INTRODUCTION

TO

2068 MACHINE CODE

by

Dr. Lloyd Dreger

A Self Study Manual For The
Beginning Assembly Language Student
That Bridges The Gap Between
Advanced Basic and Machine Code

Distributed by S.M.U.G.--Sinclair Milwaukee Users Group
Box 101, Butler, WI 53007

This Manual was typed on the Timex/Sinclair 2068 Computer using MSCRIPT, an AERCO Printer Interface with a Gemini Star 10X Dot Matrix Printer. It was stored on disks using an AERCO disk drive system.

INDEX

LET'S START AT THE VERY BEGINNING

I am assuming that my readers know the BASIC language of the T/S 2068 When you start to learn to walk you don't run the four minute mile the first day or even the first year. First you crawl, then you stand and take your first steps, finally walking (Beginning Basic) and then running (Advanced Basic).

Machine code is the equivalent of flying. When you try to fly, the first thing a sane person would do is to study aerodynamics. Then, getting your courage up, you climb a small bluff, put on your wings, get a running start over the precipice and, hopefully, soar down to the bottom of the bluff without crashing. Less sane types would just jump off the bluff.

This manual is a self study course in machine code aerodynamics as applied to T/S 2068 wings. This is what you should (or have to) know before you ever start writing machine code programs. We will start with some simple examples (easy low bluffs) but we won't get into advanced code as that is the topic for the next manual. Portions of this book have been pretested on my present m/c class. Their helpful suggestions were greatly appreciated.

A MISCONCEPTION

M/C is a new language. Just like flying uses different rules, different coordination and different muscles than walking, so is machine code different from Basic. It is not an extension of Basic nor is it an emulation of Basic commands--that is the function of the ROM in your computer. When one writes a M/C program one plans and thinks altogether differently than one does in Basic.

WHY LEARN MACHINE CODE?

There are only three good reasons for using machine code:

1. Basic is too slow.
2. Basic can't do it or is too cumbersome.
3. Basic is too long.

You may wish to add another:

4. Because it's there.

Which means that you're just curious as to what somebody else's code is doing. And you know what curiosity does...

BUT TO GET STARTED

Some of this is going to be very basic and elementary--if it is, just skip that section and go on to one you don't know. I have tried to cover all the bases and leave nothing to chance.

CHAPTER 1

NUMBERS AND COUNTING

THE DIGITAL COMPUTER

The T/S 2068 is a digital computer. All it can do is handle numbers and that is ALL it can do--nothing more. But, it handles numbers very rapidly and very accurately doing exactly as it's told at about a million instructions per second. Therein lies the problem. You are now embarking on writing instructions for a computer. In Basic there was enough error trapping to just stop a program and tell you you goofed. In machine code it crashes. 99 times out of 100 the first time you run your program it's going to crash irretrievably. So the first law of m/c programing is: ALWAYS SAVE YOUR PROGRAM BEFORE RUNNING IT...unless you don't mind the frustration of reentering your program a zillion or so times before you get it right.

The computer is "dumb" and has no brain of its own. It follows your instructions to the letter. Pardon me, to the number. If you tell it to do something wrong or something stupid it will do it as it doesn't know any better. There is no, "Well, you know what I mean". It's still the same old adage, "GARBAGE IN, GARBAGE OUT". The machine code programmer has to be meticulous and exacting or his/her program will never give the correct results. There is no margin for error. There is talk of making a thinking computer but these types of programs are complex and far far too advanced for you at this point--let's soar first, then try hang gliding and maybe a little flying like a turkey before we try to soar like an eagle or go supersonic like an SST.

A STUMBLING BLOCK

I said that the computer can only handle numbers. That is true and later on I will show you that that is all it really does. This is the problem however. Everything is numbers and sometimes the numbers stand for numbers, sometimes they stand for instructions, sometimes they stand for symbols and sometimes they stand for pictures--it all depends upon where you are in the computer. We will find out what they mean at all these different times. But first let's look at how the computer stores a number.

THE EIGHT BIT BYTE (pronounced bite)

Oh, oh! Two new words! If you are going to learn a new language you are going to have to learn the terminology so don't be afraid of new terms as they will be fully explained as they are introduced. You can't be a Chemist without knowing the symbols

they use for elements and how they combine them to represent molecules, or Physics without terms such as momentum and inertia. So it is with programming.

The 2068 is an "electronic" digital computer--not a mechanical computer. In fact, it's nothing but a very huge collection of on-off switches called transistors packaged together very densly in LSIC (Large Scale Integrated Circuits) called chips which have from 2 to 64 leads coming out of them for connections to other things. They look like little black boxes. We are not going to investigate what exactly is inside these little black boxes as we leave that to the "Hardware Hackers". We are just interested in how to make them work properly. We aren't even interested in what signal to send down what wire as far as that goes.

BINARY NUMBERS

The memory section of the 2068 consists of 65536 sections each of which is 8 bits (switches) long. Each of these 65536 sections is called a byte. Where in this array of 65536 positions a particular byte is located is called its ADDRESS and is designated by a number from 0 to 65535. Zero is an absolutely good number as far as a computer is concerned. The bits inside a byte also are named by numbers. The lowest bit is called "Bit 0" (zero, not the letter "O"), with the next called "Bit 1", etc. to Bit 7 for the highest. Since we are used to reading numbers from left to right we would write the names of the bits as the following string:

7 6 5 4 3 2 1 0

By convention, we designate a zero as the "off" state of a switch and "1" as the "on" state. Then the number 0 would look like:

0 0 0 0 0 0 0 0

Horray! It even looks like zero!

Flipping Bit 0 on to represent a 1 condition makes our string of bits:

0 0 0 0 0 0 0 1

And that looks like 1. So far so good.

But now we run into a snag since if we try to add another number to the right position for the number 2 we can't. There are only two positions for this switch, "on" and "off"--it's not a 10 position switch which is what we would need to write decimal numbers. Our only recourse is to start using the Bit 1 switch. Turning that on and turning off the Bit 0 switch gives us:

0 0 0 0 0 0 1 0

Which looks like 10 doesn't it?

If we think about our digital (literally fingers) number system, which is a base 10 number system for obvious reasons, it starts using the 10's position on the 10th number (not counting 0). Now, since the computer starts using the 10's position on the 2nd number (again not counting 0), we say that the computer is using a base 2 number system--or binary (bi means 2 as in bi-cycle).

Well, how do we write 3? Simply as 2 + 1 or:

0 0 0 0 0 0 1 1

We are again, as they say, "full up" so the number 4 has to start using Bit 2--the third bit:

0 0 0 0 0 1 0 0

With 5 as: 0 0 0 0 0 1 0 1
and 6 as: 0 0 0 0 0 1 1 0
and 7 as: 0 0 0 0 0 1 1 1

At 8 we have to start using Bit 3 as:

0 0 0 0 1 0 0 0

But a shortcut. If we write down the values at which a switch is first used together with the switch or bit numbers we get an interesting sequence:

Bit #	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

Each bit is double that of the one to its right. Write this little table down somewhere so that you can refer back to it from time to time.

And, since we saw that to do a 3 we actually did 2 + 1, if we add all these values together we come up with 255 which is the highest value we can store in a byte. That's not a very big number. What do we do for bigger numbers?

We use another byte. But we don't just add the two together as that would only get us to 510. Let's put the 2nd byte in front of, or to the left of, the "low" byte. Like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32768	16384	8192	4096	2048	1024	512	256	128	65	32	16	8	4	2	1

If we continue our series of doubling, we get the numbers listed below the bit numbers as the first time that bit is used. Adding all these together gives us 65535 as the maximum for two bytes. Adding the number 0 gives us a total of 65536 storage spaces or bytes in memory. We call the upper byte of this 2 byte

"word" the "high" byte.

We have already used this terminology in Basic when we did:

PRINT PEEK 23627 + 256*PEEK 23628.

Look at the value of Bit 8--256. To get it to read 1 we have to divide it by 256. To get Bit 9 to 2 we also have to divide by 256. Since the byte can only hold numbers up to 255 we have to multiply the high byte by 256 for the right answer. Omitting that 256* gives the wrong answer.

But shouldn't it be PRINT 256*PEEK 23627 + PEEK 23628?--high byte first? Nope! A quirk of m/c code is that it's always "low byte first, high byte last" for "word" size numbers. We were actually asking our computer to give us the starting address of the VARS (variables) area storage.

We will be using double byte word storage quite often in m/c BUT we do not extend it to 3 or more bytes. When we need numbers bigger then 65535 or numbers with fractions, the computer goes to what is called floating point notation which handles numbers in quite a different manner.

CONVERTING BINARY TO DECIMAL NUMBERS AND DECIMAL TO BINARY

Get that little table I told you to write down. You may wish to extend the table by adding the high byte numbers we added on page 5. Use a slash between the high and low byte. Let's convert:

0 1 0 0 1 1 1 1

back to decimal. We will write down the numbers corrresponding to the 1 bits only and add these up. The left position is 0 so no 128. Next a 1 so we have 64. 2 more zeros so skip 32 and 16 but write down 8, 4, 2 and 1 for the last 4 1's. Adding them up we get 64 + 8 + 4 + 2 + 1 = 79.

Now try these: 1 0 1 0 1 0 1 0, 01100110, 10101, 1111.

Did I confuse you? Generally programmers don't space binary numbers nicely but run them all together as a string of 1's and 0's. They also drop all leading zeros. You should have gotten 170, 102, 21 and 15 for your answers.

Let's go the other way and convert a decimal number to binary. First, write down the number. We'll use 89. Refer to your table and try to subtract off 128. We can't, so write down a "0". Repeat with 64. It works so write down a "1" after the "0". Subtracting 64 leaves 25. Obviously 32 can't be subtracted so another 0. 16 can, so a 1 leaving 9. Another 1 for 8 leaves 1 so no 2 or 4 (00) and a 1 for the final 1 to finish off the number: 01011001.

Okey! Try 180, 56 and 219. (The answers are on the bottom of the page to make it a little more difficult for you to cheat.)

ADDING AND SUBTRACTING IN BINARY

Adding and subtracting binary numbers is quite like it is in decimal notation if we remember that we can NEVER have numbers bigger than 1. So if we get a 2 in addition we write a zero and carry 1 to the next column. For example.

```
 01111110     126
 00011111     _31
 10011101     157
  ccxxxc
```

In the above example, the columns marked with a "c" resulted in an addition of 2 so we write a 0 and carry a 1. In the columns marked by an "x" we had a carry and an add to give us a 3 so we write a 1 and carry a 1 to the next column.

Now try these:

```
 01010101     00111011     00111110     11000001
+10011111    +00111101    +01111100    +01000000
```

The answers are on the bottom of the page again. But what about that last number? 193 + 64 = 257--that's bigger than 255 and our actual answer is 1. Only the 8 low bytes count. We have what is known as an "overflow". No, the computer won't crash if it happens but instead will turn on the CARRY FLAG to warn itself that a byte has gone past the zero mark. Thus, we sometimes call the carry flag the 9th bit of a byte. The computer automatically uses it when adding or subtracting word long numbers to get any carrys from the low byte to the high byte.

SUBTRACTION: In subtraction, borrowing a 1 through a 0 results in a 1 remainder, not a 9 as we're used to. As an example:

```
   ccc2
 10000011
-00101010
 01011001
```

The columns marked with a "c" have a remainder of 1 as the carry continues with a final carry of 2 over to the last column.

Now, try these subtractions. Careful, the last one is a zinger.

```
 11001100     11100011     11111000     00011110
-00110011    -10011111    -10101010    -11100000
```

ANSWERS: 180 = 10110100, 56 = 00111000, 219 = 11011011
ADDITION ANSWERS: 11110100, 01111000, 10111010, c00000001
SUBTRACTIONS ANSWERS: 10011001, 01000100, 01001110, M00111110

That last carry has an "M" before it indicating that it's a minus or negative number. And, you guessed it, another flag went up--the minus flag indicating that this time we had an "underflow". The carry flag also goes on...unless it was on in which case that "1" was used and the carry flag is now "off"...but that can only happen when you use SBC (subtract with carry) instruction. You will learn more about that later.

BINARY MULTIPLICATION

It's like regular multiplication by 1 and 0 with the add rules applying when you add. Since your computer can only add two numbers we will be adding our partial answers as we go along. We will assume we have as many bits as necessary for the answer and not concern ourselves with handling overflows at this point.

```
  00010110      22
 x00001101     x13
  00010110      66
 000101100_     22
 0001101110    286
 00010110___
 00100011110
```

```
  0000110101101111      3439
 x0000000011101111     x239
  0000110101101111     30951
 0000110101101111_     10317
 0001010000100110 1    6878__
 0000110101101111__    821921
 0001011110000010 01
 0000110101101111___
 0001100100110000001
 0000110101101111____
 0001001110111011000 01
 0000110101101111______
 0001011101001100100001
 0000110101101111_______
 0001100100001010100100001
```

Try these. No ringers this time.

```
  00001111      00010001      111101101101101
 x00001111     x00010001     x0000001111101001
```

The answers are below.

BINARY DIVISION

It's like regular division and subtraction combined with carrys on the subtractions as we learned above.

MULTIPLICATION ANS: 11100001, 100110010, 1101000101000001101 01

```
      ___10011
1101/11111111
     1101
       10111
        1101
        10101
         1101
remainder 1000
```

```
      ______1110111110
and 110101/1100011001010110
           110101_
           1011100
            110101_
            1001110
             110101__
             1100110
              110101_
              1100011
               110101_
               1011100
                110101_
                1001111
                 110101_
                  110101
                  110101
                       0
```

In the first case above we stopped when we hit the decimal point but there was no reason why we should do so. We could have kept right on going. We as yet have not discussed binary fractions so you have to go to a later section of this chapter to find out how a binary fraction is converted back to a decimal fraction.

Try these divisions.

1000001/1111, 11001100110/11001100, 111111111/101010.

You really won't be doing much adding, subtracting, multiplying, or dividing in binary in m/c but it lays the groundwork for floating point numbers which will be discussed later on.

HEXADECIMAL COUNTING

And that is about all you can do with it is count. But, many programmers write in hexadecimal code and as such we have to discuss it. It has only 2 advantages:

1. All 8 bit bytes can be expressed with 2 and always 2 symbols.
2. Everyone does it. Well, almost everyone.

The second is the lamest excuse I have ever seen. It smacks of a "lemming" attitude.

I have never seen a scheme of doing simple arithmetic operations in hexadecimal. However, it has some historical, or is it hysterical, significance. It is the reason why the byte has 8 bits.

