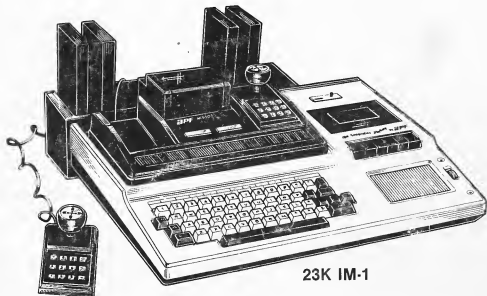


PRICE 19.95

APF™



23K IM-1

PROGRAMMING AND TECHNICAL ASSISTANCE MANUAL

- HIGH RESOLUTION GRAPHICS
- MACHINE LANGUAGE
- MEMORY MAPS
- SCHEMATICS AND PARTS LIST

PROTECTO ENTERPRIZES

BOX 550, BARRINGTON, ILLINOIS 60010

Phone 312/382-5244

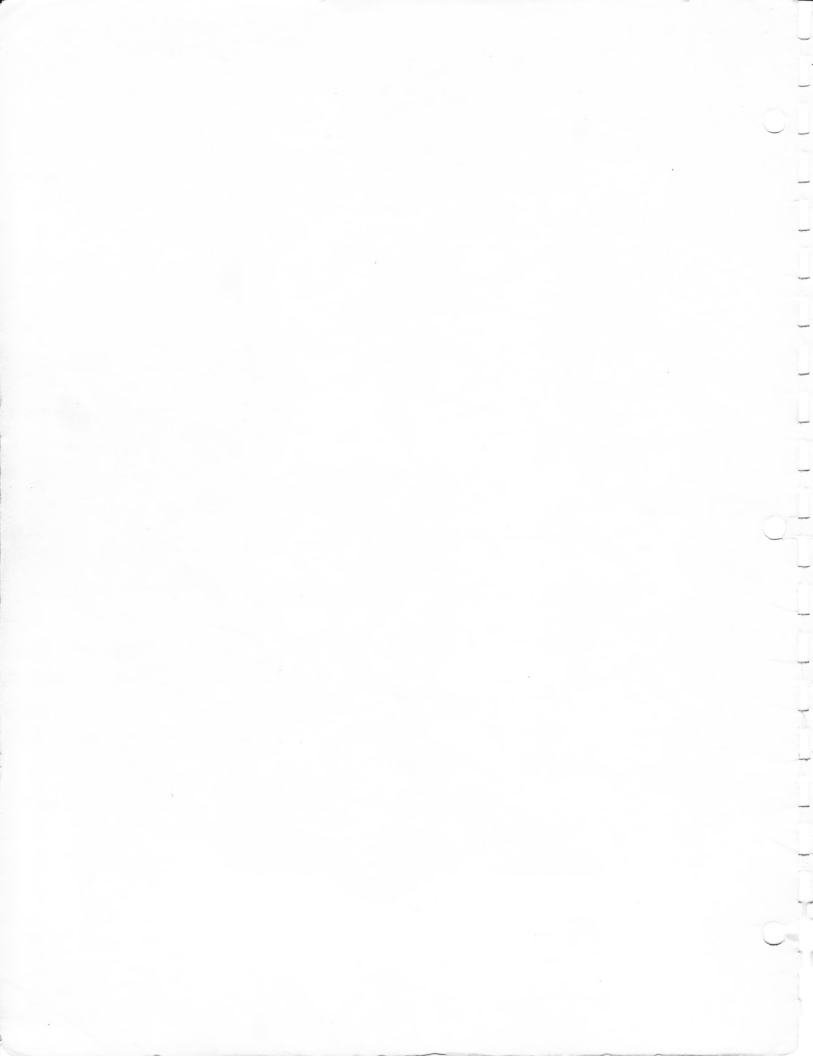


TABLE OF CONTENTS

Chapter

	Introduction
1	The MP1000
2	The MPA-10
3	Memory Maps
4	Memory Useage for Programs and Variables
5	Entering and Using Machine Language
6	Some Useful Routines
7	The Tape System
8	High Resolution Graphics
9	Saving Time and Space

APPENDICES

A	MC6800 Instruction Set Summary
B	Machine Language Reference Mode
C	Schematics and Parts Layout
D	ASCII Character Set/Screen Codes

INTRODUCTION

This APF Imagination Machine Technical Reference Manual is written for those of you who would like to "know" more about the inner workings of the Imagination Machine. It assumes the reader is a technical person who has some general understanding of microcomputers (hardware, software, or both).

Chapters 1-2 and Appendix C supply all of the available electrical schematics. Also, a brief operations description is given. The rest of the manual goes into areas such as memory maps, details of machine language programming, and high resolution graphics mode.

The book is filled with details of information. Read through it all or just the sections that interest you, and then get ready to experiment with your imagination.

Note: Please do not infer from this manual that APF can or will make available design engineers for your circuits or ideas. We present this information so you can enhance your uses and fun of the Imagination Machine. Once you open the cabinets you void the warranties and are on your own.

All information presented in this manual is believed to be accurate and correct.

CHAPTER 1

THE MP1000 DESCRIPTION

Figure 1-2 shows a block diagram of the MP1000. The MP1000 contains the following sections:

1. The Microprocessor Unit (The MPU)
2. The Video Display Generator (The VDG)
3. The T.V. Driver (The MC1372)
4. The Internal ROM
5. The Peripheral Interface Driver (PIA)
6. The IK Read Write Memory for Screen Image
7. Power Supply

THE MPU

The brains of the system is the microprocessing unit (MPU). The MP1000 uses an 8 bit microprocessor - the MC6800. Fuller details of this are available from several semiconductor suppliers. The MPU gets its instructions from the Read Only Memory, processes these instructions and data, stores codes in the read write memory for the VDG to interpret and put out screen patterns, and sends and reads codes from the PIA.

All data is transferred over an 8 bit bidirectional data bus. Addresses are sent out from the MPU on a 16 bit bus (65,536 unique addresses).

The rate at which instructions are executed and data is transferred is set by a biphasic clock input to the CPU. These are $\phi 1$, and $\phi 2$. The MP1000 has a clock rate of 894886 KHZ or a cycle time of 1.1174605 microseconds. This clock is derived by dividing the 3.579545 mhz xtal frequency by 4

MP1000 CLOCK TIMING

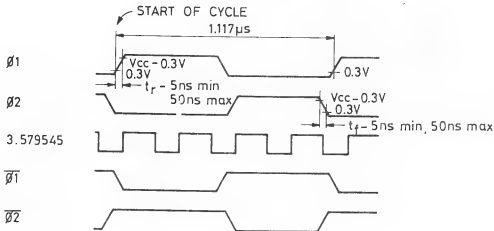


Figure 1-1

APF ELECTRONICS INC.
BLOCK DIAGRAM - MP1000

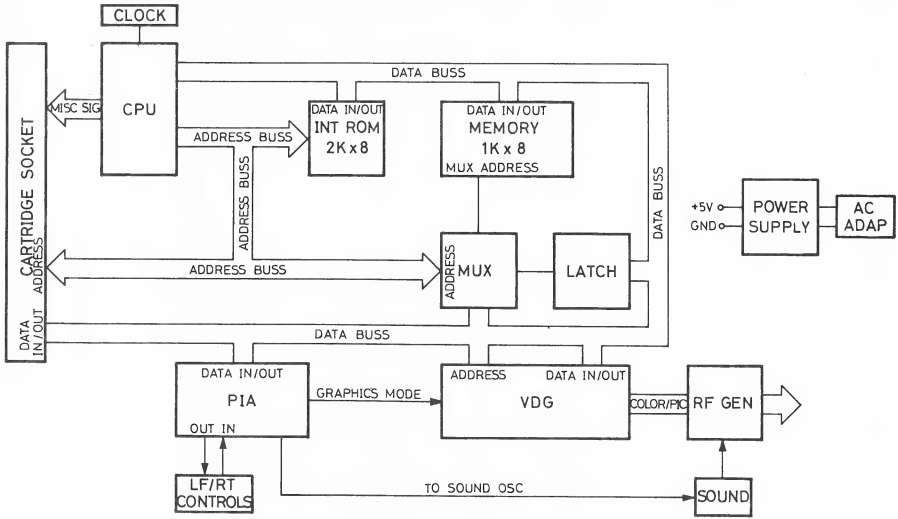


Figure 1-2
MP1000 Block Diagram

During operation of the MPU, instructions are fetched from memory, executed, and the next instruction is fetched. The sequence of which instructions are fetched is determined by the program flow. There are 2 exceptions to this -

Power Up or Reset

Whenever power is turned on or the Reset button is depressed, PIN 40 of the MPU receives a signal which directs it to start a reset or initialization sequence. The starting address of this sequence is stored in ROM memory at locations Hex FFFF and FFFE. The MPU always goes to these locations to find its reset program starting address.

Interrupts

The other time the sequence of instructions can be changed is if an interrupt request is granted. The MPU stops what it is doing and finds the address of the interrupt routines which is stored at Hex FFF8 and FFF9.

VIDEO DISPLAY GENERATOR (VDG)

The VDG is a large scale integrated circuit that scans memory to produce a composite video signal and generates alphanumeric or graphics displays. It always scans memory during Phase 1 so as to not interfere with the MPU. Although the VDG can have up to 14 modes of operation, it is implemented in the MP1000 to have a maximum capability with minimum parts count. A description of the input/output signals is:

Address Output Lines (DA0-DA12) - Thirteen address lines are used by the VDG to scan the display memory. The starting address of the display memory corresponds to the upper left corner of the display screen. As the television signal sweeps from the left to right and top to bottom, the VDG increments the RAM display address.

Data Inputs (DD0-DD7) - Eight data lines are used to input data from RAM to be processed by the VDG. The data is interpreted and transformed into luminance Y (PIN 28) and color outputs ϕA and ϕB (PIN 11 and PIN 10).

Power Inputs - V_{CC} requires +5 volts. V_{SS} requires zero volts and is grounded.

Video Outputs (ϕA , ϕB , Y, CHB) - These four analog outputs are used to transfer luminance and color information to a standard NTSC color television receiver via the MC1372 RF modulator.

LUMINANCE (Y) - This six level analog output contains composite sync., blanking and four levels of video luminance.

ϕA - This three level analog output is used in combination with ϕB and Y outputs to specify one of eight colors.

ϕB - This four level analog output is used in combination with ϕA and Y outputs to specify one of eight colors. Additionally, one analog level is used to specify the time of the color burst reference signal.

CHROMA BIAS (CHB) - This pin is an analog output and provides a D.C. reference corresponding to the quiescent value of ϕA and ϕB . CHB is used to guarantee good thermal tracking and minimize the variation between 1372 and 6847.

Synchronizing Inputs (\overline{MS} , CLK)

Three-State Control - (\overline{MS}) is a TTL compatible input which, when low, forces the VDG address lines into a high impedance state.

Clock (CLK) - The VDG clock input (CLK) uses a 3.579545 MHz (standard) TV crystal frequency square wave. The duty cycle of this clock must be between 45 and 55 percent since it controls the width of alternate dots on the television screen. The MCL372 RF modulator supplies the 3.579545 MHz clock and has provisions for a duty cycle adjustment.

Synchronizing Outputs (\overline{FS} , \overline{HS} , \overline{RP}) - Three TTL compatible outputs provide circuits, exterior to the VDG, with timing references to the following internal VDG states:

FIELD SYNC - (\overline{FS}) - The high to low transition of the \overline{FS} output coincides with the end of active display area. During this time interval an MPU may have total access to the display RAM without causing undesired flicker on the screen. The Low to High transition of \overline{FS} coincides with the trailing edge of the vertical synchronization pulse.

HORIZONTAL SYNC - (\overline{HS}) - The \overline{HS} pulse coincides with the horizontal synchronization pulse furnished to the television receiver by the VDG. The high to low transition of the \overline{HS} outputs coincides with the leading edge of the horizontal synchronization pulse.

ROW PRESET - (\overline{RP}) - An external character generator ROM may be used with the VDG. An external four bit counter must be added to supply row selection. The counter is clocked by the \overline{HS} signal and cleared by the \overline{RP} signal.

Mode Control Lines (Input) ($\overline{A/G}$, $\overline{A/S}$, $\overline{INT/EXT}$, $\overline{GM0}$, $\overline{GM1}$, $\overline{GM2}$, \overline{CSS} , \overline{INV}) - Eight TTL compatible inputs are used to control the operating mode of the VDG. \overline{CSS} and \overline{INV} are changed on a character by character basis. The \overline{CSS} pin is used to select between two possible alphanumeric colors; when the VDG is in the alphanumeric mode and between two color sets when the VDG is in and full Graphic mode.

TABLE OF MODE CONTROL LINES (INPUTS)

<u>A/G</u>	<u>A/S</u>	<u>INT/EXT</u>	<u>INV</u>	<u>GM2</u>	<u>GM1</u>	<u>GMO</u>	<u>ALPHA/GRAPHIC MODE SELECTION</u>
0	0	0	0	X	X	X	Internal Alphanumerics
0	0	0	1	X	X	X	Internal Alphanumerics Inverted
0	0	1	0	X	X	X	External Alphanumerics (not used)
0	0	1	1	X	X	X	External Alphanumerics Inverted (not used)
0	1	0	X	X	X	X	Semigraphics 4
0	1	1	X	X	X	X	Semigraphics 6 (not used)
1	X	X	X	0	0	0	64 x 64 Color Graphics (not used)
1	X	X	X	0	0	1	128 x 64 Graphics (not used)
1	X	X	X	0	1	0	128 x 64 Color Graphics(not used)
1	X	X	X	0	1	1	128 x 96 Graphics (not used)
1	X	X	X	1	0	0	128 x 96 Color Graphics(not used)
1	X	X	X	1	0	1	128 x 192 Graphics (not used)
1	X	X	X	1	1	0	128 x 192 Color Graphics
1	X	X	X	1	1	1	256 x 192 Graphics

X = DON't CARE

Basically, the MPU places 8 bit codes into the screen memory area. As the VDG outputs the composite video signal, it fetches from memory the appropriate 8 bit code for where it is up to on outputting to the screen (like an X,Y coordinate) and interprets the code based upon the selected VDG mode.

The details of each mode as implemented in the MP1000 are as follows:

Major Mode 1

Alphanumeric/Semigraphics Mode

These modes always occupy an 8 x 12 dot character matrix box, and there are 32 x 16 character boxes per TV frame. This mode

is entered by changing \bar{A}/G to alphamode. Then all submodes are selectable on a character by character basis.

Alphanumeric - internal ROM of VDG generates one of 64 ASCII displays characters in a 5 x 7 box. One of two colors can be selected with an inverse mode.

Semigraphics - 8 x 12 dot box is broken into 4 small boxes, each of which are 4 dots wide by 6 dots high. The 8 x 12 box is given a color and each of the 4 small boxes can be on (luminance) or off (no luminance) with that color.



Alphanumeric Box - 5 x 7 ASCII character in an 8 x 12 box.



Semigraphics Box - 4 small boxes (B0-B3), each 4 dots by 6 dots.

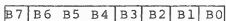
Interpretation of 8 Bit Screen Map Word in Alphanumeric/
Semigraphics Mode

Alphanumeric



0 Inverse 6 Bit
For and Code for
Alpha Color ASCII Character
Mode Select

Semigraphic



1 Color Luminance on (1)
For For or off (0) for
Semi Large each Small Box
Mode Box

Color Codes for Semigraphics

B6	B5	B4	Color
0	0	0	Dark Green
0	0	1	Yellow
0	1	0	Blue
0	1	1	Red
1	0	0	White
1	0	1	Cyan
1	1	0	Purple
1	1	1	Orange

Major Mode 2

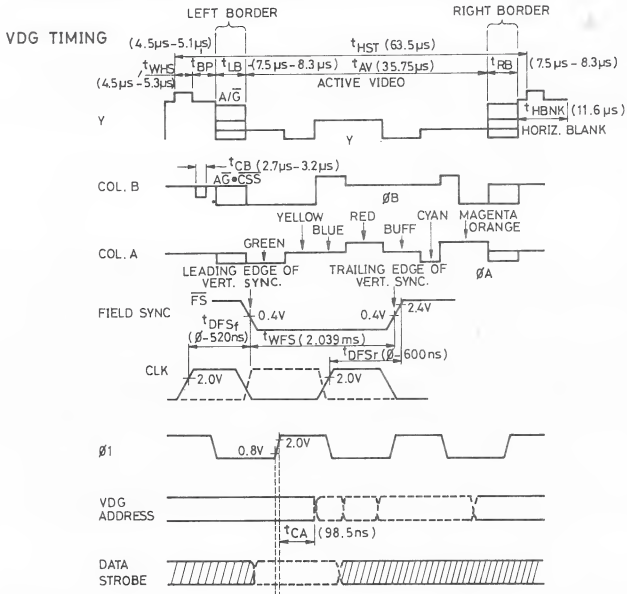
High Resolution Graphics (128 x 192 or 256 x 192)

This mode is implemented similar to the alphanumeric mode except the character shapes are not predefined in ROM like the ASCII characters are. Instead they are defined in RAM by the program. The screen is mapped to have 32 x 12 character boxes. Each box is 8 dots or 4 dots wide by 16 dots high.

