembler.* Assembly language programs can be created on a UNIX* time-sharing system. These programs can then be loaded into memory and executed on the MAC Tutor.
- *C Compiler.* A UNIX system can be used to create programs in the C programming language.

* UNIX is a trademark of Bell Laboratories.

4.4 Available Programs

The following descriptions include the starting address on the title line, input parameters, constraints, and abnormal conditions.

4.4.1 Move Memory - *022F

This routine moves a copy of a block of memory from one place to another. The input consists of setting registers b8, b9, and b10 as follows:

- b8 The address of where to move the block of memory.
- b9 The address of the first location to move.
- b10 One more than the last address that is to be moved.

For example, to move a copy of everything in locations 1900 through 19FF to locations 1800 through 18FF, refer to Table 4-13.

TABLE 4-13. MOVE MEMORY EXAMPLE STEPS

User Type Input	Output	
	1800	DC
/B8'cr'	B8	0888
1800'cr'	B9	0999
1900'cr'	BA	0AAA
1A00'cr'	BB	0BBB
*022Fg'cr'	MAC Tutor Exec 1.0 (cr)	
	1800	00

Table Note: If the "to" address is greater than the "from" address and the blocks overlap, only the locations between these two addresses will be moved properly. All the rest of the destination block will consist of repetitions of this small block.

4.4.2 Write a PROM - *0541

This program writes the contents of any contiguous 1024-byte block of memory into a 2708 PROM. The program that writes the PROM uses register and stack space in the upper 1024-byte block of RAM (starting address 1800); therefore, it is not possible to successfully program an entire 2708 PROM unless the lower 1024-byte block of RAM (starting address 1400) is used to contain the program to be written. This restriction is not too severe, since approximately the first 950 bytes in the upper 1024-bytes of RAM (starting address 1800) can be written into the PROM.

The only input required consists of setting register b9 to the starting address of the block of 1024-bytes of RAM to be written into the 2708 PROM. It takes approximately two minutes to write the PROM.

When the program is complete, it indicates whether or not the PROM is correctly written by setting registers b13 and b14 to the following values:

- b13 If the lower 1024-bytes of RAM (starting address 1400) were used, this register contains 400 if the PROM is written correctly. If incorrectly written, b13 will contain a number from 0000 to 03FF, indicating the first location in the PROM that is in error. If the upper 1024-bytes of RAM (starting address 1800) were used, b13 should contain at least 03b9.

b14 If b13 does not contain 400, then the left two digits in b14 are what should have been stored in the PROM at the location specified by b13. The right two digits in b14 show what is actually there.

To write locations 1000 to 1400 into a PROM that has been placed in the PROM programming socket, refer to Table 4-14.

TABLE 4-14. WRITE PROM LOCATIONS EXAMPLE STEPS

User Type Input	Output
	1800 DC
/B9'cr'	B9 1234
1000'cr'	BA 5678
*0541g'cr'	

4.4.3 Verify a PROM - *057B

This program verifies that the contents of the PROM match what is in a block of memory. The only input is to set register b9 to the beginning address of a 1024-byte block of memory. (Since a blank PROM and nonexistent memory both contain FF, a PROM can be zero verified by specifying a nonexistent block of memory, such as 2000.)

When complete, the program indicates the results of the verification operation by setting registers b13 and b14 as follows:

- b13 If this register contains 400, the 1024 bytes in the PROM are the same as the 1024 bytes in memory. If the register does not contain 400, it will contain a number from 0 to 3FF, indicating the first location in the PROM that did not match.
- b14 If register b13 does not contain 400, the left two digits of register b14 indicate what was stored in the PROM. The right two digits are in memory.

To check whether the contents of locations 1800 to 1BFF are the same as what is in the PROM, refer to Table 4-15.

TABLE 4-15. VERIFY PROM CONTENTS EXAMPLE STEPS

User Type Input	Output
	1800 DC
/B9'cr'	B9 1234
1800'cr'	BA 5678
*057Bg'cr'	

4.4.4 Dump to Audio Tape - *06C6

This routine dumps a block of memory to an audio tape file. The input consists of:

- a8 File ID, a unique number from 1 through FE identifying this file. It is recommended that IDs 0 and FF not be used.
- b9 The address of the first location to be stored on the tape.

- b10 One more than the last address to be written onto the tape.

The sequence of events necessary to write a file out to tape is:

- Set registers a8, b9, and b10.
- Start tape recorder and set to record mode.
- Wait until tape leader has been skipped.
- Execute the tape dump program.
- Stop the tape recorder when the program is completed.

Note: While the program is executing, the leftmost digit of the LED display indicates what is happening. For the first 5 seconds it will show two vertical bars, the right bar being one-half the height of the left bar (this indicates that the 100 sync characters which begin every file are being written out). After this is completed, the right two vertical segments should be lighted, the top horizontal segment should be off, and all other segments should flicker (this indicates that data is being written out to the tape). If the display indicates a pattern of bars with none of the segments varying, one of two things has happened:

- All the data to be stored on tape is the same. This situation is possible, but rather unlikely.
- The data being written out are all Fs. This was probably caused by putting the wrong starting address in b9 and writing out a nonexistent program.

To store locations 1000 to 13FF on tape using the file ID 10, refer to Table 4-16.

TABLE 4-16. TAPE STORE LOCATION EXAMPLE STEPS

User Type Input	Output	User Action
	1800 DC	
/B8'cr'	B8 1234	
0010'cr'	B9 5678	
1000'cr'	BA 9ABC	
1400'cr'	BB DEF0	
*06C6g'cr'	MAC Tutor Exec 1.0 1800 DC	Start tape recorder. Skip leader. Stop tape recorder.

4.4.5 Read from Audio Tape - *05EE

This routine reads information stored on tape back into MAC Tutor memory. The input consists of:

- a8 File ID of data to be read in from tape.
- b9 Address of the first location to be stored in memory. (This parameter is used only if the file ID is FF.)

Since special meanings have been assigned to certain file IDs, actions will take place as follows:

- 0 The next file on the tape will be read into the address stored as part of the file.
- 1-FE The first file with the same ID as this will be read into the address stored as part of the file.

- FF The next file on the tape will be read into the address specified in register b9.

The following steps are required to read in a file from tape:

- Set register a8 and possibly register b9.
- Start execution of the tape load program.
- Start tape recorder.
- Upon completion of program, stop tape recorder.

As with the dump program, the leftmost digit of the LED display gives some indication of what the program is doing. If the display is a random pattern that does not change or only changes very slowly every one or two seconds, the program is waiting for the next data file to appear. If there are two vertical bars (the right one half the height of the left one), the 100 sync characters that begin a data file are being read. The data file is actually being loaded when the right two vertical segments are lighted, the top horizontal segment is off, and all other segments are flickering too fast to be understood.

The program can detect the following types of errors:

- *Vertical parity error.* A parity bit is stored with each character in the data file to enable detection of a change in one bit of the character.
- *Longitudinal checksum error.* The last character of a file, called a checksum, gives the tape load program one more way of checking that a file is read in properly.

When the tape load program terminates, register b14 contains the number of vertical parity errors in the upper two digits. The lower two digits contain the computed checksum for the file. If register b14 contains all zeros, no errors were detected. If register b14 has no zeros, data that was read in should be viewed with suspicion. An error has occurred but there is no way to determine where it has occurred.

The tape read program will ignore anything on the tape that it does not recognize as a data file. As a result, a short voice description of a data file can precede that file on the tape without causing any problems for the load program.

For example, if a data file with file ID 53 has been stored on tape, refer to Table 4-17 to read that file back.

TABLE 4-17. TAPE READ EXAMPLE STEPS

User Type Input	Output	User Action
/A8'cr'	1800 DC	
53'cr'	A8 12	
*05EEg'cr'	A9 34	
	MAC Tutor Exec 1.0	Start tape recorder.
	1800 DC	
/BE'cr'	BE 0000	Stop tape recorder.

Table Note: The tape can be started from the very beginning and the program will skip everything until it comes to the right file. It is also possible to manually position the tape with the fast forward and rewind controls to just before the desired file. If the position of the file is known, either through voice information on the tape or tape recorder counter, this technique can be used to speed up tape file processing.

4.5 Testing and Diagnosing

MAC Tutor testing and diagnosing approaches consist of:

- Self-test program
- Truth table excitation
- Manual logic analyzer

The self-test approach consists of running a program that checks out each portion of the MAC Tutor. This program requires that a set of straps be plugged into the I/O and PROM programming sockets. Then, by feeding the outputs to the inputs of the various elements, the program verifies the operation.

The truth table excitation approach makes use of a logic tester to excite the various elements in the MAC Tutor and to logically compare the appropriate outputs. This test requires that the MAC-8 be removed from the socket and the logic tester be connected to the MAC-8 and I/O sockets.

The third approach for testing and diagnosing the MAC Tutor is through the use of a logic analyzer (e.g., Hewlett-Packard Model 1600A or equivalent). The address and data buses are available for monitoring purposes at connectors J1 and J2. Through the use of the memory map shown in Table 4-18, the read and write cycles for the various memory devices can be monitored and verified. Typically, the first items that are checked out involve the integrity of the control signals to and from the MAC-8. These include the reset, memory read, memory write, and clock signals. If these signals check out, the ROM, RAM, and I/O follow in sequence. These are checked out by determining the integrity of the chip select signal of these devices and the data bus contents.