DIVISION ANSWERS: 101, 1000 remainder 110, 11000011000.

If you think about binary numbers, it takes 4 bits for the number 9, written as 1001B (the B in back of the number will be our way to designate a binary number). An H behind a number will signify a Hexadecimal number. Decimal numbers sometimes have a D suffix or none at all. Because all three systems are used we have to specify which we are using.

Historical: Back when computers had expensive memories (hand wired ferrite cores!) the byte was only 4 bits long. Just long enough to express a single decimal digit if you wish, with a little to spare--up to 15. Programmers didn't want to waste this extra space so the base 16 number system was devised. The 16 numbers used in this system are 0-9, A, B, C, D, E and F. You can see that a lot of imagination, intuition and ingenuity went into designing those symbols by BIG BLUE. It was in the era of IBM and Sperry Univac that I first saw the terminology used so I'll blame BIG BLUE as I first saw it in their publications. Even if IBM didn't do it, shame on them if they never bothered to fix it. IBM mania has led to other unnecessary complexities as well.

The 4 bit byte became known as the nybble (pronounced nibble). When computers grew, they put 2 nybbles into a byte making it 8 bits long. The future goes to the 16 and 32 bit computers with CPU's (Central Processing Units) already designed and in use so there is no chance of anyone ever deviating from the pattern.

In Hex, A = 10, B = 11, C = 12, D = 13, E = 14 and F = 15. The low 4 bits of a byte are expressed in the right symbol, the high 4 bits in the left symbol. If a nybble is zero, a leading or trailing 0 must be written. Thus F0 = 240 and 0F = 15.

To convert Hex to Decimal: Take the first symbol value and multiply by 16 and add the value of the 2nd symbol...if you have an IQ higher than 135 you can do it quite rapidly in your head.

To convert Decimal to Hex: Divide the decimal by 16, convert to the proper first symbol and then convert the remainder to the 2nd symbol.

For double byte numbers converted either way the important numbers to remember are 4096, 256, 16 and 1. I dare most of you to do that mentally.

For those of my readers who have normal IQ's the following table will help to convert from Hex to Decimal and Decimal to Hex.

To get from hex to decimal: Find the first symbol in the left column and read across to the column of the second symbol as found on the top. AA = 170. AB = 171. BA = 186.

To go from decimal to hex: Find the number in the table and read the first symbol on the left and the 2nd symbol on the top.

SECOND SYMBOL

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
F	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	1	16	17	18	18	20	21	22	23	24	25	26	27	28	29	30	31
R	2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
S	3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
T	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
S	6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
Y	7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
M	8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
B	9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
O	A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
L	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

OTHER NUMBER SYSTEMS

There is still another base number system that has been used from time to time called OCTAL--base 8, but we won't be using that system.

NEGATIVE NUMBERS

Up to this point we have been discussing positive or unsigned numbers. We now go to signed numbers.

Humans use the minus sign in front of a number to designate a negative number and sometimes a plus sign in front of positive numbers--but at least always a minus sign in front of negative numbers. Unfortunately, the computer doesn't recognize a minus

sign unless it's given a number. In addition, the sign of a number should be stored with that number so it doesn't get lost. In signed numbers one of our precious 8 bits must be used as a sign designator. By convention, it's Bit 7. When zero, the number is positive, when 1, negative.

Let's see now. That limits positive numbers from 1 to 127 as 128 turns on bit 7. That leaves 127 spots for negative numbers.

But if 00000001B is 1, 10000001B should be -1. It isn't.
_11111111B is -1. I have deliberately written it exactly
100000000B under the 1 and added them together.

Take another 2 numbers: 00110001 49
11001111 -49
Adding them: 100000000.

Notice that in both pairs the 0's have turned to 1's and the 1's to 0's...except for the low bit. If we took exact opposites and added we would always get 11111111B which is only 255. If -1 were 11111110, then 11111111B would be minus 0. I'm not enough of a mathematician to tell you the difference between a +0 and a -0. To avoid this, the number is complemented to 256 by merely flipping all the bits and adding 1. This is called twos complementing a number. In this way we only have one zero, not two. Simply flipping the bits is called complementing.

Since we will be working with single byte negative numbers the table on the next page will help convert a negative number to its 2's complement. The left column gives the tens with the units across the top. The table is dual conversion. Just use the right lines for decimal or Hex conversion. The table is al- so reproduced in the appendixes for your convenience.

Word long numbers use the same method by 2's complementing to 65536 with the first bit of the high byte indicating sign.

As an aside, the word complement comes from Plane Geometry where we had pairs of angles known as complementary angles which always added up to exactly 180 degrees.

Now for the question of the month. How does the computer know when it's dealing with signed numbers or unsigned numbers?

The answer is, it depends upon the instruction. If the instruction requires a signed number one must be used. Signed numbers are used in JUMP RELATIVE statements where a negative number means jump backwards and a positive number means jump forward. The computer looks at Bit 7 and then makes appropiate operations based on the rest of the number.

NEGATIVE NUMBER CONVERSION TABLE
(DECIMAL AND HEX)

	0	1	2	3	4	5	6	7	8	9
0	0	255	254	253	252	251	250	249	248	247
	00	FF	FE	FD	FC	FB	FA	F9	F8	F7
10	246	245	244	243	242	241	240	239	238	237
	F6	F5	F4	F3	F2	F1	F0	EF	EE	ED
20	236	235	234	233	232	231	230	229	228	227
	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3
30	226	225	224	223	222	221	220	219	218	217
	E2	E1	E0	DF	DE	DD	DC	DB	DA	D9
40	216	215	214	213	212	211	210	209	208	207
	D8	D7	D6	D5	D4	D3	D2	D1	D0	CF
50	206	205	204	203	202	201	200	199	198	197
	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5
60	196	195	194	193	192	191	190	189	188	187
	C4	C3	C2	C1	C0	BF	BE	BD	BC	BB
70	186	185	184	183	182	181	180	179	178	177
	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1
80	176	175	174	173	172	171	170	169	168	167
	B0	AF	AE	AD	AC	AB	AA	A9	A8	A7
90	166	165	164	163	162	161	160	159	158	157
	A6	A5	A4	A3	A2	A1	A0	9F	9E	9D
100	156	155	154	153	152	151	150	149	148	147
	9C	9B	9A	99	98	97	96	95	94	93
110	146	145	144	143	142	141	140	139	138	137
	92	91	90	8F	8E	8D	8C	8B	8A	89
120	136	135	134	133	132	131	130	129	128	
	88	87	86	85	84	83	82	81	80	

ADDING AND SUBTRACTING NEGATIVE NUMBERS

Notice in the above examples that -1 and +1 add up to zero as does -49 and +49--a very good reason for doing a 2's complement to negate a number. Now, if you recall your algebra--you do don't you? Anyway, when you subtract a negative number you change the sign and add. In m/c that means 2's complement and

add. If you remember only that you will have no trouble subtracting negative numbers. Adding negative numbers just makes the results more negative--which means that if you run through the zero on the bottom side you have a number LESS than -255 and you have to go to word length numbers carrying the extra bit into the high byte and complement the high byte and 2's complement the low byte. This may be a bit confusing at this point but I have put it here to be reviewed when you are referred back to this chapter later in the book.

NUMBERS THAT AREN'T REALLY NUMBERS

Kindly open your Operators Manual to page 240. That's the book you got with your computer so I know you have a copy..somewhere. You are going to be sleeping with pages 239-245 and pages 262-265 so it would be a good idea to get copies made of these pages and put them inside plastic folders to save wear and tear on your Manual. You may also want to copy page 252 if you plan on doing a lot of screen work.

1. CHARACTER CODES

In Basic you learned that, "PRINT CHR$ 65" will give you a capital A on the screen. On page 241, the number 65 has an "A" in the CODE column. 66 gives you a CAP B and 32 a space. All the numbers from 165-255 will give you the appropriate token spelled out. Therefore, all the numbers from 32 to 255 are printable. Those under 32 are not, giving the familiar "?". In m/c we can even use these CONTROL characters in PRINT operations. Notice that the PRINT comma (6) is a different code than the comma (44). The PRINT comma is our TAB 16 or TAB 0 half line move.

2. PIXELS

In Basic you learned how to design your own graphics by designating eight 8 bit numbers as a character, symbol or drawing. You now know enough about memory storage to figure out that the computer uses 8 bytes of binary to store the character. If you had a good Basic course you would even know where they are stored. You would also know where the display file is kept. For those of you not so fortunate, don't worry, just read the next chapter. Pixels are stored as numbers. Another example of a number that isn't a number.

3. INSTRUCTIONS

A. TOKENS. Basic instructions like PRINT (CHR$ 245) are stored as single numbers as well. You were religiously instructed never to type in all the letters of PRINT but to hit the right token key. In this respect your 2068 saves a lot of space as it only uses one byte to store the word PRINT rather than 6 (extra space for the space after PRINT). It also lets your computer run faster as it only

has to decode one number rather than a string. The first space after a line number or a ":" is always a token.

B. MACHINE CODE. The instructions we are to learn all about (assembly language) like: EX AF, AF' all have to be encoded to numbers. EX AF, AF' has the code number 8. The 8, not EX AF, AF' is what is stored. All machine code is nothing but a string of numbers--some of them instructions, some are real numbers, some are symbol or pixel numbers.

When we do a "RANDOMIZE USR #" or a "PRINT USR #" or a "LET A = USR #" or a "LIST USR #" (# = address) statement in Basic we are telling the computer to set its Program Counter, a special counter that keeps track of what address is holding the next instruction, to the address we give it and start doing the instructions from there on. There is NO BASIC INTERPRETER in the computer. It is just reading machine code that causes it to operate in a manner we call Basic language.

BCD NUMBERS

BCD (Binary Coded Decimal) numbers are numbers that are not true binary numbers. Remember that we said the nybble (4 bits) was just big enough to hold one decimal digit? Therefore, in BCD the high nybble is a true binary number and the low nybble is a true binary number but the combination is not. An 8 bit byte holds 2 decimal numbers from 0 to 9 in each nybble, 00 to 99 in each byte. 12345678 in BCD is 12, 34, 56, 78 in separate bytes but looks like 0001 0010, 0011 0100, 0101 0110, 0111 1000 in binary bits spaced into nybbles. The 2068 only uses binary coded decimal numbers at one point in floating point calculations when converting a binary number back to decimal. This is only temporary as it always stores numbers one to the byte in ASCII coded format or true binary.

THE SLUG

After writing a line and hitting ENTER, your computer edits the line before putting it in the program, a feature you don't really appreciate until you work on another computer and get all those syntax errors when trying to run a program. Another thing the computer does before entering a line is SLUG the number. If you want to know where I got this word check code 14. When it detects a number it goes to the end of the number and puts in a 14 code, sends the digits it has found (together with any sign, decimal point and E)to the floating point calculator for encoding into a 5 byte long number which is then inserted after the slug indicator. The first byte of the number is the exponent of the number telling the computer how many binary places left or right of the first binary digit to put the decimal point (I suppose we shouldn't use the term "decimal point" when talking about binary numbers but it is used for the same purpose as in

decimal--to differentiate the integer from the fraction.) The next 4 bytes are the 32 most significant binary digits of the number. The computer has done some of the calculations that it would normally have to do when running the program. Obviously this speeds up execution of the program.

However, we never see these 6 bytes printed to the screen when we LIST the program as the print routine detects the 14 SLUG code and skips it and the next 5 bytes. We will see how this works when we look at the storage of a typical Basic line a little later in the book. The Slug is another example of a number that is not a number.

DECODING THE SLUG

Fractional Binary Numbers: Before we can decode the slug we have to discuss fractional binary numbers. For our example, let's assume the decimal point is at the left of the high bit. The bits then have the values:

1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512
.5	.25	.125	.0625	.03125	.015625	.0078125	.00390625	.001953125

The decimal equivalents are given under the fractional ones. The fractions should look familiar--they are the same numbers we used for whole numbers only now they are in the demoninators. To help us convert fractions and whole numbers to binary we will use the table on the next page.

We have only listed the 48 binary digits either way from the decimal point. The table, of course, can be extended. We would have to that if we wanted to encode numbers like 6.023E+23 or 9.1085E-28.

You are also wondering why I extended the decimal numbers out to the very end when we all know that the 2068 is only accurate to 10 places. The reason is that if we don't carry out subtractions or additions to the bitter end errors creep into the numbers. We use the abbreviations, 9 0's and 12 0's to condense the fractions in the bottom part of the table to reduce the length of numbers for printing on an 80 column printer.

Let's convert 0.1000000000 to binary. This looks like a simple number but watch what happens. Take an empty sheet of paper and write down 0.100000000 on the top. Go down the table writing zeros for each number we pass because it is bigger than our number. In our case, our binary number would start with .0001 as we go down to .0625. Subtracting leaves us a remainder of .0375. Again we look for a number equal to or just smaller than our remainder. We find it in the next number at .03125. Our binary beomes .00011 with a remainder of .00625. We skip two numbers to get to .00390625 so our binary is now .00011001 with a remainder of .00234375. On to .000110011 binary using .001953125 for a remainder of .000390625. A few more binary bits to .000110011001

BINARY BIT CONVERSION TABLE

1	.5
2	.25
4	.125
8	.062,5
16	.031,25
32	.015,625
64	.007,812,5
128	.003,906,25
256	.001,953,125
512	.000,976,562,5
1,024	.000,488,281,25
2,048	.000,244,140,625
4,096	.000,122,070,312,5
8,192	.000,061,035,156,25
16,384	.000,030,517,578,125
32,768	.000,015,258,789,062,5
65,536	.000,007,629,394,531,25
131,072	.000,003,814,697,265,625
262,144	.000,001,907,348,632,812,5
524,288	.000,000,953,674,316,406,25
1,048,576	.000,000,476,837,158,203,125
2,097,152	.000,000,238,418,579,101,562,5
4,194,304	.000,000,119,209,289,550,781,25
8,388,608	.000,000,059,604,644,775,390,625
16,777,216	.000,000,029,802,322,387,695,312,5
33,554,432	.000,000,014,901,161,193,947,656,25
67,108,864	.000,000,007,450,580,596,923,828,125
134,217,728	.000,000,003,725,290,298,461,914,062,5
268,435,456	.000,000,001,862,645,149,230,957,031,25
536,870,912	. 9 0's ,931,322,574,615,478,515,625
1,073,741,824	. 9 0's ,465,661,287,307,739,257,812,5
2,147,483,648	. 9 0's ,232,830,643,653,869,628,906,25
4,294,967,296	. 9 0's ,116,415,321,826,934,814,453,125
8,589,934,592	. 9 0's ,058,207,660,913,467,407,226,562,5
17,179,869,184	. 9 0's ,029,103,830,456,733,703,613,281,25
34,359,738,368	. 9 0's ,014,551,915,228,366,851,806,640,125
68,719,476,736	. 9 0's ,007,275,957,614,183,425,903,320,312,5
137,438,953,472	. 9 0's ,003,637,978,807,091,712,951,660,156,25
274,877,906,944	. 9 0's ,001,818,989,403,545,856,475,830,078,125
549,755,813,888	. 12 0's,909,494,701,772,928,237,915,039,062,5
1,099,511,627,776	. 12 0's,454,747,350,886,464,118,957,519,531,25
2,199,023,255,552	. 12 0's,227,373,675,443,232,059,478,759,765,625
4,398,046,511,104	. 12 0's,113,686,837,721,616,024,739,379,882,812,5
8,796,093,022,208	. 12 0's,056,843,418,860,808,012,369,689,941,406,25
17,592,186,044,416	. 12 0's,028,421,709,430,404,006,184,844,970,703,125
35,184,372,088,832	. 12 0's,014,210,854,715,202,003,092,422,485,351,562,5
70,368,744,177,664	. 12 0's,007,105,427,357,601,001,546,211,242,675,781,25
140,737,488,355,328	. 12 0's,003,552,713,678,800,500,773,105,621,337,895,625

and a remainder of .000146484375. Counting the bits we have in our binary number we are up to 12. Looking at our remainder we only have 3 place accuracy (3 zeros). Fortnunately, we see a pattern developing in our binary number which indicates that it has a repeating set of numbers and will NEVER come out even. That is, in fact, the case. Our binary number will look like:

.000110011001100110011001100110011001 with a remainder of:
0.000,000,000,100,708,386...