The object shapes are defined in one section of memory, and the image map in another.

The system is forced to do 2 fetches from memory before sending data to the VDG for interpretation. The first fetch gets the object number, and the second gets the details of the particular row of the object.

For more details in programming in this mode, see Chapter VIII.



MCl372

The MCl372 color TV video modulator is used to generate an RF TV signal from baseband color-difference and luminance signal supplied by the VDG.

The MCl372 also supplies the system 3.579545 MHz clock. The device contains a chroma subcarrier oscillator, lead and lag networks, suppressed carrier DSB chroma modulator; and RF oscillator and modulator.

In the MP1000 the luminance, chroma and sound carrier signals are inputted to the MCl372. The output is a modulated RF signal whose carrier frequency is set by the LC tank circuit.

ROM

The ROM contains sequences of MC6800 instruction codes and data. It tells the MPU what to do, how to do it, and where to put it when it's done.

The ROM contains basically 2 groups of programs. One is the internal rocket patrol game and the other are known as house-keeping routines. The housekeeping routines are a reset or startup routines, keyboard strobing routines, interrupt servicing routines, screen creation routines, sound generator routines, etc.

The ROM used in the MP1000 is a 2K x 8 ROM but capability for up to 8K x 8 is provided.

PIA

The PIA (MC6821) is a universal peripheral interfacing device. On one side it has bus signals to interface with the MPU. On the other it has two 8 bit ports which can be programmed as inputs or outputs, plus 4 control/interrupt lines.

In the MP1000 it is used as follows

- 1) 4 lines are used to drive the MP1000 controller matrix (keyboard and joystick). The 8 lines of the PA Register are the inputs from these keyboard lines. This is the same way that calculator keyboards are read.

All keyboard decoding and debouncing is performed by the software routines. The MPU sets the 4 PB lines so 1 of them is logic level zero and the other 3 are logic level high. Then it looks at the PA inputs. If any are low, then one of the keys on the driver PB line is closed. It then performs

decoding and debouncing of that key. If not, it changes the 4 PB lines so the next one is low with the other 3 high and, again, looks at the PA inputs.

- 2) PB6 - drives the GMO input of the VDG.
- 3) PB7 - selects alphanumeric or graphics mode.
- 4) CA2 - controls the object latch register.
- 5) CB2 - generates a sound oscillator
- 6) CB1 - inputs field sync from the VDG and passes it to the MPU as an interrupt signal.

Register control of the PIA is given in Table 1-1.

MEMORY AND ADDRESS MULTIPLEXING

The screen memory consists of 1K x 8 bytes. It is comprised of 2 x 2114 (1K x 4) memory chips. Access time on these parts is 200 NS which is very critical.

Only the MPU can write data to memory, but there are 4 possible addressing modes for reading. These are

Mode	Memory	Address	Input	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1				MPU Address Bits A9 - A0									
2			1	VA8	VA7	VA6	VA5	VA4	VA3	VA2	VA1	VA0	
3			0	VA12	VA11	VA10	VA9	VA4	VA3	VA2	VA1	VA0	
4			1	VD4	VD3	VD2	VD1	VDO	VA8	VA7	VA6	VA5	

AN - Address bit from CPU
 VAN - Address bit from VDG
 VDN - Address bit from object latch

Mode 1 - Occurs during phase 2 always, and is the MPU time slot to address memory. The CPU can read or write to memory.

Mode 2 - This is the VDG access during alphanumerics/semigraphics mode. This occurs during a phase 1.

Mode 3 - This is the first access on graphics mode of the screen memory map. The object number is latched halfway from the start of phase 1 so it can be used during the 2nd half of phase 1. Data fetched during this mode does not go into the VDG.

Determine Active CA1 (CB1) Transition for Setting Interrupt Flag IRQA(B)1 – (bit b7).

- b1 = 0 : IRQA(B)1 set by high-to-low transition on CA1 (CB1).
b1 = 1 : IRQA(B)1 set by low-to-high transition on CA1 (CB1).

IRQA(B) 1 Interrupt Flag (bit b7)

Goes high on active transition of CA1 (CB1). Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

CA1 (CB1) Interrupt Request Enable/Disable

- b0 = 0 : Disables IRQA(B) MPU Interrupt by CA1 (CB1) active transition.
b0 = 1 : Enable IRQA(B) MPU Interrupt by CA1 (CB1) active transition.
1. IRQA(B) will occur on next (MPU generated) positive transition of b0 if CA1 (CB1) active transition occurred while interrupt was disabled.

b7	b6	b5	b4	b3	b2	b1	b0
IRQA(B)1 Flag	IRQA(B)2 Flag	CA2(CB2) Control			DDR Access	CA1(CB1) Control	

IRQA(B)2 Interrupt Flag (bit b6)

CA2 (CB2) Established as Input (b5 = 0). Goes high on active transition of CA2 (CB2); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.
CA2 (CB2) Established as Output (b5 = 1): IRQA(B)2 = 0, not affected by CA2 (CB2) transitions.

Determines Whether Data Direction Register Or Output Register is Addressed

- b2 = 0 : Data Direction Register selected
b2 = 1 : Output Register selected.

CA2 (CB2) Established as Output by b5 = 1

b5	b4	b3	(Note that operation of CA2 and CB2 output functions are not identical)
1	0	→ CA2 b3 = 0 : <u>Read Strobe With CA1 Restore</u> CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next active CA1 transition. b3 = 1 : <u>Read Strobe with E Restore</u> CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next high-to-low E transition.	
			→ CB2 b3 = 0 : <u>Write Strobe With CB1 Restore</u> CB2 goes on low on first low to high E transition following an MPU Write into Output Register B; returned high by the next active CB1 transition. b3 = 1 : <u>Write Strobe With E Restore</u> CB2 goes low on first low-to-high E transition following an MPU Write into Output Register B; returned high by the next low-to-high E transition.
b5	b4	b3	
1	1	→ Set/Reset CA2 (CB2) CA2 (CB2) goes low as MPU writes b3 = 0 into Control Register. CA2 (CB2) goes high as MPU writes b3 = 1 into Control Register.	

CA2 (CB2) Established as Input by b5 = 0

b5	b4	b3	
0		→ CA2 (CB2) Interrupt Request Enable/Disable b3 = 0 : Disables IRQA(B) MPU Interrupt by CA2 (CB2) active transition. b3 = 1 : Enables IRQA(B) MPU Interrupt by CA2 (CB2) active transition. 1. IRQA(B) will occur on next (MPU generated) positive transition of b3 if CA2 (CB2) active transition occurred while interrupt was disabled.	
			→ Determines Active CA2 (CB2) Transition for Setting Interrupt Flag IRQA(B)2 – (bit b6) b4 = 0 : IRQA(B)2 set by high-to-low transition on CA2 (CB2). b4 = 1 : IRQA(B)2 set by low-to-high transition on CA2 (CB2).

Table I – 1
PIA Control Register Format

Mode 4 - This is the 2nd access in graphics mode. The row number is determined by VA5-VA8 (1 of 16) and the object # (from the latch) are sent to memory. The resulting data is clocked into the VDG. This mode occurs during the 2nd half of phase 1 during graphics mode.

MODE SELECTION

Depending upon whether the VDG is in graphics or alphanumeric mode, the memory is mapped differently. The two maps are shown in figure 3-2 in Chapter 3.

POWER SUPPLY

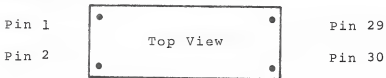
The power supply consists of an external A.C. adaptor and internal circuits.

The A.C. input (approximately 9.6 volts) is rectified, filtered and regulated to provide a D.C. voltage of 5 volts +/- 5%. This is used to supply all of the semiconductors in the MP1000. Maximum current capability is approximately 1 amp.

CARTRIDGE SOCKET

The cartridge socket provides all of the signals to the outside world.

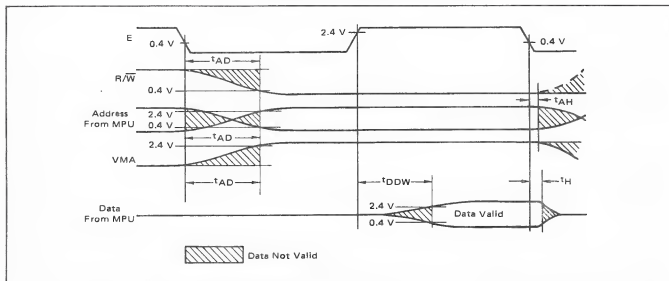
These pinouts are as follows



Pin	Signal	Pin	Signal
1	D0	16	A7
2	A0	17	D3
3	D1	18	A15
4	A1	19	Read/Write
5	D2	20	A8
6	A2	21	Phase 2
7	Ground	22	A14
8	A3	23	EN 89
9	D7	24	A9
10	A4	25	VMA
11	D6	26	A13
12	A5	27	+5 Volts
13	D5	28	A10
14	A6	29	A11
15	D4	30	A12

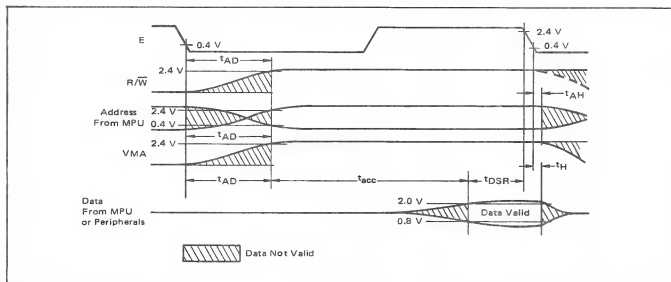
SYSTEM TIMING

WRITE DATA IN MEMORY OR PERIPHERALS



- $t_{AD} = 270$ ns maximum
- $t_{acc} = 530$ ns maximum
- $t_{DSR} = 100$ ns maximum
- $t_{AH} = 20$ ns maximum
- $t_H = 20$ ns maximum
- $t_{DDW} = 165$ ns maximum

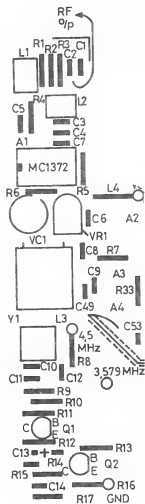
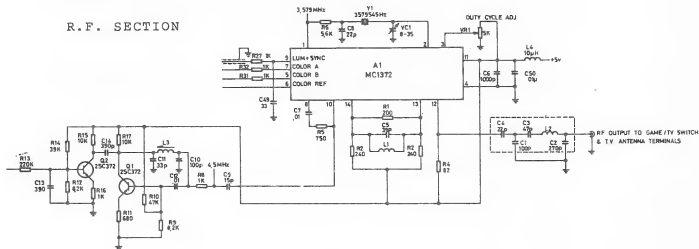
READ DATA FROM MEMORY OR PERIPHERALS



ADJUSTMENTS AND TUNING

Inside the MP1000 there are several adjustments that can be made

R.F. SECTION



R.F. OUTPUT

- A. 1. The R.F. Output Frequency is set by L1 and C5. L1 is an air core coil and will allow approximately a ± 6 MHz adjustment from the factory set frequency of 61.25 MHz (with C5 = 39 PF). By changing C5 to 27 PF, the resonant frequency becomes approximately 67 MHz (U.S. Channel 4).
2. The R.F. Output goes through a sideband filter whose bandwidth is approximately 6 MHz. It can be adjusted by tuning L2 to give a peak output of the upper sideband.
- B. Tuning Procedure for R.F. Output (Channel 3)
1. Connect R.F. output cable of MP1000 to spectrum analyzer or calibrated field strength meter.
 2. Switch MP1000 power on.
 3. Adjust coil L1 to read 61.25 MHz (TV Channel 3).
 4. Adjust coil L2 to get maximum output signal strength then adjust the tuning point of L2 to 1 MHz higher than TV Channel 3 signal (ie, 62.25 MHz) or tuning of L2 to reduce maximum output signal level by about 1 db. This also gives a tuning point close to 62.25 MHz.

CRYSTAL OSCILLATOR

- A. VC1 is a variable capacitor to allow adjustment of the xtal oscillator frequency.
- This clock should be adjusted to give exactly 3.579545 MHz.
- B. VR1 allows an adjustment of the duty cycle. It should be adjusted to give a duty cycle between 45 and 55%.

SOUND OSCILLATOR

The sound subcarrier frequency can be adjusted by tuning L3. It should be adjusted to give an unmodulated frequency of 4.500 MHz.

The MPA-10 base unit adds the following to the MP1000

1. An interconnection (the J Connector)
2. A main unit with 8K of RAM and keyboard
3. A power supply for the MPA-10 circuits and expansion
4. A tape deck for both audio and digital recording/playback
5. A cartridge with a Basic interpreter
6. Provision for expansion

J CONNECTOR

The MP1000 and MPA-10 are electrically joined by the J Connector (this is so named because its physical configuration looks like the letter J).

Besides connecting the signals between the 2 units, it buffers the address and data lines.

MAIN UNIT

The main unit of the MPA-10 has the following subsections

1. A memory section
2. A PIA
3. A decoding section

The memory is comprised of 8 - 8K x 1 dynamic memory chips.

The MC3242 chips multiplexes the 13 addresses to the memory as 7 row addresses, and 7 column addresses. The control of these is determined by ROWEN. The MC3242 also performs the task of memory refresh during Phase 1 when the MPU is not addressing memory. The rest of the memory section is comprised of timing and control implemented using a TTL delay line to develop RAS (row address strobe), CAS (column address strobe) and ROWEN.

Memory timing is in Figure 2-2.

The PIA is used to strobe the keyboard lines (similar to the MP1000 strobing) and to control the tape system. A 3-bit code is put into PB0-PB2, and decoded by a 74LS145 (1 of 8 decoders). These become the strobe lines and are looked at by the PA0-PA7 inputs of the PIA. As in the MP1000, all decoding and debouncing of the keyboard is performed by the software. To deal with tape are the following signals:

PB3 - AUDEN - Enables or disables audio section of tape deck
PB4 - MOTEN - Enables or disables tape deck motor. This can be
 overridden by the fast forward or rewind buttons
PB5 - WREN - Indicate to tape whether to read or write to digital
 track.
PB6 - WRDATA - Digital data to tape deck from the MPU
PB7 - READDATA - Digital data from tape deck to the MPU

Decoding Section - The rest of the MPA-10 base unit has address decoding and expansion bus signal generation.

Keyboard - The keyboard consists of 53 keys. It is set up as a 7 x 8 matrix. All reading of keyboard, decoding and debouncing is performed in software.

Tape Deck/Power Supply

The MPA-10 has its own power supply. The supply receives an A.C. input from the A.C. adaptor. 4 D.C. voltages are developed

+ 5 volts $\pm 5\%$
+ 12 volts $\pm 5\%$
- 12 volts
- 5 volts

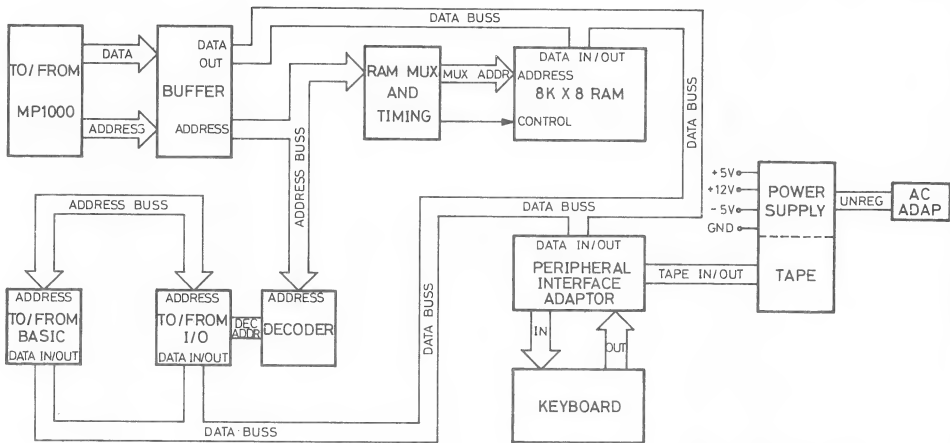
These supplies feed all the circuits of the MPA-10 plus go to the expansion bus.

