THE HACKER'S GUIDE TO ADAM™

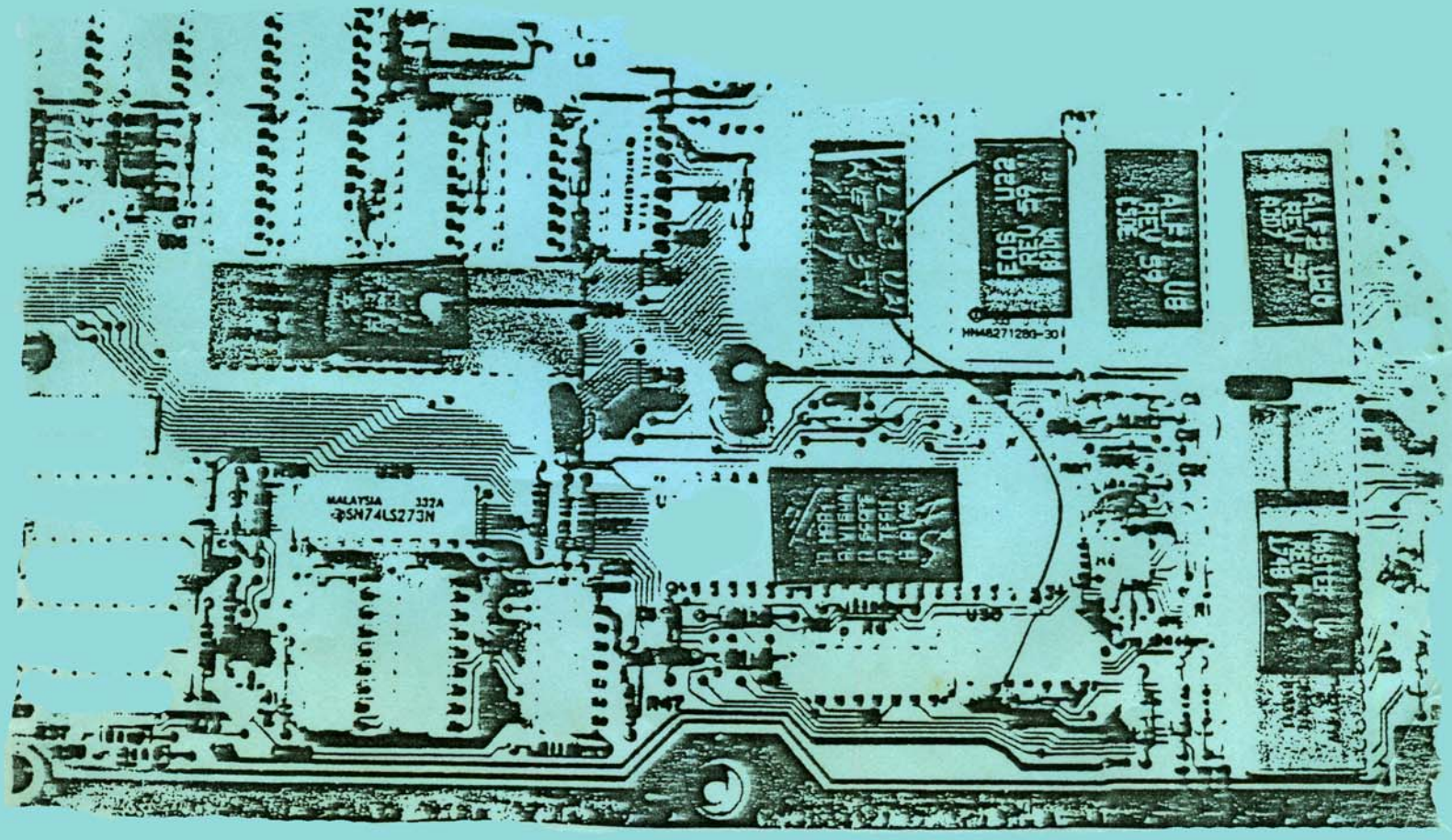
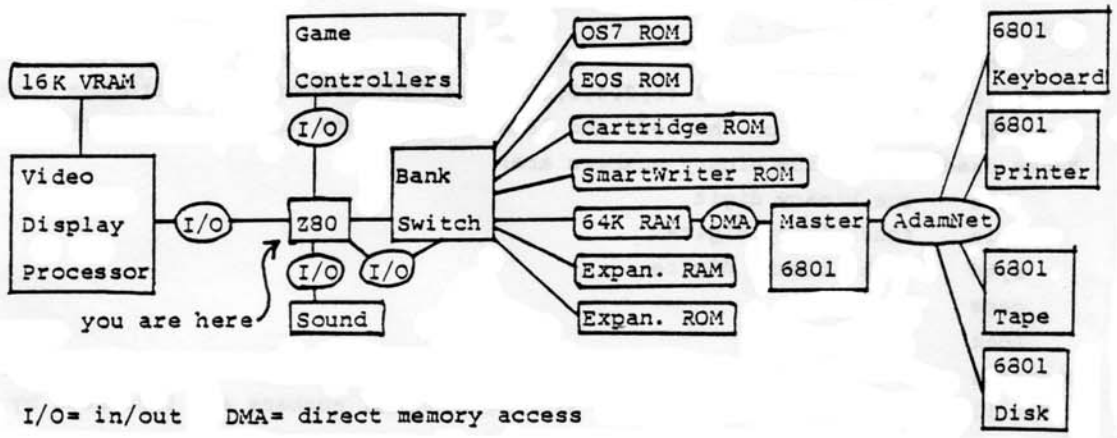


Table of Contents

Program Index

Chapter 1. System Overview.

The Adam has two main circuit boards, the Colecovision board on top and the Adam board on the bottom. The Colecovision board has the Z80 microprocessor that you can program, the video display processor, sound processor, game controllers, and some ROM. The Adam board has the 64K RAM, master 6801 microprocessor that runs AdamNet, ROM`s, and the tape 6801. The system is very complex, much more so than other home computers, because Adam is an expansion of a game board and because it was designed to load tapes during a game while the screen is still active, something computers with only one microprocessor cannot do. The diagram below is an outline of the components that will be discussed in more detail in separate chapters.



I/O= in/out DMA= direct memory access

The Z80 and master 6801 can both read and write the same 64K RAM space, and data is passed from one to the other by leaving it in special locations in RAM where it will be picked up later. This is good for games because action is not interrupted, but seems unnecessarily complicated for computer applications. The Z80 addresses the game controllers, video display processor, and sound in a special in/out space, only accessible from machine language. It can also change the memory it looks at in its normal 64K space by a bank switch controlled from the in/out space.

As a machine language programmer you must think of yourself as being located at the Z80 and writing numbers to, or reading numbers from, the various memory locations and devices. Usually it is only necessary to write a short machine language routine to accomplish a desired task, and the major part of your program can be in BASIC.

CHAPTER 2. Numbers

Several ways of representing numbers are used with computers, which may be a pain at first but is convenient. The numbers actually handled by the Z80 and stored in RAM are in binary (base 2), where 0 is represented by 0.5 to 1 volts and 1 is represented by 3.5 to 4.5 volts. Thus binary is the natural number system for computers because they have two states, just as decimal is the natural number system for us because we have ten fingers. Binary numbers are not used directly to program the Adam, however, because they are quite awkward. Instead several number systems are used, called hexadecimal (base 16), two's complement, and floating point, in addition to the usual decimal used in BASIC. The easiest way to convert numbers from binary to decimal or vice versa is to first convert binary to hexadecimal and then hexadecimal to decimal. Conversion of hexadecimal to decimal is done using the table or subroutine for programs shown later. Such subroutines are never there when you need them, however, and the best way to solve the numbers problem is to buy a hexadecimal-decimal calculator.

BINARY

The binary numbers in the Adam are stored in 8 bit units called bytes. The digits represent powers of 2 (1,2,4,8,16,32,64,128), represented with the most significant bit (128) on the left and least significant bit (1) on the right. Terms used to describe binary numbers are:

bit	one binary digit
nibble	four binary digits
byte	eight binary digits
page	256 bytes
block	four pages

Examples of 8 bit binary numbers are 178 = 10110010 = B2, 55 = 00110111 = $37, 239 =11101111 = EF, 17 = 00010001 = $11. Hexadecimal numbers are indicated by $ when necessary. Binary numbers are not used often by programmers except when certain bits have to be changed or when making shape tables (unless you use a shape-maker program).

Variables in BASIC that are specified as integers by following the name with % (eg. DIM A%(30)),are stored as 2 byte binary numbers, the least significant byte first. Thus the range of possible values is from 0 to FFFF, or 0 to 65,535 decimal. Strings of letters, numbers (0 to 9), and symbols are stored as one byte binary numbers which correspond to the letters etc. according to ASCII code (see the Coleco BASIC manual).

HEXADECIMAL

Hexadecimal representation is convenient when programing in machine language because each digit corresponds to 4 bits in binary, and a byte can always be represented by two hexadecimal digits. Furthermore, addresses in memory are often divided into pages of 256 bytes, and all 64K (65,535) bytes of RAM can be specified by four hexadecimal digits (0000 to FFFF). The problem comes, however, when BASIC is used, since all access to memory (PEEK and POKE) are in decimal. Conversions between hexadecimal and decimal can be made with the table below, finding the decimal number in the table from the first and second hexadecimal digits in the lefthand column and top row, respectively. The

reverse conversion is also convenient. Four digit hexadecimal numbers can be easily converted to decimal by looking up the left two digits, multiplying the decimal equivalent times 256, and adding the result to the decimal equivalent of the right two digits.

Hexadecimal to decimal conversion.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

The table was generated by the following program in BASIC. The "NOT" statements are needed to line up the columns because the TAB command only works to 31, appropriate for the screen but not the printer. It is probably worth printing some of these tables so you can always have one handy.

```
  3 PR #1
  4 PRINT
  5 h$ = "0123456789ABCDEF"
  7 PRINT "    ";
 10 FOR x = 1 TO 16
 20 PRINT MID$(h$, x, 1); "   ";
 30 NEXT: PRINT
 40 FOR x = 1 TO 16
 50 PRINT MID$(h$, x, 1); " ";
 60 FOR y = 1 TO 16
 65 PRINT " ";
 70 IF  NOT INT(n/100) THEN  PRINT " ";
 80 IF  NOT INT(n/10) THEN  PRINT " ";
 90 PRINT n; : n = n+1
100 NEXT y: PRINT: NEXT x
```

The conversion of hexadecimal and decimal numbers in programs can be done with the following subroutines.

```
  5 REM  hex to dec converter
 10 x$ = "0123456789ABCDEF"
 20 INPUT b$
 30 FOR x = 1 TO 2: FOR k = 1 TO 16
 40 IF MID$(b$, x, 1) = MID$(x$, k, 1) THEN  b(x) = k-1: k = 16
 50 NEXT k, x
 55 PRINT b(1)*16+b(2)
 60 GOTO 20
```

TWO`S COMPLEMENT BINARY.

This convention is used to represent positive and negative numbers in binary or hexadecimal, and is used for relative jumps on the Z80. Positive numbers 0 to 127 decimal (01111111 or 7F) are the same as usual for 8 bits. Negative numbers are made by pretending that the byte is the odometer on your car and driving backwards starting at zero. Thus -1 = 11111111, -2 = 1111110, etc. To complement a binary number means to change all the 1`s to 0`s and 0`s to 1`s. Doing just that is called 1`s complement. 2`s complement is 1`s complement plus 1, and the 2`s complement of a number from (decimal) 1 to 127 is the negative of the number. Thus in decimal 255 to 128 are negative numbers in this convention. This is logical because arithmatic in 2`s complement works if you ignore the carry. For example, adding +9 and -2 gives +7.

+9	00001001
-2	11111110
+7	00000111

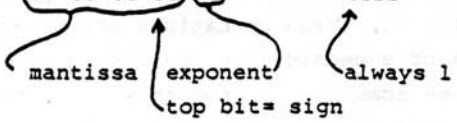
Relative jumps on the Z80 are a little more complicated (as usual) because +2 is added to the offset before the jump.

FLOATING POINT

Numerical variables that are not followed by % are stored in floating point representation, which allows a wide range of values. It is similar to "scientific notation" of calculators or BASIC, with a mantissa times the number base to a power or exponent. For most practical purposes the scale can be regarded as continuous, but it is actually 2^{40} discrete numbers, half of which are between -1 and +1. Zero cannot be represented exactly. The mantissa can take values between 1/2 and (almost) 1, in binary 0.10000... and 0.11111..(the "." being the binary equivalent to a decimal point) ,positive or negative. The exponent is from 0 to 127, positive or negative. There are many different formats for the actual representation in RAM. On the ADAM the mantissa is four bytes and the exponent one byte with the following format. The mantissa bytes are stored in RAM in reverse order, with the least significant first. The most significant byte is strange in that the top bit (left) is assumed to be 1 for the purpose of calculating the number but is in fact used to specify the sign,1=-,0=+. The sign of the exponent is specified by the top bit (1=+, 0=-). thus $80=0, $81=1, $78=-2, etc. The following examples should make this clear. To try other numbers add a line to the printmem program which sets a variable to the number and then look on page 206 or 207 for the number in RAM (see BASIC chapter).

decimal	floating point (hex)	top 4 bits	decimal
1	00 00 00 00 81	1000	1/2 * 2^+1
2	00 00 00 00 82	1000	1/2 * 2^+2
3	00 00 00 40 82	1100	3/4 * 2^+2
4	00 00 00 00 83	1000	1/2 * 2^+3
5	00 00 00 20 83	1010	5/8 * 2^+3
6	00 00 00 40 83	1100	3/4 * 2^+3
7	00 00 00 60 83	1110	7/8 * 2^+3
8	00 00 00 00 84	1000	1/2 * 2^+4
9	00 00 00 10 84	1001	9/16* 2^+4
10	00 00 00 20 84	1010	5/8 * 2^+4

0.5	00 00 00 00 80	1000	1/2 * 2^+0
0.25	FF FF FF 7F 7E	1111etc.	1 * 2^-2
0.001	98 6E 12 03 77	-	- * 2^-9
100	00 00 00 48 87	11001	100/128*2^+7
-1	00 00 00 80 81	1000	-1/2 * 2^+1
-10	00 00 00 A0 84	1010	-5/8 * 2^+4
-0.25	FF FF FF FF 7E	1111	-1 * 2^-2



To translate a floating point number into hexadecimal, write it out in binary, set the top bit, and place the binary point. Then return to hexadecimal starting at the binary point. For example, the number in the floating point accumulator printed out by Printmem is: 00 00 90 7C 8E. Why? Convert to binary:

7 C 9 0 0etc
0111 1100 1001 0000 0000...

Set the top bit and place the point at 14 (8E):

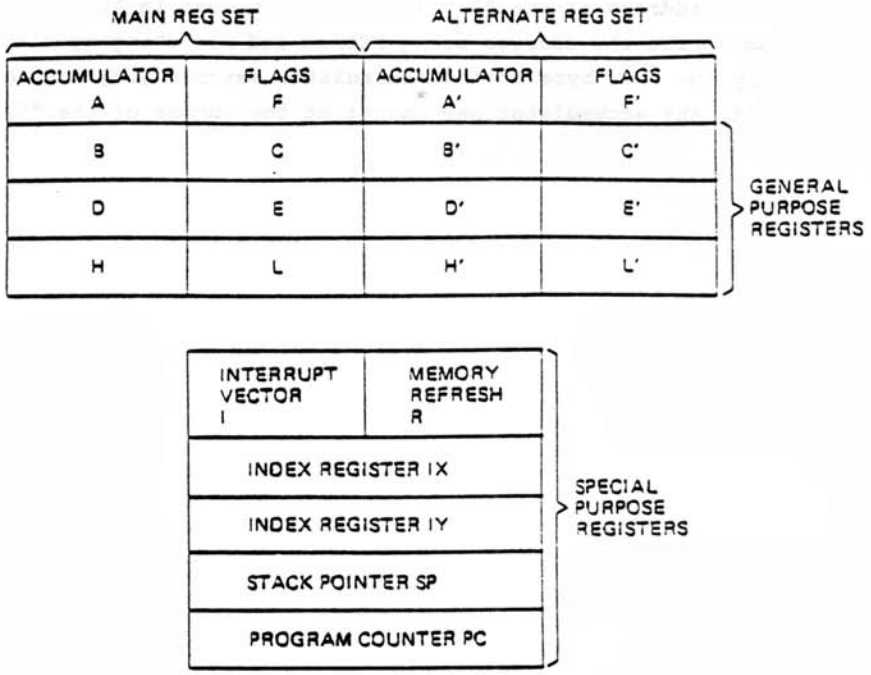
1111 1100 1001 00.00 0000
3 F 2 4

3F24 is the address of the "90" byte of the number in RAM, so the FP accumulator held the address being PEEKed and was changing with each PEEK. Since only the "90" byte of the accumulator was changing during the program at that point, the accumulator was caught at the number of the "90" address.

CHAPTER 3. The Z80

The Z80 microprocessor is the central processing unit (CPU) of the Adam. It steps along programs in RAM, executing simple machine language instructions, much as a calculator is programmed by pushing buttons. The machine language instructions are a series of 8 bit numbers that represent operations that move 8 bit numbers from one register to another, or add two 8 bit numbers, etc. For people to understand what is going on, these operations are usually represented in "assembly language", a series of mnemonics for the instructions which correspond to the machine language numbers. A program which takes mnemonics and turns them into machine language numbers is called an assembler. A program which takes machine language and turns it into mnemonics is called a disassembler. A disassembler, which is given in chapter 4, is useful to print out the machine language programs in the ADAM, which are BASIC, the operating system and SmartWriter, in a form that is reasonable to understand. This chapter will give a brief outline of the Z80 which should be enough to allow understanding of a disassembly listing and simple machine language programming. If more advanced information is needed a complete book on the Z80 such as Rodnay Zaks' "How to program the Z80" should be consulted.

The Z80 has several registers, as shown below.



Z80 Registers

The A register, or accumulator, is the central register and is used in most operations. The F register contains flags, or bits that are set to 1 when certain results of operations occur. The flags are C,Z,P/V,S,N,H.

C=carry flag. C=1 on overflow of arithemitic operations.

Z=zero flag. Z=1 if result of operation is zero.

S=sign flag. S=1 if the MSB of result is 1.

P/V= parity or overflow flag. For parity P/V=1 if the result is even, 0 if it is odd. For overflow, P/V=1 if operation produces overflow.

H=half carry flag. H=1 if add or subtract produce carry or borrow from bit 4 of the accumulator.

N=add/subtract flag. N=1 if the operation was subtract.

The flags are used for conditional branch instructions. In the mnemonics branch occurs on: Z=zero, NZ= not zero, NC= no carry, C= carry, PO= parity odd, PE= parity even, P= plus, and M= minus.

The B,C,D,E,H, and L registers are general purpose and are used individually as 8 bits in some instructions and in pairs (DE, BC, HL) as 16 bits in others. The I (interrupt vector) and R (memory refresh) registers are for special purposes and can be ignored for most applications. The IX and IY registers are 16 bit index registers that are used in some instructions to point to and step through tables etc. The SP (stack pointer) register points to the memory location that is the top of the stack, a last-in-first-out memory area similar to the stack in BASIC that stores addresses to return to after GOSUB's, etc. The PC(program counter) register points to the next location in memory for execution of machine language instructions. All of the special purpose registers (F,I,R,IX,IY,SP,PC) essentially take care of themselves in most short programs and can be ignored.