Counting the zeros in our remainder, we find that we have 9 place accuracy. Exactly as advertsied, the 2068 only is capable of 9 place accuracy. Well, yes and no, sometimes it's less.

Try, LET z = 1.001010010010 01: PRINT z. The answer was 1.001001. That's only 7 digits long. You did get 9 digit accuracy as the number would be 1.00100100 but the trailing two zeros are never printed. The 2068 never prints trailing zeros which is a nuisance when it comes to dealing with dollars and cents amounts.

Try, LET w = 123456789: PRINT w. You get 1.2345679E+8...8 digits long. Whoops, an error! Instead of getting 1.23456789E+8, our answer was truncated at 8 digits and rounded. The 2068 rounds whenever the truncated number is greater than 5. Therefore, although the 2068 has 9 place accuracy, you don't always get 9 digit numbers.

The true mathematician could have told us that it would take approximately 3.32 binary digits for every decimal digit. Therefore, 32 bits of binary is only 9.63 decimal digits worth.

Going from binary number is easy with the use of the table. Start at the decimal point and write down the integer of the "1" bits only and then add them up. Then do the fractional part by again writing down the fractional equivalents of the "1" bits and add them up as well.

Try these:
Encode to binary: .444444444, 321.798 and 1,678.79.
Decode to decimal: 101.10110110110110110110110110110,
1110000000001011.000011110000111, and 111.00111000000011111111
Round the decimals to the 10 most significant figures.

DOLLARS AND CENTS

We learned in Basic that to round a number to dollars and cents

ANSWERS: .444444444 = .01110001 11000111 00011100 01101011,
321.798 = 10100000 1.1100110 00100100 11011101,
1678.79 = 1101001 110.11001 01010001 11100110,
Binary to Decimal: 5.71418299, 113699.059, 7.21894318

we do a:

```
LET a = (INT((a +.005)*100)/100)
```

where "a" is the number we want to round to the nearest penny. However, try the following values of "a" and see what you get when you PRINT a: 0.005; 2.000 and 1.005. I got .01, 2 and 1 repectively. Well, the first two are right although the 2nd answer didn't print the ".00". The third answer is even wrong as it should be 1.01! Try 1.005001 and you get the right answer. It's only the value 1.005 that comes out wrong.

If you want the 2068 to print even dollars with trailing ".00" you have to trick it into doing so. The following program will do it for you (except for 1.005). A is your variable.

```
 5 LET a = (INT ((a +.005)*100)
10 LET a$ = STR$ a
15 PRINT a$( TO LEN a$-2);".";a$(LEN a$-1 TO )
20 LET a =a/100
```

Line 20 is not necessary if you have no more calculations to do on your number. If you want your printout in neat columns, you have to do a calculated TAB based on LEN a$.

By now you have also noticed that the 2068 doesn't print numbers with commas every 3 spaces. In fact, entering numbers with commas in them will give you a syntax error. The 2068 doesn't have a PRINT USING funtion like many computers have. Of all the missing commands that would be beneficial I have yet to see someone do this one. However, once again, by putting the number into a string we can trick it into inserting the commas where we want them. We have that extra piece of information called the LEN of the string to help us out. We leave it to the student to modify the above program to print a "$" in front of numbers and add commas every 3rd digit.

A further caution about INT. It always rounds down. Thus 1.99 becomes 1; 0.999 becomes 0 and -1.999 becomes -2 as does -1.01.

THE SLUG EXPONENT

In our example of fractions, we didn't have a full number ahead of it. From the examples I gave you we combined integers with fractions. The decimal point occurred anywhere in the binary byte. That is an impossible situation as EVERY BYTE MUST BE AN INTEGER. We get out of this situation by moving the decimal in front of the first significant bit--the first "1" and use an exponent byte to indicate how many places we moved it. That means that numbers smaller than .5 are going to have the zeros between the decimal and the first "1" chopped off as we move the decimal right. For full numbers the decimal will have to move left. In other words, we need positive and negative exponents. BUT, it's not the negative numbers we just learned about. This time Sin-

clair uses 80H (128D) as zero with numbers greater than 128 as positive and numbers under 128 negative (127 is -1).

If you remember, 0.100000000 in binary started as .0001100110... The first significant digit is the 4th binary number. Thus we chop off the 3 leading zeros and move the remaining digits left 3 bits. The exponent becomes 128 - 3 = 125.

What is the exponent for 31, 11111. binary? 128 + 5 = 133. The 32 binary digits are padded out with zeros as .11111000 00000000 00000000 00000000. We show the position of the moved decimal point.

SIGNED FLOATING POINT NUMBERS

Since the first bit of the binary part of the number is always a 1, this position is superfluous. The 2068 takes full advantage of this fact and stores the SIGN of the number there...a "0" for positive and a "1" for negative. Our number 31 becomes .01111000 in its first byte. Our decimal .1 becomes .01001100 in its first byte.

Let's convert our .1 to slug form. We know the exponent is 125 and the binary part was .01001100 11001100 11001100 11001100 (we had to add 3 more digits on the end when we chopped off the 3 leading zeros.) Converting the bytes to decimal gives us the slug 14, 125, 76, 204, 204, 204. Remember slugs start with a 14 which really isn't part of the number.

TRANSLATING SLUGS INTO NUMBERS:

The 2068 uses two notations for slugs. We have just given you the floating point notation. Integers are stored as 14, 0, 0, low, high, 0. Translation is direct.

Since you have probably PEEKed these numbers, they will be 5 decimal numbers after a 14. The first after the 14 is the exponent. Skip that for the time being and convert the next 4 bytes into binary. Now look at the leading bit. If it's a zero convert it to a "1", if a 1 leave it and write a minus sign in front of the bit. Now subtract 128 from the exponent and move the decimal that many places right if positive or add that many zeros in front of the first byte if negative. Now refer back to the table on page 16 and convert the binary to decimal.

Alternate translation: Sometimes the above method gets to be really horrendous as one even moves off the table. We can translate the binary part of every number as a whole number starting at bit 32 in the table. Once we have this number we now adjust the number for the SHIFTED exponent. Take the exponent and subtract off (128 + 32 = 160)--the extra 32 is because we started conversion at bit 32 assuming our exponent was 32. With our remainder go down the table that amount of numbers and if negative read the fraction (if your exponent was still positive, read the

integer number). MULTIPLY by this number. If you have a good hand calculator you can probably do it on that without difficulty unless you have a really big or small number.

The student will have surmised from all this that the same binary number can represent 256 different numbers just because there is a change in the exponent. The exponent of 126 turns our same binary number that gave us 0.1 into 0.2. An exponent of 127 would give us 0.4. 128 would give us 0.8. 129 would give us 1.6, etc.

SCIENTIFIC NOTATION

Rememer when we did LET w = 12345678 : PRINT w, and got an answer of 1.2345679E+8? We get this scientific notation every time a number is more than 8 digits long--not like your Manual says over 10^13! You will remember from Basic that the "E" means "times 10 to the power of". What it really says is move the decimal point over that many places--left if negative, right if positive.

In the 2068 ROM there is a check exponent routine that finds how far away from 128 the exponent is and, if greater than 32 either way, automatically goes into E notation. It is very important to keep this in mind when dealing with dollars and cents, as in doing bookkeeping where rounding and truncating numbers is a "no-no". Once you get to about $100,000.00 there is a distinct possibility of messing up the number. It really doesn't take a very big business to have assets over that amount.

LIMITS OF NUMBER SIZE

It is now obvious that 2^127 and 2^-128 are the largest and smallest numbers that our computer can ever handle as that is the point where the exponent hits 255 and 0 repectively. These numbers are 1.7014118E+38 and 2.9387359E-39. Big but not huge. The 2068, like any other computer, can be made to handle much bigger numbers but not without writing your own m/c program to do so. We discuss the problems of doing this in the chapter on floating point.

DOUBLE PRECISION NUMBERS

Some computers have less precision than 8 decimal numbers in their original setups but then add double precision numbers that allow accuracy without rounding out to 16 or more significant figures. The 2068 doesn't have this feature much to the irony of some owners. What obviously has to be done is that a double precision floating point calculator must be written using an 8 byte mantissa (in machine code of course as Basic is too slow). This is no simple task and not a project for even the most ardent of machine code novices. The routine to print a floating point number from our binary data is not simple to understand much less rewrite.

CHAPTER 2

MEMORY MAPPING

TYPES OF MEMORY

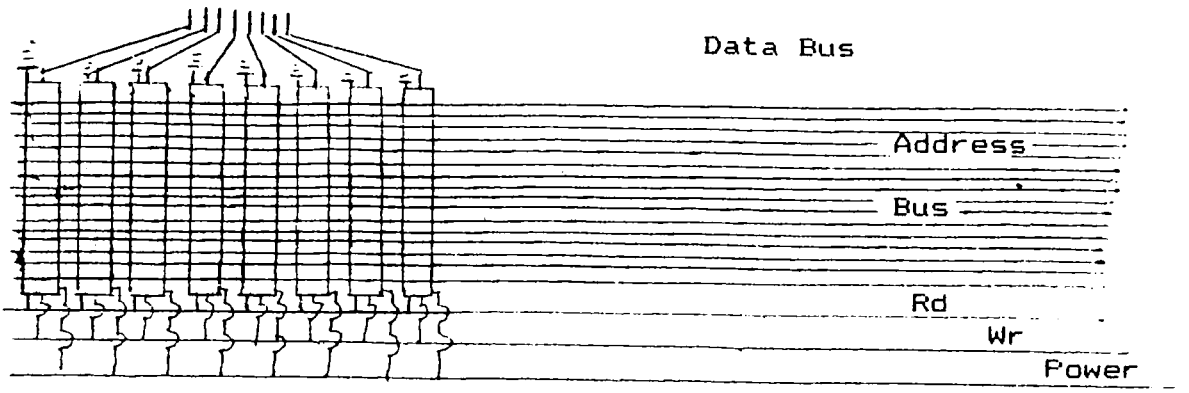
It's about time to find out where things are stored in our computer. Your 2068 has 64k of RAM (Random Excess Memory). This is the maximum amount of memory it can address at any one time. If you recall from the last chapter, the largest number your computer can handle in two byte word form was 65535 (adding zero makes 65536 bytes). That little k after the 64 means kilo or 1000 (from the metric system of measurement). Well, 64,000 isn't 65536. In computerese it is. The "k" is really 1024 bytes long--which happens to be the closest power of 2 (2^10) to 1000. 1024 x 64 is 65536.

RAM can be written to or read from. That means it can be changed as you desire. It would soon fade, much like a TV tube picture does, if it wasn't constantly refreshed to keep it holding its memory. Naturally when you turn the computer off all RAM memory is erased.

Bank Switching: One of the special features of the 2068 is its ability to switch to a different "bank" of memory and use that instead. In fact, it can be programed to switch to anyone of 256 different 64k banks if you want to add that much more memory to the back of the 2068. This would give you a whopping 16,777,216 bytes of memory. Banks are further subdivided into eight 8k CHUNKS. These chunks are labeled 0 to 8 like the bits.

What does a 64k RAM look like physically? Generally manufacturers make memories 64k long and only 1 bit wide. This means you have to use 8 of these chips side by side in parallel to get your 8 bit byte wide memory. Your computer uses an 8 bit wide data bus (8 lines side by side) to get data to and from memory. It also uses a 16 bit address bus (16 lines) to tell each memory chip what address it wants read or written to. Thus each memory chip will have all 16 address lines attached to it but only one data line from the data bus. Now since we have to read and write to RAM each chip further has to have a READ line and a WRITE line running to it. And of course we need a refresh line, a power line and a ground line. Generally the address lines are made to run from chip to chip underneath them. The whole assembly would look something like the diagram on the next page.

The appropriate signal goes down the address lines, another signal goes down either the read or write line telling the chip to



send (read) or change (write) the memory address given. In the case of write, the data lines contain the material to be entered into memory. This kind of processing is called parallel processing as all 8 bits are handled at once. It is obviously at least 8 times faster than the one-after-the-other handling of bits that your tape recorder uses (serial processing).

In fact, the tape recorder is even slower as what it really does is record 2 tones--one means zero, the other 1. They are an octave apart in frequency. It's the switching of tones that further limits the transmission speed of the tape recorder and lowers it down to only 1200 baud (bits/second). The T/S 1000 and the T/S 1500 are even worse at 300 baud. Those of you who have dot matrix printers or disk drives will have ribbon cables for a "centrononics" parallel type port sending 8 bits in parallel to the device at 8X the baud rate. In addition, the signal is electrical and not sound so can be a lot faster. No wonder disk drives load and save programs so much faster.

Okay, if RAM is erasable every time we turn the computer off, where are the permanent instructions kept? They are kept in a different type of memory unit called ROM (Read Only Memory). Your computer has two ROM chips. One is 16k by 8 bits and the other is 8k by 8 bits. As the name says, we can't write or change this memory, only read what is there. It is permanent, errors and all and, of course, doesn't erase when the computer is turned off. Because it can't be changed we can say that ROM is "etched in stone" more or less.