TABLE 4-18. ADDRESS ASSIGNMENTS/MEMORY MAP

Device	Physical Location											Hex Addresses			
		A15	A14	A13	A12	A11	A10	A4	A3	A2	A1		A0		
32AG ROM } 32AAF ROM }	N01	0	0	0	0	0	X								0000-07FF
2708 PROM	K01	0	0	0	0	1	0								0800-0BFF
2708 PROM	G01	0	0	0	0	1	1								0C00-0FFF
2708 PROM	D01	0	0	0	1	0	0								1000-13FF
9131 RAM	D05-K05	0	0	0	1	0	1								1400-17FF
9131 RAM	G05-N05	0	0	0	1	1	0								1800-1BFF
8255 I/O	D24	0	0	0	1	1	1	0	0	0					1F00-1F03
8255 I/O	D16	0	0	0	1	1	1	0	0	1					1F04-1F07
8255 I/O	D20	0	0	0	1	1	1	0	1	0					1F08-1F0B
74LS273 I/O	C10	0	0	0	1	1	1	0	1	1	0	1			1F0D
74LS273 I/O	C13	0	0	0	1	1	1	0	1	1	1	0			1F0E

- Table Notes:
1. X designates either logical 1 or 0. Blank areas indicate future expansion.
 2. Unit comes equipped with one of the two listed ROMs

Chapter 5
GLOSSARY

5. GLOSSARY

<i>Addend</i>	A number to be added to another.
<i>Addressing Mode</i>	A way of forming the effective memory address(es) for the operand(s) in an instruction.
<i>Architecture</i>	A design or orderly arrangement of a microprocessor.
<i>Augend</i>	A quantity to which an addend is added.
<i>Autobaud</i>	To automatically adjust to a given baud rate.
<i>Baud Rate</i>	A measure of data flow. The number of signal elements per second based on the duration of the shortest element. When each element carries one bit, the the baud rate is numerically equal to bits per second (bps).
<i>Bit</i>	A binary digit (logical 1 or 0).
<i>Byte</i>	A sequence of adjacent binary digits (usually shorter than a word) operated on as a unit. Sometimes referred to an 8-bit byte.
<i>C Compiler</i>	A unit that translates C language source programs into machine language codes.
<i>Central Processing Unit (CPU)</i>	The heart of any computer system. A basic CPU consists of an arithmetic and logic unit, control block register array, and input/output.

<i>Checksum</i>	The last character of a data file that is used for error detection purposes.
<i>Clock</i>	A pulse generator that controls the timing of microprocessor switching circuits.
<i>Command</i>	The portion of an instruction that specifies the operation to be performed.
<i>C Program</i>	An organized set of instructions written in the C programming language.
<i>CPU</i>	See <i>Central Processing Unit</i> .
<i>Data Bus</i>	A group of lines each capable of transferring one bit of data. It is bidirectional and can transfer data to and from the CPU, memory storage, and peripheral devices.
<i>Debug</i>	To search for and eliminate errors in a computer program.
<i>Decrement</i>	A programming instruction that decreases the contents of a storage location.
<i>DIP Connector</i>	Dual In-line package connector.
<i>Dump</i>	To transfer the contents of memory to an output device.
<i>Dyadic Operation</i>	An operation performed using two operands, the source and destination.
<i>EPROM</i>	See <i>Erasable Programmable Read-Only Memory</i> .
<i>Erasable Programmable Read-Only Memory (EPROM)</i>	Usually consists of a mosaic of undifferentiated cells that is electrically reprogrammable and erasable by ultraviolet irradiation.

<i>Fetch</i>	To obtain data from a memory location. Reading an instruction from memory and entering it in the instruction register is often referred to as an instruction fetch.
<i>Frequency Shift Keying (FSK)</i>	A form of frequency modulation in which the modulating wave shifts the output frequency between predetermined values (usually called a mark and space).
<i>FSK</i>	See <i>Frequency Shift Keying</i> .
<i>Hardware</i>	The electrical, mechanical, electronic, and magnetic components of a computer.
<i>Hexadecimal</i>	Whole numbers and letters in positional notation using the decimal number 16 as a base. The least significant hexadecimal digits read: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.
<i>Increment</i>	A programming instruction that increases the contents of a storage location.
<i>Input/Output (I/O)</i>	Package pins connect directly to the internal bus network to interface the microprocessor with the "outside world."
<i>Interface</i>	A common boundary between adjacent component circuits or systems enabling the devices to yield or acquire information from one another. (Buffer, handshake, and adapter are used interchangeably with interface.)
<i>Interrupt</i>	Suspension of the normal programming routine of a microprocessor in order to handle a sudden request for service.
<i>I/O</i>	See <i>Input/Output</i> .
<i>LED</i>	Light-emitting diode.

<i>Memory</i>	Core, disk, drum, or semiconductor systems into which information can be inserted and held for future use. (Memory and storage are interchangeable terms.)
<i>Microprocessor</i>	A central processing unit fabricated on one or two chips consisting of arithmetic and logic unit, control block, and register array. The inputs and outputs of the associated sections are joined to a memory storage system.
<i>Modem</i>	An acronym for modulator-demodulator. A device that converts data to a form that is compatible between data processing and transmission equipment.
<i>Monadic Operation</i>	An operation performed using only one operand.
<i>Opcode</i>	An acronym for operation code; that part of the coded instruction designating the operation to be performed.
<i>Operand</i>	A quantity of data in which an operation is performed; usually one of the instruction fields in an addressing statement.
<i>Peripheral</i>	Auxiliary function (devices not under direct computer control).
<i>PPI</i>	See <i>Programmable Peripheral Interface</i> .
<i>Program</i>	A procedure for solving a problem. Frequently referred to as software.
<i>Programmable Peripheral Interface (PPI)</i>	An integrated circuit that can be programmed to interface with a variety of peripheral equipment.
<i>Programmable Read-Only Memory (PROM)</i>	A programmable mosaic of undifferentiated cells. Program data is stored in the PROM.

<i>PROM</i>	See <i>Programmable Read-Only Memory</i> .
<i>Pushdown Stack</i>	A register array used for storing and retrieving data on a last-in, first-out basis.
<i>RAM</i>	See <i>Random-Access Memory</i> .
<i>Random-Access Memory (RAM)</i>	Memory in which access to any storage location is provided immediately by means of vertical and horizontal coordination. Information can be "written in" or "read out" in the same rapid manner.
<i>Read-Only Memory (ROM)</i>	A storage device in which stored data cannot be altered by computer instructions (sometimes called firmware).
<i>Register</i>	A device for temporary storage of one or more bits involved in arithmetical, logical, or transferral operations. The number of registers in a microprocessor is considered one of the most important architecture features.
<i>ROM</i>	See <i>Read-Only Memory</i> .
<i>Routine</i>	A sequence of instructions for performing a particular task.
<i>Single-Step</i>	A command that executes only one instruction at a time.
<i>Software</i>	The internal programs or routines prepared to simplify computer operations. Software permits the programmer to use a language such as C or mathematics to communicate with a computer.
<i>Storage</i>	Any device that retains information. The word storage is used interchangeably with memory.

Subroutine

Part of a master routine that can be used at will to accomplish a specific task (the object of a branch or jump command).

Teletypewriter (TTY)

The teletypewriter uses electromechanical functions to generate codes (Baudot) in response to manual inputs from a typewriter keyboard.

Transistor-Transistor Logic (TTL)

A logic-circuit design method that uses inputs from multiple emitter transistors. Sometimes referred to as multiemitter transistor logic.

TTL

See *Transistor-Transistor Logic*.

TTY

See *Teletypewriter*.

Word

A number of bits that are treated as one unit, where the number depends on the CPU.

APPENDIX
RESIDENT EXECUTIVE PROGRAM

```

#define IOPAGE 017400
#define PCNTRL *(b11+3)
#define QCNTRL *(b11+7)
#define PAIO *b11
#define PBIO *(b11+1)
#define PCIO *(b11+2)
#define PDIO *(b11+4)

#define SETMOD 0200
#define AINP 020
#define BINP 02
#define CINP 011
#define CLINP 01
#define CUINP 010
#define DINP 010

#define SSTKEY 24
#define NOKEY 28
#define OFFDIG 15
#define ALLDIG 0177400
#define KDOWN 2

#define BAUD *015776

#define NUMDEL 3
#define SSTDEL 0x7f

#define RAMORG 0x1800
#define RAMLEN 04000
#define SYSREG 0x1be0
#define USERRG 0x1bc0
#define USERB11 *015726 /* 0x1bd6 */
#define USERB12 *015730 /* 0x1bd8 */
#define USERB13 *015732 /* 0x1bda */
#define USERB14 *015734 /* 0x1bdc */

#define NBIT 50

#define A 10
#define B 11

#define STARTCH "*"
#define ENDCH "/"
#define CHECKSUM 0x2a /* "*" */
#define EOT 004
#define SYNC 026

#define BIT1 24<<8 | 12
#define BIT0 12<<8 | 12
#define CYCLE0 13
#define CYCLE1 6
#define NOISE 040

int USERB11;
int USERB12;