The tape circuits are comprised of 2 parts

Audio Section - This is for monaural record or playback of audio signals. The 2 changes from standard designs are:

- a) A half track erase head that only erases the audio portion of the tape.
- b) An enable or disable to the audio section (from the MPA-10 PIA). This enables/disables both recording of audio or playback.

Digital Section - Saturated recording is used to write digital data. Sufficient current is driven through the record head to fully polarize the tape in one direction or the other. All encoding of digital ones and zeros is performed in software. The digital recovery circuits take the magnetic field from the tape and recover them into logic levels which then go back to the PIA. All decoding of digital data is performed by the software.



APF ELECTRONICS INC.
BLOCK DIAGRAM - MPA10

Figure 2-1
MPA-10 Block Diagram

THE ROM CARTRIDGE

This consists of a total of 12 kilo bytes of memory (comprised of an 8K x 8 and 4K x 8 ROM chips). It contains the Basic interpreter as well as certain I/O driver routines. It plugs into the ROM cartridge socket of the MPA-10.

THE EXPANSION BUS

Provision is made to expand the system further. There is a 50 Pin bus that comes out through the expansion ports. Its pinouts are shown in Figure D-3.

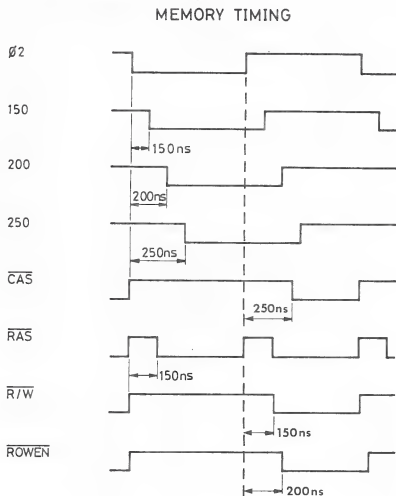


Figure 2-2

CHAPTER III

THE IMAGINATION MACHINE MEMORY MAP

Address		
Hex	Decimal	Description/Useage
0000-03FF	0-1023	MP1000 internal memory. 1K of memory is used. See Figure 3-2 for details of useage.
0400-1FFF	1024-8191	Each 1K block is same as 0000-03FF.
2000-2003	8192-8195	Peripheral interface adaptor (Motorola MC6821) used in MP1000. See Figure 3-3 for details.
2004-3FFF	8196-16383	Each 4 consecutive address same as 2000-2003.
4000-5FFF	16384-24575	Internal ROM of MP1000
6000-6003	24576-24579	Peripheral interface adaptor used in MPA-10 section.
6004-63FF	24580-25599	Each 4 consecutive address same as 6000-6003.
6400-67FF	25600-26623	For external I/O devices.
6800-77FF	26624-30719	Basic interpreter ROM cartridge (4K).
7800-7FFF	30720-32767	For ROM expansion
8000-9FFF	32768-40959	Basic interpreter ROM cartridge (8K).
A000-BFFF	40960-49151	Read/Write memory. See Chapter IV for details.
C000-DFFF	49152-57343	Expansion read/write memory
E000-FFFF	57344-65519	Not used.
FFF0-FFFF	65520-65535	MC6800 reset/interrupt vectors - ROM.

Figure 3-1

1K Internal MP1000 Usage

\$ = Hexadecimal; Otherwise Decimal

<p>\$0000 0000</p> <p style="text-align: center;">384 Bytes</p> <p style="text-align: right;">0383 \$017F</p>	<p>Graphics Mode - used for screen map.</p> <p>Alpha/Semi Mode - not used.</p> <p>Not used by Basic interpreter</p>
<p>\$180 384</p> <p style="text-align: center;">128 Bytes</p> <p style="text-align: right;">511 \$01FF</p>	<p>Used only by ROM cartridge games as scratch pad and stack area.</p> <p>Not used by Basic interpreter.</p>
<p>\$0200 512</p> <p style="text-align: center;">512 Bytes</p> <p style="text-align: right;">1023 \$03FF</p>	<p>Alpha/Semi Mode - used for screen image</p> <p>Graphics Mode - used for object shape definitions.</p>

Figure 3-2

MP1000 Peripheral Interface Adaptor Addressing (Addresses in Hex)

<p>\$2000 - Data Register A</p>
<p>\$2001 - Control Register A</p>
<p>\$2002 - Data Register B</p>
<p>\$2003 - Control Register B</p>

Figure 3-3

CHAPTER IV

HOW IS MEMORY USED AND ALLOCATED

The 8192 bytes of memory from A000-BFFF (decimal 40960-49151) is used for all storage except screen maps. It is allocated as follows:

<u>Decimal</u>	<u>Hex</u>	
40960	A000	SYSTEM VARIABLES, LABEL TABLES, SIMPLE VARIABLES
	↓	
41727	A2FF	
41728	A300	I/O BUFFER
	↓	
41983	A3FF	
41984	A400	PROGRAM TEXT
	↓	
		COMPLEX VARIABLE STORAGE
		↓
		FREE MEMORY
		↑
49151	BFFF	STACKS

PROGRAM STORAGE

Program steps are stored in the following format:

First 2 bytes are for the line # (in packed BCD Code).

Then ASCII and token code for statement with all spaces removed except those in quotes, print using definitions or remarks.

Note all keywords are stored as a 1 byte token code. See Appendix D for the complete list.

Finally, carriage return symbol (Hex OD).

A400 and A401 are used to point to the next locations to store a statement (they start off upon initialization set with Hex A402). Actual program storage starts at A402.

As an example of program storage, let's enter the following program:

```
10 PRINT 123
```

To look at memory, do a CALL 28672. This enters the machine language monitor mode. You now should get an * instead of the cursor.

Using the M Command (Examine/Change Memory) type:

* M A400 A4 — You type these
 — And see the contents of A400

Next, press / (Don't press Return after the A4 is shown.).
This command will show the next memory address and its
contents.

A401 09

The contents of A400 and A401 are A409. This is the next free
location. If we went back to Basic and entered another state-
ment, its storage would start at A409. Before returning to
Basic, let's look at the next 8 memory locations. Just keep
pressing the Slash Key. You will see the following:

<u>Hex</u> <u>Address</u>	<u>Hex</u> <u>Data</u>	
A402	00/	Line Number - This is packed BCD Code --
A403	10/	Line Number is 0010.
A404	91/	Token for print Command
A405	31/	
A406	32/	ASCII Codes for 12?
A407	33/	
A408	0D/	Carriage return - end of statement line delimiter
A409	00	

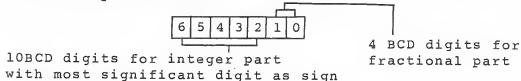
VARIABLES

The variable list or label table (for 26 variable names) is
stored starting at hexadecimal A0C3. Each label or name takes
9 bytes as follows:

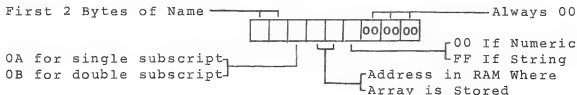
First 2 bytes are for variable name. Variable names can be
longer, but only the first 2 characters are stored.

Then 7 bytes:

If non subscript numeric variable (such as I), they are 7
byte value in packed BCD.



If the variable is of the subscripted type (dimensioned), then the 9 bytes are as follows:



Let's try an example. Clear the machine and enter the following:

```
10 I = 14
20 DIM A(6), B$(7,4)
30 J = 12345678.99
40 DIM C(1,1)
```

If we go examine the label table now, we will find it empty. The label table will only contain entries when statements using variables are executed, not when they are keyed in.

Type RUN

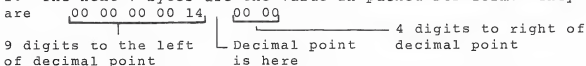
Now let's go to the examine memory mode
CALL 28672

As opposed to examining single memory bytes at a time, we will use the D Command to display 16 bytes at a time. We get

```
*DA0C3 49 00 00 00 00 00 14 00 00
41 00 DA A4 36 00 00
```

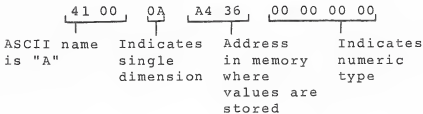
Examining the first 9 bytes only

The first 2 are 49 00. The 49 is ASCII for the Letter I and 00 is null. So, the first entry in the table is the variable I. The next 7 bytes are the value in packed BCD form. They are

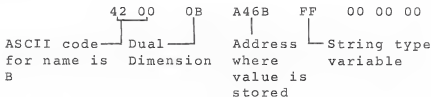


Let's continue with the next entry in the table. Its entry starts at A0CC (A0C3+9).

Using the D Command again (DA0CC), let's look at the 9 bytes starting at A0CC



Continuing in the table, the next entry is at A0D5. The 9 bytes will be



If you continue on your own, you will find the entries for J and then C.

MEMORY AMOUNT USED BY VARIABLES

The pointers for where to store dimension variables are at 41009, 41010 (Hex A031, A032). As dimension statements are executed, the pointers direct the interpreter where to allocate space and are updated with each allocation.

A RUN command initializes 41009, 41010 to have the same numbers as 41984 and 41985 respectively.

For all dimensioned variables there are always 4 bytes preceding the actual stored values. These bytes contain the actual dimensions of the array and are called the overhead.

- Simple (numeric non subscript) - 9 bytes in name table only
- String (single dimension) - 1 byte per character dimensioned plus 4 bytes for overhead.
- Numeric Array-single subscript - 7 bytes per dimension plus 4 bytes for overhead
- Numeric Array-two subscripts - 7 bytes each element plus 4 bytes overhead. Ex: DIM(5,4) is 6 x 5 = 30 elements x 7 bytes = 210 plus 4 overhead = 214 bytes
- String-two subscripts - 1 byte per character dimensioned plus 4 bytes overhead.

CHAPTER V

ENTERING AND USING MACHINE LANGUAGE PROGRAMS

Machine Language Programs (those written in MC6800 code) can be entered and used as part of a Basic program. They also can be saved on tape with a Basic program.

Machine Language Programs are useful where speed is essential (and a program written in Basic is too slow) or to implement routines that are not available in the Basic language.

ASSEMBLING AND ENTERING MACHINE LANGUAGE PROGRAMS

MC6800 programs for machines with a cassette only have to be hand assembled. Units with disks can use the APF Assembler/Editor.

Appendix A gives a table of MC6800 machine code. For more details refer to a MC6800 programming reference manual. Once a program is assembled, it can be entered into the machine in hexadecimal format by using the machine Language Reference Mode. This is entered by a Call Statement (See Appendix B for details).

METHOD 1 - THIS METHOD IS FOR AN ENTIRELY WRITTEN MACHINE LANGUAGE PROGRAM.

Statement #10 will always be CALL 42200 (42200 is Hex A4D8). Then the machine program is entered starting at A4D8.

A problem that must be overcome is that a RUN command will clear to "00" all memory locations from the end of program storage to the end of memory. If we just type in statement 10 as above, then key in our machine language program, a RUN command will wipe out the program since the Basic Interpreter only knows about line 10 in the program and clears everything after it to 0. The solution is very simple. After entering the machine language codes, we change the end of program pointer so it points past our machine language program. The end of program pointer is 2 bytes contained at A400 and A401. They must be set to BF and F0 respectively before giving a RUN command. They also are saved to tape so once changed they will stay changed.

As an example - to write a program to fill the screen with all blue. This, of course, could be done with HLINE, but Machine Code will be faster and serves as a simple example.

First reset the system and key in

```
10 CALL 42200
15 STOP
```

This will be the entire Basic program and a Run Command will execute it. (Don't type RUN yet since we haven't entered the program at 42200.)

Now enter the machine language mode (CALL 28672). The machine program will start at Hex A4D8 (decimal 42200). The program is:

<u>ADDRESS</u>	<u>CONTENTS</u>	<u>INSTRUCTION</u>	<u>COMMENT</u>
A4D8	CE	LDX #\$200	Load X reg with starting screen address
A4D9	02		
A4DA	00		
A4DB	86	LDAA #\$DF	Load A reg with code for blue square
A4DC	DF		
A4DD	A7	STAA 0,X	Store A indexed
A4DE	00		
A4DF	08	INX	Increment X register
A4E0	8C	CPX #\$400	Is X equal to end screen address yet
A4E1	04		
A4E2	00		
A4E3	26	BNE -8	If not, do next address
A4E4	F8		
A4E5	7E	JMP #\$8894	Jump back to Basic
A4E6	88		
A4E7	94		

Enter the machine language program by using the M Command. See Appendix B if you are unclear on how to use this mode.

Before returning to Basic, change A400/A401 to BF and F0 respectively so a RUN command doesn't wipe out your program.

MA400 BF/
A401 F0 Return Key

Next return to Basic with G8894.

Now run the program and then save it to tape if you want.

Note: If you try to list the program now, you will get garbage on the screen. The List command will go from beginning of program memory to end of program memory (and we changed the end pointer) and interpret everything as an ASCII Code or Token.

METHOD 2 - YOU CAN USE A MACHINE LANGUAGE ROUTINE AS PART OF A BASIC PROGRAM. YOU CAN ENTER ONE OR MORE MACHINE LANGUAGE ROUTINES, ACCESS THEM FROM A BASIC PROGRAM AND PASS VARIABLES BETWEEN THEM.

The way this is done is to enter "dummy" Remark Statements in the program. In the comment field of the Remark Statement type in enough characters to allow space for the machine program (it is suggested you use a single letter, repeated in the Remark Statement so it is easy to find exactly where in memory it is stored).

As an example, let's do a program to set the screen to have 1 single character code fill it up. The code we put to the screen will be found in memory location 0000. (We can POKE location 0 with a code.) The steps are:

1. Write the machine language routine first so we can get a count of the number of bytes it will use up. Don't try to figure the actual addresses in memory where it will go since we have to put the "Basic" program in first. The machine language program will be:

CE 0200	LDX#\$0200	Start of screen adding
96 00	LDAA 0	Character to be put to screen is stored at Location 0.
A7 00	STAA 0,X	Store A indexed
08	INX	Increment index register
8C 0400	CPX #\$400	Is X equal to end screen address yet
26 F8	BNE -8	If not, do next address
39	RTS	Return from subroutine

Adding up the number of bytes for the above routine, we get 14.

2. Next we enter the Basic program

```

10  REM      AAAAAAAAAAAAAA
20  POKE    0,223
30  CALL    11111

```

Line 10 is the Dummy Remark Statement. It is in the space occupied by Line 10 in memory that we will put in the machine language program. We put in the Remark Statement 14 letter A's (you could put in 14 of anything). Line 20 pokes to Location 0,223. Location 0 is used by our machine program to contain the value to be stored to the screen (223 is Hex DF, which is a blue square). Line 30 is the Call to machine language routine. Right now we have called 11111. We don't know yet exactly where in memory the routine will be located, so again we use a temporary number (it will be a five-digit address).

3. We are now ready to enter the machine language routine. Type CALL 28672 then press Return Key. We enter the monitor mode now to find where the Remark Statement of Line 10 is stored.

Do a D A400 and you see

```

* A400 A4 27 00 10 94 20 20 41 41
  41 41 41 41 41 41 41 41

```

These numbers are the 16 bytes of memory starting at A400. The first 2 (A427) are the end of program storage pointers.

The next 2 (0010) are the first line number in packed BCD Code.

The 94 20 20 is the token code for a remark command (A remark uses 3 bytes for token storage).

Next we see a series of 41. This is the ASCII Code for the letter A, and it is here we want to put our machine routine. Since the first 41 is the 8th byte in the display, its address is A407. Going back to our machine language routine, we can now fill in address

ADDRESS	CODE
A407	CE 0200
A40A	96 00
A40C	A7 00
A40E	08
A40F	8C 0400
A412	26 F8
A414	39

Press Return and type

M A407 You see the data is 41. Change it to CE and hit
/. You will see the next line.
A408 41 Change this to 02 and hit /.

Continue with this till you enter the 14 bytes of program.

After keying in the program, it is a good idea to check it.

Type DA407 CE 02 00 96 00 A7 00 08 8C
04 00 26 F8 39 0D 00

If the memory matches the above, press Return. If not,
go to the incorrect address (use the M Command) and correct
it.

Let's return to Basic.

G8894

4. We are almost ready. We just have to change the Call
Address in line 30. Our program starts at A407 which is
41991 in decimal.

Change Line 30 to be

30 CALL 41991

5. If you followed everything OK, then let's give a Run
Command.

RUN Return

The screen should have turned blue almost instantly, and
the cursor is back. (The cursor is blue, so press Return
a couple of times to clear the screen.)