ADDRESSING MODES

The most complicated aspect of the Z80 is the addressing modes. The address in RAM or the Z80 registers can be specified in various ways. The following types of addressing are described and illustrated with examples. To understand the examples better it will probably help to look ahead where mnemonics are described. An important convention to understand is that if a register or number is enclosed in parentheses, eg. (HL) or (nn), then the number used is the number stored at the address in RAM given by the register or the number following the op code.

IMPLIED ADDRESSING

In this mode the address is implied by the instruction. Examples are " LD A,B" which copies the B register into the accumulator, and "AND H" which ands the H and A registers, the A register being implied.

IMMEDIATE ADDRESSING

In this mode the number to be used is specified in the machine code. Examples are "LD A,n" which copies the next number in RAM into the accumulator, and "LD HL,nn" which copies the 16 bit number nn into the HL register.

ABSOLUTE ADDRESSING

In this mode the address in RAM to be used is specified in the two bytes following the op code in machine language. Examples are "LD A,(nn)" which copies the contents of the memory location with address nn to the A register, and "JP nn" which jumps the program to address nn. The 8 bit numbers of the address are put in memory in reverse order with the low order byte before the high order byte. Thus the instruction "JP 34A8" in machine code is "C3 A8 34" (in hexadecimal).

RELATIVE ADDRESSING

In this mode the byte following the op code is a two's complement number which is added to the program counter + 2 to cause a relative jump. An example is "JR z,e", jump relative on result zero. Values of e from 0 to 7F cause a forward jump and values from 80 to FF cause a backward jump. The disassembler calculates the address jumped to.

INDEXED ADDRESSING

In this mode the address is formed by adding the byte following the op code (called the displacement, or d) to the number in an index register (IX or IY). An example is "LD A,(IX+d)" which loads the number in RAM location specified by adding the contents of index register IX to the displacement d into the A register.

INDIRECT ADDRESSING

In this mode the address is the number in a 16 bit register pair (BC,DE, or HL). An example is "LD A,(BC)" which loads the contents of the memory location specified by the BC register into the A register.

BIT ADDRESSING

A single bit in a byte may be set to 1 (SET), reset to 0 (RES), or tested to set the zero flag (BIT). Various addressing modes may be used to specify the byte. Examples are "SET 3,(HL)", "RES 4,A" and "BIT 7,(IY+d)". The register after the mnemonic specifies the byte to be acted upon.

INSTRUCTION SET

After addressing modes, all there is to learn about the Z80 is the instruction set mnemonics. A list of these with definitions follows.

ADC Add with carry two specified registers. 8 bit additions are made between the A register and any other register or memory location with the result left in the A register. 16 bit additions are between the HL register and other 16 bit registers with the result in HL. In each case the carry flag is added to the result and the carry flag is set if the result exceeds the size of the register.

ADD Add without carry. This instruction is similar to ADC except that the carry flag is not added to the result. The carry flag is set if the result exceeds the size of the register.

AND Logical "AND" the A register with the specified register, number or memory location. Logical AND gives a result where bits in binary are 1 only if they are 1 in both numbers. For example, in binary 10110001 AND 01101001 = 00100001, or in hexadecimal B1 AND 69 = 21, or in decimal 177 AND 105 = 33.

BIT tests the specified bit of the register or memory location addressed and sets the zero flag if the result is zero.

CALL Call subroutine. The program counter is stored on the stack and the address given after the CALL instruction is loaded into the program counter. CALLs may also be conditional.

CCF Complement (reverse) the carry flag.

CP Compare register or memory location with the accumulator. Sets zero flag if the numbers are equal.

CPD Compare with decrement. A is compared with the memory location specified by HL and HL and BC are decremented by 1. The zero flag is set if A = (HL).

CPDR Block compare with decrement. Like CPD but continues until a match is found (A = (HL)) or BC = 0.

CPI Compare with increment. Compares A with (HL), sets zero flag if equal, increments HL by 1 and decrements BC by 1.

CPIR Block compare with increment. Like CPI but continues until A = (HL) or BC = 0.

CPL Complement accumulator. All bits that are 1 are set to 0 and vice versa.

DAA Decimal adjust accumulator. Used in binary coded decimal arithmetic.

DEC Decrement register or memory.

DI Disable interrupts.

DJNZ Decrement B and jump relative on nonzero.

EI Enable interrupts.

EX Exchange specified registers.

EXX Exchange BC, DE, and HL registers with the alternative set.

HALT CPU executes NOP's until an interrupt or reset.

IM Set interrupt mode.

IN Input number to register from port specified by the C register,(C), or number, (n).

INC Increment register or memory location.

IND Input with decrement. Loads (HL) with input from (C), decrements B and decrements HL.

INDR Block input with decrement. Like IND but repeats until B = 0.

INI Input with increment. Loads (HL) with input from (C), increments HL and decrements B.

INIR Block input with increment. Like INI but repeats until B = 0.

JP Jump.

JR Jump relative.

LD Load or copy the contents of a register or memory location to another.

LDD Load with decrement. HL loaded to memory location (DE), DE, HL, and BC are decremented.

LDDR Block load with decrement. Like LDD but repeats until BC = 0.

LDI Load with increment. (HL) is copied to (DE), DE and HL are incremented and BC is decremented.

LDIR Block load with increment. Repeats LDI until BC = 0.

NEG Negate accumulator in two`s complement.

NOP No operation. Fills in spaces in machine code and delays about 1 microsecond.

OR Logical OR accumulator with specified register. Logical OR acts on bits. For example,in binary, 10101100 OR 00010111 = 10111111. In hexadecimal, AC OR 17 = BF. In decimal, 172 OR 23 = 191 (same example each time). 1 OR 1, 1 OR 0,and 0 OR 1 all equal 1. 0 OR 0 = 0.

OTDR Block output with decrement. Like OUTD but repeated until B=0.

OTIR Block output with increment. Like OUTI but repeated until B=0.

OUT Output register specified to port given by the C register, (C), or number, (n).

OUTD Output with decrement. The memory location addressed by the HL register is outputted to the C port. The B and HL registers are decremented.

OUTI Output with increment. The memory location addressed by the HL register is outputted to port C. The HL register is incremented and the B register decremented.

POP Pop specified register (16 bit) from stack, as in BASIC.

PUSH Push register (16 bit) to stack.

RES Reset. The specified bit is set to zero.

RET Return from subroutine. The program counter is popped from the stack, low byte, high byte.

RETI Return from interrupt. Like RET.

RETN Return from non-maskable interrupt. Like RET.

RL Rotate register left through carry flag.

RLCA Rotate accumulator left with branch carry.

RLC Rotate register or memory location left with branch carry.

RLD Rotate left decimal (for BCD).

RR Rotate register or memory location right through carry flag.

RRC Rotate right with branch carry.

RRD Rotate right decimal (for BCD).

RST p Restart at location p*8 in zero page.

SBC Subtract with borrow.

SCF Set carry flag.

SET Set to 1 specified bit of register or memory.

SLA Arithmetic shift left. This multiplies the register or memory location by 2.

SRA Arithmetic shift right.

SRL Logical shift right.

SUB Subtract register specified from the accumulator, the result appearing in the accumulator.

XOR Exclusive OR accumulator and specified register. For example, in binary 10110100 XOR 10001110 = 00111010, or in hexadecimal B4 XOR 8E=3A, or in decimal 180 XOR 142 = 58. XOR A is used to set the accumulator to zero.

How do you use all these codes? To start with you hand assemble some machine language. Some people think you need an assembler to write machine language, but starting with an assembler would be like starting to write english with a word processor. Its unnecessarily complicated.

To illustrate a short machine language program I will show a way around the limitation in BASIC that POKE will not work above 54160. To POKE to higher memory the load commands of the Z80 work fine. In assembly language we write a subroutine as follows:

```
LD A,n
LD (nn),A
RET
```

The code for LD A,n found in the alphabetical assembly language table that follows, is $3E (or 62 in decimal) followed by the 8 bit value of n. The code for LD (nn),A which loads the first n that is now in the accumulator into memory location nn, is $32 (or 50 in decimal). The code for RET (return from subroutine) is $C9 (or 201 in decimal). We can now POKE the decimal numbers into pokable memory as shown in the first five lines of the following program:

```
 5 REM  HIPOKER
10 DATA 62,0,50,0,0,201
20 FOR x = 0 TO 5
30 READ d
40 POKE 210+x, d
50 NEXT
60 INPUT "start address high byte"; adh
70 INPUT "start address low byte"; alo
80 INPUT "number"; n
90 POKE 211,n :POKE 213, alo :POKE 214, adh
```

```
100 CALL 210
110 PRINT n; "  "; PEEK(adh*256+alo)
120 alo = alo+1
130 GOTO 80
```

In this case the program was stored in an unused part of zero page. You can put them anywhere they do not erase a necessary part of BASIC or the operating system (the copywrite statement and "hi Cathy" on page 4, for example). Most programs would be best in the same area as shape tables, above BASIC and below the stack (see pages C-16 and C-20 in the BASIC manual). Such an area must be reserved with a HIMEM* command at the beginning of the BASIC program.

It is not necessary to PUSH registers on the stack at the beginning of a routine called from BASIC and POP them at the end, because the CALL routine does that for you.

Since the limitation to the POKE command is not natural (is not in Apple BASIC, for example), and was simply added by Coleco programmers to try to impede Hackers, it can also be removed by reversing the limitation in BASIC. In our version this is at 3F15, and simply POKing 255 into 16149 and 16150 will allow you to POKE any number in the 64K RAM.

The following table gives a complete list of op codes in alphabetical order which can be used for hand assembly of short machine language routines. The disassembler in this book could also be modified to be a simple assembler to look up op codes for you.

* NOTE: LOMEM is permanent until changed by a new LOMEM command. HIMEM is reset each time a program is RUN.

Z80 op codes (Courtesy of Zilog) 05=d, 8405=nn, 20=n, 2E=e

8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
08	DEC	BC
0D	DEC	C
15	DEC	D
18	DEC	DE
1D	DEC	E
25	DEC	H
28	DEC	HL
DD28	DEC	IX
FD28	DEC	IY
2D	DEC	L
38	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

Z80 op codes (Courtesy of Zilog) 05=d, 8405=nn, 20=n, 2E=e

EDA9	CPD	
EDB9	CPDR	
EDA1	CPI	
EDB1	CPIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
EDBB	OTDR	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

Z80 op codes (Courtesy of Zilog)

05=d, 8405=nn, 20=n, 2E=e

CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB0586	RES	6,(IX+d)
FDCB0586	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	PO
C8	RET	Z
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
0F	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n
9F	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EF	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052F	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

Chapter 4. A Disassembler

The disassembler listing which follows will translate machine code into assembly language. It is essentially several tables of pointers by which the machine language op code points to the assembly language mnemonic and register or address information. These tables are entered as data statements of letters and symbols which are converted to numbers by the ASCII code because it is shorter and requires less typing. The information is then put into string arrays which are: nm$= mnemonics, t$= names of registers etc.; a$(x), b$(x), c$(x) which have pointers to nm$,t$,t$, respectively;d$(x),e$(x) and f$(x) like a$,b$,c$ when the op code begins with ED; and g$, h$, i$, for op codes which begin with CB. Line 23 prints the address in hexadecimal. Line 25 prints the op code. Lines 30-60 check for special codes and gosub appropriately. In lines 100 and 110 n is the number of bytes expected following the op code. The variables pa,pb,and pc are the pointers as numbers extracted from the string arrays. Lines 3000 to 4000 fill the string arrays when the program is first run. Lines 5000 to 5095 are a decimal to hexadecimal conversion subroutine. Lines 6000 to 6020 calculate and print relative jumps.

When you run the program it asks for a starting address, which should be in decimal. It then prints out the disassembled listing until you stop it by typing control s or c. If you have fan-fold paper you can leave it going for hours (plan on leaving the house if you have sensitive ears). To avoid disassembling ASCII, tables and garbage etc., consult the memory map and print out relevant areas of RAM with printmem first because it is much faster. Typical output lines are as follows:

```
]start addr
57344 E000  C5        PUSH  BC
57345 E001  EB        EX    DE,HL
57346 E002  CDE9E1    CALL  nn     E1E9
57349 E005  69        LD    L,C                    i
57350 E006  C1        POP   BC
57351 E007  EB        EX    DE,HL
57352 E008  79        LD    A,C                    y
57353 E009  4B        LD    C,E                    K
57354 E00A  50        LD    D,B                    P
57355 E00B  14        INC   D
57356 E00C  47        LD    B,A                    G
57357 E00D  B7        OR    A
57358 E00E  2806      JR    Z,e  E016              (
57360 E010  EDA3      OUTI
57362 E012  00        NOP
57363 E013  00        NOP
57364 E014  20FA      JR    NZ,e  E010
57366 E016  15        DEC   D
57367 E017  20F7      JR    NZ,e  E010
57369 E019  C9        RET
57370 E01A  C5        PUSH  BC
57371 E01B  EB        EX    DE,HL
57372 E01C  CDE7E1    CALL  nn     E1E7
57375 E01F  69        LD    L,C                    i
57376 E020  C1        POP   BC
57377 E021  EB        EX    DE,HL
57378 E022  79        LD    A,C                    y
57379 E023  4B        LD    C,E                    K
57380 E024  50        LD    D,B                    P
57381 E025  14        INC
```

The address is first printed in decimal and then in hex. The opcode is then printer in hex, followed by the mnemonic and any numbers or jump addresses in hex. On the far right the ASCII of the code is printed to help identify tables, etc.

If you type the program in and it runs alright you may still have made an error by adding an extra data element. To check for that type "? i$(255)" in the immediate mode after running the program. The result should be "@". Checking for substitution errors could be done by driving the program with a for-next loop to generate all op codes and comparing them with the listing at the end of chapter 3.