```

```

int USERB13;
int USERB14;

int BAUD;
_ASSEM "BAUD = 15776";

_ASSEM "PCNTRL = 17403";
_ASSEM "QCNTRL = 17407";
_ASSEM "PAIO = 17400";
_ASSEM "PBIO = 17401";
_ASSEM "PCIO = 17402";
_ASSEM "PDIO = 17404";

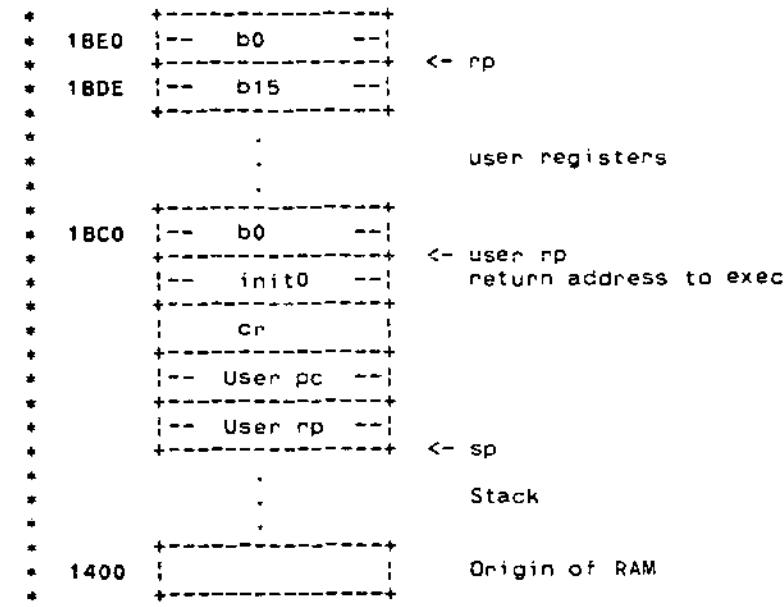
77A 7E 30 6D 79      char tfmt[]      { 0176, 0060, 0155, 0171 }
77E 33 5B 5F 70      { 0063, 0133, 0137, 0160 }
782 7F 73 77 1F      { 0177, 0163, 0167, 0037 }
786 0D 3D 4F 47      { 0015, 0075, 0117, 0107 };

78A 18 14 10 0C      char tnum[]      { 24, 20, 16, 12 }
78E 0D 0E 0F 19      { 13, 14, 15, 25 }
792 15 11 08 09      { 21, 17, 8, 9 }
796 0A 0B 1A 16      { 10, 11, 26, 22 }
79A 12 04 05 06      { 18, 4, 5, 6 }
79E 07 1B 17 13      { 7, 27, 23, 19 }
7A2 00 01 02 03      { 0, 1, 2, 3 };

7A6 65 01 E2 01 F0 01 int tfnc[]      { &numb, &plus, &minus, &star }
7AC D1 01              { &equal, &exec, &neg, &breg }
7AE DE 01 0E 02 A7 01 { &rptr, &sst, &sst, &init0 }
7B4 C3 01              { &tty };
7B6 FE 01 15 02 15 02
7BC 29 00
7BE 44 02

/* Mac8 Tutor Executive
*
* Global memory allocation
*
* +-----+
* 1BFE |-- b15 --| BAUD rate counter
* +-----+
* |-- b14 --| contents of (b13)
* +-----+
* |-- b13 --| current address
* +-----+
* |-----| on digits flags
* | a12 | exec state flags
* +-----+
* |-- b11 --| address of 10 page
* +-----+
*
* .
* .
*

```



```

*/
main()
{
0  7F          nop();
1  7F          nop();
2  7F          nop();
3  58 03      goto 1f;
5  59 00 0B   goto *0x800;
8  47          1: push(rp);
9  4D 0F E0 1B  rp = SYSREG;
D  C0 BF 00 1F  b11 = IOPAGE;
11 82 BF 07 8B  QCNTL = 0213;
15 82 BF 04 FF  PDIO = 0377;
19 82 BF 03 93  PCNTL = 0223;
1D 86 0B 02     a0 = PCIO;
20 52 03 31     if (!bit(3,a0)) goto 1f;
23 7F          nop();
24 59 F4 07     goto powon;
27 60 F0       reset: b15 = 0;
29 C1 FF D6 1B 00 1F  init0: USERB11 = IOPAGE;
2F C1 FF D8 1B 02 FF  USERB12 = ALLDIG | KOOWN;
35 4D 0F E0 1B  init:  rp = SYSREG;
39 C0 BF 00 1F  b11 = IOPAGE;
3D 0D 0F B9 1B  sp = USERRG - 2 - 1 - 2 - 2;
41 C2 FF 05 35 00  *(dsp + 5) = &init;
46 22 F0 04     *(sp + 4) = 0;
49 C2 FF 02 00 1B  *(dsp + 2) = RAMORG;
4E C2 FF 00 C0 1B  *(dsp + 0) = USERRG;
53 82 BF 03 90  1:  PCNTL = 0220;
57 82 BF 02 10  PCIO = 020;

```

```

58 C6 DF 02
5E C5 ED
60 C0 CF 00 FF
64 01 02
66 C5 0F FE 1B
6A 61 F1 44 02
6E 79 AC 00
71 B0 0F 18
74 4B F1 04
77 80 CF 02
7A 79 DF 00
7D 79 AC 00
80 B0 0F 1C
83 40 F1 06
86 98 CF FD
89 58 EF

8B 5A C1 ED
8E 90 CF 02
91 B0 0F 0F
94 40 F9 07
97 C0 20
99 C0 0F 0F 00

9D A0 0F 0F
A0 30 01
A2 EB 0F A6 07
A6 C5 30
A8 69 3F
AA 58 CE

```

```

b13 = *(dsp + 2);
b14 = *dt3;
man2: b12 = ALLDIG;
      set(zero); /* mac7 hardware error */
      b0 = BAUD;
      if (!zero) tty();
      rdkey();
      a0 = SSTKEY;
      if (zero) goto 1f;
      a12 = KDOWN;
1:    disp();
      rdkey();
      if (a0 == NOKEY) {
          a12 = & 0375;
          goto 1b;
      }
      if (bit(1,a12)) goto 1b;
      a12 = ! KDOWN;
      if (a0 <= 15) {
          b2 = b0;
          b0 = 15;
      }
      a0 = - 15;
      a0 = * 2;
      b0 = + &tfnc;
      b3 = *d0;
      +b3();
      goto 1b;
}
/* rdkey - read keyboard
 *
 * entry - entry - none.
 *
 * uses - a 0,1,3
 *        b 0,1
 *
 * calls - none.
 *
 * exit - a0 = number from 0 to 27 indicating
 *         which key was pressed.
 */
rdkey()
{
    PCNTRL = 0202;
    QCNTRL = 0233;
    b0 = (-7)&0377;
    a3 = 0357;
1:    a0 = + 7;
      a3 = >>> 1;
      if (!neg) return;
      PAIG = a3;
      a4 = PDIO;
      a4 = & 0177;

```

```

AC 82 BF 03 82
B0 82 BF 07 9B
B4 C0 0F F9 00
B8 80 3F EF
BB A8 0F 07
BE 34 3F
C0 65 00
C2 B1 B3
C4 86 4B 04
C7 98 4F 7F

```

```

CA 88 4F 7F
CD 48 F1 EC
D0 0C 14
D2 28 18
D4 A8 01
D6 C0 1F 8A 07
DA 75 10
DC 85 01
DE 66

```

```

a4 = ^ 0177;
if (zero) goto 1b;
a1 = flo(a4);
--a1;
a0 += a1;
b1 = &tnum;
b1 += logical(a0);
a0 = *b1;
return;
}
/* disp - display numbers in 7-segment displays
*
* entry - a12(15-8) = bit mask indicating on
*          digits.
*          b13 = first four digits to display
*          b14 = last four digits
*
* uses - a 0,2,5
*        b 0,5
*
* calls - dsp4, delay.
*
* exit - 7-segment displays refreshed.
*/
disp()
{
    nob();          /* historical allignment */
    QCNTL = 0213;
    PDIO = 0;
    PCNTL = 0220;
    swap(b12);
    a2 = 0;
    b0 = b13;
    dsp4();
    b0 = b14;
    dsp4();
    b5 = NUMDEL;
    delay();
    PBIO = OFFDIG;
    swap(b12);
    return;
}
/* dsp4 - display 4 digits
*
* entry - b0 = 16-bit number to display
*
* uses - a none.
*        b 0
*
* calls - dsp2.
*
* exit - next 4 digits displayed.
*/
dsp4()

```

```

DF 7F
E0 82 BF 07 88
E4 22 B0 04
E7 82 BF 03 90
EB 6A C0
ED 20 20
EF C0 0D
F1 79 07 01
F4 C0 0E
F6 79 07 01
F9 C0 5F 03 00
FD 79 5D 01
100 82 BF 01 0F
104 6A C0
106 66