6. You have now successfully done a machine language routine.
It can be saved on tape with the rest of the program (just
give a CSAVE).

You can add to the program. Just don't add anything in front
of Line 10. That would shift Line 10's code in memory, and
the CALL address in Line 30 would become wrong.

Some Guidelines on Writing Machine Language Routines

1. When writing a program in both Basic and CALLS to Machine
routines, the most important thing is to PLAN IT OUT VERY
CAREFULLY. It can be very difficult to make changes later

on. Once a Remark Statement is put in and then replaced by a machine routine, do not put in any "Basic" Statements with lower line numbers than the Remark Statement. An insert of a "Basic" Statement will shift all memory contents upward, and you will have to change your CALL Statement.

2. Leave extra places in a Remark Statement (at least 3). If you later find you have to add something in machine code, you can do a jump to subroutine if you leave room.
3. The end of your routines should be an RTS (Return from Subroutine) and not a jump to 8894.
4. If you want to access dimensioned variables in machine language, FORCE them to be located where you want them. By Poking 41009 and 41010 prior to a Dimension Statement, you can force where a variable is located.
5. The next 3 chapters have lists of several machine language routines that might be useful.

CHAPTER VI

SOME USEFUL ROUTINES

You might find that there are some routines or functions not built into the Basic interpreter that you need. Most can be implemented using either PEEKS/POKES/CALLS in machine language or with subroutines written in Basic.

"PRINT AT"

If you want to print anywhere on the screen, use a routine that changes the cursor pointer and do a GOSUB to it before doing a Print Statement. The screen resides at locations 512-1023. Remember, you can print anywhere in memory, but only 512-1023 appears on the screen.

The cursor is stored as 2 bytes in memory locations 40960 and 40961 (Hex A000/A001).

```

10 GOTO 100
20 REM: ROUTINE TO MOVE CURSOR POSITION TO VALUE OF CU.
25 POKE 40960, INT (CU/256): POKE 40961, CU: RETURN
100 FOR I = 1 TO 32: PRINT: NEXT: REM CLEAR SCREEN TO GREEN
110 INPUT "LINE AND COLUMN TO START PRINT", L, C
120 CU = 512 + L * 32 + C: GOSUB 25: PRINT "HI"
125 INPUT "MORE", K: GOTO 100

```

Line 100 - will clear the screen to all green.

Line 110 - asks for a horizontal line number (L), and a vertical column number (C) where you want to start printing at. It converts these to the actual memory location on the screen. 512 is the top corner of the screen so we add to it the number of lines (L) times 32 (32 characters per line), and add the column.

Line 25 - We enter line 25 with the variable CU having the memory location we want to change the cursor pointers to have. Since the cursor pointer is a double byte location (it takes 2 bytes to point to a memory location between 0 and 65536), we have to break CU into 2 numbers. The most significant number (into 40960) is the number of 256's contained in CU. We get this by taking the integer portion of CU/256. Into 40961 we have to POKE the remainder of dividing 256 into CU. The POKE instruction automatically does this. So, into 40961 effectively goes $CU - INT(CU/256) * 256$.

Run the program and enter various numbers for L (between 0 and 15), and C (between 0 and 31). After their entry you will see the word "HI" printed on the screen. The program next says MORE?, and just press Return to run again.

HOW MUCH MEMORY IS LEFT FOR PROGRAMS AND VARIABLES

Program storage starts at 41986. Dimensioned variable storage usually occurs after the last statement.

41984, 41985 point to next location to store a step
41009, 41010 point to next location to store dimensioned variables

Using the Above:

Amount of space used for program and variable storage is
AMT = (PEEK (41009) *256 + PEEK (41010)) - 41986

Amount of program space only is
AMT = PEEK (41984) x 256 + PEEK (41985) - 41986

Amount of free space left is
AMT = 49151 - (PEEK (41009) *256 + PEEK (41010))

↑
End of memory for an 8K system

USING KEY \$(0) FUNCTION

The purpose of KEY\$(0) is to get an input from the keyboard without waiting for a Return Key (which is needed in an input statement). KEY\$ does not debounce a key, nor does it put the depressed Key's code to screen. Below is a program utilizing KEY\$(0) and does the following:

1. Will wait in a loop for a key to be pressed.
2. Will put the character code to the screen. If it is a Return, the program will stop.

```
10 DIM A$(1)
20 GOTO 100
30 A$ = KEY$(0): IF A$ = "" THEN 30
40 A = ASC(A$): RETURN

100 GOSUB 30: IF A = 13 THEN STOP
110 PRINT A$
120 IF KEY$(0) "" THEN 120
130 GOTO 100
```

Line 10 - dimensions a string variable A.

Line 30-40 - The function KEY\$(0) is called and its value is assigned to A\$. If A\$ is NULL (empty), then no key is pressed, and we remain at Line 30.

When a key is pressed, the program moves from Line 30 to 40 where the ASCII value of the key pressed is assigned to variable A. Then the subroutine returns.

Line 100 - goes to subroutine 30 to get a key input. If A (the ASCII value of the key pressed) is 13, then it was a Return Key, and we stop the program.

Line 110 - prints the value of A\$ (the key pressed). Note the use of the semicolon so we will print on the same line each time a key is pressed.

Line 120 - checks that the key has been released. Without Line 120 you will find it very difficult to press a key and get only one entry (try the program without Line 120).

Line 130 - goes to Line 100 and repeats the process.

TRIG FUNCTIONS

Although Trig Functions are not part of APF Basic, they can be easily implemented as subroutines utilizing series approximations.

The series approximations are

$$\sin(S) = \frac{S}{1} - \frac{S^3}{3!} + \frac{S^5}{5!} - \frac{S^7}{7!}$$

$$\cos(S) = 1 - \frac{S^2}{2!} + \frac{S^4}{4!} - \frac{S^6}{6!} + \frac{S^8}{8!}$$

$$\tan(S) = S + \frac{S^3}{3} + \frac{2*S^5}{15} + \frac{17*S^7}{315}$$

The angle S is in radians and less than or equal to $\pi/2$ (90°).

A simplified implementation is as follows:

```

10 GOTO 100
20 Y = S - S*S*S/6 + S*S*S*S*S/120 - S*S*S*S*S*S*S/5040:
   RETURN: REM

100 INPUT "ANGLE ", S1
110 IF S1 <= 90 THEN IF S1 >= 0 THEN 130
120 PRINT "ILLEGAL ANGLE": GOTO 100
130 S = 3.1428 * S1/180
140 GOSUB 20
150 PRINT "ANGLE"; S1, "SIN"; Y: GOTO 100

```

Line 20 - does the SIN calculation of the Angle S (in radians). For a speedier calculation, the values of 31, 51 and 1 have been put in instead of calculating them each time.

Line 100 - inputs the angle (in degrees).

Line 110 - checks that angle is in range of 0 to 90°. This line could be replaced by a calculation that converts the angle to the first quadrant (0 - 90°).

Line 130 - converts the Angle S1 which is in degrees to an Angle S in radians.

Line 140 - goes to subroutine at Line 20 and returns with Y as the SIN(S).

MOVING THE DIMENSION POINTER

The pointer that directs Basic where to allocate space for dimensioned variables is contained in 2 bytes at Locations 41009 and 41010. By using POKES, these pointers can be changed.

One use of this is there are 512 bytes of memory (Locations 0-511) that are not normally used by Basic. You can force dimensioned variables to be stored here and gain 512 bytes of memory space. Example:

```
5 GOSUB 50
10 POKE 41009, 0: POKE 41010, 0.
20 DIM A (10), B$(99).
25 GOSUB 50
30 STOP
50 PRINT PEEK(41984), PEEK(41009), PEEK(41985), PEEK(41010):
RETURN
```

When a RUN Command is given, the dimension pointers are set equal to the end of program storage pointers.

```
(41009) = (41984)
(41010) = (41985)
```

Line 5 - goes to subroutine at 50 and prints these values.

Line 10 - changes the dimension pointers, and Line 20 uses these new values for the Dimension Statements.

Line 25 - goes to subroutine 50 again. You can see the dimension pointers have allocated space for A, and B\$.

STRINGS

There are several features of APF's Strings that might differ from some other Basics.

1. All Strings must be dimensioned before usage. Otherwise the error message "ILLEGAL VARIABLE" will occur.

```
20 INPUT A$
```

Running the above will produce an error message. You must add Line 10 as follows

```
10 DIM A$(X)
```

Where X is the number of characters plus 1 that you want to allocate space for A\$. X must be a number (not a variable) and be less than 100.

2. You can dimension an array of Strings

```
10 DIM B$(3,10) - This dimension is 4 Strings, each with  
eleven characters.
```

3. You can designate the starting position of a String variable in a statement.

Ex:

```
10 DIM A$(10): REM Dimensions A$ as 11 characters.
```

```
20 INPUT A$(4): REM - This means to get an input from the  
keyboard and place it in A$ starting at  
the 5th character position.
```

```
30 INPUT A$(0): REM start inputting to the first character  
position
```

```
40 INPUT A$: REM - This is a default condition to start  
at the first character position
```

4. A String always has some value assigned to each of its character positions. The values are what is contained in memory where the String is dimensioned at. Usually the Dimension Statement places the storage area in memory that has been cleared to 0. A zero is a null character and is non-printable. When assignments of values are made into the String, they remain until another assignment is made. This means that Strings are not cleared to null automatically.

Ex:

```
10 DIM A$(5): PRINT A$
20 A$ = "ABCDEF": PRINT A$
30 A$ = "GHI": PRINT A$
```

Line 10 Dimensions A\$ and the Print will show nothing (nulls are non-printable characters).

Line 20 sets each character position of A\$ and prints it

Line 30 will change only the first 3 characters of A\$. The Print Statement will print GHIDEF.

If you want to clear out a string variable, then assign it to a clear or null string.

Add Line 15

```
15 DIM NULL$(5): REM null is dimensioned and contains nulls.
                    If we never set it equal to anything, it
                    will remain with nulls.
```

Add Line 25

```
25 A$ = NULL$: REM - This causes A$ to be cleared to nulls.
```

5. STRING CONCATENATION AND DISSECTION

String concatenation can be implemented by using the LEN Function

```
10 DIM A$(5), B$(5), C$(11), NULL$(11)
20 A$ = NULL$: B$ = NULL$: C$ = NULL$
30 INPUT A$, B$
40 C$ = A$: C$(LEN(A$)-1) = B$
50 PRINT A$, B$, C$: GOTO 20
```

Dissection (right part, left part, etc) can be done in a similar manner.

MACHINE LANGUAGE ROUTINES

There are a number of built-in subroutines in the Basic ROMS which can be accessed. Most of these cannot be called directly from Basic since they either require the A, B, or X register of the MC6800 to be set up with a certain value, or return with a result in the A, B or X register. Therefore, routines

have to be written to deal with these registers and make their inputs and outputs accessible to Basic. For the purposes of this manual we will keep these accesses very general purposes (to enter the machine language routine, CALL 28672).

Note: All addresses are in hexadecimal.

1. MOVE MEMORY BLOCK

USE: to move a block of data from one section of memory to another.

LIMITS: block of data is less than or equal to 256 bytes

SETUP: A Register - none
B Register - number of bytes to be moved
X Register - none
Memory A029/A02A - first address to move to
Memory A02B/A02C - first address to move from

CALL: JSR 7700 (Hex)

RETURNS: none

EXAMPLE: will move block of 10 bytes stored starting at Location Hex 50 to screen at Location Hex 0300.

First enter the data - we will use the ASCII Codes for the first 10 letters of the alphabet.

```
*0050 41/ (REM ASCII for 'A')
0051 42/
0053 43/
etc.
```

Now the program - We will locate the program starting at Address 0000.

```
0000 C6 LDAB#0A 10 bytes to be moved
0001 0A
0002 BD JSR 7700 go to move routine
0003 77
0004 00
0005 7E JMP 7000 Jump back to monitor mode
0006 70
0007 00
```

Next setup A029 - A02C

```
A029 03 To Address
A02A 00
A02B 00 From Address
A02C 50
```

Run the program by typing

G0000

Instantly you will see the first 10 alphabet characters appear on the screen in reverse video, and the program jumps back to the monitor mode.

2. CLEAR SCREEN TO BLACK

USE: to clear screen to all black
LIMITS: none
SETUP: none
CALL: JSR 4296
RETURNS: none

3. SET SCREEN TO HAVE ALL ONE CODE

USE: similar to 2, but instead of Hex 80 (black character), can fill screen with any character code.

LIMITS - none

SETUP: A Register - code to be put to screen.

CALL: 4298

RETURNS: none

Ex:

```
*M0000 86 LDAA #8F
0001 8F
0002 BD JSR $4298
0003 42
0004 98
0005 7E JMP to monitor
0006 70
0007 00
```

4. INPUT FROM CONTROLLERS

USE: to check if a key is pressed on either hand controller

LIMITS: none

SETUP: none

CALL: JSRS41BE - left hand controller
JSRS41D9 - right hand controller

RETURNS: carry flag of status register
If clear, no key pressed.

If set, key pressed and ASCII code for key is contained in memory Hex 01F2

5. INPUT FROM KEYBOARD

USE: to get key input from main keyboard

LIMITS: will not return a shifted keyword (CTRL Key and top 2 rows).

SETUP: none

CALL: JSR \$80CF

RETURNS: ASCII Code for key pressed in A register
If (A) = 0, then no key pressed.

6. ADD TO INDEX REGISTER

USE: allows a number to be added to the index register

LIMITS: number to be added is 256 or less

SETUP X Reg - setup
A Reg - value to be added

CALL: JSR \$771B

RETURNS: A Register added to X and result in X

7. OUTPUT TO SCREEN

USE: will take a code from the A Register and output it to the screen.

A. If token code, it will decompress it to actual token word (ex \$94 will go to screen as REM).

B. If scrolling necessary, will scroll screen.

C. If backspace code, will do a backspace.

D. If carriage return, does a return and scroll if necessary.

SETUP: A Register - code to output
B,x - none
\$A000/\$A001 - screen address

CALL: JSR \$8473

CHAPTER VII
THE TAPE SYSTEM

The Imagination Machine tape system was designed to be simple, reliable, fast, and versatile. This chapter will give some more explanations and some further possible uses of the tape system.

First the Basic commands and what they do

CSAVE - This is the save to tape command and the sequence of events is

1. The motor and audio are enabled. With the audio enabled, you can hear the digital data through the speaker when it is recorded. You can also do audio/recording through the mike jack at this time.
2. The message to REWIND TAPE, PRESS PLAY THEN RETURN is put up. There is a 2 second delay before this occurs to allow the motor to get up to full speed. The computer will now wait till the RETURN KEY is pressed before continuing.
3. There will now be approximately 11 seconds of "Header" put out to the tape. This is to allow during the read sequence of syncing up with the data.
4. After the header, 512 bytes will be put out. Depending upon the mode flag (location 41452), these 512 bytes will come from 0-511 (flag=non 0 number) or from 512-1023 (flag=0).
5. The next block of data put out will be the contents of user RAM in the system. The computer will start with the byte at 41984 (Hex A400), and send out consecutive bytes till it reaches the indicated end of memory. The end of memory pointer is contained in locations 41446-41447. This means that program storage as well as data storage is saved to tape.
6. As data has been sent out, a check sum has been calculated. This check sum byte is next written.
7. The tape motor and audio are disabled, and the computer returns to the keyboard mode.

CLOAD - This is the opposite of CSAVE and will bring data/information back to the computer.

1. The motor and audio are enabled. Any recorded audio can be heard during playback through the built-in speaker.

2. The message to REWIND TAPE, PRESS PLAY THEN RETURN is put up. Again there is a 2 second delay from motor enable till this message occurs. The computer now waits for the return key.
3. The computer will now wait 6 seconds before trying to get in sync. This is to allow the tape leader to fully pass the head as there is a chance of the computer getting a false start from the leader.
4. Next the computer syncs up with the tape data and puts the first 512 bytes to the screen. This gives the picture you see loaded to the screen from APF made tapes.
5. After the screen is filled, the computer will read data and put it in memory starting at 41984. It will keep looking for data until it has filled up all available user memory (indicated by 41446/41447).
6. Finally it looks for a checksum byte. While it has been reading it has recalculated a checksum and then compares the read and calculated checksum. If they match, it prints "OK."
7. The motor and audio are turned off.

CALL ROUTINES

There are several routines that can be called to use the tape instead of using CSAVE and CLOAD.