There may be more efficient ways to write a disassembler for the Z80, but this one works and was enough trouble to write that I am not going to change it. It has some illogical aspects, such as the listing of the mnemonic CPIR twice, that are slightly embarrassing, but still not worth changing. On the other hand it can easily be modified to input hex numbers, etc. which you are welcome to do. It could even be turned into an assembler by creating string arrays of complete mnemonic statements (complete lines) to be searched through for a match to lines typed in. It would be slow but useful . The major work of designing and typing in the data for the op code tables would be done already for the disassembler. If you work on it you are likely to discover the deadly Coleco data-bump bug that adds a space in front of data lines when they are saved on tape. It sounds harmless, but after a few sessions of revisions followed by saving the new version you will find that data is pushed off the end of the line and lost, causing an out of data error the next time you run the program. To avoid this you must go throuth the whole program and edit out the extra spaces with control right arrow every now and then.

```
]
    2 REM          Z80 disassembler by P. Hinkle,March 1984
    5 GOTO 1000
   10 INPUT "start addr"; ad
   11 PR #1
   20 PRINT: op = PEEK(ad)
   21 n = 0: n1 = 0: dc = 0
   22 PRINT ad; TAB(7);
   23 GOSUB 5000
   25 GOSUB 120
   30 IF op = 203 THEN  GOSUB 200: GOTO 150
   40 IF op = 221 THEN  GOSUB 400: GOTO 150
   50 IF op = 237 THEN  GOSUB 600: GOTO 150
   60 IF op = 253 THEN  GOSUB 800: GOTO 150
   66 GOSUB 70
   67 GOTO 150
   70 pa = ASC(a$(op))
   80 pb = ASC(b$(op))
   90 pc = ASC(c$(op))
  100 IF pb = 78 OR pb = 94 OR pc = 78 OR pc = 94 THEN  n = 2: n1 = 2
  110 IF pb = 86 OR pb = 71 OR pb = 89 OR pc = 86 OR pc = 71 OR pc = 89 THEN  n
= 1: n1 = 1
  115 RETURN
  118 ad = ad+1: op = PEEK(ad)
  120 PRINT MID$(x$, INT(op/16)+1, 1);
  130 PRINT MID$(x$, (op/16-INT(op/16))*16+1, 1);
  140 RETURN
  150 IF n > 0 THEN  ad = ad+1: n = n-1: op = PEEK(ad): GOSUB 120
  160 IF n > 0 THEN  ad = ad+1: op = PEEK(ad): GOSUB 120
  170 PRINT TAB(23)
  180 PRINT nm$(pa-49); TAB(29); t$(pb-64);
  181 IF pc = 117 THEN  GOTO 185
  183 PRINT ","; t$(pc-64);
  185 IF n1 = 2 THEN  PRINT SPC(4): GOSUB 120: op = PEEK(ad-1): GOSUB 120
  186 IF pa = 77 OR pa = 64 THEN  GOSUB 6000
  187 pp = POS(0)
```

```
 188 IF pp < 20 THEN  pp = pp+31
 189 PRINT SPC(60-pp);
 190 IF n1 = 2 THEN  GOSUB 5100
 192 IF n1 = 1 THEN  GOSUB 5100
 194 GOSUB 5100
 199 ad = ad+1: GOTO 20
 200 REM                          CB routine
 210 GOSUB 118
 230 pa = ASC(g$(op))
 240 pb = ASC(h$(op))
 250 pc = ASC(i$(op))
 260 GOSUB 100: RETURN
 400 REM                       DD routine
 420 GOSUB 118
 430 IF op = 203 THEN  GOSUB 118: GOSUB 200: dc = 1: GOTO 450
 440 GOSUB 70
 450 IF pb = 95 THEN  pb = 96: IF dc = 0 THEN  GOSUB 118
 452 IF pb = 72 THEN  pb = 76
 454 IF pc = 95 THEN  pc = 96: IF dc = 0 THEN  GOSUB 118
 456 IF pc = 72 THEN  pc = 76
 460 RETURN
 600 REM                          ED routine
 610 GOSUB 118
 630 pa = ASC(d$(op-64))
 640 pb = ASC(e$(op-64))
 650 pc = ASC(f$(op-64))
 660 GOSUB 100: RETURN
 800 REM                       FD routine
 810 GOSUB 118
 820 IF op = 203 THEN  GOSUB 118: GOSUB 200: dc = 1: GOTO 850
 830 GOSUB 70
 850 IF pb = 95 THEN  pb = 97: IF dc = 0 THEN  GOSUB 118
 852 IF pb = 72 THEN  pb = 77
 854 IF pc = 95 THEN  pc = 97: IF dc = 0 THEN  GOSUB 118
 856 IF pc = 72 THEN  pc = 77
 860 RETURN
1000 x$ = "0123456789ABCDEF"
2000 DATA    A,B,C,D,E,H,L,n,HL,BC,DE,SP,IX,IY,nn,M,NC,NZ,P,PE,PO,Z,e,(SP),(C),
(n),(IX)
2001 DATA       (IY),(BC),(DE),(nn),(HL),(IX+d),(IY+d),0,1,2,3,4,5,6,7,I,R,00H,
08H,10H,18H,20H,28H,30H,38H,?, ,AF,AF`,(A),(HL)
2002 DATA       ADC,ADD,AND,BIT,CALL,CCF,CP,CPD,CPDR,CPIR,CPI,CPL,DAA,DEC,DI,DJ
NZ,EI,EX,EXX,HALT
2003 DATA       IM,IN,INC,IND,INDR,INI,INIR,JP,JR,LD,LDD,LDDR,LDI,LDIR,NEG,NOP,
OR,OTDR,OTIR,OUT,OUTD
2004 DATA     OUTI,POP,PUSH,RES,RET,RETI,RETN,RL,RLA,RLC,RLCA,RLD,RR,RRA,RRC,RR
CA,RRD,RST,SBC,SCF
2005 DATA      SET,SLA,SRA,SRL,SUB,XOR,RETI,?,CPIR
2010 DATA      T,N,N,G,G,>,N,d,B,2,N,>,G,>,N,i,@,N,N,G,G,>,N,b,M,2,N,>,G,>,N,g,
M,N,N,G,G,>,N
2011 DATA       =,M,2,N,>,G,>,N,<,M,N,N,G,G,>,N,m,M,2,N,>,G,>,N,6
2012 DATA       N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N

2013 DATA       N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,D,N,N,N,N,N,N,N,N

2014 DATA       2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1,r,r,r,r,r,r,r,r,l,l,l,l,l,l,l,l

2015 DATA       3,3,3,3,3,3,3,3,s,s,s,s,s,s,s,s,U,U,U,U,U,U,U,U,7,7,7,7,7,7,7,7

2016 DATA       ^,[,L,L,5,\,2,k,^,^,L,t,5,5,1,k,^,[,L,X,5,\,r,k,^,C,L,F,5,t,1,k

2017 DATA       ^,[,L,B,5,\,3,k,^,L,L,B,5,t,s,k,^,[,L,?,5,\,U,k,k,N,L,A,5,t,7,k

2020 DATA       u,I,\,I,A,A,A,u,v,H,@,I,B,B,B,u,V,J,],J,C,C,C,u,V,H,@,J,D,D,D,u
,Q,H
2021 DATA       ^,H,E,E,E,u,U,H,H,H,F,F,F,u,P,K,^,K,_,_,_,_,u,B,H,@,K,@,@,@,u
2022 DATA       A,A,A,A,A,A,A,A,B,B,B,B,B,B,B,B,C,C,C,C,C,C,C,C,D,D,D,D,D,D,D,D

2023 DATA       E,E,E,E,E,E,E,E,F,F,F,F,F,F,F,F,_,_,_,_,_,_,_,u,_,@,@,@,@,@,@,@,@

2024 DATA       @,@,@,@,@,@,@,@,@,@,@,@,@,@,@,@,@,A,B,C,D,E,F,_,@,@,@,@,@,@,@,@,@

2025 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
```

```
 2027 DATA        Q,I,Q,N,Q,I,@,l,U,u,U,t,U,N,@,m,P,J,P,Y,P,J,G,n,B,u,B,@,B,t,@
 2028 DATA        o,T,H,T,W,T,H,G,p,S,_,S,J,S,t,G,q,R,v,R,u,R,v,G,r,O,K,O,u,O,u,G
,s
 2030 DATA        u,N,@,u,u,u,G,u,w,I,\,u,u,u,G,u,u,N,@,u,u,u,G,u,u,J,],u,u,u,G,u
,V,N,H,u,u,u,G,u,V,H
 2031 DATA        ^,u,u,u,G,u,V,N,@,u,u,u,G,u,V,K,^,u,u,u,G,u
 2040 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@

 2041 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,u,@,A,B,C,D,E,F,_,@

 2050 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,u,u,u,u,u,u,u,u,A,B,C,D,E,F,_,@

 2051 DATA        u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u

 2060 DATA        u,u,N,u,N,u,G,u,u,u,N,t,N,u,G,u,u,u,N,@,N,u,u,u,u,u,N,G,N,t,G,u
,u,u,N,H,N,u,u,u,u,u
 2061 DATA        N,H,N,t,u,u,u,u,N,u,N,u,u,u,u,H,N,u,N,u,u,u
 2070 DATA         F,X,l,N,S,^,E,N,F,X,↑,N,u,_,u,N,F,X,l,N,u,u,E,N,F,X,↑,N,u,u,E,
N,F,X,l,u,u,u,u,j,F,X,↑,u,u,u,u
 2071 DATA         e,u,u,l,N,u,u,u,u,F,X,↑,N,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
 2072 DATA        V,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,Q,;,J,Z,u,u,u,u
 2073 DATA        0,8,H,Y,u,u,u,u,R,v,K,W,u,u,u,u,P,9,I
 2079 DATA         A,X,H,^,u,u,b,j,B,X,H,I,u,u,u,k,C,X,H
 2080 DATA         ^,u,u,c,@,D,X,H,J,u,u,d,@,E,X,H,u,u,u,u,u,F,X,H,u,u,u,u,u,u,u,u,
H,^,u,u,u,u,@
 2081 DATA         X,H,K,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,
u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
 2082 DATA        u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
 2083 DATA         X,A,I,I,u,u,@,u,X,B,I,^,u,u,u,@,X,C,J,J,u,u,u,j,X,D,J,^,u,u,u,
k,X,E,H,u,u,u,u,u,X,F,H,u,u,u,u,u,u,u,K,K
 2084 DATA         u,u,u,u,X,@,K,^,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,
u,u,u
 2085 DATA        u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
,u,u,u,u,u
 2086 DATA        c,c,c,c,c,c,c,c,h,h,h,h,h,h,h,h,a,a,a,a,a,a,a,a,f,f,f,f,f,f,f,f

 2087 DATA        o,o,o,o,o,o,o,o,p,p,p,p,p,p,p,p,u,u,u,u,u,u,u,u,q,q,q,q,q,q,q,q

 2088 DATA        4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4

 2089 DATA        4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4

 2090 DATA        ],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],]

 2091 DATA        ],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],],]

 2092 DATA        n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n

 2093 DATA        n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n

 2100 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@

 2101 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,t,t,t,t,t,t,t,t,A,B,C,D,E,F,_,@

 2110 DATA        b,b,b,b,b,b,b,b,c,c,c,c,c,c,c,c,d,d,d,d,d,d,d,d,e,e,e,e,e,e,e,e

 2111 DATA        f,f,f,f,f,f,f,f,g,g,g,g,g,g,g,g,h,h,h,h,h,h,h,h,i,i,i,i,i,i,i,i

 2112 DATA        b,b,b,b,b,b,b,b,c,c,c,c,c,c,c,c,d,d,d,d,d,d,d,d,e,e,e,e,e,e,e,e

 2113 DATA        f,f,f,f,f,f,f,f,g,g,g,g,g,g,g,g,h,h,h,h,h,h,h,h,i,i,i,i,i,i,i,i

 2114 DATA        b,b,b,b,b,b,b,b,c,c,c,c,c,c,c,c,d,d,d,d,d,d,d,d,e,e,e,e,e,e,e,e

 2115 DATA       f,f,f,f,f,f,f,f,g,g,g,g,g,g,g,g,h,h,h,h,h,h,h,h,i,i,i,i,i,i,i,i
 2120 DATA       u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
 2121 DATA       u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u,u
 2122 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
 2123 DATA        A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@

 2124 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
 2125 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
 2126 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
 2127 DATA       A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@,A,B,C,D,E,F,_,@
```

```
3000 DIM nm$(69)
3001 DIM t$(57)
3002 DIM a$(255)
3003 DIM b$(255): DIM c$(255)
3004 DIM d$(122): DIM e$(122): DIM f$(122)
3005 DIM g$(255): DIM h$(255): DIM i$(255)
3010 FOR x = 0 TO 57: READ t$(x): NEXT
3020 FOR x = 0 TO 69: READ nm$(x): NEXT
3021 FOR x = 0 TO 255: READ a$(x): NEXT
3022 FOR x = 0 TO 255: READ b$(x): NEXT
3023 FOR x = 0 TO 255: READ c$(x): NEXT
3030 FOR x = 0 TO 122: READ d$(x): NEXT
3031 FOR x = 0 TO 122: READ e$(x): NEXT
3032 FOR x = 0 TO 122: READ f$(x): NEXT
3040 FOR x = 0 TO 255: READ g$(x): NEXT
3041 FOR x = 0 TO 255: READ h$(x): NEXT
3042 FOR x = 0 TO 255: READ i$(x): NEXT
4000 GOTO 10
5000 a = INT(ad/4096)
5010 PRINT MID$(x$, a+1, 1);
5020 b = ad-a*4096
5030 c = INT(b/256)
5040 PRINT MID$(x$, c+1, 1);
5050 d = b-c*256
5060 e = INT(d/16)
5070 PRINT MID$(x$, e+1, 1);
5080 f = d-e*16
5090 PRINT MID$(x$, INT(f)+1, 1);
5092 PRINT "  ";
5095 RETURN
5100 jj = PEEK(ad-n1)
5110 IF jj > 33 AND jj < 123 THEN  PRINT CHR$(jj);
5120 n1 = n1-1: RETURN
6000 PRINT "  ";
6010 IF op > 127 THEN  op = op-256
6015 oad = ad
6020 ad = ad+op+1: GOSUB 5000: ad = oad: RETURN
```

Printmem is a short program that prints out RAM in a convenient format to interpret before disassembling. The ASCII equivalents of the numbers are printed on the left with = signs for non-ASCII numbers. Lines of 16 hexadecimal numbers are then printed in pages of 256. The format is particularly useful for interpreting tables and variable or string areas. A sample printout of page 4 is shown following the program.

Viewer is a very short program which displays pages of RAM on the screen as ASCII and graphics characters. It is a good one to start with.

```
 1 REM    VIEWER  by P. Hinkle
 5 INPUT "page"; p
10 FOR j = 0 TO 240 STEP 16
15 PRINT "     ";
20 FOR i = 0 TO 15
30 x = p*256+i+j
40 t = PEEK(x)
41 IF t = 12 OR t = 13 OR t = 16 OR t = 128 OR t = 10 THEN  t = 61
42 IF t = 0 OR t = 7 OR t = 8 OR t = 9 THEN  t = 61
43 IF t = 22 OR t = 24 OR t = 28 THEN  t = 61
44 IF t > 159 AND t < 164 THEN  t = 61
45 IF t = 148 OR t = 151 THEN  t = 61
50 PRINT CHR$(t);
60 NEXT i
70 PRINT
80 NEXT j
90 GOTO 5
```

```
]
   1 REM  PRINTMEM
   2 PR #1
   3 h$ = "0123456789ABCDEF"
   5 INPUT "page"; p
   6 PRINT p
  10 FOR j = 0 TO 240 STEP 16
  15 PRINT "      ";
  20 FOR i = 0 TO 15
  30 x = p*256+i+j
  40 t = PEEK(x)
  41 IF t < 33 OR t > 126 THEN  t = 61
  50 PRINT CHR$(t);
  60 NEXT i
  65 GOSUB 200
  70 PRINT
  80 NEXT j
  85 PRINT: PRINT: PRINT: PRINT: PRINT
  90 p = p+1: GOTO 6
 200 PRINT TAB(30);
 210 FOR i = 0 TO 15
 220 a = PEEK(p*256+i+j)
 230 b = a/16
 240 c = INT(b)
 250 GOSUB 300
 260 c = (b-INT(b))*16
 270 GOSUB 300
 280 PRINT " ";
 290 NEXT i
 295 RETURN
 300 c = c+1
 310 d$ = MID$(h$, c, 1)
 315 ww = FRE(9)
 320 PRINT d$;
 330 RETURN
]
```

```
                page4
==:=:==:C>=:i>=:    02 1B 3A 80 3A 05 1B 3A 43 3E 1B 3A 69 3E 1B 3A
=6>=:=N>=:='>=:=    02 36 3E 1B 3A 02 4E 3E 1B 3A 02 27 3E 1B 3A 04
Hi=Cathy=FATAL=S    48 69 20 43 61 74 68 79 12 46 41 54 41 4C 20 53
YSTEM=ERROR=====    59 53 54 45 4D 20 45 52 52 4F 52 1C 0C 20 20 20
==Coleco=SmartBA    20 20 43 6F 6C 65 63 6F 20 53 6D 61 72 74 42 41
SIC=V1.0=(c)=198    53 49 43 20 56 31 2E 30 20 28 63 29 20 31 39 38
3,=Lazer=MicroSy    33 2C 20 4C 61 7A 65 72 20 4D 69 63 72 6F 53 79
stems=Inc=]==:==    73 74 65 6D 73 20 49 6E 63 01 5D 00 01 3A 01 0D
=NEXT=without=FO    10 4E 45 58 54 20 77 69 74 68 6F 75 74 20 46 4F
R=Syntax=RETURN=    52 06 53 79 6E 74 61 78 14 52 45 54 55 52 4E 20
without=GOSUB=Ou    77 69 74 68 6F 75 74 20 47 4F 53 55 42 0B 4F 75
t=of=DATA=Illega    74 20 6F 66 20 44 41 54 41 10 49 6C 6C 65 67 61
l=Quantity=Overf    6C 20 51 75 61 6E 74 69 74 79 08 4F 76 65 72 66
low=Out=of=Memor    6C 6F 77 0D 4F 75 74 20 6F 66 20 4D 65 6D 6F 72
y=Stack=Overflow    79 0E 53 74 61 63 6B 20 4F 76 65 72 66 6C 6F 77
=Undefined=State    13 55 6E 64 65 66 69 6E 65 64 20 53 74 61 74 65
```

CHAPTER 5. Memory Map (all numbers hexadecimal).

0000-00FF Zero page. interrupt routines. All C9 (return) except at 66-AB = NMI every 16.7 ms, runs FLASH.

0100 Start of BASIC

0101-0104 Pointers for version of Basic. See Coleco manual p.C23 My version has A3 3E C3 4F here.

010B-03A8 Basic word table. Format: token (1 byte), address in address table (2 bytes), number of letters in word (1 byte), word.

03A9-041F Routine address table. Format: number of addresses (1 byte), address(es) (2 bytes each).

0420-047F Hi Cathy and copyright statement.

0480- Error messages. Format: number of letters (1 byte), message in ASCII.

05B8-3ED8 Basic routines. Identify from word and address tables.

3ED9 Himem pointer.

3EDE Lomem pointer.

3EE3 Pointer to start of numeric variables.

3EED Pointer to end of numeric variables.

3EEF Pointer to start of string space.

3EF3 Pointer to end of string space.

3EFE Line number for ONERR GOTO.

3F01. Speed (FF).

3F02 USR address. CALL is better than USR. Forget it.

3F04 @ address.

3F15 POKE limit

3F22-3F26 FP accumulator (see chap. 2).

3F2B-3F2F FP operand.

3F32 number of digits in FP result.

3FA4-4045 Basic words, math. Format: number of letters, word, 88 or A8, address.

42A3 Text background color.

4EAA-4F4E Tape word table. Format: number of letters, word, address table pointer (1 byte), which gives the offset of the address from the beginning of address table.

4F4F-4FA5 Tape address table. Format: 2 byte address of routine. Pointed to by offset in word table.

4FA6-5E3F Tape routines. see tape word and address tables.

5E40-5EE8 Tape error messages. Format: number of letters, message.

6B00 Approximate location of string variable table. Format: 03 21 address (2 bytes), name (2 bytes).

6B00 After string table is the numeric variable table. Format: 03 01, address (2 bytes), name (2 bytes).

6B00 After numeric table is Basic math word table.

6C00 String space. Format: address in table, number of letters (bytes), string.

CE00-CF00 Numeric variables (see chap. 2). Numbers are preceded by letters of the name after first two.

CF00- Tokenized BASIC program (see chap. 8).

D200- Stack

D400-D700 Buffer from tape: catalog. Format: name, type, 17 bytes (see chapter 15).

D800- Buffer from tape: last program loaded.

E000 Start of operating system (see chapter 7).

FBFF OS data tables

FC30-FD5E OS jump table.

FD50- Global RAM

FD75 Keyboard input byte.

FEC0 Processor control block

FEC4- Device control block

IN/OUT space.

1E Auto dialer

3F Network reset, output odd,even =EOS enable

4F Expansion connector #2

5E Modem data I/O

5F Modem control status

7F Memory bank switch

A0-BF Video display processor.

E0-FF Sound generator. Out only.

FC-FF Game controllers. In only.

Chapter 6. Memory Bank Switches.

The Z80 can only address 64K (2^{16}) memory locations but can change blocks of memory by a technique known as bank switching. The Adam contains about 40K of ROM (read only memory) with SmartWriter and two operating systems, EOS and OS7 that can be switched into the upper or lower 32K of memory space. Expansion RAM (if you have it) and game cartridge ROM can also be switched in and out of the active memory space. Memory is normally all RAM in BASIC, the OS having been copied from ROM to RAM.

The bank switch is an OUT 7F,x command, where the lower nibble of x selects the following options:

Lower 32K	D1	D0	Upper 32K	D3	D2
SmartWriter, EOS	0	0	32K RAM	0	0
32K RAM	0	1	expansion ROM ?	0	1
expansion RAM	1	0	expansion RAM	1	0
OS7+24K RAM	1	1	cartridge ROM	1	1

For example, to select normal RAM for both the upper and lower blocks the number in binary is 0001, or 1. To select 32K of RAM on the bottom and cartridge ROM on the top, the number is 1101, or 13 (dec). A D1 and D0 of 0 will access either SmartWriter or the EOS ROM, depending upon another in/out command. Performing OUT 3F,2 before the OUT 7F,0 will access the EOS ROM. OUT 3F,0 causes OUT 7F,0 to access the SmartWriter ROM.

The following program moves code from ROM to RAM and then displays it on the screen as characters to help find the interesting parts. It is mainly an illustration of how to access the ROM's from BASIC. You must remember that the part of the program that accesses a ROM must be in the other 32K of memory space or else it will disappear from the Z80 when the ROM is switched in.

```
  1 REM   ROMVIEWER
  2 HIMEM :49999
  3 DATA   62,0,211,127,17,180,195,33,0,0,1,0,1,237,176,62,1,211,127,201
  4 REM  poke 'get page from rom' routine into mem.
  5 FOR x = 50000 TO 50019: READ d: POKE x, d: NEXT
  7 INPUT "page"; p: GOSUB 100
 10 FOR j = 0 TO 240 STEP 16
 15 PRINT "     ";
 20 FOR i = 0 TO 15
 30 x = 50100+i+j: t = PEEK(x)
 40 REM  check for return, backspace, etc.
 41 IF t = 12 OR t = 13 OR t = 16 OR t = 128 OR t = 10 THEN  t = 61
 42 IF t = 0 OR t = 7 OR t = 8 OR t = 9 THEN  t = 61
 43 IF t = 22 OR t = 24 OR t = 28 THEN  t = 61
 44 IF t > 159 AND t < 164 THEN  t = 61
 45 IF t = 148 OR t = 151 THEN  t = 61
 50 PRINT CHR$(t);
 60 NEXT i: REM  get next character
 70 PRINT
 80 NEXT j: REM  get next 16 characters
 90 GOTO 7: REM   get next page
 99 REM  get page p from rom
100 POKE 50009, p
110 CALL 50000
120 RETURN
```

Chapter 7. The Operating System.

The operating system, or EOS (E for elementary), is a collection of routines loaded from ROM to RAM from E000 to FC00. Much of EOS comes from OS7, the operating system for the ColecoVision game board, that is also available on the Adam. EOS also uses RAM from FC00 to FFFF for various tables as follows:

FC00-FC2F	EOS data tables
FC30-FD5C	EOS jump table
FD5D-FEBF	EOS RAM
FEC0-FEC3	Processor control block
FEC4-FFFF	Device control block

The jump table of routine addresses at FC30 was made so that the entry points of routines would always be the same, ie. the table, even if the routine was moved. The most useful routines for running the printer, tape, etc., are described in the relevant chapters. If you wish more information you can disassemble the operating system (it only takes 57 pages), and use the following list of titles and jump table addresses to identify and study the routines. In many cases there seem to be repeated routines, but there are minor differences, usually related to games. For example, there is a 'read one block' for the tape, a 'start read one block' and a 'end read one block'. The useful routine is just plain 'read one block', and the others are for use when you want to do something else while the block is being read. You can 'start X',do something else, and then 'end X' later. However, we do not recommend it.