```

```

107 6A 00      {
109 79 12 01      swap(b0);
10C 6A 00      dsp2();
10E 79 12 01      swap(b0);
111 66          dsp2();
                return;
            }
/* dsp2 - display 2 digits
*
* entry - a0 = 8-bit number to display
*
* uses - a 0,1,2,3,5
*        b 3,5
*
* calls - delay.
*
* exit - next 2 digits displayed.
*/
dsp2()
{
112 80 10      a1 = a0;
114 38 1F      a1 =>> 1;
116 38 1F      a1 =>> 1;
118 38 1F      a1 =>> 1;
11A 38 1F      a1 =>> 1;
11C C0 3F 7A 07 b3 = &tfmt;
120 75 31      b3 += logical(a1);
122 85 33      a3 = *b3;
124 82 BF 01 0F PBIO = OFFDIG;
128 34 C1      a12 = <<< 1;
12A 40 F5 0E      if (!odd) goto dp21;
12D 82 B3 04      PDIO = a3;
130 82 B2 01      PBIO = a2;
133 C0 5F 03 00 b5 = NUMDEL;
137 79 50 01      delay();
13A 28 20      dp21: ++a2;
13C 98 0F 0F      a0 = & 017;
13F C0 3F 7A 07 b3 = &tfmt;
143 75 30      b3 += logical(a0);
145 85 33      a3 = *b3;
147 82 BF 01 0F PBIO = OFFDIG;
148 34 C1      a12 = <<< 1;
14D 40 F5 07      if (!odd) goto dp22;
150 82 B3 04      PDIO = a3;
153 82 B2 01      PBIO = a2;
156 28 20      dp22: ++a2;
158 66          return;
            }
/* bitime, delay - delay specified time
*
* entry - b5 = delay count (picked up from
*        BAUD by bitime)
*
* uses - a 5
*        b 5

```



```

*
* calls - none.
*
* exit - eventually.
*
*/
bitime()
{
159 C5 5F FE 1B      b5 = BAUD;
15D 01 02      delay: set(zero);
15F 68 58      --b5;
161 64 00      if (neg) return;
163 58 F8      goto delay;
}
/* numb = process hex number
*
* entry - a2 = number keyed in
*
* uses - a 0,1,13,14
*        b 0,13,14
*
* calls - none.
*
* exit - number shifted into the current
*        address or data field as required.
*
*/
numb()
{
165 52 C0 12      if (bit(0,a12)) {
168 5A C2 21      if (bit(2,a12)) goto chneg;
168 38 E1      a14 = << 1;
16D 38 E1      a14 = << 1;
16F 38 E1      a14 = << 1;
171 38 E1      a14 = << 1;
173 90 E2      a14 = | a2;
175 81 DE      *b13 = a14;
177 58 11      } else {
179 5A C2 3A      if (bit(2,a12)) goto reg2;
17C C0 0D      b0 = b13;
17E 79 3B 02      shift4();
181 C0 D0      b13 = b0;
183 98 DF F0      a13 = & 0360;
186 90 D2      a13 = | a2;
188 C5 ED      b14 = *d13;
}
18A 66      return;
18B 52 C3 0A      chneg: if (bit(3,a12)) {
18E C0 0E      b0 = b14;
190 79 3B 02      shift4();
193 C0 E0      b14 = b0;
195 58 08      } else {
197 38 E1      a14 = << 1;
199 38 E1      a14 = << 1;
19B 38 E1      a14 = << 1;
19D 38 E1      a14 = << 1;
}

```

```

19F 90 E2
1A1 79 33 04
1A4 C1 0E
1A6 66
    }
    a14 =! a2;
    regad();
    *d0 = b14;
    return;
}
/* areg - set a register mode
*
* entry - none.
*
* uses - a 0,2,12,13,14
*       b 0,12,13
*
* calls - none.
*
* exit - display set up for a-register display
*
*/
areg()
{
1A7 D0 CF 04 FF
1A8 D8 CF F6 33
1AF C0 DF A0 00
1B3 20 20
1B5 98 2F 0F
1B8 98 DF F0
1B8 90 D2
1BD 79 33 04
1C0 C5 E0
1C2 66
    b12 =! ALLDIG!04;
    b12 =& 031766;
    b13 = A<<4;
    reg1: a2 = 0;
    reg2: a2 =& 017;
    a13 =& 0360;
    a13 =! a2;
    regad();
    b14 = ^d0;
    return;
}
/* breg - set b register mode
*
* entry - none.
*
* uses - a 12,13
*       b 12,13
*
* calls - areg.
*
* exit - display set up for b-register display.
*
*/
breg()
{
1C3 D0 CF 0C FF
1C7 D8 CF FE 3F
1CB C0 DF B0 00
1CF 58 E2
    b12 =! ALLDIG!014;
    b12 =& 037776;
    b13 = B<<4;
    goto reg1;
}
/* star - set address mode
*
* entry - none.
*
* uses - a 12,13,14
*       b 12,13,14

```

```

*
* calls - none.
*
* exit - display set up for memory display and
*       all further numbers keyed into the
*       address field. Address is set to
*       current user pc.
*
*/
star()
{
    a12 = & 0362;
    b12 = ! ALLDIG;
    b13 = *(dsp + 4);
    b14 = ~d13;
    return;
}
/* equal - set data mode
*
* entry - none.
*
* uses - a 12
*       b none.
*
* calls - none.
*
* exit - all further numbers keyed in get
*       stored at the current address.
*
*/
equal()
{
    a12 = ! 1;
    return;
}
/* plus - increment the current address
*
* entry - b13 = current address
*
* uses - a 2,13,14
*       b 2,13,14
*
* calls - none.
*
* exit - current address incremented and
*       address/data mode unchanged.
*
*/
plus()
{
    if (!bit(2,a12)) goto 1f;
    a2 = a13;
    ++a2;
    goto reg2;
1:
    ++b13;
    b14 = ~d13;
}
1D1  98 CF F2
1D4  D0 CF 00 FF
1D8  C6 DF 04
1DB  C5 ED
1DD  66

1DE  90 CF 01
1E1  66

1E2  52 C2 07
1E5  80 20
1E7  28 20
1E9  58 CA
1EB  68 D0
1ED  C5 ED

```

1EF 66

```
        return;
}
/* minus - decrement the current address
 *
 * entry - b13 = current address
 *
 * uses - a 2,13,14
 *        b 2,13,14
 *
 * calls - none.
 *
 * exit - current address decremented and
 *        address/data mode unchanged.
 */
```

```
minus()
{
```

1F0 52 C2 07
1F3 80 2D
1F5 28 28
1F7 58 BC
1F9 68 D8
1FB C5 ED
1FD 66

```
        if (!bit(2,a12)) goto 1f;
        a2 = a13;
        --a2;
        goto reg2;
1:      --b13;
        b14 = *d13;
        return;
}
```

```
/* rptr - display user rp
 *
 * entry - none.
 *
 * uses - a 12,13,14
 *        b 12,13,14
 *
 * calls - none.
 *
 * exit - current address set to location
 *        containing the user rp.
 */
```

```
rptr()
{
```

1FE 6F DF 00
201 7D DF 02
204 C5 ED
206 D0 CF 00 FF
20A 98 CF F2
20D 66

```
        b13 = &(sp+0);
        b13 += 2;
rptr1:  b14 = *d13;
        b12 = ! ALLOIG;
        a12 = & 0362;
        return;
}
```

```
/* exec - execute user program
 *
 * entry - b13 = starting address
 *
 * uses - a 0
 *        b 0
 *
 * calls - none.
 *
```

```

* exit - to user program.
*
*/
exec()
{
20E 44 00          b0 = pop();
210 C2 FD 02      *(dsp+2) = b13;
213 45           rp = pop();
214 67           ireturn();
}
/* sst - single step user program
*
* entry - b13 = current address to execute
*
* uses - none.
*
* calls - none.
*
* exit - None. Interrupt will automatically
*        occur before one user instruction
*        can complete.
*/
sst()
{
215 80 AF 7F      a10 = SSTDEL;
218 79 DF 00      1:  disp();
21B 28 AB        --a10;
21D 40 F0 F9      if (!neg) goto 1b;
220 44 00      sst0: b0 = pop();
222 80 CF 12      a12 = 022;
225 C2 FD 02      *(dsp+2) = b13;
228 82 BF 02 80  PC10 = 0200;
22C 7F          nop();
22D 45          rp = pop();
22E 67          ireturn();
}
/* move - move block of memory
*
* entry - b8 = fwa of destination
*        b9 = fwa of source
*        b10 = lwa+1 of source
*
* uses - a 0,8,9
*        b 8,9
*
* calls - none.
*
* exit - (b9) to (b10-1) moved to b8.
*/
move()
{
22F 01 02          set(zero);
231 F0 9A          b9 = b10;
233 64 01          if (zero) return;
}

```

```

235 87 09
237 83 80
239 58 F4

```

```

    a0 = *b9++;
    *b8++ = a0;
    goto move;
}
/* shift4 - shift b0 left by 4
 *
 * entry - b0 = 16-bit number to be shifted
 *         logically.
 *
 * uses - a 0,1
 *        b 0,1
 *
 * calls - none.
 *
 * exit - b0 shifted left 4.
 */

```

```

238 E8 00
23D E8 00
23F E8 00
241 E8 00
243 66

```

```

shift4()
{
    b0 += b0;
    b0 += b0;
    b0 += b0;
    b0 += b0;
    return;
}

```

```

7C0 4D 63 54 75 74 6F
7C6 72 20 45 78 65 63
7CC 20 31 2E 30 0A 0D
7D2 00

```

```

char header[] "McTutor Exec 1.0\n\n";

```

```

7D3 68 2A 67 73
7D7 0D 0A 21 6C
7D8 2F 69 72

```

```

char tty[] { 'h', '*', 'g', 's' }
           { '\r', '\n', '|', '|' }
           { '/', '|', 'r' };