- CALL 34040 - Motor and audio are enabled with a 2 second delay. No message is put to screen.
- CALL 34061 - Motor and audio are turned off.
- CALL 34138 - Will put out header, 512 bytes from either 0-511 or 512-1023, memory data, and checksum.
- CALL 34225 - Will read from tape. First 512 bytes go to screen, then data goes to memory. The checksum is checked.

SAVING THOSE FRONT PICTURES

All APF prerecorded programs have a front picture that is loaded to screen. There are several ways that this can be done.

- A. A picture can be created in memory 0-511 by a program. By moving the cursor pointer, all print statements can be print

in 0-511. Using POKES, colors and shapes can be added. Once the picture is placed there, you can load a tape with your program (it won't destroy memory in 0-511). Next change 41452 (the flag) to non-0 (such as 255), then do a CSAVE.

As example (remarks are not necessary to type in)

```
10 CU=0: GOSUB 500: REM MOVE CURSOR TO 0
20 FOR I=1 to 16: PRINT SPC(32) ;: NEXT I: REM ALL GREEN SCREEN
30 CU=256: GOSUB 500
40 PRINT SPC(9); "THIS IS A TEST"
50 FOR I=0 TO 31: POKE I, 191: NEXT I: REM RED SQUARE
60 FOR I=478 to 511: POKE I, 255: NEXT I: REM ORANGE SQUARE
70 CU=512: GOSUB 500: REM MOVE CURSOR BACK TO SCREEN
80 END
90 POKE 40960, INT (CU/256): POKE 40961, CU-(INT(CU/256)*256)
510 RETURN
```

Run the program

Now

POKE 41452,255 - change the flag
Do CSAVE, then try a CLOAD

B. Many of APP's front screens have been created using our Artists & Easel program. After ARTISTS AND EASEL is used to create a picture, it is relocated in memory from 0-511 (using a machine language move routine). Then the tape with the program that goes with that screen is loaded, 41452 is changed and then CSAVE is done.

USING THE TAPE TO LOAD NEW SCREEN FROM A PROGRAM

It is possible to have a program running and periodically load a new front screen from tape and then have the program continue. (With a little imagination other things can be selectively loaded.)

Since the tape system saves and loads the screen plus only INDICATED PROGRAM MEMORY, we can changed INDICATED PROGRAM MEMORY.

End of memory is contained in 2 bytes at 41446 and 41447.

It's best to illustrate this in an example. We will create a picture, save it, create a second picture and also save it. Then we will read the picture in, wait for a return key, and then read the 2nd picture in.

TO CREATE AND SAVE THE PICTURE

```
10 CALL 17046: REM CLEAR SCREEN
20 SHAPE=15: REM CREATE SCREEN THAT IS COLORED HLIN
30 FOR I=0 TO 15
40 COLOR=I: HLIN 0, 31, I: NEXT I
50 GOSUB 500: REM GOTO ROUTINE TO SAVE
70 CALL 17046: REM CLEAR SCREEN
80 FOR I=0 TO 31: REM CREATE SCREEN THAT IS COLORED VLIN
90 COLOR=I: VLIN 0, 15, I: NEXT I
100 GOSUB 500
110 END: REM END
500 POKE 41446, 164: POKE 41447, 1: REM TO CHANGE END OF MEMORY
    POINTER TO 41985 (Hex A401)
510 CALL 34040: CALL 34138: CALL 34061: REM MOTOR ON, SAVE,
    MOTOR OFF END MEMORY POINTER
520 POKE 41446, 191: POKE 41447, 255: REM CORRECT END OF MEMORY
    POINTER
530 RETURN
```

Enter the program, place a tape in the deck, rewind it and press play (it won't start without a motor enable). Now run the program.

The second part is to load the screen back in. Clear the machine (press reset), then

```
10 GOSUB 500: REM GOTO ROUTINE TO READ FIRST SCREEN
20 POKE 40960, 0: INPUT R: REM MOVE CURSOR AND WAIT FOR RETURN
    KEY
30 GOSUB 500: REM GO READ SECOND SCREEN
40 END
500 POKE 41446, 164: POKE 41447, 1: REM CHANGE END MEMORY
    POINTER
510 A=PEEK (41984): B=PEEK (41985): REM SAVE TRUE END OF
    PROGRAM
    IT WILL BE CHANGED WITH LOAD
520 CALL 34040: CALL 34225: CALL 34061: REM: READ TAPE
530 POKE 41446, 191: POKE 41447, 255 REM: CHANGE MEMORY END
    POINTER BACK
540 POKE 41984, A: POKE 41985, B: REM: CHANGE END PROGRAM POINTER
550 RETURN
```

Enter the program, then place the tape in deck, rewind and press play.

Now run the program. After the first screen is loaded press Return to get the 2nd picture.

TO SAVE PROGRAM DATA ON TAPE

When a CSAVE command is given, all of user memory is saved. This means all program statements as well as dimensioned variables. You can enter data into a program, and it can be saved on tape with the program for future use. One problem that has to be overcome is that a RUN command clears all memory from the last statement to the end of memory. Normally a RUN command clears all variable space to zero. There are several ways to get around this.

- A. Using a GOTO statement instead of a RUN command to start a program will not clear the variable area. If the first statement of your program is 10, then start the program with GOTO 10 instead of RUN.

The only thing else to note is that after a system reset, a RUN command must be executed to perform system initialization, otherwise erroneous messages occur. Once a RUN command has been executed, a GOTO may be given as a direct command. Several of APF's program tapes use this method to save and retrieve data.

- B. An alternate and more general way to save data is to

1. Determine amount of memory required for dimensioned variables.
2. Have the first statement of the program do:
POKE 41009, PEEK (41984) - X: POKE 41010, PEEK (41985) - Y
where $X * 256 + Y =$ amount of memory required for variables
3. Before the first RUN of the program
POKE 41984, PEEK (41984) + X: POKE 41985, PEEK (41985) + Y

41984, 41985 - points to end of program storage
41009, 41010 - points to where next variable is stored

As an example

```
10 POKE 41009, PEEK (41984) - 1
20 DIM NA$(30)
30 IF NA$ <> "" THEN 100
40 INPUT "NAME ", NA$
50 CSAVE: STOP
100 PRINT "NAME ", NA$
110 STOP
KEY IN THE PROGRAM
POKE 41984, PEEK (41984 + 1)
RUN THE PROGRAM
```


The first time you RUN, NA\$ is not set to anything and the program asks for an input. Then it saves the program and data to tape.

To see that data was retained:

Reset the system, type CLOAD and load the tape.

RUN the program (type RUN). Your entry for NA\$ was saved and is printed.

SAVING AND RETRIEVING SELECTED AREAS OF MEMORY

Using CALLS, PEEKS and POKES it is possible to save and retrieve selected segments of memory. (A program can bring into memory more data or statements from tape.) To do this, the following CALLS and PEEKS are needed.

Hex	Decimal	Operation
\$84F8	34040	CALL - Enable motor and audio. Wait 2 seconds, then returns.
\$850D	34061	CALL - Shuts off motor and audio.
\$8550	34141	CALL - Puts out 11 second header. Then saves memory contents to tape. Memory area saved is indicated by "High," "Low." Also mem end must equal the same as "High."
\$854B	34228	CALL - Reads from tape to memory. Where it goes in memory is set by "Low," "High."
\$A007	40967	Low - 2 bytes indicating lowest byte of memory to save or read to tape.
\$A009	40969	High - 2 bytes indicating highest byte of memory to save/read to tape.
\$A1F6	41446	Mem End - 2 bytes indicating end of memory

Let's illustrate this in an example (It's not necessary to key in Remarks)

10 DIM A(5)		Rem set up array space
15 Print "Place tape in deck and engage"		Rem print tape message
20 For J = 1 to 3		Rem will do loop 3 times
30 For I = 0 TO 5: Input A(I): Next I		Rem get inputs for 6 elements of Array A.
35 GOSUB 200		Rem: Go set up high, low, mem end.
40 CALL 34141: CALL 34061		Rem: CALL save routine, then stop motor
50 GOSUB 300		Rem restore mem end value
60 Next J		Rem: get 6 more values for A

```
70 Input "Rewind tape, press play, then return key ", K
    Rem: wait for rewind

80 For J = 1 TO 3
    Will read back 3 groups
    of data

90 GOSUB 200
    Go set high/low/mem end

100 CALL 34228
    Go read from tape

110 CALL 34061
    Stop motor

120 For I = 0 TO 5
    Print out values of A

130 Print A(I); " ";

140 Next I

150 Next J
    Go get next group of
    data from tape

160 End

200 POKE 40967, PEEK(41984)
    Rem set low pointers to end
210 POKE 40968, PEEK(41985)
    of program memory
220 POKE 40969, PEEK(41009)
    Rem set high pointer to
230 POKE 40970, PEEK(41010)
    end of variable storage
240 POKE 41446, PEEK(41009)
    Rem set memend to same
250 POKE 41447, PEEK(41010)
    as high
260 CALL 34040: Return
    Rem start motor and return

300 POKE 41446, 191
    Rem: restore mem end
310 POKE 41447, 255
320 Return
```

Place a blank tape in the deck and RUN the program. Enter values for A(I) when asked. The program will ask for 3 groups of 6 values each to use for Array A. After you enter each group it will save them to tape.

When the 3 groups are entered and saved, follow the directions and you will see the program retrieve from tape each of the 3 groups and display them.

CHAPTER VIII

HIGH RESOLUTION GRAPHICS

The Imagination Machine has two modes of high resolution graphics.

MODE 1 - 128 x 192 dots of resolution with 2 color sets, each with 4 colors per set.

MODE 2 - 256 x 192 dots of resolution with 2 color sets, each with 2 colors per set.

These are, in addition to the regular alphanumerics/semigraphics mode, used by the "BASIC" operating system. Both graphics modes are implemented as an "OBJECT DEFINED SYSTEM." An object or shape is defined, and then the screen map shows which object shape is placed in object boxes of the screen. This is analogous to the regular alphanumeric mode where the object shapes - the alphanumeric character set - is predefined in ROM as the ASCII character set. You place the object number (the ASCII code) in the screen map and the VDG decodes the object number into the appropriate video signal. The main difference in the graphics mode is the object shape is defined by the programmer in read/write memory. Therefore we need 2 sections of memory - one for object shape definitions, the other for a screen map.

In either Mode 1 or Mode 2 the screen map is divided into 32 x 12 boxes (384 total). Any one of the defined object shapes can be selected to be placed in any of these 384 boxes, and each box must have an object number assigned to it.

Each box is subdivided into 16 rows, each row with either 4 dots wide (mode 1) or 8 dots wide (mode 2).

Each object shape will require 16 bytes for its definition (1 byte for each of the 16 rows, and each byte is interpreted as 4 dots wide or 8 dots wide depending on which mode is used).

There are 512 bytes of memory allocated for object shape definition. Since each object shape requires 16 bytes for its definition, there can be a maximum of 32 objects that are defined at any one time. Since they are in RAM, they can be redefined.

THE MAP AREAS

The object shape definitions are stored in memory from Hex 200 - 3FF. The first 16 bytes are the definition for object number 0, the next 16 bytes for object #1, etc. Figure 8-1 shows the object shape map.

Figure 8-1 OBJECT SHAPE MAP

Addresses are in hexadecimal

*Object 0:	200 - 20F	Object 10:	300 - 30F
Object 1:	210 - 21F	Object 11:	310 - 31F
Object 2:	220 - 22F	Object 12:	320 - 32F
Object 3:	230 - 23F	Object 13:	330 - 33F
Object 4:	240 - 24F	Object 14:	340 - 34F
Object 5:	250 - 25F	Object 15:	350 - 35F
Object 6:	260 - 26F	Object 16:	360 - 36F
Object 7:	270 - 27F	Object 17:	370 - 37F
Object 8:	280 - 28F	Object 18:	380 - 38F
Object 9:	290 - 29F	Object 19:	390 - 39F
Object A:	2A0 - 2AF	Object 1A:	3A0 - 3AF
Object B:	2B0 - 2BF	Object 1B:	3B0 - 3BF
Object C:	2C0 - 2CF	Object 1C:	3C0 - 3CF
Object D:	2D0 - 2DF	Object 1D:	3D0 - 3DF
Object E:	2E0 - 2EF	Object 1E:	3E0 - 3EF
Object F:	2F0 - 2FF	Object 1F:	3F0 - 3FF

*Address 200 is byte to define top row of Object 0
201 is byte to define next row down of Object 0
202 is byte to define next row down of Object 0
20F is byte to define bottom row of Object 0

The Object Number Map (screen map) is located at Hex 0000-017F. It looks as follows in FIGURE 8-2.

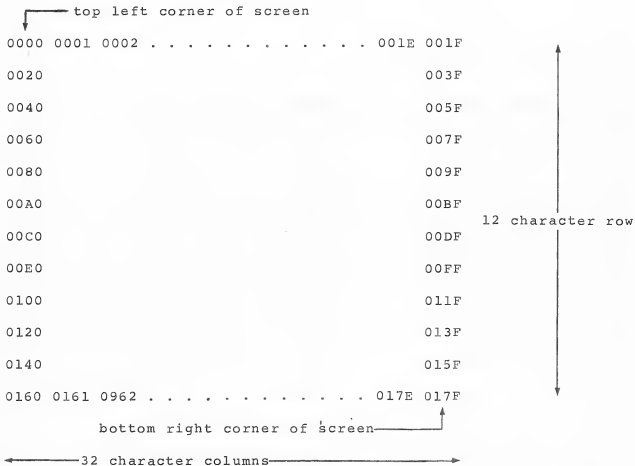
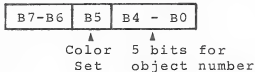


Figure 8-2

Since there are only 32 definable objects, we require only 5 bits in the screen map word to select which object is selected. One more bit of this word is used to select which color set is used. This means an objects color set is selectable on a character by character basis. The screen can actually show each of the 32 objects in both color sets. The final 2 bits of the word are not used.



COLORS OF AN OBJECT SHAPE DEFINITION

Mode 1 - In Mode 1 each byte of a definition is interpreted as 4 bit pairs. Each bit pair selects 1 of 4 colors as follows:

Bit Pair	Color Set 0	Color Set 1
0 0	Green	White
0 1	Yellow	Green
1 0	Blue	Purple
1 1	Red	Orange
Border	Green	White

BORDER COLOR - The border color (screen that is visible outside of the 384 character boxes) takes the color of green or white depending upon the color set used in the right most character box of a line. If Box 001F has color set 0, then the border on those 16 rows is green. If it has color set 1 selected, then its border is white. The top and bottom borders take the color set from the bottom right character (Box 017F).

Mode 2 - In this mode each bit of the object shape definition byte is interpreted as 1 of 2 colors.

Bit	Color Set 0	Color Set 1
0	Black	Black
1	Green	White
Border	Green	White

Border color is determined like Mode 1.

WRITING PROGRAMS IN HIGH RESOLUTION GRAPHICS

USING BASIC

High resolution screens can be created by using just Basic program statements.

Switching Between Modes

To go from alphanumeric to graphics mode, 2 POKES are required

POKE 8193, 60 - This enables the object latch. It also allows orange color set in alphanumeric mode. From this point on, this address does not have to be changed. In inverted alphanumeric mode you will get orange/black letters instead of green. If you want to disable this, then POKE 8193, 52.

POKE 8194, 158 - This will set the display to graphics (128 x 192) mode. Sets $\bar{A}/G=1$: $GMO=0$. When this is done you will no longer see alphanumeric characters. Also, 8193 must be set to 60 previously.

POKE 8194, 222 - This will set the display to graphic (256 x 192) mode. Sets $A/G=1$ and $GMO=1$
POKE with 30 to return to ALPHA/SEMI

SETTING UP SHAPES

Object shapes can be set up by using POKE instructions.

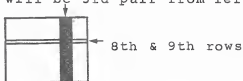
The codes to POKE can be stored as data statements, as arrays, or as absolute statements.

As an example:

To create a high resolution graphics screen where the objects are vertical and horizontal lines.

The object will be a green box with yellow lines for $\frac{1}{4}$ the screen and a white box with green lines for the other $\frac{1}{4}$. Both will be in 128 x 192 mode.

This will be 3rd pair from left



The 16 bytes for the shape will be
4,4,4,4,4,4,4,85,85,4,4,4,4,4,4,4


```
10 POKE 8193, 60: POKE 8194, 158: Rem set up graphics mode
20 For I = 512 to 518: Rem Set up object definition
30 POKE I, 4: POKE I + 9, 4
40 Next I
50 POKE 519, 85: POKE 520, 85
60 For I = 0 to 191: Rem Set object numbers to screen map
70 POKE I, 0: POKE I + 192, 64: Next I
80 POKE 40960, 1: POKE 40961, 129: Rem Move cursor off screen
90 Input K: POKE 40960, 2: Rem Wait for return key
100 POKE 8194, 30: Rem Return to alphanumeric mode
```

It is a good idea before trying to do graphic programs to sit down and carefully sketch out on paper the picture you want to create. Define the various objects and where they are to be placed.