Before discussing ROM further, there is a 3rd type of memory unit you may have heard about called EPROM (erasable programable ROM). It is essentially a ROM (permanent) until you erase it with ultraviolet light and then "burn" in a new program. The burning process is done by using a higher voltage electrical current to set the memory. Your computer can't do this as it takes a special device. New units are electrically erasable.. sometimes called EEPROMs.

THE ROM MEMORY BANKS

Each bank of memory is given a number which corresponds to its port number. The "home" or main RAM is bank #255. The first 16k

of the home RAM is "shadowed" by the 16k main ROM. This means that the ROM instructions are for all practical purposes sitting in the lower 2 chunks of the home RAM--only it still can't be changed.

Only when you are using your tape recorder to SAVE, LOAD, or VERIFY a program or change VIDEO MODE do you use the 8k EXTENDED ROM which is located in Bank #254. There is about 2k of free space in the extended ROM.

Now, the cartridge slot, under the door at the right of your keyboard, uses Bank #0 known in computerese as the "dock" bank. It is only these 3 banks that can be called by the 2068 as it comes from the factory...all the rest take special progamming.

EXTENDED ROM

The lowest 4k of the extended ROM contain the casette routines and the CHANGE VIDEO routine. You can't use the change video routine as it is as it contains a fatal error. In fact all the big errors in your computer occur in the extended ROM. The rest of the extended ROM contains the Function Dispatcher and Bank Switching routines which are programs to assist the computer in transferring data between banks and call different banks. Unfortunately, some sections are so full of errors that they are useless. If you have the "TIMEX 2068 TECHNICAL MANUAL" the errors are listed and can be corrected--or you can read further as a program to correct the errors is given later on in this book. The reason you can correct this routine is that it is not used from extended ROM but is written to 25088 of the home RAM which is then rewritable. Don't use the Function Dispatcher and Bank Switching routines without making these corrections. Unfortunately, these routines have to be switched to Chunk 7 of home ROM when using dual screen modes and there are other mistakes which reintroduce more errors each time the shift is made. These also must be corrected out each time.

Since the Function Dispatcher and Bank Switching routines actually get transferred to home RAM, there is only 4k that is added to the 64k of home RAM to give you 68k--which is where the 68 comes from in the name of the computer.

THE CARTRIDGE BANK

When you turn your computer on it takes a few seconds to set itself up before you get the copyright notice. In that time it checks to see if you have put in a cartridge by checking for setup data at the start of chunk 4 (32768) to see if a LROS (Language ROM Orientated Software)--written in machine code, or an AROS (Application ROM Orientated Software)--written in Basic, is present. AROS cartridges can only use the top 32768 bytes of their ROM as they need the routines, display file, etc. of lower memory to run the Basic. With LROS you can use every- thing but chunk 3. However, many LROS cartridges also reserve chunk 2

which contains the display (TV) file.

Don't pull a Bill!

It can be very expensive. Your manual says always to turn off the computer before you plug or unplug anything and they mean it. If you look at the spacing of the contacts, it doesn't take much misalignment to contact the wrong connections. There are voltages on some of those lines and 5 volts is enough to blow an IC (integrated circuit) without difficulty. Poor Bill, one of our S.M.U.G. members, blew his whole computer when he tried plugging in his AERCO Printer Interface without first turning off the computer.

The same holds for plugging and unplugging cartridges into the slot...only more so. If you are lucky enough to get the cartridge in place without blowing anything, the computer won't know it's there until you reinitialize it. It only checks for the cartridge upon turnon. It then knows its there and uses the ARSBuffer to store its data and call the next program line. You can restart the computer without turning it off by doing the command, "RANDOMIZE USER 0". The computer starts reading instructions at the beginning of bank 255.

THE MEMORY MAP OF HOME RAM

We have discussed everything else except the home RAM bank or the "working" bank. This one is going to get quite complex so it is time to open your Operators Manual to page 254. (See how much information is stored in those appendies!) There you have 2 maps of the home RAM. The left one is for the single screen (normal) mode, the right for the Dual screen modes.

Now, make some additions to the left map. On the left of the line separating Home ROM from Display File 1 put 16384-4000H (That's the decimal-hexadecimal notation for the start of the Display File.) Draw a line 3/4 up the display file and write above that line ATTRIBUTES. To the left of the line you drew put 22528-5800H. To the left of the next 4 hex numbers add the decimal equivalents in ascending order 23296, 23552, 24576, 25088. Opposite the line below ARSBUF write 26688-6840H. Opposite the line below CHANS write 26660-6824H. Opposite the line below PROG write 26710-6856H. Cross out that 6840H as that's wrong. In the space between STKEND and RAMTOP write SPARE. In the space above that write YOUR CODE. Opposite UDG write 65368-FF58H. At P-RAMT write good old 65535-FFFF.

On the right map write 31488 opposite 7B00H. Opposite F7C0H write 63424. Opposite F9C0H write 63936. On the bottom of the page write RAM RESIDENT CODE = FUNCTION DISPATCHER & BANK SWITCHING.

Now you have a useable Memory Map. Making a copy of it and keeping it between plastic is a good idea although not quite as use-

ful as the other pages I told you to copy earlier.

CHUNKS 0 AND 1

Since every 2000H is 8192 bytes or 8k or 1 chunk, you can see that the 2 lowest chunks of Home RAM are used by the operating code instructions to run Basic programs. We will be learning exactly what is in there a little later in the book. At the present time there is only one thing I want to add--the CHARacter Table is at 15616-3D00H. This is the table of pixels that the computer uses to write the TV screen. Each character has 8 bytes worth of pixels starting with character 32 (space) and ending with character 127 (copyright). There are no pixel bytes for the graphic symbols as the computer generates them from the graphic codes for these characters themselves which, incidentally, takes less space then the pixel bytes themselves. It was quite a shock for me not to find any as I started writing my first machine code program for the 2068.

CHUNK 2--THE DISPLAY FILE.

Is really the TV file. You learned in Basic that the "normal" TV mode prints 32 characters per line (0-31) and has 22 lines (0-21). But you have to add the lines on the bottom of the screen (2 more) so you really have 24 lines.

From your Basic UDG (remember User Defined Graphics) you learned that a character is made up of 64 pixels arranged in an 8 across by 8 down matrix. They were stored in 8 across lines. This is called pixel mapping and allows us to design our own characters be they letters, symbols or pictures. In the T/S 1000 we used character mapping--the display file only contained one byte per character. That wouldn't have been so bad as long as we could have told the computer to switch to a different character pixel table but we couldn't even do that. Additionally, the Z80 CPU (Central Processing Unit) had to stop every 1/60th of a second to refresh the screen. To do this it looked at the display file, then found the pixels in the character table and sent them to the TV. No wonder things were SLOW. We could tell it to forget about the screen and just calculate by using FAST. In the 2068, the pixels are already arranged for another chip to send to the TV screen so there is no need for slow or fast.

Now, 32 characters per line times 24 lines times 8 pixel bytes per character is a total of 6144 bytes just to print the TV screen. On top of this we have to add one attribute byte per character or 32x24 = 768 attribute bytes. That's a grand total of 6912 bytes. Contrast this to the 768 bytes of screen plus 32 end of line bytes for 800 bytes for the T/S 1000. At least they fixed one thing on the 2068--the Display File is in a fixed location, not floating above the Basic program. It also had the nasty habit of collapsing if less than 3.5k of free memory remained-- remember? The T/S1000 was never made to handle more than 32k of memory in its original design...others have found

ways around this limitation.

THE 64 CHARACTERS PER LINE SCREEN

Going to the dual screen mode you have to use the right map and get two display files. But I'll warn you it isn't what you expect. Yes, each character has 8 pixels but DISPLAY FILE 1 holds all the EVEN characters and DISPLAY FILE 2 all the ODD characters as defined by TAB. Therefore, every other character is in DISPLAY FILE 1 and every other in DISPLAY FILE 2. In addition to the errors in the Change TV Mode routine which doesn't allow it to work, we have another suprise for you. Your computer only supports DISPLAY FILE 1. You have to write machine code routines to make CLS, TAB, AT, LIST, PRINT and COPY work from screen to screen.

THE 80 CHARACTERS PER LINE SCREEN

This is even worse. You have to squeeze 16 more characters into a line. In 64 column mode you had 64x8 or 512 pixels across the screen. Dividing by 80 gives you 6.4 pixels per character. Well, fractions of a pixel don't work so we have to round down to 6 and then redesign the characters to 5 pixels wide, reserving the 6th pixel as a space between characters. Let's see, 80x6 is only 480 pixels per line. 32 are left over. Divide these 32 by 8 gives 4 bytes worth. To center we need 2 empty bytes in front of each line and 2 empty in back. We have to start with DISPLAY FILE 1 position 1 (position 0 being the first). Now, 6 bits of character 1 go into position 1 of Display File 1 together with the first two bits of character 2. Display File 2 position 1 gets the last 4 bits of character 2 and the first 4 bits of character 3. Back to Display File 1 position 2 for the last 2 bits of character 3 and all of character 4. Start Display file 2 position 2 with character 5.... It's going to be some time before you write a program that can do that. One only has 8 ink-paper colors available in dual screen mode as the attribute file isn't used. The whole screen has to be the same two colors.

THE HI-RES GRAPHICS SCREEN

Display File 1 holds all the pixels for a 32X24 normal screen but Display File 2 has an attribute for each pixel byte. This still limits you in doing beautiful art. You have only one ink and one paper color per 8 pixels and they are not in a square but a line. Again, it doesn't work without machine code.

I guess all these things were to be additions to the 2068 for future expansion...there still is about 2k of empty space in extended ROM. Some of the techniques are discussed in the "TIMEX 2068 TECHNICAL MANUAL". A beautiful plan but no followup. So we have to do it ourselves. We are, slowly but surely, as that is what users groups are all about.

THE PRINTER BUFFER

The printer buffer is only used for the T/S 2040 printer. It is only 256 bytes long which is just long enough to send 32x8 bytes of pixels--just long enough for one line. If you have ever stopped your printer in midline you will see that that is the way it works--one pixel line across the paper at a time. Dot Matrix Printers generally use their own buffers which makes this space available for other uses--like the printer driver routine itself as OLIGER does. Like AERCO, HUNTER, and WOODS, just to mention a few more, he is a 3rd party (not TIMEX related) supplier of hardware and programs. Much of what we have comes from these dedicated geniuses.

THE SYSTEM VARIABLES

If you were wondering where I was getting all those numbers from that I have been spouting about in this chapter, it's from this table located on pages 262-265 of your Operator's Manual and which I asked you to make copies of.

It is this table of 1046 bytes (23756-24297 are reserved for additional variables) that helps the computer keep track of almost everything it needs to know. It can be PEEKed at anytime--inside programs and/or in command mode. Unfortunately, it's written in computerese and takes a little knnowledge and interpretation to figure out just what some of those abbreviations mean. Then you are still at a loss unless you have a copy of the "TIMEX 2068 TECHNICAL MANUAL" or "The Timex/Sinclair 2068 ROM Manuscript" to help you out. The "Technical Manual" was available from Timex--Products Service Center, Box K, 7004 Murry St., Little Rock, AK 72203 for $25--the same place you used to send your computer to for repairs. HOWEVER, since then they have changed the repair outlet to: T/S Users Group of Cincinnati. Call (513) 271-5575, Jack Roberts before sending. The Manuals are available in limited supply from them. If they run out, there will be a slight delay for another printing.

"The Complete Disassembly of the 2068 ROM" is available through S.M.U.G. (Sinclair Milwaukee Users Group), Box 101, Butler, WI 53007. Price is $16.95 + $2.50 S&H. Wisconsin residents kindly add 5% sales tax.

We discuss the System Variables in full detail in Chapter 3 as it's quite an extensive discussion. Let's move on to the rest of the memory map.

MACHINE STACK (24576-25087)

I should say from 25087 to 24576 as the stack works from the top down. 512 bytes long, it is this section of memory that the CPU uses to store numbers, always in pairs, for further use. It also uses this stack to store data that it will need when it is going to transfer them to another bank of memory where it will need

them when working on routines there.

A simple example: In Basic, whenever you are in a routine and want to call a SUBROUTINE you do a GOSUB. Since the variables you have been using are stored in the variables table, you don't have to save any numbers for future use as they are all stored there and updated as needed. All that has to be remembered is the return address...that address to come back to when the return statement is encountered in the subroutine...this is done in Basic by updating the OLD PROG LINE and SUB OLD PROG LINE.

In machine code it is more primitive. A machine code routine may CALL (equivalent to a GOSUB) another routine. If one has numbers in the various registers that one has to save to continue with the routine after the RETURN from the CALL, then a convenient way of saving the numbers is to PUSH them onto the stack before making the CALL. They are always PUSHed onto the stack in pairs like: AF, BC, DE and HL (or IX and IY). If you don't need the values anymore to continue after the CALL you don't have to PUSH them. Finally, when you make your CALL, the CPU itself PUSHes one more set of values onto the stack--the address of the next statement, i.e., the RETURN address to come back to when it sees the RETURN statement. The CPU will POP the bottom two numbers off the stack at this point and use that as a RETURN to where it came. Of course you may use the stack to store numbers while in a subroutine BUT make sure you POP them off before the CPU gets that RETURN statement or you RETURN to whatever address that set of unPOPed numbers would make.

You have just met your first assembly instructions PUSH, POP, CALL and RETURN. As I remind my students so often, MAKE SURE YOUR PUSHES EQUAL YOUR POPS. POPing too many values is just as bad as not POPing enough.

The stack works from the TOP DOWN. A special register called the stack pointer is set with the starting address 25088. As a PUSH or CALL is encountered the first value goes into 25087 and the next value into 25086 as the SP (stack pointer) is DECREMENTED (decreased by 1) twice from 25088 to 25086. As the POP or RETURN is received, it reads out the value at the stack pointer and the one above it to the appropiate registers, and INCREMENTS (increases by 1) the value of the stack pointer twice. NOTE: The values are still in those addresses and will only be overwritten by the next PUSH or CALL statement. This important fact can sometimes be useful in programming.

RAM RESIDENT CODE (25088-26688)

Better known as the Function Dispatcher and the Bank Switching routines. It is a disgrace to TIMEX as it is full of errors. Chapter 10 discusses the Function Dispatcher and Bank Switching routines in full detail after it shows you how to correct it. Let it suffice at this point just to mention that it is this set of routines that allows the 2068 to switch from one bank of mem-

ory to another and call routines in other banks. It is also used to run all Cartridge programs.