```

```

7DE 99 03 8D 02 A9 02
7E4 20 02
7E6 90 03 EB 03 08 03
7EC 22 03
7EE 83 02 29 00 99 02

```

```

int ttyf[] { &half, &addr, &run, &sst0 }
           { &retn, &linefd, &unix, &load }
           { &raddr, &init0, &rpoint };

```

```

/* tty - main teletype controller
 *
 * entry - none.
 *
 * uses - a 0,7,10,13,14
 *        b 0,10,13
 *
 * calls - baud, prstring, rdtty, prtty, ktype
 *
 * exit - none. (it doesn't)
 */
tty()

```

```

244 44 00
246 79 A1 04
249 80 7F 40
24C 80 CF 02
24F 79 76 04
252 C0 DF C0 07
256 79 91 04
259 C6 DF 02
25C 79 05 04
25F 79 C6 04
262 98 EF 7F
265 49 F1 27 00
269 5A C4 04
26C 79 14 05
26F 80 EF 41
272 48 F8 0A
275 80 EF 5A
278 40 F9 04
27B 90 EF 20
27E 79 3F 04
281 58 DC

```

```

283 98 CF 12
286 90 CF 68
289 80 DF A0
28C 66

```

```

{
    b0 = pop();
    baud();
    a7 = 0100;
    a12 = 02;
    lfcn();
    b13 = &header;
    prstring();
    b13 = *(dsp + 2);
    prloc();
1:  ndtty();
    a14 = & 0177;
    if {zero} goto reset;
    if (bit(4,a12)) goto 2f;
    prtty();
2:  a14 = 'A';
    if (!t) goto 3f;
    a14 = 'Z';
    if (!iteq) goto 3f;
    a14 = ' 040';
3:  ktype();
    goto 1b;
}
/* raddr - set register mode
 *
 * entry - none.
 *
 * uses - a 12,13
 *        b none.
 *
 * calls - none.
 *
 * exit - state bits set up for register
 *        operations.
 */
raddr()
{
    a12 = & 022;
    a12 = ' 0150';
    a13 = A<<4;
    return;
}
/* addr - '*' key => set up to input address
 *
 * entry - none
 *
 * uses - a 12
 *        b 10,13
 *
 * calls - none.
 *
 * exit - current address set to origin of ram
 *        and ready to be changed.
 */

```

```

280 C6 AF 04
290 C0 DA
292 98 CF 12
295 90 CF 20
298 66

addr()
{
    b10 = *(dsp + 4);
    b13 = b10;
    a12 = & 022;
    a12 = | 040;
    return;
}
/* rpoint - set address to rp
 *
 * entry - none.
 *
 * uses - a 12,13
 *       b 13
 *
 * calls - lfcn, p1oc
 *
 * exit - current address set to base of stack
 *        which contains the user rp.
 */
rpoint()
{
    b13 = &(sp + 0);
    b13 = + 2;
    a12 = & 022;
    lfcn();
    p1oc();
    return;
}
/* run - set execute bit in status byte
 *
 * entry - none.
 *
 * uses - a 12
 *       b none.
 *
 * calls - none.
 *
 * exit - execute bit set so next return
 *        will start execution.
 */
run()
{
    a12 = | 0200;
    return;
}
/* pnum - print 8-bit number on tty
 *
 * entry - a9 = number to be printed
 *
 * uses - none.
 *
 * calls - pnti.
 *

```

```

299 6F DF 00
29C 7D DF 02
29F 98 CF 12
2A2 79 76 04
2A5 79 05 04
2A8 66

```

```

2A9 90 CF 80
2AC 66

```



```

    * exit - number printed as two hex digits.
    */
prnum()
{
    prn1();
    prn1();
    return;
}
/* prn1 - print 4-bit number as hex digit
 * entry - a9 = upper 4 bits is number to be
 *         printed.
 *
 * uses - a 9,14
 *        b none.
 *
 * calls - prtty.
 *
 * exit - digit printed and a9 shifted left
 *        by 4.
 */
prn1()
{
    a9 = <<< 1;
    a9 = <<< 1;
    a9 = <<< 1;
    a9 = <<< 1;
    a14 = a9;
    a14 = & 017;
    a14 = 9;
    if (lreq) goto 1f;
    a14 =+ 'A' - '0' - 10;
1:   a14 =+ '0';
    prtty();
    return;
}
/* tnumb - number input from tty
 *
 * entry - b10 = current number being built up
 *         a14 = character input
 *
 * uses - a 0,10,12,13,14
 *        b 0,10,13
 *
 * calls - shift4.
 *
 * exit - new digit shifted into the right
 *        of b10.
 */
tnumb()
{
    a14 = & 017;
    if (!bit(3,a12)) goto 2f;

```

```

2AD 79 B4 02
2B0 79 B4 02
2B3 66

```

```

2B4 34 91
2B6 34 91
2B8 34 91
2BA 34 91
2BC 80 E9
2BE 98 EF 0F
2C1 80 EF 09
2C4 48 F9 04
2C7 A8 EF 07
2CA A8 EF 30
2CD 79 14 05
2D0 66

```

```

2D1 98 EF 0F
2D4 52 C3 1C

```

```

2D7 52 C6 10
2DA 98 CF BF
2DD 80 EF 08
2E0 65 01
2E2 90 CF 04
2E5 80 DF 80
2E8 66
2E9 52 C5 07
2EC 98 DF F0
2EF 90 DE
2F1 66
2F2 C0 0A
2F4 79 3B 02
2F7 C0 A0
2F9 98 AF F0
2FC 90 AE
2FE 5A C5 05
301 90 CF 01
304 66
305 C0 DA
307 66

```

```

        if (!bit(6,a12)) goto 1f;
        a12 = & 0277;
        a14 = 11;
        if (!zero) return;
        a12 = | 04;
        a13 = 8<<4;
        return;
1:      if (!bit(5,a12)) goto 2f;
        a13 = & 0360;
        a13 = | a14;
        return;
2:      b0 = b10;
        shift4();
        b10 = b0;
        a10 = & 0360;
        a10 = | a14;
        if (bit(5,a12)) goto 1f;
        a12 = | 1;
        return;
1:      b13 = b10;
        return;
}

```

```

/* unix - listen to modem
 *
 * entry - none.
 *
 * uses - a 14.
 *        b none.
 *
 * calls - rdttty.
 *
 * exit - when a '!' is received from the modem.
 *
 */

```

```

unix():

```

```

308 82 BF 02 01
30C 79 C6 04
30F 98 EF 7F
312 80 EF 21
315 40 F1 F5
318 22 80 02
31B 79 14 05
31E 79 76 04
321 66

```

```

        PC10 = 01;
1:      rdttty();
        a14 = & 0177;
        a14 = '!';
        if (!zero) goto 1b;
        PC10 = 0;
        prtty();
        lfcn();
        return;
}

```

```

/* load - load hex file from modem
 *
 * entry - none.
 *
 * uses - a 7,8,9,13,14
 *        b 13
 *
 * calls - rdmod, lfcn, prloc, getbyt.
 *
 * exit - next location that would have been

```

```

322 82 BF 02 01
326 80 7F 20
329 20 80
32B 79 0D 05
32E 99 EF 7F
331 80 EF 3A
334 40 F1 F5
337 79 71 03
33A 80 9A
33C 40 F1 10
33F 22 B0 02
342 80 7F 40
345 81 D8
347 79 76 04
34A 79 05 04
34D 66
34E 79 71 03
351 80 DA
353 79 71 03
356 6A D0
358 80 DA
35A 79 71 03
35D 28 98
35F 48 F0 08
362 79 71 03
365 83 DA
367 58 F4
369 79 71 03
36C 48 F1 BD
36F 58 CE

```

```

*      load is printed. Only ':00' is
*      printed from the last line of the
*      hex file if the load is successful.
*      Otherwise there was a checksum error
*      in the last line listed.
=
*/

```

```

load()
{
    PCID = 01;
    a7 = 040;
    a8 = 0;
1:  rdmod();
    a14 = & 0177;
    a14 = ':';
    if (!zero) goto 1b;
    getbyt();
    a9 = a10;
    if (!zero) goto 3f;
2:  PCID = 0;
    a7 = 0100;
    *b13 = a8;
    lfcn();
    prloc();
    return;
3:  getbyt();
    a13 = a10;
    getbyt();
    swap(b13);
    a13 = a10;
    getbyt();
3:  --a9;
    if (neg) goto 4f;
    getbyt();
    *b13++ = a10;
    goto 3b;
4:  getbyt();
    if (zero) goto 1b;
    goto 2b;
}
/* getbyt - accumulate 8-bit byte from hex file
*
* entry - none.
*
* uses - a 8,10
*        b none.
*
* calls - digit.
*
* exit - a10 = byte read
*        a8 = current value of check sum.
*
*/
getbyt()
{
    digit();