WRITING PROGRAMS IN HIGH RESOLUTION GRAPHICS USING MACHINE LANGUAGE

The examples shown do not give specific addresses where to locate but are general relocatable programs.

Example 1

Entering into the Various Modes

To enter Mode 1 or 2, the PIA in the MP1000 must be changed. There are 3 signals that control graphics mode.

$\overline{A/G}$ - Alphanumeric or Graphic select - must be logic "1" for graphics

GMO - Logic "0" - Mode 1 (128 x 192)
Logic "1" - Mode 2 (256 x 192)

\overline{CLR} - Must be 1, to enable objects codes

The program to enter from semigraphics to graphics mode is

HEX CODE	INSTRUCTION	COMMENTS
B6	LDAA \$2002	Load "A" with PIA Data Register B
20		
02		
84	ANDA #\$3F	Set $\overline{A/G}$ high and GMO low
3F		
8A	ORAA #\$80	If ORAA with \$C0, will set GMO high and go into Graphics Mode 2. ORAA with \$80 goes into Graphics Mode I.
80		
E7	STAA #\$2002	
20		
02		
B6	LDAA \$2001	Load "A" with PIA control Register A
20		
01		
84	ANDA #\$C7	Will set CA2=1 (which is \overline{CLR} Signal).

HEX CODE	INSTRUCTION	COMMENTS
C7		
8A	ORAA #\$38	
38		
B7	STAA \$2001	
20		
01		

EXAMPLE 2 Routine to set same value to a consecutive group of addresses.

Routine is at \$477C (in internal ROM)

Enter routine with

X REGISTER ← start address in memory to get set

B REGISTER ← number of consecutive bytes ., from x register addresses that get set

A REGISTER ← value to be stored

As a simple example, set the top of screen to all have Object 3 in them. First set Object 3 to have all bytes at \$AA.

HEX CODE	INSTRUCTION	COMMENTS
CE	LDX #\$0230	Load x with first address Object #3 starts at \$0230
02		
30		
C6	LDAB #\$10	Load B with count
10		
86	LDAA #\$AA	Load A with value
AA		
BD	JSR \$477C	Jump to subroutine

HEX CODE	INSTRUCTION	COMMENTS
47		
7C		
CE	LDX #\$0000	Load X with top of screen address in graphics mode
00		
00		
C6	LDAB #\$20	Load B with count (32)
20		
86	LDAA #\$03	Load A with value. Value is object 3 code number.
03		
BD	JSR \$477C	Jump to subroutine
47		
7C		

INTERRUPTS

The Imagination Machine has a built-in IRQ Interrupt Servicing Routine. This can only be used during a machine language program. Never allow the interrupts to be enabled while Basic statements are being executed. (The system initialization routine disables the interrupt mask of the 6800 status register.)

The interrupt system is driven by the field sync output of the VDG. This occurs every 1/60 of a second. It is fed to the MC6800 IRQ input via the MP1000 PIA.

INTERRUPT ENABLING/DISABLING

1. TO ENABLE THE INTERRUPTS

- A. CB1 of the PIA must be programmed to accept and input. CB1 Mode is set by the CONTROL REGISTER SB which is Hex Address 2003.

2003 ← Hex 35 - IRQ set by high to low transition of field sync

2003 ← Hex 37 - IRQ set by low to high transition of field sync

- B. A CLI instruction must be given (clear interrupt mask bit in status register).

2. TO DISABLE THE INTERRUPTS

- A. Set 2003 with Hex \$34
- B. Give an SEI instruction

INTERRUPT SERVICING ROUTINE

The built-in interrupt servicing routine does the following

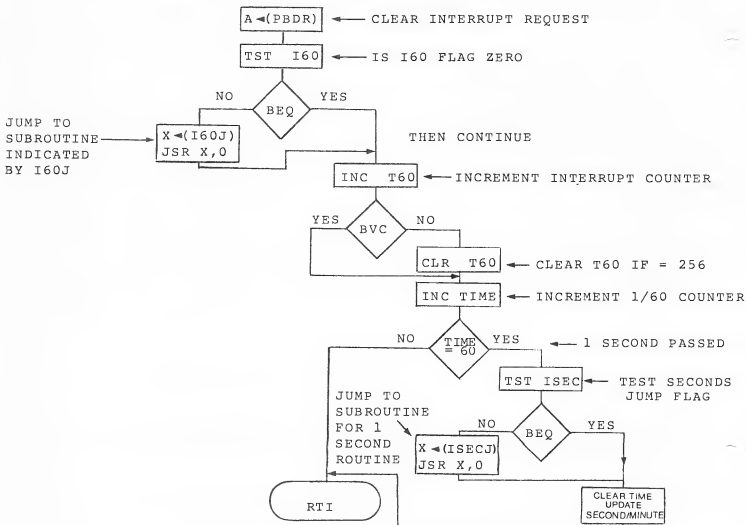
1. Allows jumps to subroutines whose addresses are set by the user.
2. Keeps count of number of interrupts as well as seconds and minutes.

There are several flags and counters used by the Interrupt Routine as follows

- I60 - (\$01FC) - if non zero, causes an immediate JSR from interrupt routine. The JSR address is contained in I60J. Both I60 and I60J are user set.

- I60J - (\$01C5-01C6) - JSR address if I60 is non zero.
- ISEC - (\$01FD) - if ISEC is non zero, then every 60 interrupts (1 second) a JSR will occur. JSR address is contained in ISECJ. Both ISEC and ISECJ are user set.
- ISECJ - (01C7-01C8) - JSR address if ISEC is non zero and 1/60 second counter overflows.
- T60 - (01F8) - incremented by each interrupt. Clears to zero when overflows (at 256th interrupt) and starts count again.
- TIME - (01FB) - keeps count of 1/60 of seconds. Clears when reaches 60 and then causes SECOND to be incremented.
- SECOND - (01F9) - incremented every SECOND. Clears when reaches 60 and increments MINUTE.
- MINUTE - (01FA) - increment every 60 SECONDS. Clears when reaches 60.

Below is a flowchart of the interrupt servicing routine.



EXAMPLE OF INTERRUPT USEAGE

As an example to show how to use the interrupts

1. Each 60th of a second we will add a character to the screen.
2. After 5 seconds we will clear the screen and return to the monitor routine.

SOLUTION

1. First we need 2 routines for the interrupts.

A. 1/60 interrupt - put character to screen

```

0010      DE 00      LDX(00)      SCREEN POINTER
0012      96 02      LDAA (02)    CODE TO SCREEN
0014      A7 00      STAA, 0, X  STORE IT
0016      08        INX          INCREMENT SCREEN POINTER
0017      DF 00      STX 00
0019      4C        INCA        INCREMENT CODE
001A      97 02      STAA 02
001C      39        RTS         RETURN
  
```

B. The 1 second interrupt routine

```

0020      B6 01F9    LDA# SECOND
0023      81 04      CMP# #4
0025      2C 01      BGE +1
0027      39        RTS
0028      0F        SEI
0029      BD 4296    JSR 4296
002C      7E 7000    JMP MONITOR
  
```

2. The initialization and main routine

```

0030      CE 0010    LDX# 0010      SET UP JSR ADDRESSES
0033      FF 01C5    STX 01C5      FOR INTERRUPT ROUTINES
  
```

```
0036    CE 0020    LDX# 0020
0039    FF 01C7    STX 01C7
003C    86 35     LDAA 35     SET UP PIA
003E    B7 2003    STAA 2003
0041    4F        CLRA        CLEAR COUNTERS
0042    B7 01F8    STAA T60
0045    B7 01FB    STAA TIME
0048    B7 01F9    STAA SECOND
004B    97 02     STAA 02     SET CHARACTER CODE
004D    4C        INCA
004E    B7 01FC    STAA I60    CLEAR FLASS FOR INTERRUPT
0051    B7 01FD    STAA ISEC   ROUTINE
0054    CE 0200    LDX# 0200   SET SCREEN ADDRESS
0057    DF 00     STX 00
0059    0E        CLI        ENABLE INTERRUPT
005A    3E        WAI        WAIT FOR INTERRUPT
005B    20 FD     BRA -3
```

If you have everything loaded in correctly, type G0030.

You will see the screen fill up with characters. Since it is putting up 1 character every 60th of a second, for 5 seconds, there will be 300 characters and then the screen clears and goes back to the monitor.

SAVING SPACE AND TIME

There are several things that can be done in a program to save memory space and speed up programs.

SAVING SPACE

1. After a section of a program is running, all Remark statements should be removed to save space.
2. Use multistatements per line. Each new line takes 3 extra bytes. A line can have up to 128 characters, and all keywords are only 1 character.
3. Using subroutines can save space instead of retyping a common used routine.
4. Do not over-dimension strings and arrays.

SPEEDING UP PROGRAMS

1. Place all subroutines at the beginning of a program (lowest step numbers). All statements are stored in ascending order of statement number and when a GOSUB is executed, it starts at the beginning of the program looking for the correct line number. This means a subroutine at step 9000 is found only after the machine looks at all line numbers preceding 9000.
2. Remove Remark statements if possible.
3. Try to use nonsubscripted variables in for/next loops or in frequent calculations. It takes about 3 times as long to find a subscripted variable's value as opposed to a non-subscripted variable.
4. Use multistatements per line.
5. Use and define frequently used variables early in program execution. The machine develops variable lists. Those first used are first on the list and are found quickest.
6. Don't over-dimension strings and arrays.
7. Use machine language routines if possible. Remember Basic is an interpreted language and not compiled.

MC6800 Instruction Set

		ADDRESSING MODES												BDOL/AN/ARITHMETIC OPERATION	COND. CODE REG.								
ACCUMULATOR AND MEMORY		IMMED			DIRECT			INDEX			EXTNO			INHER			(All register labels refer to contents)	S	4	3	2	1	0
OPERATIONS	MINEMONIC	DP	~	#	DP	~	#	DP	~	#	DP	~	#	DP	~	#	H	I	N	Z	V	C	
Add	ADDA	88	2	2	98	3	2	AB	5	2	BB	4	3				A + M → A	1	*	1	1	1	1
	ADDB	C8	2	2	D8	3	2	EB	5	2	FB	4	3				B + M → B ₁	1	*	1	1	1	1
Add Acmltr	ABA													1B	2	1	A + B → A	1	*	1	1	1	1
Add with Carry	ADCA	88	2	2	98	3	2	A8	5	2	88	4	3				A + M + C → A	1	*	1	1	1	1
	ADCB	C8	2	2	08	3	2	E8	5	2	F8	4	3				B + M + C → B	1	*	1	1	1	1
And	ANDA	84	2	2	94	3	2	A4	5	2	94	4	3				A & M → A	1	*	1	1	1	1
	ANDB	C4	2	2	04	3	2	E4	5	2	F4	4	3				B & M → B	1	*	1	1	1	1
Bit Test	BITA	95	2	2	95	3	2	E5	5	2	F5	4	3				A & M	1	*	1	1	1	1
	BITB	C5	2	2	05	3	2	E5	5	2	F5	4	3				B & M	1	*	1	1	1	1
Clear	CLRA													4F	2	1	00 → A	1	*	1	1	1	1
	CLRB													5F	2	1	00 → B	1	*	1	1	1	1
Compare	CMPA	B1	2	2	91	3	2	A1	5	2	B1	4	3				A - M	1	*	1	1	1	1
	CMPB	C1	2	2	01	3	2	E1	5	2	F1	4	3				B - M	1	*	1	1	1	1
Compare Acmltr	CBA													11	2	1	A - B	1	*	1	1	1	1
Complement, 1's	COM							63	7	2	73	6	3				~ M	1	*	1	1	1	1
	CDMA													43	2	1	~ A → A	1	*	1	1	1	1
	COMB													53	2	1	~ B → B	1	*	1	1	1	1
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 - M → M	1	*	1	1	1	1
	NEGA													40	2	1	00 - A → A	1	*	1	1	1	1
	NEGB													50	2	1	00 - B → B	1	*	1	1	1	1
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	1	*	1	1	1	1
Decrement	DEC							6A	7	2	7A	6	3				M - 1 → M	1	*	1	1	1	1
	OECA													4A	2	1	A - 1 → A	1	*	1	1	1	1
	OEGB													5A	2	1	B - 1 → B	1	*	1	1	1	1
Exclusive OR	EDRA	98	2	2	98	3	2	AB	5	2	88	4	3				A ⊕ M → A	1	*	1	1	1	1
	EDRB	C8	2	2	08	3	2	EB	5	2	FB	4	3				B ⊕ M → B	1	*	1	1	1	1
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	1	*	1	1	1	1
	INCA													4C	2	1	A + 1 → A	1	*	1	1	1	1
	INCB													5C	2	1	B + 1 → B	1	*	1	1	1	1
Load Acmltr	LDA	80	2	2	90	3	2	AE	5	2	BE	4	3				M → A	1	*	1	1	1	1
	LDAB	C0	2	2	00	3	2	EE	5	2	FE	4	3				M → B	1	*	1	1	1	1
Dr. Inclusive	ORAA	8A	2	2	9A	3	2	AE	5	2	BA	4	3				A + M → A	1	*	1	1	1	1
	ORAB	CA	2	2	0A	3	2	EA	5	2	FA	4	3				B + M → B	1	*	1	1	1	1
Push Data	PSHA													36	4	1	A → Mgr, SP - 1 → SP	1	*	1	1	1	1
	PSHB													37	4	1	B → Mgr, SP - 1 → SP	1	*	1	1	1	1
Pop Data	PULA													32	4	1	SP + 1 → SP, Mgr → A	1	*	1	1	1	1
	PULB													33	4	1	SP + 1 → SP, Mgr → B	1	*	1	1	1	1
Rotate Left	RDL							69	7	2	79	6	3				M	1	*	1	1	1	1
	RDLA													49	2	1	A	1	*	1	1	1	1
	RDLB													59	2	1	B	1	*	1	1	1	1
Rotate Right	RDR							66	7	2	76	6	3				M	1	*	1	1	1	1
	RORA													46	2	1	A	1	*	1	1	1	1
	RORB													56	2	1	B	1	*	1	1	1	1
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	1	*	1	1	1	1
	ASLA													48	2	1	A	1	*	1	1	1	1
	ASLB													58	2	1	B	1	*	1	1	1	1
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	1	*	1	1	1	1
	ASRA													47	2	1	A	1	*	1	1	1	1
	ASRB													57	2	1	B	1	*	1	1	1	1
Shift Right, Logic	LSR							64	7	2	74	6	3				M	1	*	1	1	1	1
	LSRA													44	2	1	A	1	*	1	1	1	1
	LSRB													54	2	1	B	1	*	1	1	1	1
Store Acmltr	STAA				97	4	2	A7	6	2	07	5	3				A → M	1	*	1	1	1	1
	STAB				07	4	2	E7	6	2	F7	5	3				B → M	1	*	1	1	1	1
Subtract	SUBA	80	2	2	90	3	2	AE	5	2	BE	4	3				A - M → A	1	*	1	1	1	1
	SUBB	C0	2	2	00	3	2	E0	5	2	F0	4	3				B - M → B	1	*	1	1	1	1
Subtract Acmltr	SBA													10	2	1	A - B → A	1	*	1	1	1	1
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A - M - C → A	1	*	1	1	1	1
	SBCB	C2	2	2	02	3	2	E2	5	2	F2	4	3				B - M - C → B	1	*	1	1	1	1
Transfer Acmltr	TAB													16	2	1	A → B	1	*	1	1	1	1
	TBA													17	2	1	B → A	1	*	1	1	1	1
Test Zero or Minus	TST							60	7	2	70	6	3				M - 00	1	*	1	1	1	1
	TSTA													4D	2	1	A - 00	1	*	1	1	1	1
	TSTB													5D	2	1	B - 00	1	*	1	1	1	1

INDEX REGISTER AND STACK

POINTER OPERATIONS	MNEMONIC	IMMED		DIRECT		INDEX		EXTND		INNER		BOOLEAN/ARITHMETIC OPERATION					
		OP	~ #	OP	~ #	OP	~ #	OP	~ #	OP	~ #	H	I	N	Z	V	C
Compare Index Reg	CPX	BC	3 3	BC	4 2	AC	6 2	BC	5 3								
Decrement Index Reg	OEX									08	4 1						
Decrement Stack Ptr	OES									34	4 1						
Increment Index Reg	INX									08	4 1						
Increment Stack Ptr	INS									31	4 1						
Load Index Reg	LOX	CE	3 3	DE	4 2	EE	6 2	FE	5 3								
Load Stack Ptr	LOS	BE	3 3	BE	4 2	AE	6 2	BE	5 3								
Store Index Reg	STX			OF	5 2	EF	7 2	FF	8 3								
Store Stack Ptr	STS			BF	5 2	AF	7 2	BF	8 3								
Index Reg → Stack Ptr	TXS									35	4 1						
Stack Ptr → Index Reg	TSX									30	4 1						