EOS JUMP TABLE

FC30	EOS start initialization
FC33	console display
FC36	console initialization
FC39	display character on screen
FC3C	delay after hard reset
FC3F	end print buffer. (check if printer done)
FC42	end print character (same)
FC45	end read one block
FC48	end read character device (check DCB)
FC4B	end read keyboard
FC4E	end write one block
FC51	end write to character device
FC54	find DCB
FC57	get DCB address
FC5A	get PCB address
FC5D	hard initialization (cold start)
FC60	hard reset AdamNet
FC63	print buffer
FC66	print character
FC69	read one block
FC6C	read keyboard
FC6F	read keyboard, return code
FC72	read printer, return code
FC75	read return code

FC78 read tape return code
FC7B relocate PCB
FC7E request status
FC81 request keyboard status
FC84 request printer status
FC87 request tape status
FC8A scan AdamNet
FC8D soft initialization
FC90 soft reset device
FC93 soft reset keyboard
FC96 soft reset printer
FC99 soft reset tape
FC9C start print buffer
FC9F start print character
FCA2 start read one block
FCA5 start read character device
FCA8 start read keyboard
FCAB start write one block
FCAE start write character device
FCB1 synchronize Z80 and master 6801 clocks
FCB4 write one block
FCB7 write to character device
FCBA initialize file manager
FCBD initialize tape directory
FCC0 open file
FCC3 close file
FCC6 reset file
FCC9 make file
FCCC read directory for file
FCCF set file in directory
FCD2 read file
FCD5 write file
FCD8 set date
FCDB get date
FCDE rename file
FCE1 delete file
FCE4 read device dependent status
FCE7 goto word processor (called by block 0 on tapes)
FCEA read EOS
FCED trim file
FCF0 check FCB
FCF3 read block
FCF6 write block
FCF9 mode check
FCFC scan for file in directory
FCFF file query
FD02 position file
FD05 EOS 1
FD08 EOS 2

POINTERS

FBFF interrupt vector table
FC17 memory configuration table
FC27 memory switch port
FC28 AdamNet reset port
FC29 VDP control port
FC2A VDP data port
FC2B controllers
FC2D strobe set port
FC2E strobe reset port
FC2F sound port
FEC0 PCB

FD0B EOS 3
FD0E cv A
FD11 get in/out ports
FD14 bank switch memory
FD17 put ASCII to VDP
FD1A write VRAM
FD1D read VRAM
FD20 write VDP register
FD23 read VDP register
FD26 fill VRAM (one character)
FD29 init VRAM table
FD2C put VRAM
FD2F get VRAM
FD32 calculate offset
FD35 point to pattern position
FD38 load ASCII to VDP
FD3B write sprite attribute table
FD3E read game controllers
FD41 update spinner
FD44 decrement low nibble
FD47 decrement high nibble
FD4A high nibble to low nibble
FD4D add A and (HL) 16 bits
FD50 sound init
FD53 sound off
FD56 start song
FD59 sound
FD5C effect over

EOS RAM TABLE

FD60 revision #
FD61 VDP mode
FD63 VDP status byte
FD64 sprite attribute table
FD66 sprite generator table
FD68 pattern name table
FD6A pattern generator table
FD6C color table
FD6E cursor bank
FD6F current device
FD70 current PCB
FD72 device ID
FD73 file name address
FD75 keyboard buffer
FD76 print buffer
FD86 sectors to init
FD87 sector #
FD88 DCB image
FDAC query buffer
FDBA FCB buffer
FDD4 file count
FDD5 mod file count
FDD6 retry count
FDD7 file #
FDD8 file name cmps
FDD9 directory block #
FDDB found entry
FDDC volume block size
FDE0 year
FDE1 month
FDE2 day
FDE3 file mnger dir ent
FE01 fmum
FE02 bytes reg
FE04 bytes to go
FE06 user buffer
FE08 buffer start
FE0A buffer end
FE0C blocks reg
FE10 user name
FE12 start block
FE16 new hole start
FE1A new hole size
FE58 and down, EOS stack
FE58 spin switch 0
FE59 spin switch 1
FE5A personal debounce
FE6E and down, temp stack
FE6E pointer to list of sounds
FE70 ptr to sound 1 data block
FE72 ptr snd 2 db
FE74 ptr snd 3 db
FE76 ptr noise db
FE78 save ctrl sound
FE79 old chr
FE7A x min
FE7B x max
FE7C y min
FE7D y max
FE7E line buffer
FE9F # of lines
FEA0 # of columns
FEA1 upper left
FEA3 ptr to name table
FEA5 cursor
0147 clear RAM size

CHAPTER 8 BASIC

BASIC is loaded from tape to RAM as outlined in the memory map. To identify routines where different commands are carried out use the tables of words which point to routines in RAM. These routines can then be called directly from machine language programs, although in most cases it is easier to do everything in machine language yourself because the routines from BASIC require extensive setup, which we have not figured out yet.

The first table is on pages 1-3, beginning with GOSUB, GOTO, etc. Print out these pages of RAM with printmem and you will see the following pattern: number of word (token), address (2 bytes low, high), number of letters in word, word. For example, 02 AD 03 05 47 4F 53 55 42, means 2=token, 03AD=address, 5 letters, and GOSUB in ASCII. Token 1 has no letters and the same address as LET, which presumably means "ignore it". The address of GOSUB, 03AD, is to a table in page 3 after the word table which gives the number of routines (in this case 1), and the address (in this case 3D8C). In this way all the routine addresses can be obtained, except a group including STOP, NEW, etc. that have 03D0 which points to a 0, ie. no address. At the end of the word table there are some words and symbols which are used in conjunction with other words. These are given tokens only, with no addresses.

The next table of BASIC words is on page 3F (63), which also holds various pointers, the floating point accmulator (3F22-6), etc. This table of math functions is organized as: number of letters, word, 88 or A8, address.

A table of tape key words is on pages 4E and 4F. These words (OPEN, APPEND, READ, etc.) do not have tokens, and the address of each command is listed in order in the address table following the name table. Thus in our copy of BASIC, OPEN is at 4E03, APPEND at 4E0F, etc. If you experiment with these routines do not use a tape you care about.

BASIC programs are stored in RAM on page CF (207) by line number (2 bytes low, high), followed by an address in page D0, D1 or higher. At the address is the tokenized line, based on the tokens in the first BASIC table and others. Print out pages 207-209 with printmem and compare it with a listing of printmem. Add new lines which do not do anything and print pages 207-209 again to see how the new line is stored.

Numeric variables are stored in pages CF, CE, etc. just below the tokenized program. The first two letters of each variable are in a table in page 6B (107) which lists the address of the variable. If variables have more that two letters, the remaining letters are in page CF (207) or vacinity. String variables are also listed in the variable table on page 6B, and are stored on page 6C and following. All these tables are in different locations if HIMEM or LOMEM are used, but they still point to each other in the same way.

Input from tape is stored directly in a buffer in pages D4 (212) to D8 (216). This area contains the CATALOG of the last tape and the last program loaded, which appears exactly as it was typed in. The CATALOG lists the name of a file, the type, and 17 bytes described in the tape chapter.

One simple way to modify BASIC that can be fun to surprise people who know BASIC, is to change the key words in tables by poking new ASCII into RAM. It is easiest if the number of letters is not changed. After such changes BASIC will only respond to the new words.

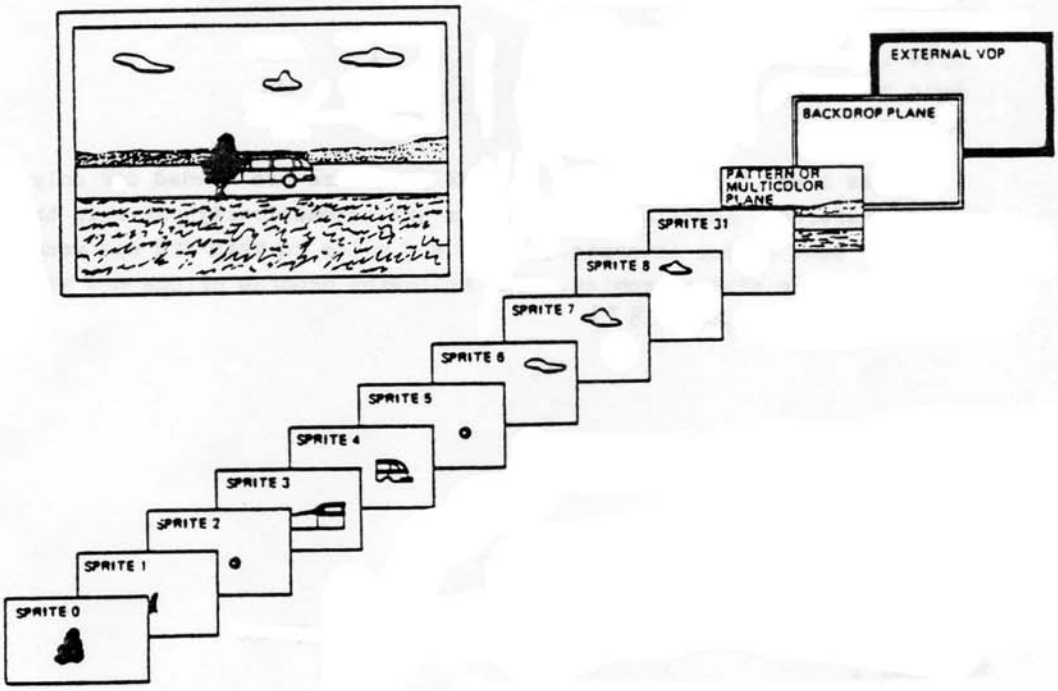
Two pokes that are useful are 17115 followed by "TEXT" which changes the screen (low nibble) and text (high nibble) colors, and 16149, 16150 which holds the limit on poke addresses.

Some memory locations in BASIC that control the format of the screen in text mode are as follows:

$4261	16993	line # from which scrolling starts.
$4262	16994	column # at which scrolling stops.
$4263	16995	top margin for scrolling.
$4264	16996	column # not scrolled.
$4265	16997	left margin.
$4269	17001	cursor line #.
$426A	17002	cursor column #.
$432E	17198	# of lines.
$432F	17199	# of columns.
$4331	17201	top margin.
$4332	17202	left margin.

CHAPTER 9 The Video Display Processor

The video signal to the TV is produced in the ADAM by the Texas Instruments video display processor (VDP), TMS9918A. It is very different from the Apple graphics in BASIC, and has modes, patterns, backgrounds, and sprites. We learned about this chip from an article in August, 1982 Byte by Steve Ciarcia and from a book sent free from Texas Instruments, Semiconductor Group, P.O.Box 1143, Houston TX, 77001. This book is hard to relate tothe ADAM, and has all examples in 9900 assembly language. The VDP is organized as multiple screens (or planes) in series, as shown below. The sprites are in the forground and can be used for moving or stationary objects. Sprites can be moved by simply changing their x and y coordinates in a table. They move cleanly without changing the colors of nearby objects, as occurs with Coleco's implementation of Apple graphics.



Behind the 32 sprites is a pattern plane which is a matrix of blocks, each 8x8 pixels that can be defined by the user. These pattern blocks are used to form the text in BASIC, but could also be used for landscapes etc. Behind the pattern plane is a backdrop plane which specifies the color of all pixels not set by the previous planes. Throughout, transparency is a possible "color". Finally, behind the background plane is the possibility, not implemented on the ADAM, of having the output of the VDP viewed on top of the output of another VDP chip.

The 9918A is a very complex chip which is connected to 16K of RAM,"VRAM", for its own use. It has four modes of operation which, together with the arrangement of tables in VRAM and a few other things, are specified by eight control registers which can be written to but not read. The chip is mapped in the Z80 in/out space at 160, 161 to 190,191 (decimal) even-odd pairs. We will

use 190 and 191. The control registers, a read-only register, and VRAM are accessed by the Z80 according to the following table.

Operation	Bits	CSW	CSR	Mode	in/out (dec)
write to register					
byte 1:data	D7----------D0	0	1	1	191
byte 2:reg.sel.	1 0 0 0 0R2R1R0	0	1	1	191
Write to VRAM					
byte 1:address	A7----------A0	0	1	1	191
byte 2:address	0 1 A13------A8	0	1	1	191
byte 3:data	D7----------D0	0	1	0	190
Read from register 8					
byte 1:data	D7----------D0	1	0	1	191
Read from VRAM					
byte 1:address	A7----------A0	0	1	1	191
byte 2:address	0 0 A13------A8	0	1	1	191
byte 3:data	D7----------D0	1	0	0	190

Bytes 1 and 2 of the write to VRAM procedure are needed for only the first byte transfered. Additional data bytes are automatically put into the next address. Some of our attempts to read and write to VRAM did not work, probably because of some timing problem. The following program prints out VRAM for you to analyze. (See Page 37)

CONTROL REGISTERS

Register 0
contains two option control bits.
bit 1, M3=1 specifies graphics mode 2
bit 0, EV=1 enables external input. Keep EV=0.

Register 1
contains seven option control bits.
bit 7, 4/16K RAM. Keep at 1 (16K).
bit 6, 0 blanks display. Keep at 1.
bit 5, interrupt enable. 1= enabled.
bit 4, M1=1 specifies text mode.
bit 3, M2=1 specifies multicolor mode.
bit 2 always =0.
bit 1, size. 0= 8x8 sprites, 1= 16x16 sprites.
bit 0, mag. 0= sprites x1, 1= sprites x2.

Register 2
The upper 4 bits are always 0. The number in the lower 4 bits (0 to 15) times $400 (1024) is the base address in VRAM of the pattern name table. Each byte in the name table corresponds to a region on the screen, and the number in the table specifies the pattern to be displayed there.

Register 3

This number (0 to 255) times $40 (64) is the base address in VRAM of the color table.

Register 4

This number (0 to 7) times $800 (2048) is the base address in VRAM of the pattern generator table.

Register 5

This number (0 to 127) times $80 (128) is the base address in VRAM of the sprite attribute table.

Register 6

This number (0 to 7) times $800 (2048) is the base address in VRAM of the sprite pattern generator table, where shapes of sprites are defined.

Register 7

The upper 4 bits (0 to 15)x16 specify the color of text in the text mode (not used by Coleco). The lower 4 bits (0 to 15) specify the background color in text mode and backdrop color in other modes.

Register 8

This is the status, read-only register. It contains three flags and a fifth sprite number and can be read during programs to check certain conditions. Reading the register clears all flags to 0.

bit 7, flag F. Interrupt flag, is set to 1 at the end of the last raster scan on the TV.

bit 6, fifth sprite flag (5S). Only four sprites are allowed on any given horizontal scan line. When a fifth sprite crosses a horizontal line this flag is set to 1 and the number of the sprite is placed in the lower 5 bits of the register.

bit 5, flag C. This coincidence or collision flag is set to 1 when two sprites collide. Collisions are checked only 60 times per second adn so may be missed.

COLOR CODES

The colors that are specified for sprites, backgrounds,etc. have the following codes.

0 transparent	8 medium red
1 black	9 light red
2 medium green	10 dark yellow
3 light green	11 light yellow
4 dark blue	12 dark green
5 light blue	13 magenta
6 dark red	14 gray
7 cyan	15 white

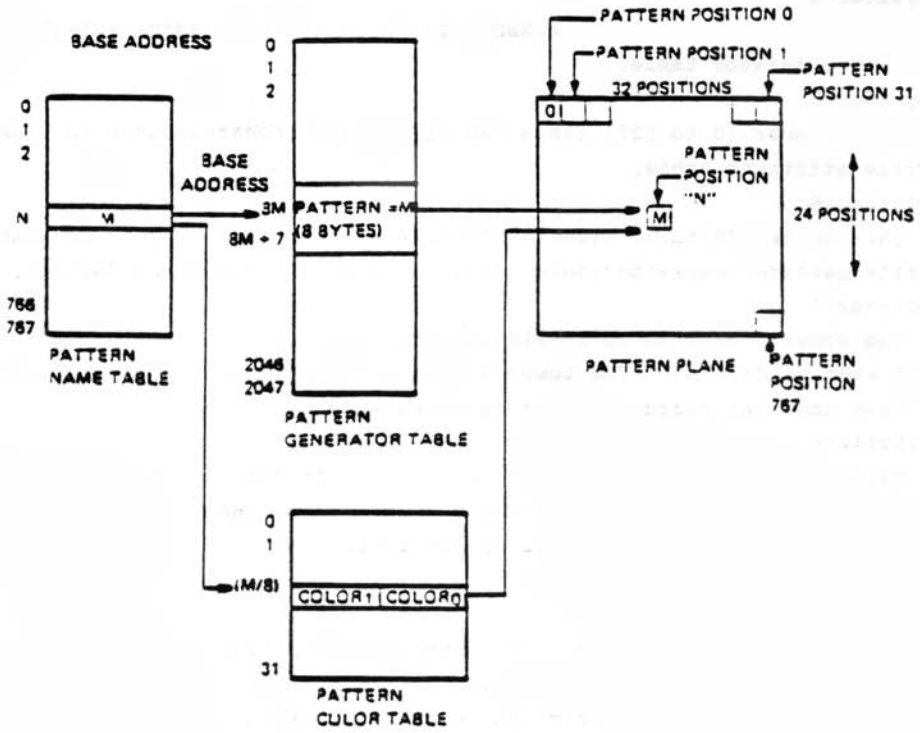
To change the color of text and background include "POKE 17115,x" followed by "TEXT" in your programs. The top nibble of x is the text color, and the low nibble is the background color.

MODES

Graphics mode 1. (M1,M2,M3=0)

This is the simplest graphics mode and, strangely, is used by BASIC to display text. The pattern plane is divided into 32 columns by 24 rows of blocks (768) each containing 8x8 pixels. Three tables in VRAM are used to create the pattern plane, as shown below.

Graphics mode 1.



The pattern name table is a 768 byte block of VRAM beginning on a 1K boundary pointed to by control register 2. Each byte corresponds to a region of the screen (ordered from left to right and top to bottom) and specifies the number of the pattern in the pattern generator table and the n/8th entry in the pattern color table to be displayed at that point. More that one pattern name table can be made, allowing rapid switching between pattern planes by simply changing the number in control register 2. The color table has only 32 numbers, and is pointed to by control register 3 times $40. Each number specifies the color of 1`s in the pattern by the top 4 bits and of 0`s by the bottom 4 bits. One number in the color table applies to 8 patterns in the pattern generator table, so patterns of the same colors should be grouped together.

The pattern generator table, pointed to by control register 4, consists of 8 bytes which form an 8x8 matrix of 1`s and 0`s as illustrated below.