```

```

371 79 86 03

```

```

374 80 AE          a10 = a14;
376 79 86 03      digit();
379 38 A1          a10 = << 1;
37B 38 A1          a10 = << 1;
37D 38 A1          a10 = << 1;
37F 38 A1          a10 = << 1;
381 90 AE          a10 = | a14;
383 A8 8A          a8 =+ a10;
385 66             return;
}
/* digit - read hex digit from hex file
*
* entry - none.
*
* uses - a 14
*       b none.
*
* calls - rdmod.
*
* exit - a14 = binary value of hex digit read
*/
digit()
{
386 79 0D 05      rdmod();
389 98 EF 7F      a14 = & 0177;
38C 80 EF 39      a14 = '9';
38F 48 F9 04      if (!teq) goto 1f;
392 A8 EF 09      a14 =+ 9;
395 98 EF 0F      1: a14 = & 017;
398 66             return;
}
/* half - set half duplex mode
*
* entry - none.
*
* uses - a 12
*       b none.
*
* calls - none.
*
* exit - bit set in a12 to indicate no echoing
*       of input.
*/
half()
{
399 88 CF 10      a12 = ~ 020;
39C 66             return;
}
retn()
{
39D 79 33 04      negad();
3A0 79 1F 04      store();
3A3 80 EF 0A      a14 = '\n';
3A6 79 14 05      prtty();

```

```

3A9 52 C7 04
3AC 59 0E 02
3AF 52 C3 33
3B2 80 0D
3B4 5A C5 03
3B7 28 00
3B9 98 0F 0F
3BC 98 DF F0
3BF 90 D0
3C1 79 82 04
3C4 80 9D
3C6 79 AD 02
3C9 79 11 05
3CC 79 33 04
3CF C5 90
3D1 52 C2 08
3D4 6A 90
3D6 79 AD 02
3D9 6A 90
3DB 79 AD 02
3DE 79 11 05
3E1 60 A0
3E3 66
3E4 5A C5 1C
3E7 68 D0
3E9 58 17

```

```

        if (!bit(7,a12)) goto 1f;
        goto exec;
1:      if (!bit(3,a12)) goto 2f;
        a0 = a13;
        if (bit(5,a12)) goto ret0;
        ++a0;
ret0:   a0 = & 017;
        a13 = & 0360;
        a13 = ! a0;
        crdel();
        a9 = a13;
        prnum();
        prsp();
        regad();
        b9 = *a0;
        if (!bit(2,a12)) goto 1f;
        swap(b9);
        prnum();
        swap(b9);
1:      prnum();
        prsp();
        b10 = 0;
        return;
2:      if (bit(5,a12)) goto 1nf1;
        ++b13;
        goto 1nf1;
    }
    /* linefd - print previous location
    *
    * entry - b13 = current address
    *         a10 = possible value to store in
    *             current address.
    *
    * uses - a 8,9,10,13,14
    *        b 9,10,13
    *
    * calls - store, prtty, prnum, prsp.
    *
    * exit - any pending values are stored in
    *         the location if necessary and
    *         the previous location is displayed.
    */
    linefd()
    {
        regad();
        store();
        a14 = '\n';
        prtty();
        if (!bit(3,a12)) goto 1f;
        a0 = a13;
        --a0;
        goto ret0;
1:      --b13;
1nf1:   crdel();
prloc:  b9 = b13;

```

```

3E8 79 33 04
3EE 79 1F 04
3F1 80 EF 0D
3F4 79 14 05
3F7 52 C3 07
3FA 80 0D
3FC 28 08
3FE 58 B9
400 68 08
402 79 82 04
405 C0 9D

```

```

407 6A 90          swap(b9);
409 79 AD 02      prnum();
40C 6A 90          swap(b9);
40E 79 AD 02      prnum();
411 79 11 05      prsp();
414 85 9D          a9 = *b13;
416 79 AD 02      prnum();
419 79 11 05      prsp();
41C 60 A0          b10 = 0;
41E 66            return;
}
/* store - store value in current location
 *
 * entry - b0 = register address if necessary
 *         b13 = current address
 *         a10 = value to be stored there
 *
 * uses - a 10,12,13
 *        b 10,13
 *
 * calls - none.
 *
 * exit - if necessary, value stored in current
 *         location and status updated to
 *         indicate no value to be stored.
 */
store()
{
41F 52 C0 0F      if (!bit(0,a12)) goto 2f;
422 52 C3 0A      if (!bit(3,a12)) goto 1f;
425 81 0A          *b0 = a10;
427 52 C2 07      if (!bit(2,a12)) goto 2f;
42A C1 0A          *d0 = b10;
42C 58 02          goto 2f;
42E 81 DA          1: *b13 = a10;
430 60 A0          2: b10 = 0;
432 66            return;
}
/* negad - calculate user register address
 *
 * entry - a13(0-3) = register number
 *         *(dsp+4) = user register pointer
 *
 * uses - a 0
 *        b 0
 *
 * calls - none.
 *
 * exit - b0 = address of desired register.
 */
negad()
{
433 80 0D          a0 = a13;
435 D8 0F 0F 00    b0 = & 017;

```

```

439 38 01
43B EE 0F 04
43E 66

```

```

43F 80 EF 30
442 48 FB 08
445 60 EF 39
448 49 F9 D1 02
44C 80 EF 61
44F 48 FB 0D
452 B0 EF 66
455 40 F9 07
458 A8 EF 09
45B 59 D1 02
45E C0 1F D3 07
462 C0 2F DC 07
466 7D 2F 02
469 87 01
46B 64 01
46D B0 0E
46F 40 F1 F5
472 C5 22
474 49 2F

```

```

a0 = << 1;
b0 =+ *(dsp + 4);
return;
}
/* ktype - determine key type
*
* entry - a14 = ascii character input
*
* uses - a 0,1,2,14
*        b 1,2
*
* calls - space, addr, go, retrn, linefd, unix,
*         load, tnumb.
*
* exit - to appropriate processing routine.
*        In the case of 'tnumb' characters
*        'a' - 'f' are converted into easy
*        to convert values.
*/
ktype()
{
    a14 = '0';
    if (lt) goto 1f;
    a14 = '9';
    if (!teq) goto tnumb;
1:  a14 = 'a';
    if (lt) goto 1f;
    a14 = 'f';
    if (!lteq) goto 1f;
    a14 =+ 9;
    goto tnumb;
1:  b1 = &tty;
    b2 = &ttyf - 2;
1:  b2 =+ 2;
    a0 = *b1++;
    if (zero) return;
    a0 = a14;
    if (!zero) goto 1b;
    b2 = *d2;
    goto *b2;
}
/* lfcn - output linefeed and carriage return
*
* entry - none.
*
* uses - a 0,14
*        b none.
*
* calls - prtty, bitime
*
* exit - carriage moved to new line and delay
*        done to allow time for this.
*/
lfcn()

```

```

476 80 EF 0A      {
479 79 14 05      a14 = '\n';
47C 80 EF 0D      prtty();
47F 79 14 05      prcr: a14 = '\n';
482 98 CF 9E      prtty();
485 80 0F 32      crdel: a12 = & 0236;
488 79 59 01      a0 = NBIT;
48B 28 08      1: bitime();
48D 40 F1 F9      --a0;
490 66           if (!zero) goto 1b;
                    return;
                    }
/* prstring - print out '\0' terminated string
 *
 * entry - b13 = pointer to string.
 *
 * uses - a 13,14
 *       b 13
 *
 * calls - prtty, crdel.
 *
 * exit - string printed out on terminal.
 */
prstring()
{
491 87 ED      1: a14 = *b13++;
493 64 01      if (zero) return;
495 79 14 05      prtty();
498 80 EF 0D      a14 = '\n';
49B 69 F1 82 04  if (zero) crdel();
49F 58 F0      goto 1b;
}
/* baud - determine baud rate of terminal
 *
 * entry - BAUD = 0 => baud rate unknown
 *
 * uses - a 0,5
 *       b 0
 *
 * calls - none.
 *
 * exit - BAUD contains delay count that enables
 *        bitime to wait one bit time.
 */
baud()
{
4A1 82 BF 03 92  PCNTRL = 0222;
4A5 C5 0F FE 1B  b0 = BAUD;
4A9 65 01      if (!zero) return;
4AB 58 B6 14      if (bit(6,PAIO)) goto 3f;
4AE 53 B6 FE      1: if (!bit(6,PAIO)) goto 1b;
4B1 7D 0F 03      1: b0 =+ 3;
4B4 80 5F 02      a5 = 2;
4B7 80 55      a5 = a5;

```



```

4B9 2B 5B
4BB 40 F0 FC
4BE 5B B6 F1
4C1 C1 F0 FE 1B
4C5 66

```

```

4C6 20 10
4C8 20 E0
4CA 86 6B 02
4CD 82 BF 03 92
4D1 82 B6 02
4D4 85 0B
4D6 98 07
4D8 88 01
4DA 48 F1 FB
4DD C5 5F FE 1B
4E1 03 0B
4E3 6A 50
4E5 3C 5F
4E7 6A 50
4E9 3C 5F
4EB 79 5D 01
4EE 80 6F 09
4F1 28 68
4F3 48 F1 14
4F6 79 59 01
4F9 38 EF
4FB 85 0B
4FD 98 07
4FF 88 01
501 40 F1 EE
504 90 EF 80
507 58 E8
509 79 59 01
50C 66

```

```

2:    --a5;
      if (!neg) goto 2b;
      if (bit(6,PAIO)) goto 1b;
3:    BAUD = b0;
      return;
}
/* rdtty - read character from tty
 *
 * entry - none.
 *
 * uses - a 0,1,5,6,14
 *        b 5
 *
 * calls - delay, bitime.
 *
 * exit - a14 = character read
 *
 */
rdtty()
{
rdt0:  a1 = 0;
      a14 = 0;
      a6 = PCIO;
      PCNTRL = 0222;
      PCIO = a6;
1:    a0 = PAIO;
      a0 = & a7;
      a0 = ^ a1;
      if (zero) goto 1b;
      b5 = BAUD;
      clear(carry);
      swap(b5);
      a5 =>>$ 1;
      swap(b5);
      a5 =>>$ 1;
      delay();
      a6 = 9;
1:    --a6;
      if (zero) goto 2f;
      bitime();
      a14 =>> 1;
      a0 = PAIO;
      a0 = & a7;
      a0 = ^ a1;
      if (!zero) goto 1b;
      a14 = 0200;
      goto 1b;
2:    bitime();
      return;
}
/* rdmod - read character from modem
 *
 * entry - none.
 *
 * uses - a 1.
 *        b none.