ARSBUF (26688-?) AROS LINE BUFFER

No space is normally reserved for the AROS LINE BUFFER. When a Cartridge is inserted under the front door of the computer and the computer senses that an AROS type cartridge is present, it moves up CHANS to provide room for a buffer. When running an AROS cartridge with Basic in it, the computer finds the next line to be executed in AROS ROM and copies it down to this buffer. It then switches back to the home ROM and executes the line. If the line has a READ statement in it, the computer goes back to the DOCK bank (0) and finds the appropiate DATA line and copies that to the ARSBUF as well. Since there is no program located in the PROGram part of the home RAM when running a cartridge, the computer further starts the VARiable table at 32553. It does NOT float upwards so care must be taken not to write too long an AROS Basic line or an AROS Data line as you may start over writing the VARS table.

CHANS (26688-26709) CHANNELS TABLE

Without a cartridge, the CHANNELS TABLE resides here. With a cartridge, it is moved up depending upon the length of the AROS line being copied. It is this table that is consulted by the STREAMS to find out how to route data--to the screen or the printer.

PROGram (26710-?)

The start of the Basic Program. It extends upward as far as necessary to accommodate the full length of the program. This value can always be found by PEEKing PROG 23635-23636.

VARS (??) VARIABLE TABLE

It always immediately follows the PROGRAM and initially starts without anything in it. As the program is run, each new variable is added at the end. The table expands upwards so room must be left for this expansion. UNLIKE most other computers, your 2068 (as well as all other SINCLAIR computers) save the VARS with the program when SAVING (tape) or MOVING (disk) a program. This allows starting a program in midstream with a GOTO statement. One can also write variables directly to this table by entering them in the COMMAND mode. Many programs which are cramped for space do this with constants and "set" strings. Doing a CLEAR or a RUN always starts by clearing out the variable table before running the program which would be disaster for a program with set constants in the VARS table.

How variables are stored in this table (their codes) is discussed in Chapter 4. The start of the VARS table can always be found by PEEKing VARS 23627-23628.

E LINE (??) EDIT LINE

When you are typing in a program line it appears on the bottom of the screen and is written to this space, the space expanding as you type in more of the line. When you finally hit ENTER, the line is checked for syntax errors, the numbers are slugged and if okay and there is a line number, is inserted in the proper line number sequence of the program. If no line number, the line is immediately executed.

Should you LIST your program and use EDIT to bring a line down to the bottom screen for changes (Editing), it again goes to this area as well. Any changes you make are executed and upon ENTER the above sequence again takes place only this time the old line is replaced by the new line which may be shorter or longer than the old line. If you changed the line number, the new line may replace another line with that number or become a new line addition to the program.

After ENTERing or executing a line, E LINE is erased and the space it occupied recovered. E LINE can always be found by PEEKing 23641 and 23642 in the usual manner.

WORKSP (??) WORKSPACE

The workspace floats above E-LINE and is used by the computer to ENTER the data you are typing in from an INPUT (Rather than putting it in E-LINE and processing it as a line). When the Workspace is active, E-LINE has collapsed down to nothing. Workspace collapses to nothing when not in use as well. Workspace can be found by PEEKing 23649-23650.

STKBOT-STKEND (??)

These two areas are used by the computer to do floating point calculations. They also collapse to zero space when not in use. Since the floating point calculator uses Forth notation by pushing numbers onto a stack (this time a 5 byte wide stack and right side up), STKBOT keeps track of the start of the stack and STKEND being the top working end of the stack. Although calcuations can only occur between the two top numbers on the stack, the stack can be preloaded with as many numbers as necessary with calculations then taking place in a group rather than push a number, do a calculation, push another number etc. More is said about this in Chapter 8 where we actually go through a calculation using the stack. Again, these areas float on top of the E-LINE. Their location can be found at anytime by PEEKing STKBOT 23651-23652 or STKEND 23653-23654.

FREE MEMORY

STKEND is the end of the computer used space--above this resides any extra memory that is still left over...remember that the

computer uses more memory as the program is RUN as it is building a variable table as it goes along. The available free memory can drop very rapidly if you start DIMensioning variables. E-LINE, WORKSP and the F. P. stack require some space but can usually get by with about 2-400 bytes.

The amount of Free Memory at the momemnt, is always given by FREE. It is calculated as the space between STKEND and RAMTOP.

RAMTOP

Ramtop is the upper limit of memory available to a Basic Program as you set it. Should your Basic program expand to the point where it needs more memory than that to continue functioning, like adding another variable to the variable table, you will get an OUT OF MEMORY error code. Upon setup, RAMTOP is set at 65367 leaving 168 spaces above it for USER DEFINED GRAPHICS (UDG). Anything above Ramtop is NEVER saved with a Basic program. It, however, can be saved as code.

Ramtop can be set anywhere you like. You can thus reserve space for your machine code program and always be assured that it is never overwritten by the Basic program by lowering Ramtop. This can be done in two ways. The simplest is just to do a CLEAR followed by the address you want Ramtop set to. Your computer puts a marker at this point so this address is not available for your code but the one immediately above is. The trouble with CLEAR is that it also clears the variable table which you may not want done. The way around this is to POKE 23730 and 23731 with the low and high values of the address respectively. PEEKing these addresses obviously tells one where Ramtop is set.

Saving things above Ramtop must be done with another save as it is not saved with a Basic program. Both your code and the UDG can be saved at the same time using: SAVE "name" CODE starting address, length. "name" is any name up to 10 characters in length and is usually the same name as the Basic program. Starting address is one above your Ramtop setting and the length is calculated from there to the end of memory at 65535.

UDG (65368-65535) USER DEFINED GRAPHICS

Your own designed symbols as you learned in Basic. You are allowed 21 of them using Graphics A to U. If you don't use them they still contain CAPS A to U. But, with the start of the UDG table designated by PEEKing 23675 and 23676, one is really not limited to 21 UDG figures--just 21 at a time. And, no limit to what appears on the screen at any one time. This is because you could design a 2nd set and put them just below the 1st set. When you want your program to print to the screen from the 2nd set all you have to do is POKE 23675 and 23676 with the starting address of your 2nd set. If you want the 1st set back a little lower in the screen just POKE the same two addresses with the address of the 1st set (65368). One can alternate between as

many sets as one wants just by making sure the program is looking at the right set when told to print to the screen. Why does this work? Because your computer has a PIXEL mapped Display File. Once the right pixels are in the display file they automatically go from there to the screen. AND from the screen to the printer with COPY.

SPRITES

Sometimes a single character space is not big enough for your graphic and you will want to use several together to form a bigger graphic which you will move around the screen as a unit. These type graphics are called sprites. There is a routine in the "2068 Technical Manual" which gives some of the techniques to handle sprites in machine code. Moving sprites by Basic is slow and jerky as you have to move them a character space at a time.

P RAMTOP (65535)

Physical Ramtop is 65535 with a perfect memory. If you have a bad memory cell in your computer it can be less. The first thing that your computer does upon startup is check all HOME RAM by writing a 2 to each cell and reading it back. If it finds other than a 2, it sets P Ramtop just below the bad cell and reduces your memory accordingly. P Ramtop's location can be checked by PEEKing 23732 and 23733.

DUAL SCREEN MODE

What we have discussed above is the complete Home RAM from bottom to top. The right hand diagram on page 254 is how the Home RAM shifts with the addition of Display File 2 above the system variables. The RAM Resident code and the machine stack are shifted to high memory above the UDG. This isn't quite enough space so the machine code variables and CHANS as well as PROG are moved up a bit for the rest of the space.

CHAPTER 3

SCREEN PRINTING

THE DISPLAY FILE MAP--SCREEN MAP

From Basic UDG and Chapter 2 we found out that our Dislay File was pixel mapped, i.e., the bytes containing the print pixels were stored there, not the character codes. However, we did not discuss in what sequence these pixel bytes were stored. That sequence and the use of various print statements and commands associated with screen printing is the subject of this chapter.

We will assume a normal screen of 32 columns by 24 lines. We also know from Chapter 2 that the Display File starts as Chunk 2, address 16384. We have to have some sequence to these bytes so how about starting with Pixel 1 of Character 1 at 16384 followed by Pixel 1 of Character 2 in the next address and so forth for 32 spaces to finish the top row of pixels for the screen. (Note that in all of this we are calling Line 0-Column 0, Line 1-Column 1.) Logic would tell us at this point to continue with Pixel 2 of Character 1 and do all the "2" Pixels for the top row.

Not so. The 33 byte, address 16416, holds Pixel 1 of Character 1 of Line 2 followed in the next address by Pixel 1 of Character 1 of Line 2. Okay, we're flexible. We do all the #1 pixels for all the screen positions before moving on to #2 pixels. Right!

Wrong! We do it for the top 8 lines only and then do all the #2 pixels for the top 8 lines as we did the #1 pixels. This is followed by the #3 pixels for the top 8 lines, the #4 pixels for the top 8 lines, etc. When we finally get done with the top 8 lines we do the same with the center 8 lines and finally, once more with the bottom 8 lines. Remember the screen has 24 lines, not the 22 we are used to thinking of. Part of our Display File will look something like this, address by address per character.

```
      1     2     3     4     5                    32
L 1 16384 16385 16386 16387 16388 --- --- --- 16415
I 2 16640 16641 16642 16643 16644 --- --- --- 16671
N 3 16896 16897 16898 16899 16900 --- --- --- 16927
E 4 17152 17152 17153 17154 17155 --- --- --- 17183
  5 17408 17409 17410 17411 17412 --- --- --- 17439
1 6 17664 17665 17666 17667 17668 --- --- --- 17695
  7 17920 17921 17922 17923 17924 --- --- --- 17951
  8 18176 18177 18178 18179 18180 --- --- --- 18197
L 1 16416 16417 16418
I 2 16672 16673 16674
```

Please note that the "seam" between the top 8 and the 2nd 8 print lines occurs at 18431-2. 18432 starts the 2nd 8.

Why such a strange way of doing things? What is the logic? Is there some advantage to doing it this way? Let's see. At 32 characters per line by 8 lines we get a string of 256 #1 pixels before we start the #2 pixels. With this clue, the advanced student already knows the answer. The beginning student has to think back to Chapter 1 when we discussed holding an address in 2 byte long words. The dividing line was 256. Therefore, if we were holding our print address in the H and L registers (How nice to have registers with H = high and L = low.) of the CPU all we have to do to add 256 to our address is INCrement H. Even the beginning student can see that putting the write to HL and INC H inside a loop and doing it 8 times will get the whole character printed to the Display File.

That's nice and fast for the first character but how do we get to address of pixel #1 of character #2? We are way off base in left field from where we have to be. I suppose we could subtract off exactly 1791 to get the starting address for the next character. We are in great trouble if that character happens to be AT 8,0 which is one of the seams in the screen.

The computer doesn't know if its going to print 1, 2 or a string of characters, so instead of storing the address of the next character, it really stores an AT value and recalculates the starting Display File address from that for each chracter. Of course, it increments AT after it has calculated a needed address. Then, if you don't use a ";" at the end of a print statement, all it has to do to start a new line is increment the line number and set the column number back to zero to be ready for the next print statement.

Calculating the correct Display File address from AT must take into account the 8 line seams in the File. The top 8 lines are easy as all you have to do is multiply the line number by 32 and add the column number and 16384 to it. The second 8 needs 8 subtracted from the line number so that the same routine as we used for the first 8 lines can be used (multiply by 32, add column # plus 16384) but we also must add 2048. Similarly, the bottom 8 lines need 16 subtracted and an addition of 4096 rather than 2048 and the same routine can be used on the remaining portion of line numbers. It's beyond the power of the beginning student to see how this can be done in a minimum of bytes but advanced students should know how. HINT: Use SET for 2048, 4096, and 16384. From this it can be seen why AT begins at 0,0 and not 1,1...nothing has to be added for the start. The beginning student should start to realize at this point that thinking for machine code is somewhat different than Basic programing.

The top screen address of the next character is stored in DF CC 23684-23685 (Display File Current Character).

PLOT

Now that we know how the screen pixels are stored, how does PLOT work? From Basic, we learned that 0,0 for PLOT is in the lower left hand corner of the screen. X, the horizontal coordinate, goes from 0 (left) to 255 (right) across the screen. Y, the vertical coordinate, from 0 (bottom) to 176 (top). These coordinates are stored at COORDS 23677-23678. X in 23677, y in 23678.

How does the computer get the right pixel from these numbers?

Let's PLOT x = 142, y = 97 and see which pixel of what address must be turned on. Looking at x first, we realize that every 8 bits across is going to be another byte's worth. Dividing 142 by 8 gives us 17 full bytes with 6 left over. We are in the 18th byte.

But, we recall that the bit numbers are 7,6,5,4,3,2,1,0 and we want the 6th bit from the left (bit 2). A zero remainder would have meant bit 0 of the 17th byte.

Y is the real problem as it counts from pixel byte # 8 of line 21 on up to pixel byte 1 of line 0. It is much easier counting from the top down by doing a 175 - y. This has the effect of moving 0,0 from the lower left to the upper left corner of the screen. It does not effect calculations on x. 175 - 97 = 78. Dividing by 8 gives us line 9, pixel byte 6. Writing that in binary:

	128	64	32	16	8	4	2	1
y =	0	1	0	0	1	1	1	0
	3rd sec	2nd sec	line in section			bytes down		

Notice how the 64 bit really translate into the 2nd section of the screen and how the 128 bit would be section 3. Notice how the 32, 16 and 8 bits give us the line number in the section. Our number was line 9 = section 2, line 1 so we expect a 1. Lastly, the 4, 2 and 1 bits gives us the pixel byte to use.

Now, let's try writing our high byte of address. Referring back to Chapter 1, we note the following values of bits:

32728 16384 8192 4096 2048 1024 512 256

We remember that the display file starts at 16384. How convenient, all we have to do is turn on that bit. Our high byte looks like:

0,1,0,0,0,0,0,0

Moving down a full section of 8 lines (32x8x8) or 2048 bytes worth. Real convenient to have another bit just equal that:

0,1,0,0,1,0,0,0 .

(If our example would have been in section 3, it would mean add 4096 rather than 2048 which just happens to be a nice number also.)

The extra line (we were in line 9) will be going to the low byte of our address so let's skip that for the present. The byte number of a character goes in the 3 low bits. (Rememeber those INC H's we were talking about earlier?) So our high byte address becomes:

0,1,0,0,1,1,1,0

Translating back to decimal gives us 19968.