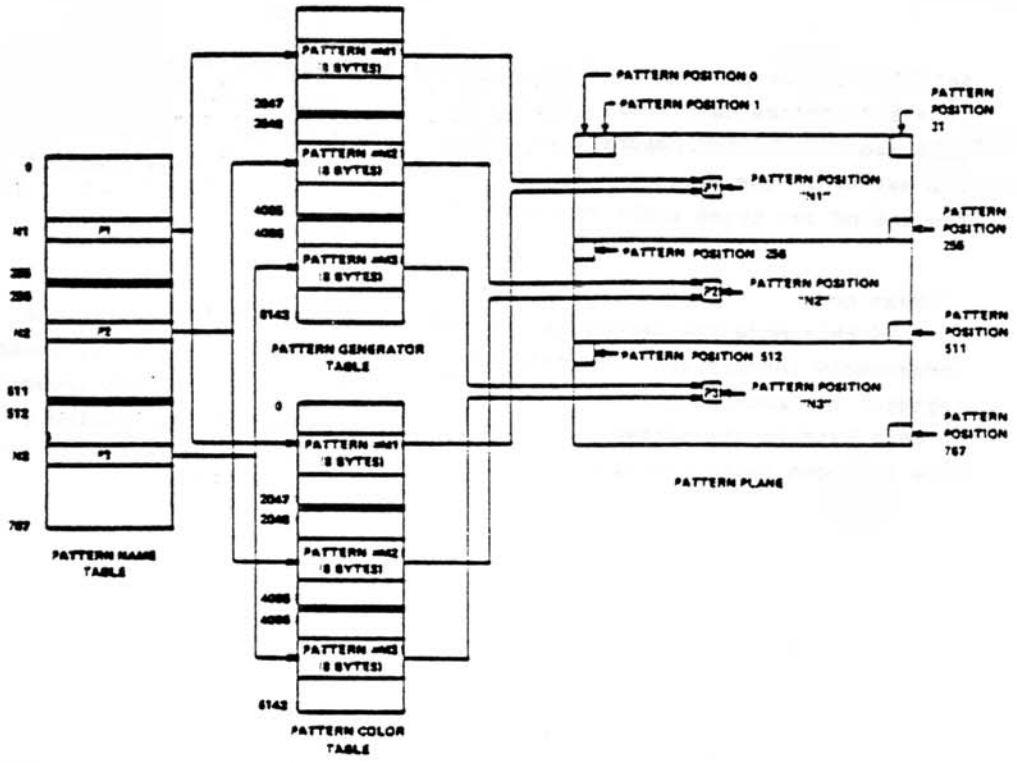
BYTE	BINARY	HEX
0	0 0 1 1 1 1 0 0	3C
1	0 1 1 1 1 1 1 0	7E
2	1 1 1 1 1 0 1 1	FB
3	1 1 1 1 1 1 1 1	FF
4	1 1 1 1 1 0 0 0	F8
5	1 1 1 1 1 1 0 0	FC
6	0 1 1 1 1 1 1 0	7E
7	0 0 1 1 1 1 0 0	3C

The same type of 8x8 matrix is used for sprites. As many as 256 patterns can be defined, taking 2048 bytes, but any smaller number can also be defined. An all-0 pattern should be included to point to for blank areas of the screen. Sprites can be used in all graphics modes, and the only limitation in mode 1 is that each 8x8 block in the pattern plane can have only two colors.

Graphics mode 2

Graphics mode 2 enhances the resolution over mode 1 by increasing the length of the pattern generator table from 2048 bytes to 6114 bytes (x3), and increasing the color table from 32 bytes to 6144 bytes. This allows every pixel to be set independantly and the color to be specified every 4 pixels (equal numbers of pattern and color bytes means 4 bits of color, or 1 color, for 4 bits of pattern, or 4 pixels). The pattern groups of 8 bytes are addressed by the name table as shown below.

Graphics mode 2

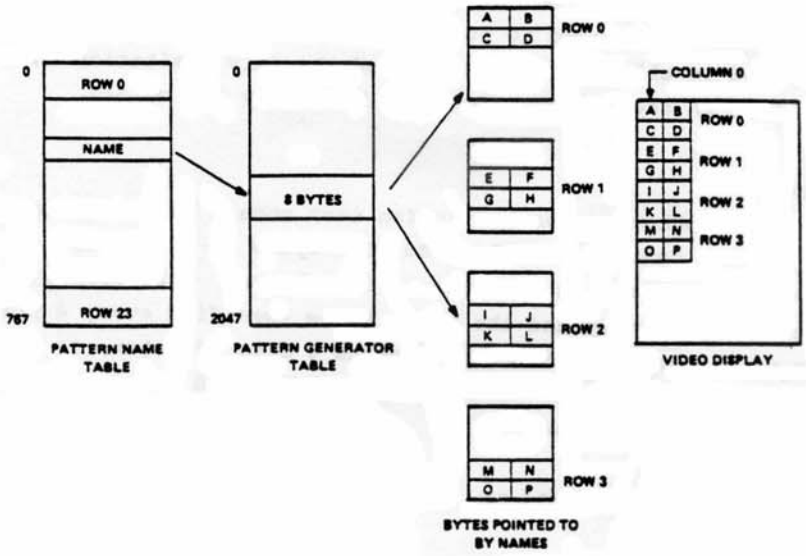


This mode is used for hires in BASIC but is awkward for such use because it was designed for backgrounds only. Sprites can be used in mode 2, and it would be ideal to combine sprite routines with BASIC hires. To do this it will be necessary to deduce VRAM allocation in BASIC hires.

Multicolor Mode

This mode is like lores graphics in BASIC, but gives a 64x48 block (of 4x4 pixels) display with any color allowed for any block. The blocks are specified as shown below.

Multicolor Mode

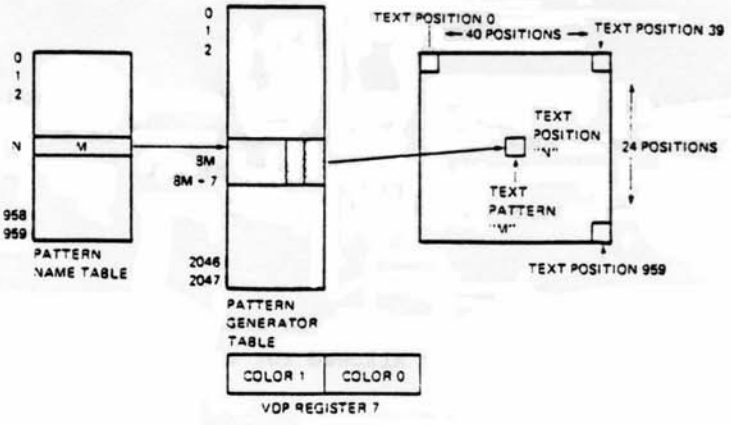


An entry in the pattern name table specifies 4 blocks, such as ABCD in row 0. If a byte in the name table which is in row 1 addresses the same pattern generator block, the colors will be EFGH, given by the third and forth bytes in the pattern. The first two bytes in a pattern apply to rows 0,4,8,12,16,20. The second two bytes apply to rows 1,5,9,13,17,21, etc.

Text Mode

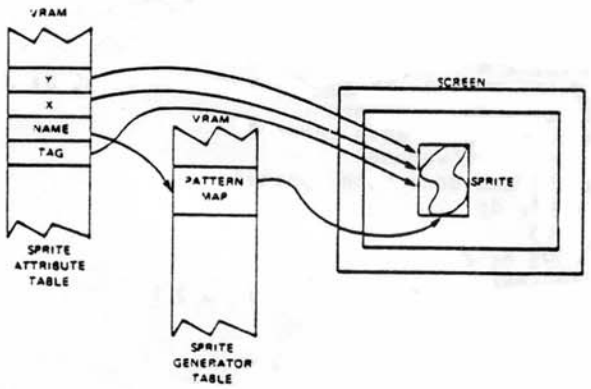
In this mode the screen is divided into a grid of 40x24 patterns (presumably letters and numbers), and the colors are specified by control register 7. Each pattern is 6 pixels across by 8 down, and the lowest two bits of each byte in the pattern generator table are ignored.The mapping in text mode is shown below. Sprites are not available in text mode.

Text Mode



SPRITES

Sprites are controlled by 4 bytes in the sprite attribute table, which specify the position ofthe sprite on an approximately 256x92 grid, point to the sprite generator table block, and specify the color of the sprite. The addressing mechanism is shown below.



In the sprite attribute table a sprite is defined by 4 bytes. The first byte is the vertical position, and the second byte is the horizontal position. The third byte is the sprite name which points to an 8 bit block in the sprite generator table. The forth byte has the sprite color in the lower 4 bits, 0's in bits 4,5, and 6, and something called the early clock bit in the top bit. When this bit is 1 the sprite is moved 32 pixels to the left, and it can probably be safely ignored. The sprite attribute table is ended by the number 208 decimal, so that the number of sprites showing can easily be changed from a maximum of 32 to less by inserting 208 in the vertical position byte of one sprite, blockiny display of it and all further sprites in the attribute table.

The size and resolution ofsprites is controlled by the size and mag bits in control register 1, as follows.

SIZE	MAG	Area	Resolution	Bytes/pattern
0	0	8x8	single pixel	8
1	0	16x16	single pixel	32
0	1	16x16	2x2 pixels	8
1	1	32x32	2x2 pixels	32

SPRITE EDITOR

The following program allows you to create a sprite file to be stored on tape as a binary file. You can make up to 32 sprites in the 8*8 or 16*16 format. It pokes the number of sprites into 51000, and the sprite length (8 or 32) into 51001. From 51002 to 51000 + # of sprites times sprite length, the sprites are stored. When you are done making the sprites the program asks whether you want to store them, and if so under what name.

```
]    2 REM     sprite-editor by B. Hinkle
     3 HIMEM :50999: ra = 51002
     5 TEXT: PRINT: PRINT: INPUT "How many sprites would you like to have (1-32)?
"; n: IF n < 1 OR n > 32 THEN 5
    10 PRINT: PRINT: PRINT: PRINT "Would you like to have:": PRINT
    12 PRINT "  1. 8x8 sprites": PRINT "  2. 16x16 sprites": PRINT: INPUT "(1,2)?
"; s
    20 IF s < 1 OR s > 2 THEN  TEXT: GOTO 10
    25 POKE 51000, n: POKE 51001, s^2*8: REM     # of sprites and length of sprite

    30 rb = s*8+11: bb = s*8+1: FOR d = 1 TO n
    50 GR: COLOR  = 10: x = 11: y = 1
    60 VLIN 0, bb AT 10: VLIN 0, bb AT rb: HLIN 10, rb AT 0: HLIN 10, rb AT bb
    65 REM     print commands on screen
    70 PRINT "     arrow keys to move cursor"
    80 PRINT "'a'-plot", "'d'-erase"
    90 PRINT "'return' when done with sprite"
    95 PRINT "sprite #"; d;
    99 REM     main loop
   100 COLOR  = 6: PLOT x, y
   110 GET a$: p = ASC(a$)
   120 IF e = 1 THEN  COLOR  = 8: PLOT x, y: GOTO 140
   130 COLOR  = 0: PLOT x, y
   135 REM     check for special commands
   140 IF p = 97 THEN  COLOR  = 8: PLOT x, y
   150 IF p = 100 THEN  COLOR  = 0: PLOT x, y: e = 0
   155 IF p = 13 THEN 200
   157 REM     check for arrow keys
   160 IF p = 163 AND x-1 > 10 THEN  x = x-1: e = 0
   165 IF p = 161 AND x+1 < rb THEN  x = x+1: e = 0
   167 IF p = 160 AND y-1 > 0 THEN  y = y-1: e = 0
   170 IF p = 162 AND y+1 < bb THEN  y = y+1: e = 0
   180 IF SCRN(x, y) = 8 THEN  e = 1
   190 GOTO 100: REM     go back to main loop
   199 REM     poke sprite into memory
   200 IF s = 2 THEN 280
   205 REM     8*8 sprite poking
   210 aa = 8: ab = 1: ac = 18: ad = 11: GOSUB 230
   220 NEXT d: GOTO 500
   229 REM     poke an 8*8 block into 51000+
   230 FOR y = ab TO aa: i = 0
   240 FOR x = ac TO ad STEP -1
   250 IF SCRN(x, y) = 8 THEN  i = i+2^(ac-x)
   260 NEXT x: POKE ra, i: ra = ra+1: NEXT y
   270 RETURN
   279 REM     16*16 sprite poking
   280 aa = 8: ab = 1: ac = 18: ad = 11: GOSUB 230
   290 aa = 16: ab = 9: ac = 18: ad = 11: GOSUB 230
   300 aa = 8: ab = 1: ac = 26: ad = 19: GOSUB 230
   310 aa = 16: ab = 9: ac = 26: ad = 19: GOSUB 230
   320 NEXT d
   499 REM     save sprites on tape
   500 TEXT: PRINT: PRINT: INPUT "Would you like to save the     sprites (y/n)?";
 a$
   510 IF a$ <> "y" AND a$ <> "n" THEN 500
   520 IF a$ = "n" THEN  PRINT "End of program": END
   530 INPUT "Type in the name for the file:"; a$: ra = ra-51000
   540 PRINT CHR$(4); "bsave "; a$; ",a51000,l"; ra
   550 PRINT "done"
```

The second program is an example of how to use a sprite file made by the sprite editor. It asks the name of the file you want and loads it into RAM at 51000. It then sets up the sprite generator and attribute tables in VRAM, peeks the generator pointer at 64870 and the attribute pointer at 64868 to set up VRAM addresses to be changed, and loads the VRAM generator table with the sprite table in RAM at 51002. It then moves the sprites by sending X and Y to VRAM. Analogous programs can be written to do various things with a sprite table.

```
    3 REM  demo of sprite-editor program by B. Hinkle
    5 HIMEM :49999: DIM y(32), x(32), yd(32), xd(32): HGR
   10 DATA    62,0,211,191,201,62,0,211,190,201
   20 FOR x = 50000 TO 50009: READ d: POKE x, d: NEXT
   30 INPUT "name of sprite file to be used?"; a$
   40 PRINT CHR$(4); "bload "; a$
   45 a = PEEK(64870): GOSUB 1000: a = PEEK(64871)+64: GOSUB 1000
   50 FOR x = 0 TO PEEK(51000)*PEEK(51001): b = PEEK(51002+x): GOSUB 1100: NEXT
x: REM    -load gen. table
   60 x = RND(-PEEK(17011)): FOR i = 1 TO PEEK(51000)
   65 REM  set up random directions for each sprite
   70 yd(i) = RND(1)*10: xd(i) = RND(1)*10: x(i) = 150: y(i) = 100+i
   80 NEXT i
   85 a = PEEK(64868): GOSUB 1000: a = PEEK(64869)+64: GOSUB 1000: REM  set up a
ddr. for att.table
   89 REM  move sprites around
   90 FOR i = 1 TO PEEK(51000)
  100 IF y(i) < 30 OR y(i) > 150 THEN  yd(i) = -yd(i)
  110 IF x(i) < 100 OR x(i) > 240 THEN  xd(i) = -xd(i)
  120 x(i) = x(i)+xd(i): y(i) = y(i)+yd(i)
  140 b = y(i): GOSUB 1100: b = x(i): GOSUB 1100: b = i-1: GOSUB 1100
  150 b = 15: GOSUB 1100: NEXT i: b = 208: GOSUB 1100: GOTO 85
  999 REM  'out' to 191
 1000 POKE 50001, a
 1010 CALL 50000: RETURN
 1099 REM  'out' to 190
 1100 POKE 50006, b
 1110 CALL 50005: RETURN



]
    3 REM    SCRN for hgr/hgr2 by B. Hinkle
   10 DATA    17,0,0,213,6,3,203,58,203,59,16,250,205,50,253,253,33,1,0,33,240,0
,62,3
   20 DATA     205,47,253,33,240,0,209,122,230,7,133,111,123,230,7
   30 DATA    71,62,8,144,71,175,55,143,16,253,166,33,240,0,40,3,54,1,201,54,0,2
01
   40 FOR x = 1 TO 61: READ d: POKE x+172, d: NEXT
   50 HGR: HCOLOR  = 12: HPLOT 10, 10 TO 200, 10
   55 HPLOT 100, 100
   57 HPLOT 0, 0 TO 255, 191
   60 INPUT "x:"; x: POKE 174, x
   70 INPUT "y:"; y: POKE 175, y
   80 CALL 173: p = PEEK(240)
   90 PRINT p: GOTO 60


 1 REM  PRINTVRAM by Ben Hinkle
 4 PR #1
 5 x$ = "0123456789ABCDEF"
10 DATA   62,0,211,191,62,00,211,191,0,0,0,0,219,190,50,32,203,201
20 FOR x = 51400 TO 51417
30 READ d: POKE x, d: NEXT x
32 INPUT "page?"; p: POKE 51405, p: PRINT p
35 FOR e = 0 TO 15
40 FOR s = 0 TO 15
50 POKE 51401, e*16+s
60 CALL 51400
65 g = PEEK(52000)
67 PRINT MID$(x$, INT(g/16)+1, 1); MID$(x$, (g/16-INT(g/16))*16+1, 1); " ";
70 NEXT s
80 PRINT: NEXT e
```

The previous two short programs, SCRN and printvram, are further examples of using direct access to the video RAM. The SCRN routine to line 50 could be included in your own program and called if you want to know whether a bit is set on the hires screen.

The following program is a simple sprite demo that illustrates making a sprite and moving it on the HGR2 screen.

```
    5 REM       SPRITE DEMO
    6 HGR2
   10 HIMEM :51399
   19 REM       load machine language code
   20 DATA       62,0,211,191,201,62,00,211,190,201
   30 FOR x = 51400 TO 51409: READ p: POKE x, p: NEXT
   34 REM
   35 REM    background
   36 FOR s = 1 TO 25
   37 HCOLOR  = 15*RND(9)
   38 HPLOT 100+3*s, 0 TO 10*s, 191
   39 NEXT
   40 REM
   50 REM       load sprite generator
   55 a = 0: GOSUB 1000: a = 120: GOSUB 1000
   60 DATA   60,126,195,219,219,195,126,60
   70 FOR x = 1 TO 8
   80 READ d: GOSUB 1100
   90 NEXT
  100 REM       load sprite attribute
  110 a = 128: GOSUB 1000: a = 127: GOSUB 1000
  120 d = 70: GOSUB 1100: GOSUB 1100: d = 0: GOSUB 1100: d = 7: GOSUB 1100: d =
208: GOSUB 1100
  199 REM
  200 REM       load control registers
  230 a = 127: GOSUB 1000
  240 a = 133: GOSUB 1000
  250 a = 7: GOSUB 1000
  260 a = 134: GOSUB 1000
  299 REM
  300 REM       MOVE IT
  310 t = t+.05
  320 x = 60*SIN(t)+70
  330 y = 60*COS(t)+70
  340 a = 128: GOSUB 1000
  350 a = 127: GOSUB 1000
  360 d = INT(x): GOSUB 1100
  370 d = INT(y): GOSUB 1100
  380 GOTO 310
  999 REM
 1000 POKE 51401, a
 1010 CALL 51400
 1020 RETURN
 1100 POKE 51406, d
 1110 CALL 51405
 1120 RETURN
```

BASIC GRAPHICS

The graphics modes in BASIC use the VDP in unusual ways to try to copy Apple II graphics. This section describes the BASIC graphics modes and how to work with or around them.

BASIC TEXT mode.

VRAM is set up as follows:

Pages	Table
0-7	pattern gen.(characters)
8-10	name 1
24-26	name 2
32	color

In BASIC text mode the VDP is in graphics mode I and the ASCII code is in the name table which points to the character set in the pattern gen table. Actually there are two name tables which alternate with the frequency of the blinking cursor. To make a character appear steadily the ASCII must be in both name tables at the same location.