```

```

*
* calls - rdttty.
*
* exit - through rdttty. a14 = character read.
*
*/
rdmod()
{
50D 80 17          a1 = a7;
50F 58 B7          goto rdt0;
}
/* prtty - print character to tty.
*
* entry - a14 = character to be printed
*
* uses - a 0,6,14
*        b none.
*
* calls - bitime.
*
* exit - character written out to terminal.
*
*/
prsp()
{
511 80 EF 20          a14 = ' ';
514 86 68 02          prtty: a6 = PCIO;
517 82 BF 03 82          PCNTRL = 0202;
51B 82 B6 02          PCIO = a6;
51E 21 80             PAIO = 0;
520 79 59 01          bitime();
523 80 6F 08          a6 = 8;
526 80 0F 20          1:  a0 = 040;
529 5A E0 03          if (bit(0,a14)) goto 2f;
52C 20 00             a0 = 0;
52E 34 EF             2:  a14 =>>> 1;
530 81 80             PAIO = a0;
532 79 59 01          bitime();
535 28 68             --a6;
537 40 F1 ED          if (!zero) goto 1b;
53A 81 BF 20          PAIO = 040;
53D 79 59 01          bitime();
540 66               return;
}

/* prom - write a prom
*
* entry - b9 = starting address
*
* uses - a 1,7,12
*        b none.
*
* calls - verify, zapall.
*
* exit - 1024 bytes starting at b9 are written
*        out, then the prom is verified to

```

```

*           see that everything was written out
*           correctly.  If everything is OK the
*           address displayed will be 400.
*
*/
prom()
{
541 82 BF 03 90          PCNTRL = SETMOD ; AINP;
545 82 BF 07 80          QCNTRL = SETMOD;
549 80 7F 6E            a7 = 110;
54C 79 8B 05            zapali();
54F 28 78              --a7;
551 40 F1 F9            if (!zero) goto 1b;
554 79 7B 05            verify();
557 22 B0 02            netd: PCIO = 0;
55A 0D 0F BB 1B          sp = USERRG - 2 - 1 - 2;
55E C2 FF 03 35 00      *(dsp + 3) = &init;
563 22 F0 02            *(sp + 2) = 0;
566 C2 FF 00 00 1B      *(dsp + 0) = RAMORG;
56B 47                  push(rp);
56C 4D 0F E0 1B          rp = SYSREG;
570 C5 DF DA 1B          b13 = USERB13;
574 C5 EF DC 1B          b14 = USERB14;
578 59 60 00            goto man2;
}
/* verify - verify information in prom
*
* entry - a12(4) = zero/data verify
*         b9 = starting address for data verify
*
* uses - a 0,1,3,4,9,13,14
*        b 0,9,13
*
* calls - none.
*
* exit - if no errors then return.  If error
*        then return one level up and set
*        b13 to prom address in error,
*        b14(15-8) to prom data, and
*        b14(7-0) to expected data.
*/
verify()
{
578 60 D0              b13 = 0;
57D 20 30              a3 = 0;
57F 80 4F 20          a4 = 040;
582 82 BF 03 90          PCNTRL = 0220;
586 82 BF 07 9B          QCNTRL = 0233;
58A 82 84 02          PCIO = a4;
58D C0 5F FF 3F        b5 = 0x3fff;
591 79 5D 01          delay();
594 82 B3 01            1:  PBIO = a3;
597 82 84 02          PCIO = a4;
59A 86 EB 04          a14 = PDIO;
59D 87 19              a1 = *b9++;

```

```

59F 80 1E
5A1 48 F1 07
5A4 6A E0
5A6 80 E1
5A8 58 AD
5AA 68 D0
5AC 28 30
5AE 40 F1 E4
5B1 A8 4F 02
5B4 80 4F 28
5B7 48 F8 D8
5BA 66

5BB C0 D9
5BD 20 30
5BF 20 40
5C1 87 00
5C3 82 B3 01
5C6 82 B4 02
5C9 82 B0 04
5CC 88 4F 40
5CF 82 B4 02
5D2 C0 5F 1C 00
5D6 79 5D 01
5D9 88 4F 40
5DC 82 B4 02
5DF 28 30
5E1 40 F1 DE
5E4 A8 4F 02
5E7 80 4F 08
5EA 48 F8 D5
5ED 66

```

```

a1 = a14;
if (zero) goto 2f;
swap(b14);
a14 = a1;
goto retd;
2:
++b13;
++a3;
if (!zero) goto 1b;
a4 =+ 02;
a4 = 050;
if (!t) goto 1b;
return;
}
/* zapall - write all locations in prom
 *
 * entry - b9 = starting address to write
 *
 * uses - a 0,3,4,5,13
 *        b 5,13
 *
 * calls - none.
 *
 * exit - all locations of the prom hit with a
 *        1 msec. write pulse.
 */
zapall()
{
    b13 = b9;
    a3 = 0;
    a4 = 0;
1:
    a0 = *b13++;
    PB10 = a3;
    PC10 = a4;
    PD10 = a0;
    a4 = ^ 0100;
    PC10 = a4;
    b5 = 28;
    delay();
    a4 = ^ 0100;
    PC10 = a4;
    ++a3;
    if (!zero) goto 1b;
    a4 =+ 02;
    a4 = 010;
    if (!t) goto 1b;
    return;
}
/* loadt - load file from tape
 *
 * entry - a8 = file id
 *        b9 = fwa of load
 *
 * uses - a 1,3,7,8,9,12,13,14
 *        b 13
 *
 *

```

```

* calls - rdbit, rdchar, rdbyte, rdlib
*
* exit - file with matching id from tape
*         loaded into ram.
*
*/
loadt()
{
    PCNTRL = SETMOD ; AINP;
    QCNTRL = SETMOD;
    b13 = b9;
    a12 = & 037;
    a9 = a8;
    if (!homog) goto 1f;
    a12 = | 040;
    if (zero) goto 1f;
    a12 = | 0100;
1:   PDIO = a12;
sync: rdbit();
    a1 =>> 1;
    a1 = | a0;
    a1 - SYNC;
    if (!zero) goto sync;
    a3 = 10;
1:   rdchar();
    a1 - SYNC;
    if (!zero) goto sync;
    --a3;
    if (!zero) goto 1b;
1:   rdchar();
    a1 - STARTCH;
    if (zero) goto 1f;
    a1 - SYNC;
    if (!zero) goto sync;
    goto 1b;
1:   b8 = CHECKSUM; /* initialize check sum character */
    rdbyte();
    if (bit(5,a12)) goto 1f; /* accept anything */
    a7 = a9;
    if (!zero) goto sync; /* wrong id */
1:   rdbyte();
    swap(b7);
    rdbyte();
    swap(b7);
    if (bit(6,a12)) goto 1f; /* ignore addr on tape */
    b13 = b7;
1:   rdchar();
    a1 - ENDCH;
    if (zero) goto 1f;
    rdlib();
    *b13++ = a7;
    goto 1b;
1:   rdchar();
    b14 = b8;
    a14 = & 0177;
    return;
}
5EE 82 BF 03 90
5F2 82 BF 07 80
5F6 C0 D9
5FB 98 CF 1F
5FB 80 98
5FD 40 FB 0A
600 90 CF 20
603 48 F1 04
606 90 CF 40
609 82 BC 04
60C 79 B2 06
60F 38 1F
611 90 10
613 B0 1F 16
616 40 F1 F4
619 80 3F 0A
61C 79 8B 06
61F B0 1F 16
622 40 F1 E8
625 28 38
627 40 F1 F3
62A 79 8B 06
62D B0 1F 2A
630 48 F1 09
633 B0 1F 16
636 40 F1 D4
639 5B EF
63B C0 8F 2A 00
63F 79 72 06
642 5A C5 06
645 B0 79
647 40 F1 C3
64A 79 72 06
64D 6A 70
64F 79 72 06
652 6A 70
654 5A C6 03
657 C0 D7
659 79 8B 06
65C B0 1F 2F
65F 48 F1 08
662 79 75 06
665 83 D7
667 5B F0
669 79 8B 06
66C C0 E8
66E 98 EF 7F
671 66