JUMP AND BRANCH

OPERATIONS	MNEMONIC	RELATIVE		INDEX		EXTND		INNER		BRANCH TEST							
		OP	~ #	OP	~ #	OP	~ #	OP	~ #	H	I	N	Z	V	C		
Branch Always	BRA	20	4 2														
Branch If Carry Clear	BCC	24	4 2														
Branch If Carry Set	BCS	25	4 2														
Branch If = Zero	BED	27	4 2														
Branch If > Zero	BGE	2C	4 2														
Branch If >= Zero	BGT	2E	4 2														
Branch If Higher	BHI	22	4 2														
Branch If < Zero	BLE	2F	4 2														
Branch If Lower Or Same	BLS	23	4 2														
Branch If <= Zero	BLT	2D	4 2														
Branch If Minus	BMI	28	4 2														
Branch If Not Equal Zero	BNE	26	4 2														
Branch If Overflow Clear	BVC	28	4 2														
Branch If Overflow Set	BVS	29	4 2														
Branch If Plus	BPL	2A	4 2														
Branch To Subroutine	BSR	BD	B 2														
Jump	JMP			GE	4 2	7E	3 3										
Jump To Subroutine	JSR			AD	B 2	BD	B 3										
No Operation	NOP									01	2 1						
Return From Interrupt	RTI									3B	10 1						
Return From Subroutine	RTS									39	5 1						
Software Interrupt	SWI									3F	12 1						
Wait for Interrupt	WAI									3E	9 1						

CONDITIONS CODE REGISTER

OPERATIONS	MNEMONIC	INNER		BOOLEAN OPERATION	CONDITION CODE REGISTER NOTES												
		OP	~ #		H	I	N	Z	V	C							
Clear Carry	CLC	DC	2 1	0 → C	*	*	*	*	*	R							
Clear Interrupt Mask	CLI	DE	2 1	0 → I	*	R	*	*	*	*							
Clear Overflow	CLV	DA	2 1	0 → V	*	*	*	*	R	*							
Set Carry	SEC	DD	2 1	1 → C	*	*	*	*	*	S							
Set Interrupt Mask	SEI	DF	2 1	1 → I	*	S	*	*	*	*							
Set Overflow	SEV	DB	2 1	1 → V	*	*	*	*	S	*							
Accumr A ← CCR	TAP	06	2 1	A ← CCR													
CCR → Accumr A	TPA	07	2 1	CCR → A													

LEGEND:

- OP Operation Code (Hexadecimal);
- ~ Number of MPU Cycles;
- # Number of Program Bytes;
- + Arithmetic Plus;
- Arithmetic Minus;
- Boolean AND;
- Msp Contents of memory location pointed to by Stack Pointer;
- + Boolean Inclusive OR;
- Boolean Exclusive OR;
- ⌘ Complement of M;
- Transfer Into;
- 0 Bit = Zero;
- 00 Byte = Zero;
- H Half carry from bit 3;
- I Interrupt mask;
- N Negative (sign bit);
- Z Zero (byte);
- V Overflow, 2's complement;
- C Carry from bit 7;
- R Reset Always;
- S Set Always;
- 1 Test and set if true, cleared otherwise;
- *

CONDITION CODE REGISTER NOTES:

- (Bit set if test is true and cleared otherwise)
- Ⓚ (Bit V) Test Result = 10000000?
- Ⓛ (Bit C) Test Result = 00000000?
- Ⓛ (Bit C) Test Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set)
- Ⓛ (Bit V) Test: Operand = 10000000 prior to execution?
- Ⓛ (Bit V) Test: Operand = 01111111 prior to execution?
- Ⓛ (Bit V) Test: Set bit to result of N = C after shift has occurred
- Ⓛ (Bit N) Test: Sign bit of most significant (MSB) byte of result = 1?
- Ⓛ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
- Ⓛ (Bit N) Test: Result less than zero? (Bit 15 = 1)
- Ⓛ (All) Load Condition Code Register from Stack. (See Special Operations)
- Ⓛ (Bit II) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state
- Ⓛ (All C) Set according to the contents of Accumulator A.

Hexadecimal Values of Machine Codes

00	.		40	NEG	A	80	SUB	A	IMM	C0	SUB	B	IMM
01	NOP		41	.		81	CMP	A	IMM	C1	CMP	B	IMM
02	.		42	.		82	SBC	A	IMM	C2	SBC	B	IMM
03	.		43	COM	A	83	.	.	.	C3	.	.	.
04	.		44	LSR	A	84	AND	A	IMM	C4	AND	B	IMM
05	.		45	.		85	BIT	A	IMM	C5	BIT	B	IMM
06	TAP		46	ROR	A	86	LDA	A	IMM	C6	LDA	B	IMM
07	TPA		47	ASR	A	87	.	.	.	C7	.	.	.
08	INX		48	ASL	A	88	EOR	A	IMM	C8	EOR	B	IMM
09	DEX		49	ROL	A	89	ADC	A	IMM	C9	ADC	B	IMM
0A	CLV		4A	DEC	A	8A	ORA	A	IMM	CA	ORA	B	IMM
0B	SEV		4B	INC	A	8B	ADD	A	IMM	CB	ADD	B	IMM
0C	CLC		4C	INC	A	8C	CPX	A	IMM	CC	.	.	.
0D	SEC		4D	TST	A	8D	BSR	.	REL	CD	.	.	.
0E	CLJ		4E	.		8E	LDS	.	IMM	CE	LDX	.	IMM
0F	SEI		4F	CLR	A	8F	.	.	.	CF	.	.	.
10	SBA		50	NEG	B	90	SUB	A	DIR	D0	SUB	B	DIR
11	CBA		51	.		91	CMP	A	DIR	D1	CMP	B	DIR
12	.		52	.		92	SBC	A	DIR	D2	SBC	B	DIR
13	.		53	COM	B	93	.	.	.	D3	.	.	.
14	.		54	LSR	B	94	AND	A	DIR	D4	AND	B	DIR
15	.		55	.		95	BIT	A	DIR	D5	BIT	B	DIR
16	TAB		56	ROR	B	96	LDA	A	DIR	D6	LDA	B	DIR
17	TBA		57	ASR	B	97	STA	A	DIR	D7	STA	B	DIR
18	.		58	ASL	B	98	EOR	A	DIR	D8	EOR	B	DIR
19	DA		59	ROL	B	99	ADC	A	DIR	D9	ADC	B	DIR
1A	.		5A	DEC	B	9A	ORA	A	DIR	DA	ORA	B	DIR
1B	ABA		5B	.		9B	ADD	A	DIR	DB	ADD	B	DIR
1C	.		5C	INC	B	9C	CPX	.	DIR	DC	.	.	.
1D	.		5D	TST	B	9D	.	.	.	DD	.	.	.
1E	.		5E	.		9E	LDS	.	DIR	DE	LDX	.	DIR
1F	.		5F	CLR	B	9F	STS	.	DIR	DF	STX	.	DIR
20	BRA	REL	60	NEG	IND	A0	SUB	A	IND	E0	SUB	B	IND
21	.		61	.		A1	CMP	A	IND	E1	CMP	B	IND
22	BHI	REL	62	.		A2	SBC	A	IND	E2	SBC	B	IND
23	BLS	REL	63	COM	IND	A3	.	.	.	E3	.	.	.
24	BCC	REL	64	LSR	IND	A4	AND	A	IND	E4	AND	B	IND
25	BCS	REL	65	.		A5	BIT	A	IND	E5	BIT	B	IND
26	BNE	REL	66	ROR	IND	A6	LDA	A	IND	E6	LDA	B	IND
27	BEQ	REL	67	ASR	IND	A7	STA	A	IND	E7	STA	B	IND
28	BVC	REL	68	ASL	IND	A8	EOR	A	IND	E8	EOR	B	IND
29	BVS	REL	69	ROL	IND	A9	ADC	A	IND	E9	ADC	B	IND
2A	BPL	REL	6A	DEC	IND	AA	ORA	A	IND	EA	ORA	B	IND
2B	BMI	REL	6B	.		AB	ADD	A	IND	EB	ADD	B	IND
2C	BGE	REL	6C	INC	IND	AC	CPX	.	IND	EC	.	.	.
2D	BLT	REL	6D	TST	IND	AD	JSR	.	IND	ED	.	.	.
2E	BGT	REL	6E	JMP	IND	AE	LDS	.	IND	EE	LDX	.	IND
2F	BLE	REL	6F	CLR	IND	AF	STS	.	IND	EF	STX	.	IND
30	TSX		70	NEG	EXT	B0	SUB	A	EXT	F0	SUB	B	EXT
31	INS		71	.		B1	CMP	A	EXT	F1	CMP	B	EXT
32	PUL	A	72	.		B2	SBC	A	EXT	F2	SBC	B	EXT
33	PUL	B	73	COM	EXT	B3	.	.	.	F3	.	.	.
34	DES		74	LSR	EXT	B4	AND	A	EXT	F4	AND	B	EXT
35	TXS		75	.		B5	BIT	A	EXT	F5	BIT	B	EXT
36	PSH	A	76	ROR	EXT	B6	LDA	A	EXT	F6	LDA	B	EXT
37	PSH	B	77	ASR	EXT	B7	STA	A	EXT	F7	STA	B	EXT
38	.		78	ASL	EXT	B8	EOR	A	EXT	F8	ADC	B	EXT
39	RTS		79	ROL	EXT	B9	ADC	A	EXT	F9	ADC	B	EXT
3A	.		7A	DEC	EXT	BA	ORA	A	EXT	FA	ORA	B	EXT
3B	RTI		7B	.		BB	ADD	A	EXT	FB	ADD	B	EXT
3C	.		7C	INC	EXT	BC	CPX	.	EXT	FC	.	.	.
3D	.		7D	TST	EXT	BD	JSR	.	EXT	FD	.	.	.
3E	WAI		7E	JMP	EXT	BE	LDS	.	EXT	FE	LDX	.	EXT
3F	SWI		7F	CLR	EXT	BF	STS	.	EXT	FF	STX	.	EXT

Notes: 1. Addressing Modes: A = Accumulator A IMM = Immediate
 B = Accumulator B DIR = Direct
 REL = Relative
 IND = Indexed

2. Unassigned code indicated by ****

MACHINE LANGUAGE REFERENCE

The Imagination Machine contains a machine language reference mode. You can use this to create, display, change, and execute machine language programs.

To use this appendix, you must be able to write programs in 6800 machine language. You must also have a working knowledge of hexadecimal notation.

CALL 28672

This BASIC statement takes you out of BASIC. You are now talking to the Imagination Machine Monitor. The monitor puts a "*" at the beginning of each line on the screen. When you see the "*", you can enter one of the three monitor commands:

D nnnn where nnnn is a hexadecimal address
 G nnnn where nnnn is a hexadecimal address
 M nnnn where nnnn is a hexadecimal address

D nnnn - DISPLAY MEMORY

This command will display the 16 bytes of memory beginning at address nnnn. To display the next 16 bytes, press the "/" key. To end the command, press the RETURN key.

```
Example: * D 9B3C
          * 9B3C 20 E0 B6 A0 58 BD 9A B6 CE
          *   9C 7E 9A 28 7C a0/
          * 9B4C AA 20 D4 86 04 CE A0 AA 0C
          *   69 00 09 8C 80 9C 26 (Return)
          *
```

G nnnn - GOTO MEMORY ADDRESS

This command acts much like the BASIC GOTO statement except the value NNNN is a four digit hexadecimal memory address. The

computer immediately begins executing the machine language program at that address.

*G 8894

Address 8894 is the start of the Imagination Machine's BASIC. This is how you reenter BASIC. If you had a BASIC program in memory when you called the monitor, it should still be there.

M nnnn - MODIFY MEMORY

This command immediately displays the contents at memory address nnnn. You can do one of four things:

reply with the "/" key and the command proceeds by displaying the next position in memory.

reply with " " key and the command proceeds to display the previous memory position.

reply with the RETURN key and the command is ended.

reply with a two-digit hexadecimal number and the RETURN key and the command stores this new number in the current memory position. Then you can press Return, /, or with the results as above.

If the M command cannot change the memory location, it will respond with a "?."

APPENDIX C

Schematics/Parts Layouts

FIG

D-1	MP1000 Schematic
D-2	MP1000 Parts Layout
D-3	MPA-10 Schematic
D-4	MPA-10 Parts Layout
D-5	J Connector Schematic
D-6	J Connector Parts Layout
D-7	ROM Cartridge Schematic
D-8	ROM Cartridge Parts Layout
D-9	Tape-Power Board Schematic
D-10	Tape-Power Board Parts Layout
D-11	Keyboard Matrix
D-12	Keyboard Layout