Going on to the low byte of our address and working with that remaining line, we must multipy by 32 which is equivalent to putting it into 3 high bits as is. (0,0,1). The 5 low bytes of the low address will be our full pixels from x = 18 (1,0,0,1,0). So our low byte address is:

0,0,1,1,0,0,1,0

Which translates to 50. Adding both address bytes gives us 20018.

There you have the whole rationale of a PLOT routine. The whole thing seems quite complex and involved with a lot of bit fiddling. Machine code is great for that sort of thing especially with the Z80 code. Those of you who know machine code should now be able to write a routine for PLOT. The routine is given in Chapter 10. The novice is not ready to take on a full fledged bit manipulation routine such as this quite yet.

If you think PLOT is bad, just think about what DRAW would involve, or if you're really serious, how about CIRCLE with the extra argument that turns it into an ellipse. For the mathematician, how about a 4 dimensioned space form unfolding into 3 different dimensions? Or? Let your imagination run. The point of all this is to get you to start thinking in different terms then you used in Basic. Machine code is really like learning a new language. Let's go on to something easier.

THE ATTRIBUTE FILE

The attribute file is part of the Display File and immediately follows the screen map. It, for once, doesn't have any fancy way of storing but merely uses one byte per character starting with 0,0 at address 22528 and going across each row in order, row upon row for 768 bytes.

Each attribute byte contains the information about that character space's BRIGHT, FLASH, PAPER and INK. (All 8 bytes are the same.) In reality, when we are done printing the character to the screen file, we are only half done. We now have to find the correct attribute byte and change that also if necessary.

The scheme for an attribute byte is given on page 252 of your User's Manual. Again it is done bit-wise. Bit 7 is used for FLASH, being 1 if on and zero if off. Bit 6 is for BRIGHT with the same notation. Bits 5, 4 and 3 are used for the PAPER color using the scheme listed in the table in the Manual. Bits 2, 1 and 0 do the same for INK. If we examine this table, we will note that the colors are formed "gun-wise". Your color TV or color monitor has 3 guns: red, green and blue. Turning on no guns gives us black. Turning on the "1" gun turns on the BLUE gun. Similarily, the "2" gun is RED and the "4" gun is GREEN. These are the primary colors. Mixing 2 colors gives the secondary colors. Red and green give yellow. Green and blue give cyan and red and blue give magenta. All 3 guns give WHITE.

Our computer only has 2 intensities of color--dull or regular and bright...because it has to handle everything in digital form. Your TV set is an analogue device which doesn't have to digitize everything so it can process a certain percentage of red, a certain percentage of blue and a certain percentage of green together with what is called luminescence (or brightness) --essentially black, to give you any hue you want. To completely digitize a TV picture, your TV would have to handle tens of millions of bytes per second...remember that the TV is refreshed every 1/60th of a second so it's 60 pictures a second. Some of the beautiful graphic display can be done with digitizing but the memory of these computers is in the multi-megabyte range.

The attribute value in present use is stored in 4 bytes in the system variables at: 23693 ATTR P(ermanent); 23694 MASK Permanent; 23695 ATTR T(emporary) and 23696 MASK Temporary.

What's the difference between Permanent and Temporary? When we do an INK 0 : PAPER 7 without a PRINT statement we are doing a PERMANENT change to Ink and Paper. When we do INK, PAPER, FLASH or BRIGHT in a PRINT statement, we are doing a TEMPORARY change-it only lasts for the duration of the statement...UNLESS we do a ";" in which case it carries over to the next PRINT statement.

What is a mask? The mask tells the computer what parts of the ATTRIBUTE to take from the TEMPORARY ATTRIBUTE and what to take from the PERMANENT ATTRIBUTE. The two masks always compliment each other--what is "om" in one is "off" in the other.

You can easilly change the attributes by POKEing in a different value. But remember you have to have ALL the values. Knowing what we already know about binary numbers we start with the INK number--as is. Then we add to it the PAPER number MULTIPLIED by 8. To turn on BRIGHT we have to add 64, right? And, to turn on FLASH, add 128. If you try this remember that nothing is going to show on the screen until you tell the computer to print something. The attribute value is changed but it isn't used until you print something.

In M/C thinking, the easiest way to get the right ATTR address is to go back to that AT, which gets updated each and every character and make the calculations from that.

OVER and INVERSE

The missing operations not included in the attribute are OVER and INVERSE. The reason for this is that once the pixels are in the Display File the Over and Inverse is not needed--it is already accomplished. Over and Inverse are found in another type of variable called P FLAG 23697 (Print Flag). But first let's talk about a "flag".

No, it's not the "Stars and Stripes" waving somewhere, and no, it's not somebody doing semaphore signaling--but you're getting warm. It's more like pennants flying on a ship signaling something like, "Gale Winds", or, "Captain is on board". A "flag" is only 1 bit of a byte. When it's "on" it means one thing and when it's "off" it usually means the opposite. Therefore, a whole byte of flags means that up to 8 flags can be stored in that byte. The 8 flags of P FLAG are:

7 Paper complement of Ink permanent
6 Paper complement of Ink temporary
5 Ink complement of Paper permanent
4 Ink complement of Paper temporary
3 INVERT permanent
2 INVERT temporary
1 OVER (XOR) permanent
0 OVER (XOR) temporary

If you remember from Basic, doing INK 9 will give you a contrasting color to the PAPER. Doing it outside or inside a PRINT line again is the difference between permanent and temporary. Running the program sets the different flags in the P FLAG.

Your User's Manual says that the use of INVERSE prints the pixels in PAPER color and the Paper in INK color...not so. Yes, it looks like that but in reality it sends the Display File the complemented pixel byte--all the 1's are 0's and all the 0's are 1's. We will discuss how to do this type of inverting later when we talk about machine code logic.

When a character is sent to the same space already occupied by another character, the old character is simply replaced by the pixel bytes of the new...overwritten as we say. When OVER is "on" we ADD the pixels in the old byte to those of the new in all cases where one or the other is a "1". When both bytes have a "1" in the same position, a "0" is used. If you try to underline a string of lower case "y" you will see that the underline line is not continuous because of the decender, that part of the "y" below the line, goes into the bottom byte where the underline occurs.

Notice that (XOR) in the OVER flag? This flag is sometimes also called the XOR flag telling the computer to do an XOR operation ...in the case of OVER it means XOR the byte in the Display File with the new pixel byte and send the results as a relacement. XOR means EXCLUSIVE OR which we will find out about later...it's an operation that can't be done from Basic.

INVERSE vs. INVERSE and TRUE VIDEO

Why have Inverse Video and True Video as well as Inverse?

Enter and run the following program:

```
 5 LET a$ = "HELLO"
10 PRINT a$
15 INVERSE 1: PRINT a$;"HELLO"
20 PRINT "HELLO"
25 INVERSE 0: PRINT INVERSE 1;"HELLO THERE";
30 GOTO 10
```

Walking through the program, the first a$ gets printed normally as we would expect. Line 15 prints a$ in Inverse but it prints "HELLO" normally. Line 20 with INVERSE still on in permanant mode prints "HELLO" inverted. Line 25 turns off the permanent Inverse but then the PRINT substatement turns it back on so "HELLO THERE" is inverted. We forgot to turn Inverse off as we cycle back but behold, a$ still goes in normally.

The reason for TRUE and INVERSE Video is that it allows one to just invert a single character in a string. Edit line 15 by adding INVERSE VIDEO in front of the "H" and True Video in back of it. Run the program again. Notice that in line 15 where we originaly got a$ printed in inverse, this time the "H" which was in inverse video did not change to normal but stayed inverted. INVERSE VIDEO is thus absolute. That is, when INVERSE VIDEO is used it keeps it Inverse no matter what INVERSE says. You can't invert an Inverse Video.

SCREEN$

When you save a SCREEN$ you save both the screen bytes and the attributes. When you load a screen back from tape directly into the Display File, you are loading at 1200 baud (bits/second)... that's 150 bytes/second. With a load taking 6912 bytes, it takes 27+ seconds to load. If you start with the screen loaded with contrasting INK and PAPER and then overwrite with a SCREEN$ load so that you can see the printing as it's loaded you first get a load of Permanent colors printed which then gets colored as you load the attributes. Nothing fancy at all to this routine but it can give one some interesting effects as it's being done.

Another favorite trick during loading is to get rid of all the loading messages--this is simply done by making INK and PAPER the same color and designing your screen to be blank where the

message would be printed. No M/C involved in this--just knowing how your computer works.

BORDCR 23624 BORDER COLOR

Address 23624 contains the Border color times 8. A contrasting ink color is automatically used. Be careful POKEing this address as it is also used for LOAD, SAVE and VERIFY--the tape routines. If you recall, the border does different things as a program is being loaded or saved. This is because the Sinclair designers found that they could give their users a visual indication that a program is loading or saving properly from or to cassette. Other personnal computers don't bother. Also note that since the loads from or to disk are so fast there really is no need for a visual signal. The emphasis with disk loads is on speed and there isn't enough time to stop and give a signal to the screen much less have the viewer notice it.

VIDMOD 23746 VIDEO MODE

Of all the disappointments in the 2068, this is perhaps the greatest. We were supposed to be able to use 4 different types of screens and end up with only being able to use one. We cannot "get at" the rest of the modes without doing some machine code routines. Even after we set up the DUAL screen mode, we can't write to it without more code. I can't think of a better use for all the extra space in the Extended ROM than to support the additional modes for the screen and at least correct these omissions. At the same time that we are "burning in" a new Extended ROM we could make all the corrections to the Function Dispatcher and Bank Switching routines as well. In case you were wondering about the Spectrum, it only has one display mode. This was supposed to be an added feature of the 2068. The modes we were supposed to have are:

MODE 0--the one we do have is the normal 32 column by 24 line screen with the attributes working as described. It only uses Display File 1.

MODE 3--Is the first of 3 dual screen modes which use BOTH Display Files with 64 characters across the screen and 24 lines down. As already mentioned every other character is stored in the same screen file. It doesn't use the attributes as the whole screen has to be the same INK and a contrasting PAPER color. No FLASH and BRIGHT are allowed. Call this the Office or monitor mode. By changing pixel widths of the characters it can be made to display an 80 column screen as we already mentioned.

MODE 2--is the high resolution color mode. It uses the single 24 line by 32 column screen but the 2nd Dislay File stores an attribute for each pixel byte of the screen. There are still some limitations to the use of colors.

MODE 1--Is the 2 page mode. It's 2 normal screens that can be

called to the TV/Monitor alternately and could be used for such things as animation by quickly switching from one display to the other. Updates of the screen file not on display must be done in code as Basic wouldn't be fast enough. At present this mode is not supported from Basic so it's all code.

DISPLAY FILE 2

If we look at our memory map (page 254 of the User's Manual if you haven't made a copy), we find that in normal operation, the space used by Display File 2 is used by the RAM Resident Code, alias the Function Dispatcher/Bank Switching routines, as well as the Machine Code Stack and the Machine Code Variables and part of the Basic Program. Note that on the left map the PROG starts at 26710, whereas in the right map the top of Display File 2 is at 31488. Therefore, before we use the 2nd display file we have to make room for it. The Function Dispatcher/Bank Switching routines and the Machine Stack go to the top of memory as the User Defined Graphics are moved down a bit to provide room. Additionally, the Machine Code Vars, ARSBUFF, CHANS and PROG are moved up for the rest of the space needed. Cartridge programs can NEVER use Display File 2 modes as they need the Bank Switching routines along with the ARSBUF to stay in Chunk 3.

At this point you should be familiar enough with entering some codes from other programs to realize that M/C does NOT use line numbers. It, however, has GOTO and GOSUB equivalents called JUMP and CALL which really say, "jump to this address". Since addresses are ABSOLUTE, anytime one moves M/C to a new address, one has to make certain to change all the JUMPs and CALLs. When your computer goes into Dual Screen mode where it needs the 2nd display file, it moves the Function Dispatcher/Bank Switching routines and the machine stack to upper RAM and then uses another routine to correct all the jumps and calls. Unfortunately, there are more errors in that routine so that it actually adds errors to the routines.

If you want to see what a 64 column mode looks like you can do:

```
OUT 255, 62
```

You get a black screen with what looks like Chinese in the top third of the screen. What you are really looking at is the Function Dispatcher/Bank Switching routines printed to the screen as pixels. Although you opened up Display File 2 to the TV screen you did not switch out the code. At this point, if you started with a clear screen, Display File 1 is empty--notice how the Chinese is nicely separated.

Also notice the Edit line at the bottom of the screen now seems strangely staggered. Try entering a line of a program. The letters look compressed horizontally. As you enter the line note how the spacing in the top line of Chinese becomes filled with

your program line. If you look carefully, on a well focused TV, you can actually make out the letters of the line between the Chinese.

Want to experiment further? Let's go to a full 2 screen mode and get rid of the Chinese. At the same time let's do it in Hex. Enter the following Hex Code loader program: (curtesy SYNTAX)

```
  1 REM DON'T NEW AFTER THIS
  5 CLEAR 63199
 10 READ A, B, C, D, E, F
 15 DATA 10,11,12,13,14,15
 20 READ Q$
 25 LET P = 1
 30 FOR X = 63200 TO 65237
 35 LET X$ = Q$(P TO P+1)
 40 LET V = VAL (X$(1))*16 + VAL(X$(2))
 45 POKE X, V
 50 LET P = P + 3
 55 NEXT X
 60 INPUT "VIDEO MODE"; V
 65 POKE 63212, V
 70 RANDOMIZE USR 63200
 75 DATA "F3,3E,01,D3,F4,DB,FF,
CB,FF,D3,FF,3E,01,F5,FB,CD,8E,0E
,F3,DB,FF,CB,BF,D3,FF,AF,D3,F4,F
1,FE,80,20,03,32,C2,5C,FB,C9"
```

Check the numbers in the bottom DATA line to make sure they are correct, then RUN. When the "VIDEO MODE" prompt comes up enter a 62.

We got a nice black screen and no Chinese. We now have made room for Display File 2 and cleared it out. Hitting LIST shows us our program in Display File 1 only. If you want, do some direct command PRINT statements to see how it works.