```

```

}
/* rdbyte - read 8-bit byte from tape
 *
 * entry - none.
 *
 * uses - a 1,7
 *        b none.
 *
 * calls - rdchar.
 *
 * exit - a7 = 8-bit byte assembled from 2 pseudo
 *        ascii characters on the tape.
 */
rdbyte()
{
672 79 8B 06      rdchar();
675 98 1F 0F      rdnib: a1 = & 017;
678 80 71          a7 = a1;
67A 38 71          a7 = << 1;
67C 38 71          a7 = << 1;
67E 38 71          a7 = << 1;
680 38 71          a7 = << 1;
682 79 8B 06      rdchar();
685 98 1F 0F      a1 = & 017;
688 90 71          a7 = | a1;
68A 66            return;
}
/* rdchar - read pseudo ascii character from tape
 *
 * entry - none.
 *
 * uses - a 1,2,6
 *        b none.
 *
 * calls - rdbit.
 *
 * exit - a1 = Character read from tape.
 */
rdchar()
{
688 80 2F 08      a2 = 8;
68E 79 B2 06      1: rdbit();
691 38 1F          a1 = >> 1;
693 90 10          a1 = | a0;
695 28 28          --a2;
697 40 F1 F5      if (!zero) goto 1b;
69A 82 B1 04      PDIO = a1;
69D 88 81          a8 = ^ a1;
69F 80 01          a0 = a1;
6A1 98 1F 7F      a1 = & 0177;
6A4 0E 00          a0 = bitsum(a0);
6A6 88 0F 01      a0 = ^ 1;
6A9 D8 0F 01 00   b0 = & 1;
6AD 6A 00          swap(b0);
}

```

```
6AF EB 80
6B1 66
```

```
        b8 += b0;
        return;
    }
    /* rdbit - read a bit from the tape
     *
     * entry - none.
     *
     * uses - a 0
     *         b none.
     *
     * calls - none.
     *
     * exit - a0(7) = bit read.
     *
     * note - the loop at '2' is very time critical
     *        and dependant upon the way outbit puts
     *        out bits.
     */
    rdbit()
    {
```

```
6B2 20 00
6B4 53 B7 FE
6B7 28 00
6B9 5B B7 FC
6BC 80 0F 20
6BF 48 FA F1
6C2 98 0F 80
6C5 66
```

```
        a0 = 0;
1:      if (!bit(7,PAIO)) goto 1b;
2:      ++a0;
        if (bit(7,PAIO)) goto 2b;
        a0 = NOISE;
        if (!teq) goto rdbit;
        a0 = & 0200;
        return;
    }
```

```
    /* dumpt - dump file to tape
     *
     * entry - a8 = file id
     *         b9 = fwa to dump
     *         b10 = lwa+1 to dump
     *
     * Uses - a 1,4,7,9,10,13
     *        b 7,13
     *
     * calls - outch, outbyte
     *
     * exit - fwa to lwa stored on tape
     */
    dumpt()
    {
```

```
6C6 82 BF 03 80
6CA 82 BF 07 80
6CE C0 D9
6D0 C0 1F 64 16
6D4 6A 10
6D6 79 31 07
6D9 6A 10
6DB 28 18
6DD 40 F1 F5
6E0 20 90
```

```
        PCNTRL = SETMOD;
        QCNTRL = SETMOD;
        b13 = b9;
        b1 = SYNC<<8 | 100;
1:      swap(b1);
        outch();
        swap(b1);
        --a1;
        if (!zero) goto 1b;
        a9 = 0;
```

```

6E2 80 1F 2A
6E5 79 31 07
6EB 80 78
6EA 79 1A 07
6ED C0 7D
6EF 79 1A 07
6F2 6A 70
6F4 79 1A 07
6F7 01 02
6F9 F0 AD
6FB 48 F1 08
6FE 87 7D
700 79 1A 07
703 58 F2
705 80 1F 2F
708 79 31 07
70B 80 19
70D 79 33 07
710 80 1F 04
713 79 31 07
716 79 31 07
719 66

```

```

        a1 = STARTCH;
        outch();
        a7 = a8;
        outbyte();
        b7 = b13;
        outbyte();
        swap(b7);
        outbyte();
1:      set(zero);
        b10 = b13;
        if (zero) goto 1f;
        a7 = *b13++;
        outbyte();
        goto 1b;
1:      a1 = ENDCH;
        outch();
        a1 = a9;
        outchar();
        a1 = EOT;
        outch();
        outch();
        return;
}
/* outbyte - output one 8-bit byte to tape
 *
 * entry - a7 = byte to be written
 *
 * uses - none.
 *
 * calls - out4.
 *
 * exit - a7 written out as two pseudo ascii
 *        characters and contents of a7 unchanged.
 */
outbyte()
{
        out4();
        out4();
        return;
}
/* out4 - output 4-bit nibble as pseudo ascii
 *        character.
 *
 * outch - output ascii character
 *
 * entry - a7 = 4-bit nibble to be written out (out4)
 *        a1 = ascii character to output (outch)
 *
 * uses - a 0,1,2,4,7,9
 *        b 0
 *
 * calls - outbit.
 *
 * exit - one ascii character written to tape.
 */

```

```

71A 79 21 07
71D 79 21 07
720 66

```



```

721 34 71
723 34 71
725 34 71
727 34 71
729 80 17
72B 98 1F 0F
72E 90 1F 30
731 88 91
733 82 81 04
736 0E 01
738 98 0F 01
73B 88 0F 01
73E 34 0F
740 90 10
742 80 4F 08
745 C0 0F 0C 18
749 34 1F
74B 48 F0 05
74E C0 0F 0C 0C
752 80 2F 0D
755 79 66 07
758 80 2F 06
75B 6A 00
75D 79 66 07
760 28 48
762 40 F1 E1
765 66

out4()
{
    a7 = <<<< 1;
    a7 = <<<< 1;
    a7 = <<<< 1;
    a7 = <<<< 1;
    a1 = a7;
    a1 = & 017;
    a1 = | '0';
outch: a9 = ^ a1;
outchar: PAID = a1;
    a0 = bitsum(a1);
    a0 = & 1;
    a0 = ^ 1;
    a0 = >>>> 1;
    a1 = | a0;
    a4 = 8;
1: b0 = BIT1;
    a1 = >>>> 1;
    if (neg) goto 2f;
    b0 = BIT0;
2: a2 = CYCLE0;
    outbit();
    a2 = CYCLE1;
    swap(b0);
    outbit();
    --a4;
    if (!zero) goto ib;
    return;
}
/* outbit - output stream of bits to tape
 *
 * entry - a0 = twice the number of 1 bits desired
 *         a2 = length of each 1 bit
 *
 * uses - a 0,3,5
 *        b none.
 *
 * calls - none.
 *
 * exit - a square wave of proper frequency and
 *        length is written to the tape.
 */
outbit()
{
    a3 = 0;
1: a3 = ^ 020;
    PAID = a3;
    a5 = a2;
2: --a5;
    if (!neg) goto 2b;
    --a0;
    if (!zero) goto 1b;
    return;
}
766 20 30
768 88 3F 10
76B 81 83
76D 80 52
76F 28 58
771 40 F0 FC
774 28 08
776 40 F1 F0
779 66

```

```

/* powon - delay for power on
*
* entry - none.
*
* uses - a 5
*       b 5
*
* calls - reset.
*
* exit - Delay done to enable any transient
*        interrupts caused by the noisy
*        transformer to disappear. This
*        routine had to be encoded into
*        the data section since it is a patch
*        that must be at the very end of the
*        executive.
*/

```

```

7F4  C0 5B          char powon[
7F6  68 58          { 0xc0, 0x5b }          /* b5 = b11; */
7F8  40 F0 FC      { 0x68, 0x58 }          /* 1: --b5; */
7FB  59             { 0x40, 0xf0, 0xfc }     /* if (!neg) goto 1b; */
                          /* goto reset; */

7FC  27 00          int powr &reset;

7FE  CD 05          char unused[
                          { 0xcd, 0x05};

```

*** SYMBOL TABLE ***

```

1BFEa - BAUD
1F03a - PCNTRL
1F07a - QCNTRL
1F00a - PAIO
1F01a - PBIO
1F02a - PCIO
1F04a - PDIO
77AD - tfmt
78AD - tnum
7A6D - tfnc
0T - main
27t - reset
29t - init0
35t - init
60t - man2
8Bt - L...0001
9Dt - L...0003
ACT - rdkey
DFT - disp
107T - dsp4
112T - dsp2
13At - dp21
156t - dp22

```

159T - bitime
15DT - delay
165T - numb
179t - L...0005
18At - L...0006
188t - chreg
197t - L...0007
19Ft - L...0008
1A7T - aneg
1B3t - neg1
165t - neg2
1C3T - breg
1D1T - star
1DET - equal
1E2T - plus
1F0T - minus
1FET - nptr
204t - npt1
20ET - exec
215T - sst
220t - sst0
22FT - move
238T - shift4
7C0D - header
7D3D - tty
7DED - ttyf
244f - tty
283T - raddr
28DT - addr
299T - rpoint
2A9T - run
2ADT - pnum
2B4T - pn1
2D1T - tnumb
308T - unix
322T - load
371T - getbyt
386T - digit
399T - half
39DT - retn
3B9t - ret0
3EBT - linefd
402t - lnf1
405T - prioc
41FT - store
433T - negad
43FT - ktype
476T - lfc
47Ct - pncr
482T - cndel
491T - prstring
4A1T - baud
4C6T - rdtty
4CBt - rdt0
50DT - rdmod
511T - prsp

514T - prtty
541T - prom
557t - netd
57BT - verify
58BT - zapall
5EET - loadt
60Ct - sync
672T - rdbyte
675T - rdnib
68BT - rdchar
682T - rdbit
6C6T - dumpt
71AT - outbyte
721T - out4
731T - outch
733T - outchar
766T - outbit
7F4D - powon
7FCD - powr
7FED - unused