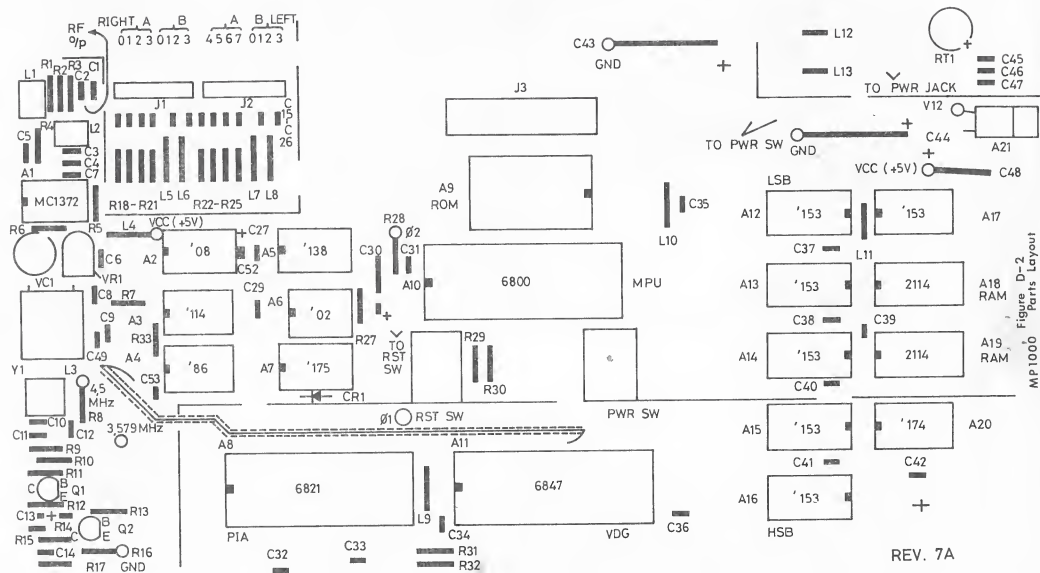


Figure D-2
MP 1000 Pairs Layout

REV. 7A

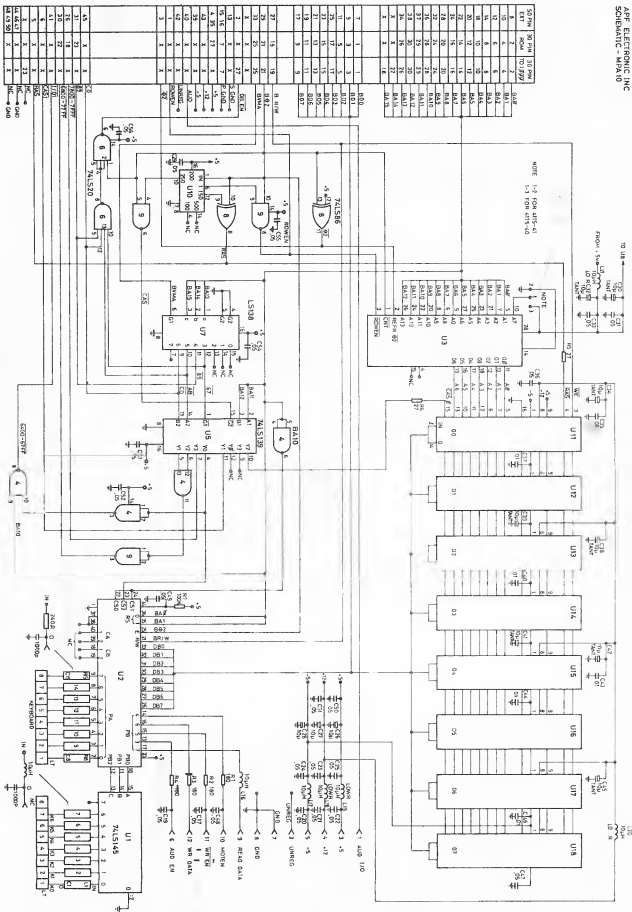


Figure D-3
MPA-10 Main Board Schematic

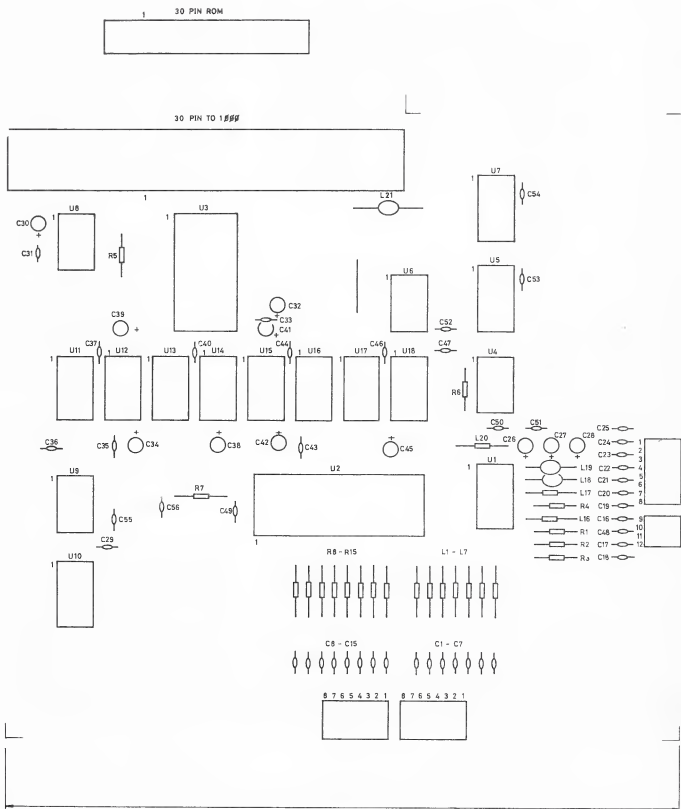


Figure D-4
MPA-10 Main Board Parts Layout

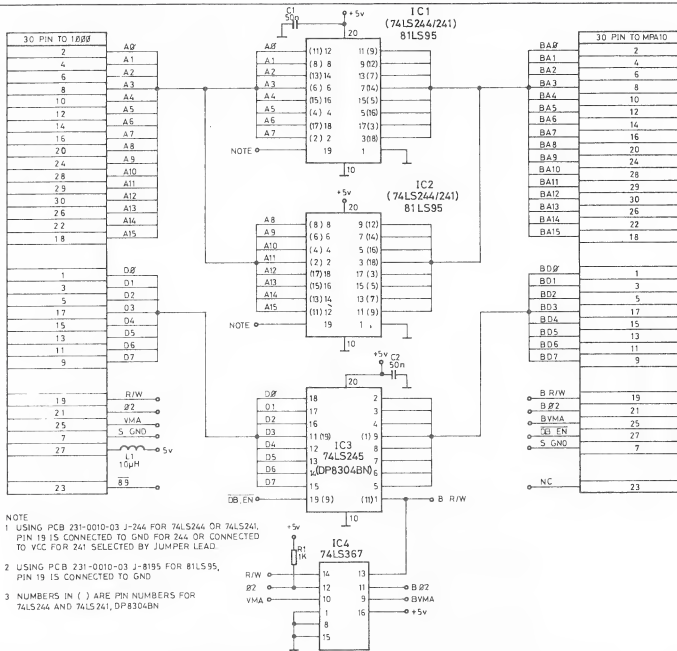
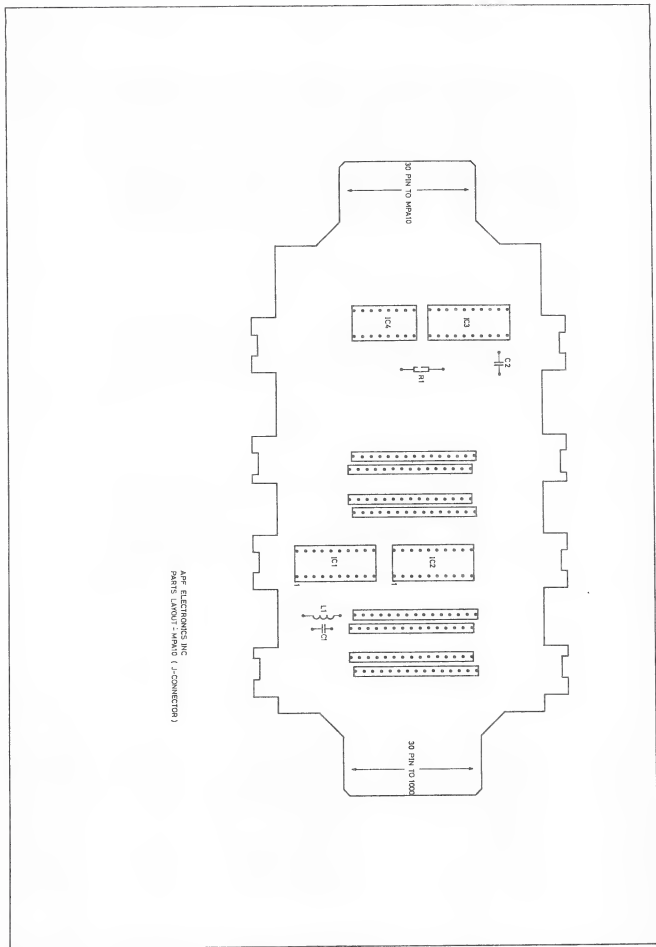


Figure D-5
J Connector Schematic



APF ELECTRONICS INC
 PARTS LAYOUT - MFR/IO (J-CONNECTOR)

Figure D-6
 J Connector Parts Layout

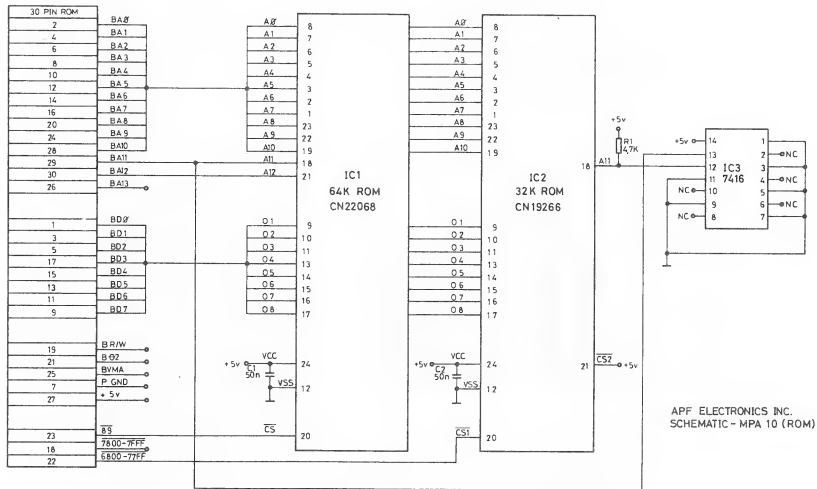
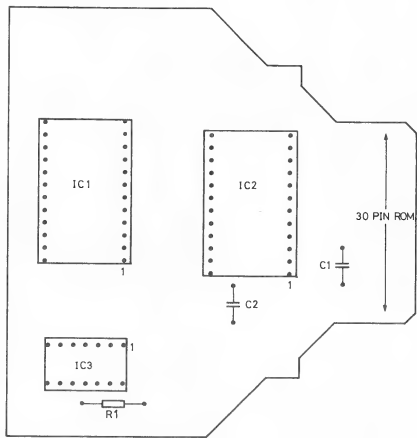


Figure D-7 Schematic
ROM Cartridge



APF ELECTRONICS INC.
PARTS LAYOUT - MPA10 (ROM)

Figure D-8
ROM Cartridge Layout

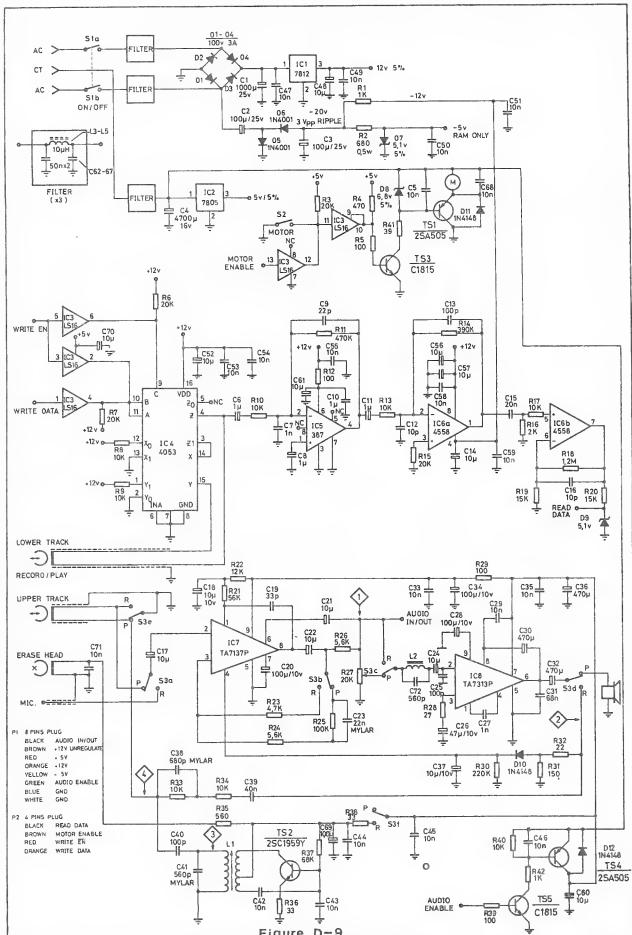


Figure D-9

Tape - Power Board Schematic

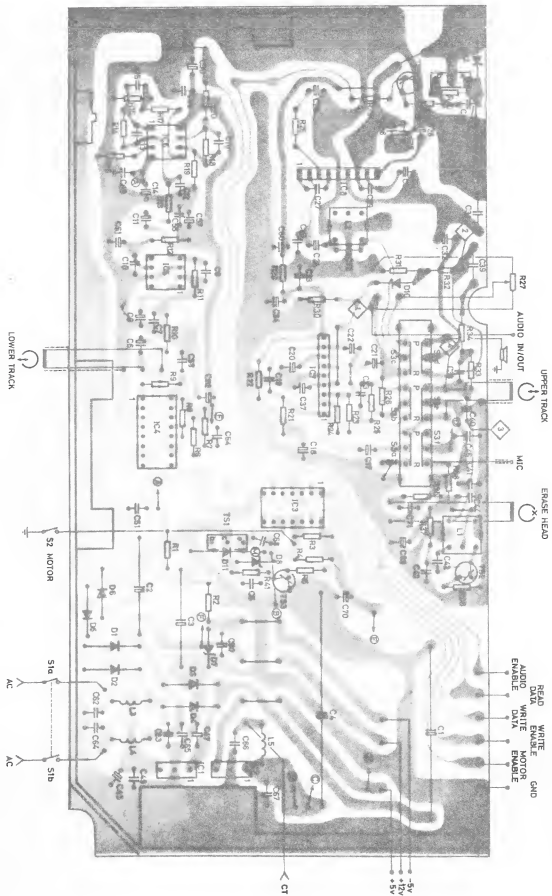


Figure D-10
Tape-Power Board Parts Layout

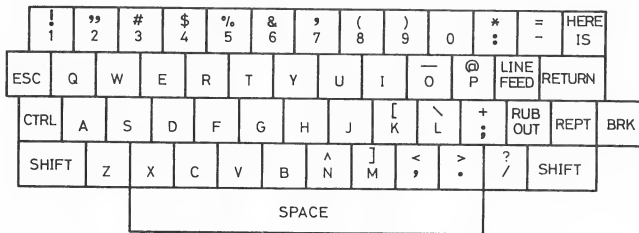
TO PIA A SIDE INPUTS

	PA0	PA1	PA2	PA3	PA4	PA5	PA6	PA7
0	X	Z	Q	2	A	1	W	S
1	C	V	R	3	F	4	E	D
2	N	B	T	6	G	5	Y	H
3	M	,	I	7	K	8	U	J
4	/	.	O	0	L	9	P	;
5	SPACE	:	RE- TURN	/	/	-	LINE FEED	RUB OUT
6	SHIFT	ESC	CTRL	REPT	BREAK	HERE IS	/	/

FROM 74LS145

APF ELECTRONICS INC.
MATRIX - MPA10 (KEYBOARD)

Figure D-11
Keyboard Matrix



APF ELECTRONICS INC.
KEYS LAYOUT - MPA10 (KEYBOARD)

APPENDIX D

ASCII CODES AND IMAGINATION MACHINE INTERNAL STORAGE

Although the Imagination Machine uses an 8-bit word (1 byte) for all memory storage, different interpretation by the machine of these codes occurs. This appendix will clarify how and when a code is interpreted.

1. For all program storage the statement is stored in standard 7-bit ASCII Code. There are 128 standard ASCII Codes (Codes 0-127). Since an 8-bit word is used in memory, there are 256 possible codes. The codes between 128 and 255 are used as "tokens" for the keywords used by Basic. (This means the words PRINT or NEXT each are represented by a single 8-bit code called their token.)
2. Since the Imagination Machine uses a color T.V. for its output and has capability for colored graphics as well as reverse video, it is the codes stored in the screen maps that have to be interpreted differently from the standard ASCII Codes. Screen Codes 0-127 will produce only 64 of the ASCII characters in 1 of 2 video modes (normal or reverse). Codes 128-255 will produce "semigraphics characters."

3. PRINT Statements

The Print Statement deals only with ASCII Codes. When the word PRINT is executed, it goes through a special routine. You can't use the word PRINT to get a semigraphics shape on the screen. Typing PRINT 123 puts the codes for 1, 2, and 3 to the screen. Typing PRINT CHR\$(132) causes the print routines to recognize the code (132) as a token code, and it expands it to its keyword. (PRINT CHR\$(132) will cause DIM to be put on the screen.)

4. POKE Statements

Since Poke Statements simply take the value and place it in memory (regardless of whether it is screen memory, program memory, or even a peripheral address), codes poked to screen memory are not interpreted as in a Print Statement. POKE 512, 132 will cause a green square with a shape of 8 to be put in the top of the screen.

RESERVED WORDS AND THEIR TOKEN CODES

<u>Reserved Word</u>	<u>Token Code (Decimal)</u>	<u>Reserved Word</u>	<u>Token Code (Decimal)</u>
ABS	170	MUSIC	164
ASC	175	NEXT	144
CALL	165	ON	136
CHR\$	174	OPEN	162
CLOAD	151	PEEK	173
CLOSE	163	PLOT	153
COLOR	156	POKE	152
CSAVE	150	PRINT	145
DATA	130	READ	143
DIM	132	REM	148
DIR	166	RESTORE	139
EDIT	158	RETURN	134
END	146	RND	176
FOR	133	RUN	161
GOSUB	128	SAVE	159
GOTO	137	SGN	171
HLIN	154	SHAPE	157
IF	140	SPC	168
INIT	160	STEP	141
INPUT	131	STOP	142
INT	169	TAB	167
KEY\$	177	THEN	135
LEN	176	TO	138
LET	129	USING	149
LIST	147	VLIN	155

ASCII CHARACTER SET (7-BIT CODE)								
M.S. CHAR L.S. CHAR	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 0000	NUL	DLE	SP	0	@	P	'	p
1 0001	SOH	DC1	!	1	A	Q	a	g
2 0010	STX	DC2	"	2	B	R	b	r
3 0011	ETX	DC3	#	3	C	S	c	s
4 0100	EOT	DC4	\$	4	D	T	d	t
5 0101	ENQ	NAK	%	5	E	U	e	u
6 0110	ACK	SYN	&	6	F	V	f	v
7 0111	BEL	ETB	'	7	G	W	g	w
8 1000	BS	CAN	(8	H	X	h	x
9 1001	HT	EM)	9	I	Y	i	y
A 1010	LF	SUB	*	:	J	Z	j	z
B 1011	VT	ESC	+	;	K	[k	{
C 1100	FF	FS	,	<	L	\	l	:
D 1101	CR	GS	-	=	M]	m	}
E 1110	SO	RS	•	>	N	↑	n	~
F 1111	SI	VS	/	?	O	↓	o	DEL

ASCII CODE

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