The easiest changes to make are text and background colors, as described on page 31. The most interesting change to make is to create your own character set. The characters are made on an 8x8 grid, but use only 5x7 to make spaces between the letters. The following program is a font editor to make new character sets. It loads the Adam set from ROM to Z80 RAM starting at 28,020 to use as a base from which to work. It then asks which character you would like to change and lets you plot it in an 8x8 grid. When you are done it puts the new character in RAM and then in VRAM using a modified OS routine. You can then save the new fonts and a short machine language program on tape as a binary file. To use the file in the immediate mode or from a program type "bload (name)", "call 28000" and "text", and the new fonts will be installed.

```
 2 REM     character editer by Ben Hinkle
 3 LOMEM :29038: HIMEM :49996: GOSUB 1010
 7 DATA   62,0,205,20,253,33,4,3,1,0,4,17
 8 DATA   100,195,237,176,62,1,205,20,253,201
10 DATA   175,50,112,225,50,113,225,50,114
13 DATA   225,50,127,225,50,128,225,50,129,225
15 DATA   50,117,225,62,17,50,118,225,62
17 DATA   116,50,119,225,62,109,50,120,225,201
18 REM   routine to modify OS
20 FOR x = 28000 TO 28037: READ d: POKE x, d
30 NEXT: CALL 28000
40 REM  main loop
50 GR: COLOR  = 10: x = 11: y = 1
53 PRINT "'q'-quit 'd'-display 's'-save"
54 PRINT "'r'-reset set", "'l'-load"
55 INPUT "Edit character # (32-126)?"; d$
56 d = VAL(d$): IF d = 0 THEN 600: REM  special command
57 IF d < 32 OR d > 126 THEN 50
58 ra = 28020+d*8: REM  addr. in table
60 VLIN 0, 9 AT 10: VLIN 0, 9 AT 19
61 HLIN 10, 19 AT 0: HLIN 10, 19 AT 9
70 PRINT "cur.keys-move", "'s'-save set"
80 PRINT "'a'-plot", "'e'-erase"
90 PRINT "'return'-done", "'q'-quit"
95 PRINT "character #"; d; " looks like:"; CHR$(d);
99 REM  edit character
100 COLOR  = 6: PLOT x, y: GET a$: p = ASC(a$)
120 IF e = 1 THEN  COLOR  = 8: PLOT x, y: GOTO 140
130 COLOR  = 0: PLOT x, y
```

```
135 REM  check for plot, done, etc.
140 IF p = 97 THEN  COLOR  = 8: PLOT x, y
145 IF p = 113  THEN 50
147 IF p = 115  THEN 500
150 IF p = 101  THEN  COLOR  = 0: PLOT x, y: e = 0
155 IF p = 13 THEN 230
157 REM  check for arrow keys
160 IF p = 163 AND x-1 > 10 THEN  x = x-1: e = 0
165 IF p = 161 AND x+1 < 19 THEN  x = x+1: e = 0
167 IF p = 160 AND y-1 > 0 THEN  y = y-1: e = 0
170 IF p = 162 AND y+1 < 9 THEN  y = y+1: e = 0
180 IF SCRN(x, y) = 8 THEN  e = 1
190 GOTO 100
220 REM  poke new character into table
230 FOR y = 1 TO 8: i = 0
240 FOR x = 18 TO 11 STEP -1
250 IF SCRN(x, y) = 8 THEN  i = i+2^(18-x)
260 NEXT x: POKE ra, i: ra = ra+1: NEXT y
270 GOTO 50
499 REM  save character set
500 TEXT: INPUT "file name ?"; a$
540 PRINT CHR$(4); "bsave "; a$; ",a28000,l1036"
550 PRINT a$; " has been saved": END
560 REM  load character set
570 HOME: INPUT "file name?"; a$
580 PRINT CHR$(4); "bload "; a$
590 GOTO 50
599 REM  special commands
600 IF d$ = "s" THEN 500
610 IF d$ = "q" THEN  TEXT: END
620 IF d$ = "d" THEN 640
625 IF d$ = "r" THEN  RESTORE: GOSUB 1010: GOTO 20
627 IF d$ = "l" THEN 570
630 GOTO 50
635 REM  display set
640 TEXT: PRINT "  "; : FOR x = 0 TO 9: PRINT x;
642 NEXT: PRINT
650 FOR x = 3 TO 9: HTAB 2: PRINT x: NEXT
660 FOR x = 10 TO 12: PRINT x: NEXT
670 VTAB 2: HTAB 5
680 VTAB 2: HTAB 5: FOR x = 32 TO 126
690 IF INT(x/10) = x/10 THEN  PRINT: HTAB 3
695 PRINT CHR$(x); : NEXT: PRINT
710 VTAB 22: PRINT " Hit any key to cont": GET a$
720 GOTO 50
1000 REM  reset table from ROM characters
1010 FOR x = 49997 TO 50018: READ d: POKE x, d: NEXT
1020 CALL 49997
1030 FOR x = 0 TO 1036
1040 POKE 28020+x, PEEK(50020+x)
1050 NEXT: RETURN
```

BASIC GR mode

The VRAM map is as follows:

Pages	Table
0-23	color
24-26	name
32-52	pattern
53-55	pattern (characters)

In the GR mode the VDP is in the graphics 2 mode. The name table is always 1,2,3,4, etc., except at the end where it is ASCII as in text mode. Similarly, the pattern table is always FC,F0,C0,FC,F0, etc., to make 6x4 pixel blocks, except at the end. The only table that is changed is the color table, which creates the lores graphics blocks.

BASIC HGR mode

The VRAM map is the same as in the GR mode. However, this time it is the pattern generator table that is changed, allowing each bit to be set. For a blank screen the pattern table is all zeros and the color table all $11 (black). As graphics are drawn the appropriate pattern and color blocks are set.

BASIC HGR2

In this mode the memory map and the mechanism of implementing graphics are the same as in HGR except that the character set is omitted and hires graphics cover the whole screen.

In any of the modes described sprites can be added in regions of VRAM not used by BASIC, or direct changes can be made in the VRAM used by BASIC.

OPERATING SYSTEM ROUTINES

The VRAM tables can be set up most easily using OS routines. The routines are described below.

FD1A Block write to VRAM. HL= address in RAM to be moved. DE= address in VRAM. BC= number of bytes to be moved.

FD1D Block read from VRAM. DE= address in RAM. HL= address in VRAM. BC= number of bytes to be moved.

FD20 Write to VDP registers. C= byte to be sent. B= register to be written.

FD23 Read register 8. The result is at FD63.

FD26 A to VRAM. HL= address in VRAM. A= byte put in VRAM. DE= number of times A is repeated.

FD29 Write to VDP registers. HL= address in VRAM to be written. A= register written to.

FD2C Write table to VRAM. HL= table address in RAM. DE= entry # in table, A= table number (0=sprite att., 1=sprite gen., 2=name, 3=pattern, 4=color). IY= number of entries to be moved.

FD2F Read table from VRAM. HL= address in Ram to be written to. The rest is the same as FD2C.

FD32 Calculate VRAM offset. D= pattern position y. E= pattern position x. Returns with DE= y*32+x.

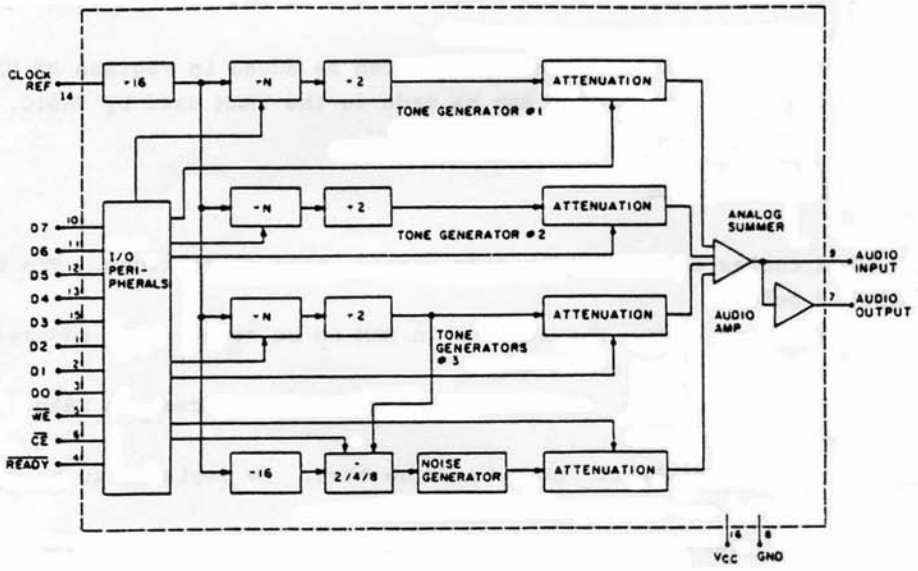
FD35 Calculate pattern position. DE= signed x or y (16 bit numbers). DE is divided by 8 and rounded from -128 to +127.

FD38 Reset character set. Loads character set from ROM to VRAM .

FD3B Writes sprites to attribute table in VRAM. A= number of sprites to be moved. DE= address of sprite attribute table in RAM. HL= address of sprite order order table in RAM. The sprite order table is a list of sprite numbers that specifies the order they will be put in the sprite table. The sprite attribute table in RAM is a duplicate of that in VRAM. The order table is all that requires changing to avoid "fifth sprite flickers".

CHAPTER 10. Sound

The sound chip on the Colecovision (top) board is the Texas Instruments SN76489A. We learned about this chip from articles in the December, 1980 Kilobaud Microcomputing by Steve Marum and in the July, 1982 Byte by Steve Ciarcia. It has three square wave tone generators and a noise generator, not nearly as sophisticated as the Commodore SID chip, but definately fun to play with. A block diagram of the chip is shown below.



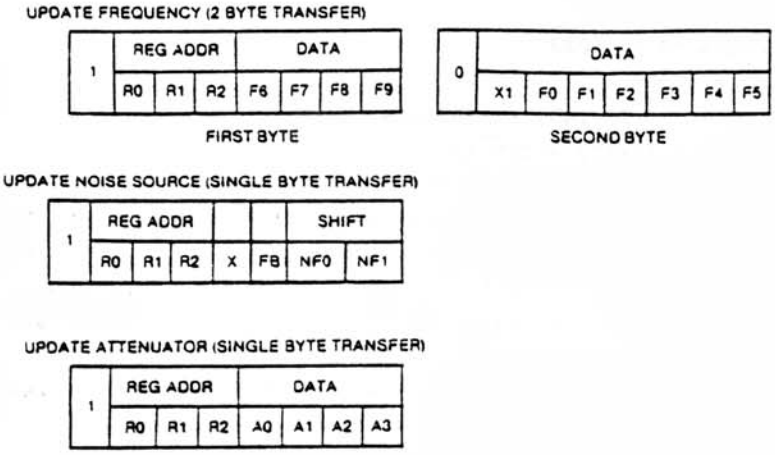
The SN76489 sound chip

Texas Instruments uses an odd convention for describing the order of bits in a byte and calls the most significant bit (MSB) 0, or D0 for the data bus. instead of 7, or D7.

The pin numbers of the SN76489A are also shown in the figure. The chip is addressed via the WE (write enable), CE (chip enable) and ready inputs. It is mapped in the IN/OUT address space of the Z80 at F0 (actually the lower 5 bits are not decoded so any number between E0 and FF, or 224 and 255 in decimal, will access the chip using "OUT" instructions in machine language). There is only one port to address and the various functions are accessed by the numbers given to the port. These 8 bit numbers are divided up, as shown below, to give a 10 bit frequency value (divided between two bytes of input), a 3 bit control register which specifies eight functions, a 4 bit attenuator value which controls the volume, a noise type bit and a 2 bit noise clock value.

When the MSB is 1 the next three bits are the control register that specifies the meaning of the lower 4 bits. When the MSB is 0 the lower 6 bits are the most significant bits of the 10 bit frequency value for the most recently specified tone generator. The frequency of the square wave produced is the clock frequency divided by 32 times the 10 bit number specified as the frequency value.

Types of data bytes sent to the SN76489.



The control register, specified by R0, R1, and R2 indicates the following functions:

0 tone 1 frequency
1 tone 1 volume
2 tone 2 frequency
3 tone 2 volume
4 tone 3 frequency
5 tone 3 volume
6 noise type
7 noise volume

The noise generator can be controlled to produce different types of noise at different volumes. The types are white (hiss) and perodic (motors). The frequency generating both noise types has 4 values specified by the 2 bit number formed by NF1 and NF0, or can be driven by voice 3, allowing continuously variable noise frequencies of phaser type sounds.

In practice it is likely that you will program the SN76489A in BASIC via a short machine language subroutine, and so the numbers you will use will be decimal. The table below shows the numbers used to control the chip in decimal.

Sound control numbers in decimal.

	Pitch		Volume	
	first byte	second byte	high	off
voice 1	128-143	0-63		144-159
voice 2	160-175	0-63		176-191
voice 3	192-207	0-63		208-223

noise	224-227	perodic	(227=voice 3)
	228-231	white	(231=voice 3)
	240-255	volume	(255=off)

Pitch control

I=frequency value =0 to 1023
note frequency = clock /32*I
for voice 1: byte 1 (128-143) = 128*I-INT(I/16)*16
byte 2 (0-63) = INT(I/16)

For voice 2 or 3 start with 160 or 192 for the first byte, instead of 128. For a chromatic scale use I=120,127,134,142,150,159,169,179,190,201,213,225,240 and multiples of these numbers. This scale was generated by dividing an octave (factor of two in frequency) into twelve notes spaced equally on a logrithmic scale. The frequency of the next note (half step) is the frequency of the current note times the twelth root of two.

To pass numbers to the SN76489 from BASIC a short machine language subroutine is needed. A simple example is:

```
LD  A,n
LD  C,F0
OUT(C), A
RET
```

This code can be poked into RAM as illustrated in the following programs. The first can be used to experiment with the chip, and the second is an interesting random music generator.

```
 5 REM   SOUNDTEST
 6 REM
10 HIMEM :53000
14 REM     poke in machine code
15 DATA         62,0,14,245,237,121,201
20 FOR x = 1 TO 7
30 READ d: POKE 53000+x, d
40 NEXT
100 INPUT "number (0-255)"; n
110 POKE 53002, n
120 CALL 53001
130 GOTO 100
```

```
 5 REM     RNDMUSIC
 6 REM
10 HIMEM :53000
14 REM     poke in machine code
15 DATA         62,0,14,245,237,121,201
20 FOR x = 1 TO 7
30 READ d: POKE 53000+x, d
40 NEXT
50 INPUT "which piece would you like?"; p
60 dm = RND(-p): dm = RND(9)
180 FOR m = 1 TO 2
190 FOR t = 300 TO 30 STEP -2
199 REM    think of note
200 v = RND(9)*220
205 REM   play note
210 POKE 53002, v
220 CALL 53001
230 REM   delay
240 FOR w = 1 TO t: NEXT
250 NEXT t: NEXT m
300 POKE 53002, 144: CALL 53001
310 POKE 53002, 176: CALL 53001
320 POKE 53002, 208: CALL 53001
340 FOR d = 1 TO 4000: NEXT
350 CALL 58321
```

The OS has routines, originally from the OS7, that act as a music editor. They are complicated, but create envelopes and frequency sweeps that are hard to make otherwise. The music system is a series of tables, described later, which define the timing and notes. The main table is the note table which holds instructions for each note. The swept notes can be used as an envelope with an initial volume period followed by a linear decrease, or as sound effects. The note table instructions have the following formats:

rest

```
  7  6  5  4  3  2  1  0
:CH# : 1:  length      :
```

simple note

```
    7  6  5  4  3  2  1  0
0, :CH# : 0  0  0  0  0  0
1,  F2-----------------F9
2, : volume   : 0  0 F0 F1
3, :      length          :
```

frequency swept note

```
   7  6  5  4  3  2  1  0
0,:CH# : 0  0  0  0  0  1
1, F2-----------------F9
2,:volume    : 0  0 F0 F1
3,   # of steps in sweep
4, step length: 1st step
5,     step size
```

volume swept note

```
    7  6  5  4  3  2  1  0
0, :CH# : 0  0  0  0  1  0
1, F2-----------------F9
2, :init vol  : 0  0 F0 F1
3,     length of note
4,   step size:   # of steps
5, step length: init length
```

volume and frequency swept note

```
   7  6  5  4  3  2  1  0
0,:CH# : 0  0  0  0  1  1
1, F2-----------------F9
2,:  volume  : 0  0 F0 F1
3,   # of steps in sweep
4 freq step len: 1st step len
5,   freq step size
6,:vol step   : vol step #
7,vol step len: vol init len
```

noise "note"

```
    7  6  5  4  3  2  1  0
0,  0  0  0  0  0  0  1  0
1,   volume   : 0  t F0 F1
2,      length
3,vol step size: vol step #
4,  step len   : 1st step len
```

special effect note

```
   7  6  5  4  3  2  1  0
0, CH# : 0  0  0  1  0  0
1, address  lo
2, address  hi
```

end or repeat song

```
    7  6  5  4  3  2  1  0
0,  CH# : 0  1  R  0  0  0
```

In each table the numbers form 7 to 0 at the top are the bits in each byte, and the numbers at the left are the byte numbers in the block. The numbers for frequency bits (eg. F5) are TI reverse nomenclature. Otherwise the four bit entries such as "step size" or "volume" can range from 1 to 15. The note table is pointed to by other tables as follows:

Table Table	Song Table	Note Table	Output Table
FE6E: song table addr.	song 1 addr.	song 1, note 1	song 1, current
FE70: output voice 1 addr.	output 1 addr.	note 2	note, (10 bytes)
FE72: output voice 2 addr.	song 2 addr.	note 3	
FE74: output voice 3 addr.	output 2 addr.	etc.	song 2, same
FE76: output noise addr.	song 3 etc.	song 2, note 1	

The output table is necessary because the sound chip makes a click each time it is written to and the sound routine checks the output table and writes to the chip only if there has been a change. The output table has the following format:

Output Table

7 6 5 4 3 2 1 0

0,CH#: song #
1, addr. next note
2, " high byte
3,F2----------F9
4,vol.:vol0 0 F0F1
5, length
6,step length:1st step
7, step size
8, vol step:vol step #
9, vol stp len: vol init len

These tables point to each other, with the only fixed address being FE6E. The OS routines that run the system are:

FD50 Init sound tables. HL= address of song table, B= number of output tables to be used.

FD53 Sound off. No setup required.

FD56 Start song. B= song # to be started.

FD59 Sounds. No setup. Call repeatedly. Sends output table to sound chip and updates output. Can be called repeatedly from BASIC or outomatically by the interrupt at $66 that is usually used for FLASH. Put CALL FD59, RETN at $66 in zero page, and replace it with RETN to stop.