At this point we have not used Display File 2. As the warning in the REM states: "DON'T USE NEW", we have to now Enter a New program by overwriting the old one...If you like, you may save the above program first.

```
  1 REM For 1st 8 lines only
  5 OUT 255, 62: REM Ink/Paper Change
per p 248 User,s Manual
 10 INPUT y$
 15 FOR x = 1 TO LEN y$
 20 LET A = CODE y$(x)
 25 LET AF = 15360
 30 FOR Y = 0 TO 7
 35 LET F = PEEK (AF + A*8)
 40 LET Z = X/2
 45 IF Z = INT (x/2) THEN POKE 1
6383 +(Z+.5)+Y+256,F
```

```
  50 IF Z <> INT (x/2) THEN POKE 2
4575+Z+Y*256,F
  55 LET AF = AF + 1
  60 NEXT Y
  65 NEXT x
```

Enter a 70 and a 75 to get rid of those lines.

To RUN this program do a GOTO 5, NOT a RUN. At the INPUT enter any message that you wish up to 16 screen lines long--it will be compressed down to 8-64 column lines when it prints to the top of the screen. Notice how slow Basic is...it seems to be drawing the characters in slow motion. Certainly not an acceptable speed and one reason for writing the program in M/C.

I hope you have your 2040 printer attached. If you do try a COPY. You got only every other character to the printer.

Okey, try CLS. Only screen 1 cleared. This is what I mean when I say only Display File 1 is supported. You can't write, COPY, CLS or anything else to Display File 2.

Well, at least we can clear the 2nd screen in a short program. Add to the bottom of your program:

```
70 STOP
75 FOR x = 24575 TO 30719
80 POKE x, 0
85 NEXT X
```

And RUN the program by entering GOTO 75.

As you can see, it would take new routines for each of the normal functions of the screen including AT, TAB, etc. Some of these routines are given in the 2068 Technical Manual although at this time they won't mean much to you. The advanced student may wish to try some of them at this time.

Obviously, the other screen modes will need routines as well.

SCREEN OUTPUTS

You have the choice of 3 different screen outputs depending upon what sort of screen you use. TV is for standard TV operating on either channel 2 or 3. Monitor is for a monitor output, while RGB output can be taken off the back bus for an RBG monitor. If you have an AERCO Disk Drive Interface you already own all the necessary hardware for a RBG monitor--all you need is the cable from the interface to the monitor. You can make this yourself but if you are like me, not a hardware hacker, you can send the specifications to AERCO and have them make a cable for you. The difference between a monitor and a RBG monitor is well worth the price.

CHAPTER 4

SYSTEM VARIABLES

Addresses 23552-24297 are used to store all the things your computer has to keep track of. We already met a few of them in the last chapter. They can be PEEKed at anytime--inside a program and/or in command mode. The complete list of these variables, 23756-24297 are reserved for additional variables, is listed on page 262-265 of your Owner's Manual. I asked you to make a copy of them as we will discuss them in detail now.

We will select our System Variables by subject matter rather than doing a top to bottom discussion. For this it is best to have a "hands on" perspective so turn on your computer for a while. If you have a disk drive inteface, kindly disconnect it as one of the commands we are going give a little further on messes up if the disk drive is present.

CHARacterS 23606-23607

Try, PRINT PEEK 23606 + 256*PEEK 23607. You get 15360 for the start of the character pixel table. But that is not the address I gave you in Chapter 2. It was 15616. Now, kindly read the note that goes along with CHARS. Then subtract 15360 from 15616 and you do indeed get 256. Why this offset? Since "space", the first printable character, and all subsequent characters have to have their CODE numbers multiplied by 8 to get them spaced the 8 bytes apart to designate the 8 pixel bytes, what is saved in CHARS is the amount to be added to the 8 times multiplication of the Character Code. For "space" it's 32x8 which is 256 added to 15360.

The important part for you to remember is that if you would like to design a different character font all you have to do is tell the computer to subtract 256 from the start of your character pixel table address and store it in CHARS and you have a new character set. The interesting thing is that doesn't necessarily have to be the alphabet and punctuation--it could be electrical symbols as I have seen done.

RAMtop 23730-23731

First a word about PEEKing and POKEing. One never harms anything by PEEKing numbers even when you have the wrong numbers--you just get a stupid answer. POKEing the wrong number with the wrong thing can get you into a crash but nothing more--just turn the computer off and back on. You don't physically damage anything. Machine code is a lot of PEEKing and POKEing but it's

done a little differently. However, this is a subject of a later chapter.

Enter the following command: PRINT PEEK 23730 + 256*PEEK 23731. You should get 65367 on the screen. By now you no doubt have figured out that the numbers in back of the titles are the address at which the particular system variable is stored. However, if you look at your memory map (back on page 252) you find RAMtop was not given a number but that it is exactly 1 address short of the UDG area. It came to the screen in decimal.

Yep, your computer gives you numbers in decimal--not Hexadecimal. It's just another reason why hexadecimal has limited usefulness since if you want to use it you now have to convert the decimal to hex, or write a program to do it for you.

Why didn't you get the full value of RAM--65535? Because by now you have figured out that the UDG graphic pixels are stored at the very top of RAM. But we just turned the computer on and we haven't designed anything as yet. Well, the computer reserves this area for UDG anyway and in the meantime it stores the pixels for CAPS A to U there until you redefine them. The full value of RAMtop is called Physical RAMtop stored at 23732-23733. We alredy talked about what happens if your computer develops a bad byte.

Why lower RAMtop? It's done to protect overwriting what is above it with anything else which could be disaster for M/C. Anything else the computer does fills up memory from the bottom up but it still has to know when it runs out of useable space, i.e., space not reserved above Ramtop.

Now enter the commands: CLEAR 64000: PRINT PEEK 23730 + 256*PEEK 23731. You got 64000 right? It should be pointed out that now 64000 is the last byte that Basic can use. The first byte of reserved space is 64001. We now can write our machine code above RAMtop and be assured that our Basic program will never overwrite it. It should be pointed out again that anything above RAMtop must be POKEd there and, furthermore, is NOT saved with the Basic program. It needs a special save that goes:

SAVE "name" CODE 64001,1535 or MOVE "name,bin",64001,1535

if you have a disk drive. Note that I have saved both the machine code and the UDG in one save. How convenient, now I can erase all the lines that defined my UDG and the loader program for the machine code as well, just saving the RANDOMIZE USER statement for the M/C call. I do, however, have to add a LOAD or CAT line in my Basic program to reload the code.

Now, suppose I were doing a musical program to play a few ditties using the SOUND command. These programs are chuck full of numbers as you will find out in Chapter 5. Writing all these numbers in DATA lines takes not only 1 byte per number symbol

(123 takes 3 bytes), but each number is "slugged" and so takes another 6 bytes. All these numbers are less than 256 which means they could be stored one to a byte as code above RAMtop and the program made to PEEK them as needed. I happen to know a few "composers" who have run out of memory. Now, since we are on the subject, how about putting the CODE on a different tape or disk. Then we could load tape #1 for one set of tunes and Tape #2 for the second--or, if you are ambitious, each movement of your Grand Symphony on a different load. It doesn't have to be music however. It could be other data as well. These are just a few ideas on "file storage"...they get out of hand fast enough the way it is.

SETTING RAMTOP WITHOUT CLEAR

There is one thing wrong with using CLEAR. It also CLEARS the Variable file. RUN does the same. Since you generally load the entire Basic program and then start running it, the first line or so contains the CLEAR Followed by the LOAD of the CODE. The CLEAR just cleared your variable file. There may sometimes be reasons why you don't want the variable file cleared. One of the best is to continue a program that is only half run. Just POKE 23730 and 23730 with where you want RAMtop set and load your code.

We already mentioned that it was low bit of address into 23730 and high bit of address into 23731. What is the low bit of an address and the high bit of the address? Simply take the address and divide by 256. You get a number with a fraction. The integer number without the fraction is the "high" byte. Take the high byte number multiply by 256 and subtract from the address to get the "low" byte. I can't stress this enough so once again it's "LOW BYTE FIRST".

We have spend considerable time discussing POKEing RAMtop, but the technique will apply to all other double register variables as well. How do we know how many bytes a variable uses? Refer to the first column of the System Variables Table. The number is the number of bytes. "N" refers to no lasting effect. "X" means to take care so you don't crash. The computer can get lost very easily.

STORAGE OF A BASIC LINE

PROGram 23635-23636 VARiableS 23627-23628
Edit LINE 23641-23642 WORKSPace 23649-23650

It's about time to find out how the computer stores a Basic line. Since your computer is still on, enter the following little program:

```
 5 FOR  X = 26710 TO 26810
10 PRINT X;" ."; PEEK X; TAB 11; CHR$ PEEK X
15 NEXT X
```

and RUN the program. Leave the first screen on the TV and DON'T SCROLL. Now turn to page 255 of your User's Manual and refer to the bottom of the page--BASIC PROGRAM LINE LAYOUT. As we explain a screen of program printout scroll the screen as needed.

Okay, so we didn't get a perfect printout of the program as there are a lot of question marks. Let's see what some of the question marks mean.

First a 0 and a 5 for the line number. Note that line numbers are NOT SLUGGED. The next 2 bytes 27 and 0 are the line length stored LSB/MSB fashion. (LSB = Least significant byte, MSB = most significant byte--low and high byte). We can't tell at this point if its right as the end of the line is on the next screen but we do know it will end with a "13", the enter code. The line length is MINUS the two bytes for line number and the two bytes for line length. Thus, when added to the address of the last byte of the line length, where the computer is when it reads the length of the line, will give the address of the enter byte at the end of the line. You can see how the computer searches the Basic program for a certain line number--it looks at the line number and if it's not what it wants adds the length of that line to get the address of the start of the next line -1.

Let's continue. Ah, something we recognize, FOR x = 26710. Since 26710 is a number, it is followed by the slug token (14) and then the 5 bytes of the slug itself. In this case, it's an integer (a number without a decimal point or a fraction) so only uses the 3rd and 4th bytes of slug number in LSB/MSB fashion. It's a coincidence that the two numbers of the slug correspond to the codes for the letters "V" and "h" so they are printed. On to "TO 26810" with another slug ("h" and "INT" are coincidences again) and finally the 13 ENTER code. That ends the first line of our program.

At address 26741 we start line 10. The two bytes for the line number and the two for line length and then PRINT x;" "; (the 32 stands for the space) PEEK x;" "; TAB 11, slugged as 14, 0, 0, 11, 0, 0 and then ; CHR$ PEEK x and 13 (enter). End of line 10.

At address 26773 we start line 15--two bytes for line number, two for line length, and the line itself NEXT x ENTER. That finishes our little Basic program. Please leave the program on the screen as we will continue with it. Consulting our memory map again (page 252) we see that immediately after the PROGram comes VARiableS. And since we have run this program, we have a value there--x. BUT, our x happens to be in a FOR statement so it is not a simple value. Kindly turn to page 257 of the User's Manual.

In translating the FOR/NEXT variable we see the three ones--meaning that bits 7, 6 and 5 are on. On the TV screen we see 248 SAVE (SAVE just happens to be the token for 248). To see if this

number is really a FOR/NEXT x we take 248 subtract off 128 for Bit 7 being on, another subtraction of 64 for Bit 6 being on, and a third subtraction of 32 for Bit 5 being on leaving us with only 24. Reading under the layout on page 257 we see "letter -60H"--in other words take the lower case letter code, subtract off 96D and add 128 + 64 + 32 to get the code. Since we are working back, we have to add 96 to 24 and get 120--the code for "x".

The next 5 numbers are the slugged PRESENT VALUE of x. Since this program was running while printing itself out, everytime we looped through it, the PRESENT VALUE got updated. Since we know that it was holding the address we were looking at, the 166/104 (LSB/MSB) should correspond to 26783...as that is what it was when we printed address 26784, it was updated to 167/104 as we moved on to the next line...but the MSB was still 104. Continuing with our translation of the FOR/NEXT variable, the next 5 bytes contain the limiting value (the value to stop at--the number after the TO) of x which is 26810. Address 26791 starts the STEP value, which we didn't specify, so it's defaulted to 1. 26796 and 26797 contain the looping line number--our program line 5. 26798 is the subline number within the line...it's a 2 not a 1 as it's the number after "TO" that we are after.

That is the end of the VARS Table, so address 26799 contains an end marker 128. Consulting our memory map again, we should now be in the Edit Line. If you did what I did to RUN the program you entered RUN in command mode. 26800 has my RUN and 26801 has the ENTER. Address 26802 contains another 128 end marker to mark the end of the Edit Line.

Consulting our memory map again, we see that 26803 should start the workspace area collapsed to nothing, followed by the floating point calculator stack. It contains the integer of 181/104 which corresponds to 26805--again the value of x as it was printing the line. The floating point stack has another value of 55 which was used at some point in the program. I just extended the program to include these last few bytes just to show you they were there.

There you have your complete program all laid out in numbers for you. If it didn't make complete sense to you the first time through go and rerun the program and reread the explanation again.

What happens if we add more lines to the program? We have to remember one thing. While the computer was printing out itself, the Edit Line was still processing its contents of RUN-ENTER. When it finished the RUN, you get the message at the bottom of the screen saying what statement the program finished on together with the OKEY. The Edit line is now collapsed back down to no space at all. The message is only in the bottom screen. As we start to enter a new line the first thing it does is a CLS-Lower to clear the lower half of the screen and start printing what

you entered there while saving the codes in the Edit Line space again expanding it as needed. Since the computer doesn't know how long the line will be that you are entering, it has to expand E Line a space at a time. Each time you press a new key it overwrites the cursor with the new key symbol, overwrites the end marker with a new cursor, adds a new end marker (Remember that a token is still stored as one byte.) and increments the values carried in WORKSPace, STKBOT and STKEND. Then it calls the PRINT TO TV routine to print the new symbol or token. If a token, a special extra routine to spell the token must be used. Once it has the code for the symbol, it now has to look up the pixels in the Character table and print these to the screen--repeating as often as needed to spell a token. If it happens to be a line number, just look up the pixels in the Character Table and print them--but keep the cursor in the K mode. If you sent it a token, it's time to change the cursor so check to see if CAPS LOCK is on and if so Print an Inverse C otherwise print an Inverse L cursor. It, of course, also has to check to see if the code was printable. If it turns out to be ENTER, a new sequence starts. All this happens almost simultaneously which gives you some idea of how fast your computer really is.