The following program illustrates how to use the music routines and tables to make a three voice music editor. Lines 4-7 fill an array with a scale. Line 10 clears the poke limit. Lines 20-70 set up the song, output, and table tables. Lines 75-140 input data for each note and lines 500-540 poke it into the note table. lines 1000-1020 call "start song", and lines 1025-1040 call "sounds" to play the notes. Notes are entered as the letters a through g, with upper case A through G for sharps. We are still quite inexperienced in the use of these routines, and you should try modifying this program to include swept notes and envelopes.

```
  3 REM          sound program by B. Hinkle
  4 DATA     213,0,127,142,0,169,190,201,225,120,134,150,159,179
  5 HIMEM :37499: DIM p(14): x(1) = 40000: x(2) = 40800: x(3) = 41600
  6 REM   read scale into an array
  7 FOR x = 1 TO 14: READ d: p(x) = d: NEXT x
 10 POKE 16149, 255: POKE 16150, 255
 20 DATA  64,156,76,154,96,159,86,154,128,162,96,154: REM  song table
 30 FOR x = 39000 TO 39011: READ d: POKE x, d: NEXT
 40 DATA  33,88,152,6,4,205,80,253,201: REM  init sound tables
 50 FOR x = 37550 TO 37558: READ d: POKE x, d: NEXT: CALL 37550
 60 DATA     88,152,76,154,86,154,96,154,106,154
 65 REM  Table table data
 70 FOR x = 65134 TO 65143: READ d: POKE x, d: NEXT
 73 REM   main loop
 75 HOME: PRINT: PRINT
 80 INPUT "pitch (q to quit,p to play)?"; p$
 83 IF p$ = "q" OR p$ = "p" THEN 150
 90 INPUT "octave (1-8)?"; oc
 95 IF oc < 1 OR oc > 8 THEN 90
100 PRINT: INPUT "voice #(1-3)?"; v: IF v < 1 OR v > 3 THEN 100
110 PRINT: INPUT "# of beats (1=1/4 note)?"; n
120 IF n < .032 OR n > 4 THEN  PRINT " length must be .032 to 4 (.032=1/32 not
e, 4=whole note)": GOTO 110
140 GOTO 500
145 REM   quit or play song
150 IF p$ = "q" THEN  END
160 POKE x(1), 16: POKE x(2), 80: POKE x(3), 149: GOTO 1000
499 REM   poke note into memory
500 IF ASC(p$) < 71 THEN  p = p(ASC(p$)-64)*oc
510 IF ASC(p$) > 96 THEN  p = p(ASC(p$)-89)*oc
520 POKE x(v)+1, p-INT(p/256)*256
530 POKE x(v)+2, INT(p/256): POKE x(v)+3, n*64-1
540 POKE x(v), v*64: x(v) = x(v)+4: GOTO 75
999 REM   play song with all 3 voices
1000 DATA  6,1,205,86,253,201: REM  calls start song
1005 RESTORE: FOR t = 1 TO 45: READ r: NEXT
1010 FOR x = 37570 TO 37575: READ d: POKE x, d: NEXT: CALL 37570
1020 POKE 37571, 2: CALL 37570: POKE 37571, 3: CALL 37570
1025 do = PEEK(64885)
1027 PRINT: PRINT "hit any key to return"
1030 CALL 64857: REM    call sounds
1035 dn = PEEK(64885): IF dn <> do THEN 75
1040 GOTO 1030
```

Chapter 11, The Game Controllers

The game controllers are read in the Z80 in/out space at FC,FE (cont. 1) and FD,FF (cont. 2), which overlaps with the sound generator but uses only IN commands instead of OUT commands. The numbers returned by IN commands in machine language are shown below in decimal. When nothing is pressed, zero is returned. Numbers are in decimal.

IN FD(#1) or FF(#2)			IN FC(#1) or FE(#2)		
Joystick	N	1	Keypad	0	5
	NE	3		1	2
	E	2		2	8
	SE	6		3	3
	S	4		4	13
	SW	12		5	12
	W	8		6	1
	NW	9		7	10
	fire	64		8	14
				9	4
				#	9
				*	6
				arm	64

Operating system routines that handle the controllers are read-controllers at FD3E and read spinner at FD41. Read-controllers reads and debounces the controllers and places the result in RAM in 12 bytes starting at FE5A. These bytes are: player # enable(2 bytes), followed by: fire, Joy, spinner, arm and keypad, first for player 1 and then for player 2. The keypad is decoded so that 5 gives 5 not 12, but the other data is as described. The spinner is used with some attachment we don't have and know nothing about.

Chapter 12. AdamNet

AdamNet is the serial bus that connects the tape, printer and keyboard to the master 6801 microprocessor. It is a half-duplex 62.5 kilobaud token-passing network with four wires: data, reset, +5V, and GND. Commands and responses, each being either data or control codes, are passed back and forth but are not accessible directly to the Z80 microprocessor. The Z80 controls what happens on AdamNet by putting numbers in RAM starting at FEC4 in blocks of 21 bytes called device communication blocks (DCB`s). These bytes have the following information:

Byte	Function
0	status
1-2	buffer address
3-4	buffer length
5-8	sector numbers
9-15	nothing?
16	device number
17-18	maximum length
19	device type
20	node type

The DCB`s can be seen by printing page 254 and the first line of 255 with printmem. The rest of page 255 is wasted waiting for new devices on AdamNet (15are possible). The status byte is usually $80 or 8C. If it is set to 1, as by the status request routine at FC7E,the status of the device is returned in A. If this byte is set to 2 the device is reset. The buffer address points to the area of RAM where the master 6801 will put data from the device. The device numbers are as follows:

0	Master 6801
1	Keyboard
2	Printer
4	Floppy disk 1
5	Floppy disk 2
8	Tape 1
$18	Tape 2

Fortunately, it is not necessary to worry about all of the above details, because operating system routines described in the chapters on the tape, printer, and keyboard will do it for you. Subroutines which directly manage AdamNet are: reset net, scan pcb`s, move pcb`s, find DCB, and status request.

Chapter 13. The Keyboard.

The keyboard has its own 6801 microprocessor with programs to scan and debounce keys and store input in a buffer. When a 'send character' command is received from the master 6801 via AdamNet one ASCII code is sent back. A reset command zeros the buffer and unlocks the shift lock. The wires to the keyboard are GND, +5V, reset, and read-transmit data. The key codes are standard ASCII as listed in the SmartBASIC manual with the following additions:

128	home	144	wild card	160	up arrow
129	I	145	undo	161	right arrow
130	II	146	move	162	down arrow
131	III	147	store	163	left arrow
132	IV	148	insert	164	cntrl up
133	V	149	print	165	cntrl right
134	VI	150	clear	166	cntrl down
		151	delete	167	cntrl left
137	shift I	152	shift wild card	168	up + right
138	shift II	153	shift undo	169	right + down
139	shift III	154	shift move	170	down + left
140	shift IV	155	shift store	171	left + up
141	shift V	156	shift insert	172	home + up
142	shift VI	157	shift print	173	home + right
		158	shift clear	174	home + down
		159	shift delete	175	home + left
				184	shift backspace
				185	shift tab

The master 6801 places the ASCII value received from the keyboard in RAM at FD75. This can be peeked in BASIC for reading the keyboard on the fly without stopping. One problem with this is that it does not notice the second time the same key is pressed. To include this possibility you can POKE FD75 with 0 (first change the POKE routine with POKE 16149,255: POKE 16150,255 so that it will work up there) after each PEEK and then if the result of the next peek is 0 you know a key has not been pressed.

An OS routine at FC6C reads the keyboard (checks the DCB) and then puts the contents of FD75 in A.

CHAPTER 14. The Printer.

The printer is on AdamNet with device number 2. It is written to via a device communication block (see AdamNet).One OS subroutine is all that is necessary to control the printer. It is located at FC63 (print buffer), and prints out RAM starting at the value of the HL registers and stopping when a 3 is reached. Most non-ASCII numbers are ignored, but some are control codes as listed below.

Printer Control Codes

Number(dec)	Function
3	Stop
8	Backspace
10	Line feed
11	Half step line feed
13	Return, forward printing
14	Reverse direction printing

The following program illustrates the use of this routine. The short machine language routine sets HL to $0011 and then calls F515. The rest loads a sample text string with control codes into ram starting at $0011 and then calls the machine language routine.

```
 1 REM   printer routine demo
 5 PRINT
10 DATA   33,17,0,205,21,245,201
20 FOR q = 0 TO 6
30 READ p: POKE 3+q, p
40 NEXT q
50 DATA   13,65,66,67,11,68,69,70,10,14,71,72,73,10,13,74,75,76,3
60 FOR x = 0 TO 18
70 READ p: POKE 17+x, p
80 NEXT
90 CALL 3
```

Chapter 15. The Tape.

Two tapes and two disk drives can be connected to the Adam via AdamNet, each using similar software but having its own device number (See Chapter 12). We have only one tape drive and will concentrate on that configuration. The 6801 that runs the tape has many functions. It controls the two 12V DC motors that maintain tape speeds of 20 in./sec. forward for reading and writing data, and 80 in./sec. fast forward or rewind, using a signal from the small wheel that touches the tape. It also checks for the presence of a tape in the drive with a pin at the top, applies brake current when the tape stops and a weak reverse pull on the motor not driving the tape to keep the tape tight, in addition to reading and writing data. Data is recorded at 1.4K bytes/sec on two tracks of 128K bytes each. On game tapes the blocks of 1K bytes are numbered consecutively from 0 to 7F on the first track and from 80 to FF on the other. On other tapes the blocks are arranged as 40-7F, 0-3F on one track and 80-FF on the other, so that the directory in block 0 is in the middle and distances from it to other files are shorter.

The tape is controlled via AdamNet by a device control block in RAM at FEC4 (see AdamNet chapter). It can be controlled most easily using OS routines outlined below.

FCF3-read block. A= device # (8 for tape). HL= table address in RAM to be written to. BCDE= sector #.

FCF6-write block. A= device #. HL= address in RAM to be read. BCDE= sector #.

FCFC-look for file. HL= address of buffer with name of file. Z is set if found.

Examples of the use of these routines from BASIC programs are at the end of this chapter.

Each tape begins in block zero with a machine-language program that is called if reset is hit. On ordinary tapes this is a jump to word processor, but on the SmartBASIC tape it is a 100 byte program that loads BASIC from the tape and then jumps to the beginning of BASIC at (C80D).

The tape directory starts at block 1, and consists of 26-byte records, the first being the volume record. The volume record consists of: name (12), directory size (1), directory check (4), volume size (4), nothing (2), and date (3), where the number of bytes is shown in parenthesis. The directory size byte has bit 7 set if the Tape is delete protected, and the lower 6 bits specify the number of blocks allocated to the directory. The directory check is a code that is always 55 AA 00 FF. Following the volume record are the file records with the following format: name (12), attribute (1), start block (4), length (2), used length (2), last count (2), and date (3). The name ends with the file type (A,a,H or h) and a 3. The attribute byte is as follows:

bit	indication
7	"permanent" protect
6	write protect
5	read protect
4	user file
3	system file
2	file deleted
1	execute protect
0	not a file

A typical user file has an attribute byte of $10. The start block bytes in the file record point to the first block of the file, and the last count gives the number of bytes in the last block of the file. The catalog command in BASIC loads the directory to RAM starting at page D4, where you can look at it with printmem. To recover deleted files you can change the directory in RAM and then put it back on the tape with the write block OS routine. This is done by the tape editor program below. The program asks what block number you want to see, reads it, and displays it half a page at a time on the screen in the same format as printmem, and asks if you would like to change any bytes. When you are done it stores the edited block back on the tape. To edit the directory ask for block 1.

```
  3 REM     -tape editor by B. Hinkle
  5 PRINT "Insert tape into drive #1": INPUT " hit return"; a$
 10 HIMEM :29999: x$ = "0123456789ABCDEF"
 20 DATA      62,8,1,0,0,17,0,0,33,184,136,205,243,252,201
 25 RESTORE: HOME: o = 9
 30 FOR x = 30000 TO 30014: READ d: POKE x, d: NEXT
 40 PRINT: INPUT "Block # to be edited?"; x
 50 POKE 30006, x: CALL 30000: REM     -read block 'x' to 35000
 60 FOR x = 35000 TO 36023 STEP 128
 65 HOME: tb = 17: bb = 17
 67 PRINT "page:"; INT((x-35000)/256+1);
 70 PRINT TAB(13); : FOR e = 0 TO 7
 80 PRINT e; " "; : NEXT e: PRINT
 83 PRINT "block:"; PEEK(30006);
 85 PRINT TAB(13); : FOR e = 9 TO 16
 87 PRINT MID$(x$, e, 1); " "; : NEXT e: PRINT
 89 o = 10-o
 90 FOR i = 0 TO 127 STEP 8
 95 REM     -print character part of block
100 FOR j = 0 TO 7
110 IF PEEK(x+i+j) < 32 THEN  PRINT "="; : GOTO 125
115 IF PEEK(x+i+j) = 128 OR PEEK(x+i+j) = 148 THEN  PRINT "="; : GOTO 125
116 IF PEEK(x+i+j) = 151 THEN  PRINT "="; : GOTO 130
117 IF PEEK(x+i+j) > 159 AND PEEK(x+i+j) < 162 THEN  PRINT "="; : GOTO 125
120 PRINT CHR$(PEEK(x+i+j));
125 NEXT j: PRINT " ";
130 IF INT(i/16) = i/16 THEN  PRINT MID$(x$, i/16+o, 1); : GOTO 133
132 PRINT " ";
133 PRINT " ";
135 REM      -print hex part of block
140 FOR j = 0 TO 7
145 w = PEEK(x+i+j)
150 PRINT MID$(x$, INT(w/16)+1, 1);
160 PRINT MID$(x$, (w/16-INT(w/16))*16+1, 1);
170 NEXT j: PRINT: NEXT i: PRINT
180 INPUT "Does this screen require changing (y/n, e to exit)?"; a$
190 IF a$ <> "y" AND a$ <> "n" AND a$ <> "e" THEN 180
195 IF a$ = "e" THEN 280
200 IF a$ = "n" THEN  NEXT x: GOTO 280
210 PRINT: INPUT "Byte # to be changed (0-FF)?"; b$
230 GOSUB 520: IF n(1) = 17 OR n(2) = 17 THEN 210
255 IF n(1) > 7 THEN  n(1) = n(1)-8
257 ad = x+16*n(1)+n(2)
260 INPUT "Change byte to (0-FF)?"; b$
263 GOSUB 520: IF n(1) = 17 OR n(2) = 17 THEN 260
265 REM     -change byte and write it to tape
270 POKE ad, n(1)*16+n(2): o = 10-o: GOTO 65
280 INPUT "Would you like these changes to be permanent on tape (y/n)?"; a$
290 IF a$ = "y" THEN  POKE 30012, 246: CALL 30000: GOTO 25
300 IF a$ = "n" THEN 25
310 PRINT "yes or no please": GOTO 280
510 REM   -change byte # to decimal form
520 FOR bc = 1 TO 2: FOR k = 1 TO 16
530 IF MID$(b$, bc, 1) = MID$(x$, k, 1) THEN  n(bc) = k-1: k = 16
540 NEXT k,bc: RETURN
```

All files, except "A" files in BASIC, start with a "header" which is usually one page (256 bytes) long. The third byte is a code that specifies the type as follows:

1	SmartWriter
2	SmartBASIC
16	FlashCard Maker

If you have some other commercial program that makes files you can't read from BASIC use the tape-edit program above to look at the header and see what code they use. The complete header of a SmartWriter file follows:

Byte	Contents
0	header size low (00)
1	header size high (01)
2	file type
3	top margin
4	bottom
5	left
6	right
7	line spacing
8-89	tab array

The following program will copy the SmartBASIC tape to a blank Adam tape. It starts with a short machine language program which sets up and calls the OS routines to read or write to the tape. The assembly language is:

```
LD BC,00
LD DE, block #
LD A,8
LD HL, buffer (31000)
CALL FCF3 (read one block)
RET
```

The routine reads one block, starting at zero, and loads it to the buffer. The program then checks to see if the block was empty and if so loads a new block into the same buffer. When a active block is loaded the buffer pointer is set to the next 1,024 bytes. This continues until 16 active blocks are in the buffer. It then asks you to put in the new tape and writes the 16 blocks to the original locations. It does this by poking the address of the write one block OS routine instead of the read address in the above machine language and calling it. The reason for this complexity is to minimize the number of tape switches you have to make, which can be as many as 16.

```
 2 REM     backup program by B. Hinkle
 5 HIMEM :29999: DIM d(15): q = 20: n = 1
10 DATA    1,0,0,17,0,0,62,8,33,24,121,205,243,252,201
20 FOR x = 30000 TO 30014: READ d: POKE x, d: NEXT
22 PRINT "switch #"; n
23 PRINT "place master tape in drive #1": PRINT "  hit return": INPUT z$
30 i = 0
35 REM  loop for reading blocks
40 POKE 30004, a: CALL 30000
55 IF PEEK((a-INT(a/16)*16)*1024+31000) = 255 THEN 70: REM  block empty
60 POKE 30010, PEEK(30010)+4: d(i) = a
65 i = i+1: IF i = 16 THEN 100: REM  16 full block have been read
70 a = a+1: IF a = 256 THEN  q = i: GOTO 100: REM  end of tape
80 GOTO 40: REM  read another block
```

```
 99 REM  loop for writing blocks
100 POKE 30012, 246: POKE 30010, 117
103 PRINT "place slave tape in drive #1": PRINT " hit return": INPUT z$
110 FOR s = 0 TO 15
112 IF s = q THEN  PRINT "end": END: REM  end of tape
200 POKE 30004, d(s): POKE 30010, PEEK(30010)+4: CALL 30000: REM  write block
300 NEXT s: REM  write 16 blocks
310 n = n+1: RESTORE: GOTO 20:  REM  Read next 16 blocks
```

The following revised cartridge-copy will put 7 cartridges on one tape, and moves the DCB`s to free up the top of RAM. It does not keep track of names, but gives each program a number.

The tape will load and run the game when you hit reset with the tape in the drive. Of course, cartridges are pretty tough and do not normally need to be backed up, but we thought it would be an interesting exercise.

```
]
  3 REM    2nd multiple cartridge copy program by B. Hinkle
  5 HIMEM :30999
 10 DATA    1,0,4,17,0,0,33,40,160,62,13,211,127,26,50,253,124
 20 DATA  62,1,211,127,58,253,124,119,35,19,11,120,177,194,33,121,201
 30 POKE 102, 237: POKE 103, 69: REM   disable interupt routine
 40 FOR x = 31000 TO 31033: READ d: POKE x, d: NEXT: REM        read cartridge

 50 DATA   62,8,1,0,0,17,0,0,33,40,160,205,246,252,201
 60 FOR x = 40100 TO 40114: READ d: POKE x, d: NEXT: REM        write tape
 70 PRINT "Insert slave tape into drive #1 and check cartridge in slot"
 72 PRINT: PRINT
 73 INPUT "Input number (1-7) of cartridge on tape?"; n
 75 REM    write 32K of cartridge to tape
 77 PRINT: PRINT "REMEMBER: keep your own list of cartridges on this tape,";
 78 PRINT " because cartridges are not listed in the directory."
 80 FOR i = 0 TO 31
 83 POKE 31005, i*4+128: POKE 40106, i+(n-1)*32+2
 90 CALL 31000: CALL 40100: NEXT i
100 REM          ml routine for block 0
110 POKE 40106, 0
120 DATA    243,49,0,72,33,0,8,62,2,205,41,253,33,0,0,62,3,205,41,253,33,0,32,
62,4
130 DATA   205,41,253,205,56,253,62,240,33,0,32,17,32,0,205,38,253,1,224,1
140 DATA   205,32,253,1,31,24,33,0,8,17,3,3,205,54,252,24,7,67,65,82,84,32
150 DATA    35,63,6,7,33,62,200,126,205,57,252,35,16,249,217,17,0,128,217
155 DATA    33,192,254,17,0,73,1,63,1,237,176,33,0,73,205,123,252,17,0,2,33,11
7,200
160 DATA  1,63,0,237,176,195,0,2,205,108,252,205,57,252,214,48,71,33,226,255
170 DATA   17,32,0,25,16,253,235,6,2,197,33,0,3,1,0,0,62,8,205,243,252,1,0,4,9

180 DATA   19,124,254,67,32,238,217,33,0,3,1,0,64,237,176,217,193,16,221,62,3,
211,127,195,0,0
200 FOR x = 41000 TO 41179
210 READ d: POKE x, d: NEXT: CALL 40100
250 PRINT: PRINT "copy #"; n; " is complete": END

]
```

We have always inserted cartridges with the machine on and have not had any problems, although we may be very lucky. If you want to do this it saves time putting multiple cartridges on one tape, and you can add these lines:

```
95 INPUT "another cartridge (y/n)?"; D$
97 IF D$="y" THEN GOTO 70
```

Some cartridges do not run from tape when copied by our program. There are at least two possible reasons for this. Some cartridges have protection routines which prevent them from working when they are in RAM instead of ROM, and AdamNet uses the top two pages of RAM which creates a conflict if the cartridge program goes that high. If a cartridge doesn't work with cart-copy you can look at it with the programs below to see what the problem is. This is not easy, but is a second game to play with the cartridge which may well be better than the first. It is also interesting to PEEK around cartridge ROM with cart-viewer because of the graffiti left there by frustrated programmers. One protected cartridge that Ben unlocked was River Raid, which checks to see if it is in RAM with a routine at 80B4-80CB. You can deactivate this routine on the tape copy with tape-edit by reading block 2 and changing byte B5 from 00 to 55. The copy will then run perfectly. Most cartridges do not use the whole 32K space they have, but all start at $8000. Most have 55 AA for the first two bytes, which is some sort of code. The start address of the program is stored at 800A,B. Cartridges use RAM at $7000-73FF so if you find a LD command to an address above $8000 it must be a protection routine. OS7 routines are from 0 to $2000. We have made the following additions to viewer and the disassembler so they will work with cartridges and can be used as deprotection tools.

Cart-viewer additions to viewer.

```
2 LOMEM: 30000:POKE 102, 237:POKE 103,69
3 DATA 62,13,211,127,17,72,113,33,0,0,1,0,1,237,176,62,1,211,127,201
4 FOR x=28000 TO 28019: READ d: POKE x,d: NEXT
7 GOSUB 100
100 POKE 28009,p: CALL 28000: RETURN
```

Cart-disass additions to disassembler.

```
3 LOMEM: 28256: POKE 102,237: POKE 103,69
10 INPUT "address?";ra: GOSUB 6100
22 PRINT ra; TAB(7)
118 ad=ad+1: GOSUB 6200: op=PEEK(ad)
150 IF n>0THEN ad = ad+1: GOSUB 6200: n=n-1: op= PEEK(ad): GOSUB 120
160 IF n>0 THEN ad=ad+1: Gosub 6200: op= PEEK(ad): GOSUB 120
199 ad=ad+1: GOSUB 6200: GOTO 20
4000 GOTO 7000
5000 a= INT(ra/4096)
5020 b= ra-a*4096
6015 oad=ra
6020 ra=ra+op+1: GOSUB 5000: ra=oad: RETURN
6100 POKE 27909, INT(ra/256)
6110 POKE 27908, ra-INT(ra/256)*256
6120 CALL 27900
6130 ad=28000
6140 RETURN
6200 ra=ra+1:  IF ad=28256 THEN 6100
6210 RETURN

7000 DATA 62,13,211,127,17,96,109,33,0,0,1,0,1,237,176,62,1,211,127,201
7010 FOR x= 27900 TO 27919: READ d: POKE x,d: NEXT
7020 GOTO 10
```

Some people have asked about the prompt "insert cartridge", when you are only supposed to insert a cartridge when the computer is turned off. We have always inserted cartridges with the computer on and have never found ill effects, but to be sure you should follow Coleco's instructions.

Disks

Disks are on AdamNet and behave just like tape drives except that they have different device numbers (4 and 5), have less storage (160 blocks), and are noticeably faster. To make our programs work with a disk you can change the device numbers in the machine code. In Backup, if you change the 8 in line 10 to a 4, the program will copy disk to disk in drive 1 (also change the 256 in line 70 to 160). If you do not change line 10, but add a new line "101 POKE 30007,4", it will copy tape to disk (not the whole tape of course). In cartridge-copy if you change the 8 in line 50 to a 4, it will copy to disk. In tape-editor if the 8 in line 20 is changed to 4 it is a disk editor.

Making your own "Digital Data Packs".

We originally made a copy of an Adam tape with a commercial dup machine often found in tape stores. Such machines copy both sides of a tape at once, take one minute and cost $4. Since then we have tried our home tape recorder and find that it works fine, but takes one hour. Any tape you copy will be exactly duplicated, including the programs. You can INIT the copy to reuse it, however. If you have an Adam tape from a non-Coleco source you can copy it as you would any audio tape (both sides, in stereo or mono), using a high volume record setting in the middle of the red on the VU meters. To copy a Coleco DDP you must drill two holes where cassettes usually have holes, and to play it in Adam you must drill two holes in the back of the copy. The best audio cassettes to use are Sony HF60's which currently cost 69 cents in Ithaca.

Chapter 16. The Power Supply

The power supply, located in the printer, provides 18V unregulated for the ribbon solenoid and the following regulated voltages: +5V (3 Amps), -5V (0.2 Amps), +12VI (2 Amps, inductive, for motors), and +12VL (0.3 Amps, noise free for logic). The regulation should hold from input "110 VAC" voltages from 108 to 132V. The pins of the cable connecting the printer to the main console are as follows:

1	brown	+12VL
2	red	+12VI
3	orange	+5V
4	yellow	-5V
5	green	GND
6	blue	AdamNet
7	violet	reset

It is hard to imagine an application where someone would want to provide another power supply, but if it is absolutely necessary we recommend using a 110VAC generator. Anything else would be very complicated, especially considering the -5V.

Chapter 17. The Expansion Connectors.

The expansion connectors were designed for use with 64K expansion RAM, modem, and expansion ROM. They can be used for any hardware project, however, such as a printer interface, speech generator, analog to digital or digital to analog converter. We do not recommend trying to design your own hardware unless you have some experience, since you could easily dammage your Adam. Unlike the Apple II bus, all three connectors are different, and there is no convenient decoding to address a slot. The control lines are defined on the next page where Z80 pinouts are given. The address (A), and data (D) lines are buffered from the Z80.

Expansion connectors (back)

RTC

GND	29	30	+5V
	27	28	A7
	25	26	
A5	23	24	A6
A3	21	22	A4
A1	19	20	A2
A0	17	18	D7
D0	15	16	D6
D1	13	14	D5
D2	11	12	D4
$\overline{\text{BMI}}$	9	10	D3
$\overline{\text{BIORQ}}$	7	8	$\overline{\text{BRD}}$
$\overline{\text{INT}}$	5	6	$\overline{\text{BWR}}$
	3	4	
	1	2	

I/O ROM

GND	43	44	+5V	
A14	41	42	A7	
A13	39	40	A12	
A6	37	38	A8	
A5	35	36	A9	
A4	33	34	A11	
A10	31	32	A3	
A1	29	30	A2	
A0	27	28	D7	
D0	25	26	D6	
D1	23	24	D5	
D2	21	22	D4	
$\overline{\text{CAS2}}$	19	20	D3	
A15	17	18		
$\overline{\text{BMRQ}}$	15	16	$\overline{\text{BMI}}$	
$\overline{\text{BRD}}$	13	14	Aux ROM CS	
	11	12	$\overline{\text{IORQ}}$	
$\overline{\text{BWR}}$	9	10		
	7	8	$\overline{\text{INT}}$	
	5	6		
	3	4		
	1	2	audio	

64K RAM

GND	29	30	+5V
A14	27	28	A12
A13	25	26	A8
A6	23	24	A9
A5	21	22	A11
A4	19	20	A3
A10	17	19	A2
A1	15	16	D7
A0	13	14	D6
D0	11	12	D5
D1	9	10	D4
D2	7	8	D3
CAS2	5	6	A7
RAS1	3	4	MUX
A15	1	2	$\overline{\text{BRW}}$

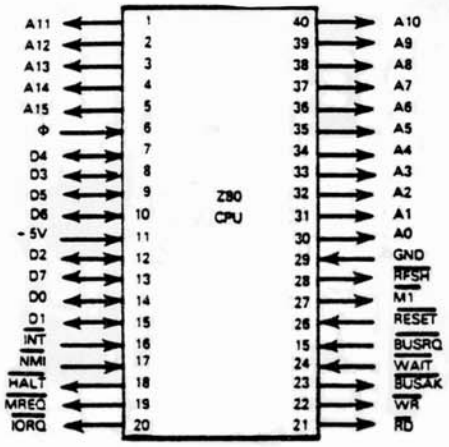
Cartridge Connector

GND	29	30	+5V
CS4	27	28	A8
A7	25	26	A9
A6	23	24	A12
A5	21	22	CS2
A13	19	20	A14
A4	17	18	CS1
A3	15	16	A10
GND	13	14	A11
A2	11	12	D7
A1	9	10	D6
A0	7	8	D5
D0	5	6	D4
D1	3	4	D3
D2	1	2	CS3

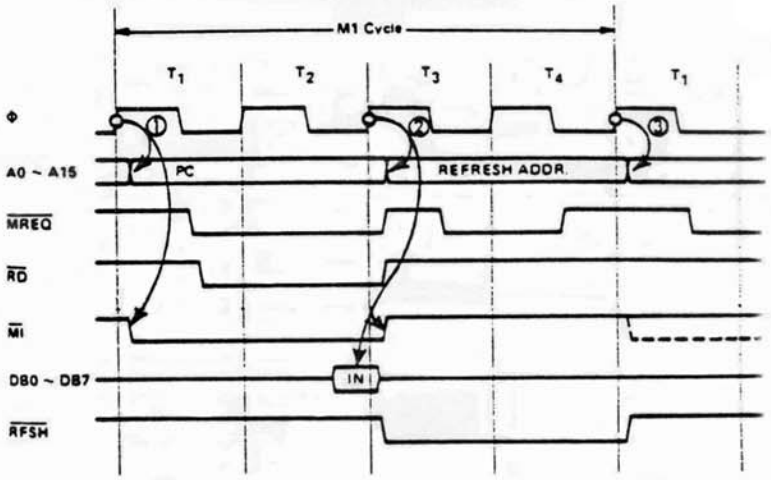
Expansion module connector (on side).

1	GND	31	Audio in, works for out
2	GND	32	Video input enable (+9V)
3	BD3	33	NTSC comp video input (+6V,1.5 VAC)
4	BA14	34	Game mode reset
5	Y2 138 decoder	35	Sound disable (0V)
6	Y1 138 decoder	36	Nothing
7	$\overline{\text{HALT}}$ input	37	BA11
8	$\overline{\text{BRW}}$ output	38	BA12
9	$\overline{\text{NMI}}$ in/out	39	VDP sync/reset input
10	Spinner int disable	40	BIORQ output
11	$\overline{\text{BUSRQ}}$ input	41	Nothing
12	BD1	42	Nothing
13	Z80 reset input	43	BA15
14	BD0	44	BA3
15	BM1 output	45	Clock 3.58 MHz
16	BD7	46	BD2
17	BD6	47	BA0
18	BA1	48	BD5
19	BD4	49	$\overline{\text{BRFSH}}$ output
20	BA2	50	$\overline{\text{WAIT}}$ input
21	BA4	51	$\overline{\text{INT}}$ input
22	BA13	52	$\overline{\text{BUSAK}}$ output
23	BA5	53	$\overline{\text{BRD}}$ output
24	BA6	54	$\overline{\text{BMREQ}}$ output
25	BA7	55	$\overline{\text{IORQ}}$ output
26	BA8	56	Audio RDY output
27	BA9	57	+12V
28	BA10	58	+5V
29	AUX decode 1, input	59	+5V
30	AUX decode 2, input	60	-5V

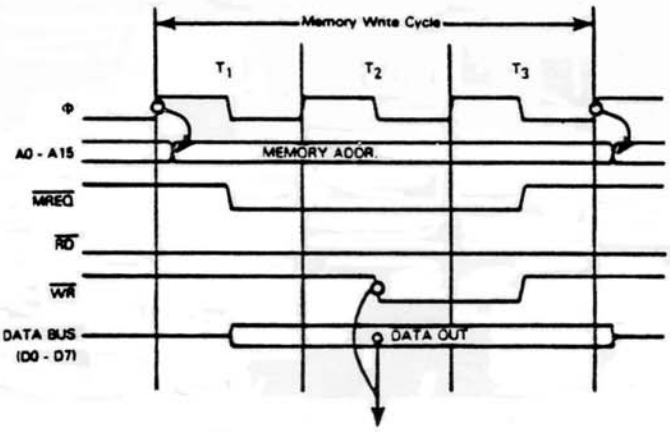
All address lines are output only, but data lines are in/out. The pins are arranged in normal connector format as shown for the expansion connectors on the previous page. Overlines indicate active low.



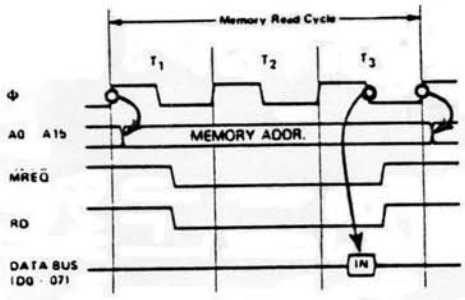
Pin Name	Description	Type
A0 - A15	Address Bus	Tristate, Output
D0 - D7	Data Bus	Tristate, Bidirectional
M1	Identifies instruction fetch machine cycle	Output
MREQ	Memory request — indicates that CPU is performing a memory access	Tristate, Output
IORQ	I/O request — indicates I/O operation in progress	Tristate, Output
RD	CPU read from memory or I/O device	Tristate, Output
WR	CPU write to memory or I/O device	Tristate, Output
RFSH	Refresh dynamic memories	Output
HALT	CPU Halt executed	Output
WAIT	Wait state request	Input
INT	Interrupt request	Input
NMI	Nonmaskable interrupt request	Input
RESET	Reset and initialize CPU	Input
BUSRQ	Request for control of Address, Data and Control Busses	Input
BUSAK	Bus acknowledge	Output
Φ	CPU clock	Input
+5V,GND	Power and ground	



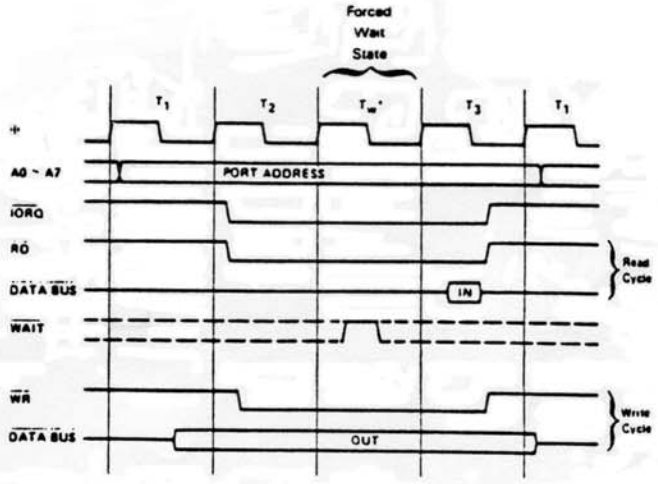
Z80 Instruction Fetch Sequence



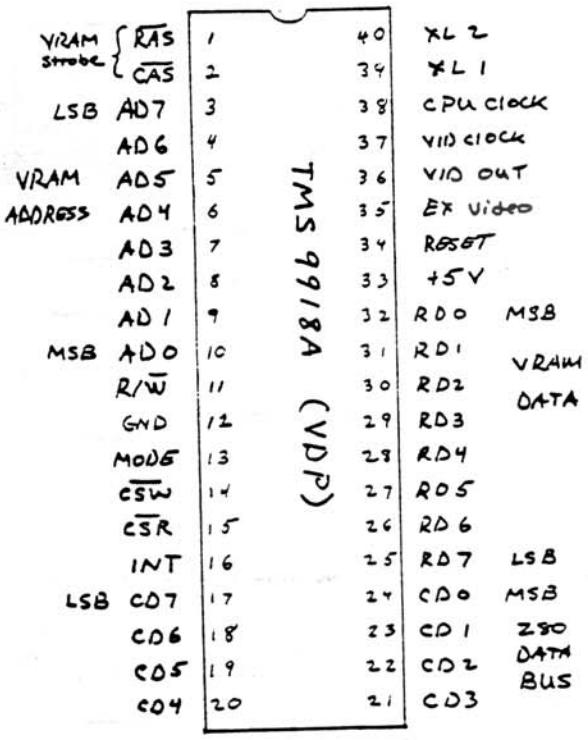
Z80 Memory Write Timing



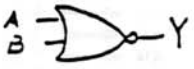
Z80 Memory Read Timing



Z80 Input Or Output Cycles

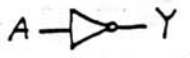


+NOR



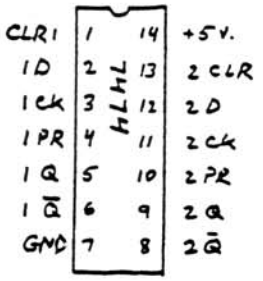
$Y = \overline{A+B}$

HEX INVERTERS



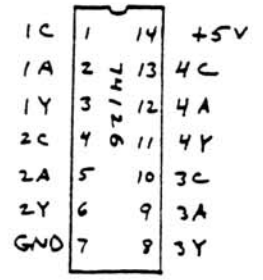
$Y = \bar{A}$

7405 has open-collector outputs

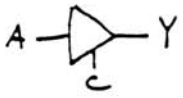


DUAL FLIP-FLOP

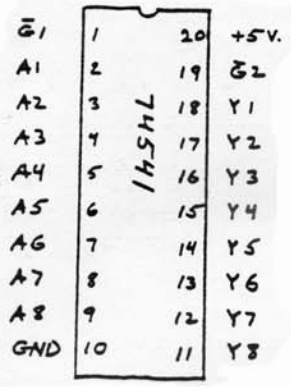
INPUTS				OUTPUTS	
PR	CLR	CK	D	Q	Q̄
L	H	X	X	H	L
H	L	X	X	L	H
H	H	↓	H	H	L
H	H	↓	L	L	H
H	H	L	X	Q_0	$\bar{Q}_0$
L	L	X	X	H*	H*



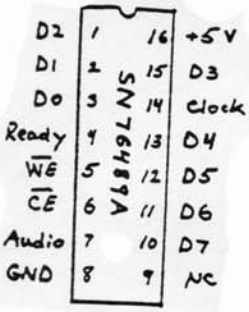
QUAD BUS BUFFER
3-state OUTPUTS

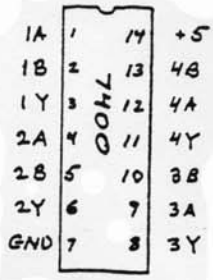


Y = A, output is disabled when C is Low.



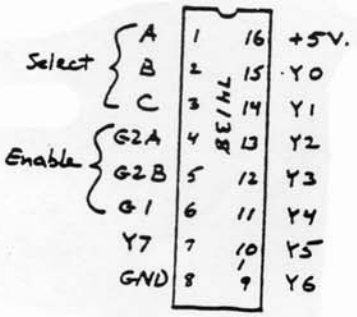
OCTAL BUFFERS
3-State OUTPUTS
NON-INVERTING





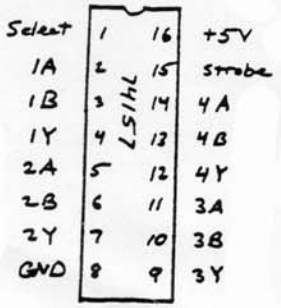
+ NAND

$Y = \overline{AB}$



3 to 8 line Decoders

Enable		Select			Output
G1	G2A+B	C	B	A	
X	H	X	X	X	all H except Specified
L	X	X	X	X	all H except Specified
H	L	L	L	L	Y0
H	L	L	L	H	Y1
H	L	L	H	L	Y2
H	L	L	H	H	Y3
H	L	H	L	L	Y4
H	L	H	L	H	Y5
H	L	H	H	L	Y6
H	L	H	H	H	Y7



QUAD 2 to 1 line DATA Selectors

Strobe	Select	A	B	OUT(Y)
L	X	X	X	L
H	L	L	X	L
H	L	H	X	H
H	H	X	L	L
H	H	X	H	H