ENTERING a line the computer checks for "syntax". That's a fancy word meaning "check for errors"...things like the same number of right and left parentheses, an even number of quotation marks, TAB number no bigger than 31, and the two AT arguments in range, for example. You don't really appreciate all this error checking until you start programing on another computer that doesn't do it. It can be quite frustrating to have it stop all the time as it finds another "simple" error. What computers don't check syntax until they run a program? Try Attari, Vic, Commodore, Texas Instruments, Tandy, Apple, IBM and its clones. But back to the 2068. If it finds an error, in goes the "?" cursor (somewhere close to the error) and the whole routine of changing the Edit line takes place. If no error is found, the numbers in the line are slugged. It then checks to see if the statement starts with a number and if so inserts the line length after the line number and inserts the line into the program either as a new line or a replacement line. If the line length is zero, it has to delete the old line. In the case of inserting a new line a position hunt for the right insertion spot must be carried out. When the correct spot if found, space for the new line must made by moving everything above it up the correct number of spaces--that includes all the higher program lines, the Variable File and Edit Line. After the line is inserted the page containing that line is LISTed to the screen. If the line has no line number it is executed immediately. After execution or insertion, the edit line is erased the the space recovered.

See how easy Basic is? You really didn't have to worry about all of this and perhaps never even thought about it until now. This, however, gives you some inkling of what all you might have to consider when you write your own machine code routines--it takes a lot of planning and it all just doesn't happen like a magic

show.

Note that the line number was originally entered one digit at a time. The computer will even let you enter a 5 digit number until it tries to enter the line. It then gets converted to the two bytes in the MSB/LSB fashion--it's the only time your computer does it--all the rest of the time it's LSB/MSB.

There is a way to force line numbers to higher than 9999 but they print in letters and punctuation and are still limited to codes less than 16384.

Try: POKE 26710, 55 followed by LIST on the program you have in the computer at the present time. Interesting line number isn't it? Now try bringing the cursor down with shifted 6 to line 10. Jumps to line 15 doesn't it. Press shift 6 again--back to line 5 --we can't get the cursor to line 10.

Now try: POKE 26710,66 and follow with LIST. Nothing--an invisible program. Well, not quite. Try running the program. No dice, right? You messed up your line number sequence for the FOR/NEXT loop. Do a NEW. We were finished with the program anyway.

PROG, VARS and E LINE are "Edit" variables--they change as you edit (change) your program. There are a few more:

E PPC 23625-23626 (Edit--Present Program Counter) When you bring a program line down to the bottom of the screen, the line number is stored here.

K CUR 23643-23644 (Keyword CURsor) The address of the K, L, C, E or G cursor which is always in the Edit line.

LIST SP 23615-23616 (LIST Stack Pointer) is the address of the Vertical Cursor "<". The one that occurs between the line number and the first token.

X PTR 23647-23648 (IX PoinTeR) This is a little ahead of the game. There is a register in the Z80 chip called IX that is used to find an error. It really stores the address behind the error cursor "?" which normally is where the error should occur.

You will note that the above 4 system variables just keep track of where the various cursors are. There are a few more associated with errors and error reporting.

ERR C 23736-23737 (ERRor, Current) The line number an error occured in which of course gets printed to the bottom of the screen with the error report. If no error then the line number the program ended on or was stopped on with a BREAK.

ERR S 23738 (ERRor Statement) The statement in the line in which

the error occured which also is in the error report.

ERR NR 23610 (ERRor NumbeR) Each error has a number--this number is always one less than that number. 255, one less than zero, is equivalent to "0--OKAY".

Since our computer has an ON ERR command, we have:

ERR LN 23734-23735 (ERRor LiNe) The line number to GOTO if we have used an ON ERR GOTO statement. I have seen this command abused to hide a lot of programming errors!

ERR T 23739 (ERRor Type) The error code for the ON ERR report.

And finally, several screen positions for the edit or LIST (which normally is used in "Editing" programs.)

S POSNL 23690-23691 (Screen POSitioN, Lower) which is the AT arguments for the lower screen. Column first, then line.

DF CCL 23684-23685 (Display File Current Character, Lower) The address of the last character in the lower screen display --which may or may not be the current position. This has to be known and updated as we add more characters to a line so that it doesn't go off the screen.

S TOP 23660-23661 (Screen TOP) The top program line listed on the screen when LISTing a program.

DF SZ 23659-23660 (Display File Size) The number of lines in the bottom "Edit Line" + 1 for the blank line...used to scroll the bottom screen when more edit space is needed.

SCROLL

While we are on Scroll let's add:

SCR CT 23692 (SCRoll CounT) The "Edit" Scroll works by looking at DF SZ and scrolling that number of lines, counted from the bottom of the screen by the number held in SCR CT less 1. Therefore, when an extra line is needed, which is done by checking DF CCL, DF SZ is used to determine how many lines scroll and the scroll is done only 1 line up. The top blank line of the bottom screen blanks the bottom line of the top screen.

Obviously, POKEing SCR CT with the number we desire, not necessarily a full screen, and then CALLing SCROLL is something we can do in machine code but not in Basic. In fact, it's possible to print INPUT to the top screen rather than the bottom. How do you think word processors work?

But you are not completely at a loss. Try this little decimal code loader program:

```
10 CLEAR 63999
15 FOR x = 64000 TO 65000
20 INPUT y
25 IF Y >= 255 THEN STOP
30 POKE x,y
35 PRINT AT 21,0;x;"  "; PEEK y
40 RANDOMIZE USER 2361
45 NEXT x
```

I use this program to enter machine code. Right now we are not interested in entering code so just RUN the program by putting in any number you wish (below 255). When you have seen enough enter a number bigger than 255. Everything should make sense in that program except line 40 RANDOMIZE USER 2361. Knowing what you now know about memory, this is a CALL to ROM. You can use ROM routines in Basic if you have everything set up...that's a big if as most of the time things can't be set up correctly from Basic. If you don't believe me, try adding: 37 POKE 23692, 3. That should scroll the screen up two lines at a time. Wrong! We didn't call the full screen scroll routine, only the scroll loop for the top screen.

How do we know what routines are where in the ROM? By PEEKing ROM and translating it--if you don't want to translate roughly 20,000 bytes of machine code and maybe get some of it wrong, you buy a book (See page 29). It's still machine code but sometimes it's great fun learning how code is written by reading someone else's interpretation of it. To tell the truth, you can spend a lifetime learning all the ins and outs of M/C.

For those of you with a copy of the "Timex 2068 Technical Manual" a listing of the major routines can be found in Appendix A (page 145). I'll warn you that some of the titles don't make much sense. For example, on page 146, the 2nd page, the 2nd column about 3/4ths the way down the page you will find PHLAF 004F. PHLAF means POP HL, POP AF which doesn't tell you a thing about what the routine is really doing.

SYSTEM VARIABLES FOR THE KEYBOARD

K STATE 23552-9 (Keyboard States) It's 8 bytes long and is used by the keyboard routines to find out what key or keys are being pressed and as a counter to count down for the repeat delay and subsequent repeats of the same key.

LAST K 23560 (LAST KEY) Contrary to what you might think, the keyboard is scanned every 1/60th of a second as the computer is running by what is called a "maskable interrupt". Maskable means that it can be defeated. This is done by using the instruction DI--Disable interrupt. BUT, if you come back into Basic without doing an EI--Enable interrupt, you are in trouble as you have a dead keyboard. The only thing that works is the "off" switch and you know

what that means. Anyway, the last key touched is stored in LAST K until you use it and clear it, remove your finger from the key, or touch another key.

REPDEL 23561 (REPeat DELay) As it says, the time in 1/60ths of seconds that the computer waits for that slow human to remove his/her finger from the key before it assumes that the human wants to type another of that key. Originally set at 35 which is too slow for touch typist at 80+ words per minute or to fire those laser guns and move other things in arcade type games. Actually the 2068 goes into a countdown loop which wastes 1/60th of a second and does it as many times as REPDEL.

REPPER 23562 (REPeatER) After waiting REPDEL/60 seconds for the first repeated key, it will only wait REPPER/60 seconds to continue with another repeat. Starts at 5. Don't go below 3 or you will be doing a lot of deletions as you get too many letters.

K DATA 23565 (Keyboard DATA) Stores the 2nd key of LAST K as when you press both CAPS SHIFT and SYMBOL SHIFT to get into Extended mode. Note that LAST K will give you the right code for the combination of keys pressed, not just the one or the other key. CAPS SHIFT and A is going to be either 65 for a CAPITOL A if the mode was L or C, or 230 for NEW if the mode was K, or 227 for READ if in the E mode, or 144 for UDG A if in G mode. CAPS and A can't ever give you FREE.

RASP 23608 (RASPberry) The length of that horrible noise you sometimes get when you mess up--I suppose somebody will find a need to change the length of it someday for some good reason.

PIP 23609 (PIP) The length of the keyboard click. This is really additional time the keyboard update is delayed. POKEing it with large values can result in longer delays than REPDEL itself. If your keyboard isn't fast enough, leave this at zero as it is when the 2068 sets itself up.

ECHO E 23682-23683 (ECHO Edit) You have all been through the use INPUT prompts to make your programs "user friendly". The input prompt uses some of the Edit Line and immediately is followed by a blinking cursor asking for the input. Echo E marks the start of this position with an AT stored at these addresses with column first, line 2nd format. The actual character codes go into the still, at this point, empty workspace area. With an enter, the workspace line gets converted to the proper variable.

SYSTEM VARIABLES FOR THE 2040 PRINTER

PPOSN 23679 (Printer POSitioN) TAB for the printer. Since it

does one line at a time, it really can't use an AT.

PR CC 23680 (PRinter Current Character) The LSB of the last character address in the buffer. When the buffer is full, the computer stores the LSB here...the MSB is still held in one of the other CPU's registers.

SYSTEM VARIABLES FOR INPUT/OUTPUT (I/O)

PORTS, STREAMS and CHANNELS

The CPU can't do much by itself. It has to interface with the outside world to various peripheral pieces of equipment. It does this through a port processing chip called the SCLD of which we will learn more about in Chapter 10. This chip has the capability of addressing 256 different ports. Ports are two way--each can send and receive data or signals. Sometimes they do it intermittently--sending a string of data and then waiting until it gets a response back before repeating with a new cycle of more data and another wait for a new response. The confusing part of all this is that there are not 256 different lines for the 256 different ports--they use the data and address buses.

Take, for example, the three peripherals you must attach to do any computing at all--the keyboard, the TV screen and the sound/joystick are internal except for the fact that you need an extension for the TV out the back. A 4th port, the tape recorder ports, one for send and one for receive, also are jacks on the back of the 2068--in fact, two jacks to the same port. The monitor jack is just another variation of the TV jack with still a third "screen" port--the RBG monitor lines existing on the back outlet bus. Still another port, the "dock" or cartridge port exists as a bus under the front cover. All the rest of the ports have to come out the back bus in various combinations of lines.

The keyboard obviously can only send signals, the TV ports obviously only receive signals. The sound portion of the sound/joystick chip obviously only receives data but sends signals to the amplifier for the speaker. The cassette recorder receive (ear) and send (mic) ports get separated although they use the same port number...they are done that way so that you don't have to constantly replug from mic to ear and back on your recorder as one would eventually plug them in the wrong way. The 2040 printer port, which has to be plugged into the back is intermittent in nature and since it's a parallel device, sends 8 bits at a time requiring a minimum of 8 ports--one for each data line. Other ports are used to control other things for the printer.

All these ports are permanently assigned as they are used in ROM routines making them impossible to change--unless you "burn in" a new ROM chip. As of 1984, Timex had assigned the ports listed in the table on the next page. MSN = Most Significant Nybble, LSN = Least Significant Nybble. There are no assignments below 70H.

		LSN (HEX)															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
M	7				9				9								
S	8	6	6	6	6				A	6	6	6	6				
B	9	6	6	6	6					6	6	6	6				
	A	6	6	6	6					6	6	6	6				
H	B	6	6	6	6					6	6	6	6				
E	C	6	6	6	6					6	6	6	6				
X	D	6	6	6	6					6	6	6	6				
	E	6	6	6	6					6	6	6	6				8
	F	6	6	6	6	2	4	5	8	6	6	6	6	7	7	1	3

1. RD Keyboard/cassette
   WR Border/Beep/cassette
2. RD/WR Dock horiz. sel.
3. RD/WR Enhancement Port
4. WR Sound chip address
5. RD/WR Sound Chip Data
6. TS 2040 Printer
7. Bank Switching
8. Micro-drive
9. Modem
A. RD/WR Centronics port

The numbers in the table correspond to the devices listed to the right. Notice that we have added the Modem, Micro-drive and Bank Switching port.

The confusing part of all these port assignments is that the same port, for example, FE(1) can read the keyboard or the cassette and write to the border, sound chip (Beep) or cassette. Obviously all these devices are hooked together as signals can come in or go out multiple jacks all at the same time. Beep gets to the speaker amplifier as well as to the cassette mic jack. How does the the computer know which one is which? It doesn't. But if it's a LOAD routine from the cassette, it will load whatever is coming in. We already talked about the keyboard being scanned every 1/60th of a second. This scan goes on "between bits" even when a load routine is being used. Thus, the 2068 can check for an INPUT from the keyboard while LOADing. It ignores all inputs except BREAK. If it reads BREAK, it aborts the LOAD, MERGE or SAVE routine it was doing.

Also, should you hook up a new device, like a disk drive system for example, you could use the same port assignments already assigned. Your routine, however, would have to be able to distinguish which device is being read at the time.

You can send things out or get things in from ports using Basic by using OUT and IN. For example, OUT 255, 1, was supposed to turn your 2068 into Dual Screen Mode, and OUT 255, 0 was supposed to turn it back into single screen mode (page 248--Owner's Manual), but these don't work because of the error in the "Change Video Mode" routine.

To Summarize: The ports used are contained in the various routines in ROM, or the routines you or the hardware dealer writes to make other peripheral equipment work.

Channels and Streams. It would be impossible to remember all these assignments all the time. CHANNELS and STREAMS simplify life. They make it unnecessary to remember what device is connected to which port if working in Basic.

Upon setup, the 2068 sets up the following streams: (K = key-

board, S = screen, P = 2040 Printer and R = workspace.)

253	K	This data is held in the System Variable STRMS 23568-23605. Each stream is 2 bytes long--a number and a channel designating letter.
254	S	
255	R	
0	K	Streams 4-15 are available for further epansion.
1	K	
2	S	Each stream is connected to a channal 5 bytes long.
3	P	This data is in the Channel Variables (26660-26709) area. Only 26688-26709 are used at setup.

Upon setup there are only 4 channels designated by the letters K, S, R and P. All the Streams with the same letter use the same channel. Therefore, Streams # 253, 0 and 1 all use channel "K". Similarly, Streams 254 and 2 both use the "S" channel. Streams 253, 254 and 255 are called hidden streams as they are used internally by the 2068 and shouldn't be changed.

Streams 0 and 1 handle the command INPUT which has an input from the keyboard to get a character and an output to the lower screen to print it there. The address of the routine to CALL for the output is stored in the "K" channel bytes 1 and 2 as LSB/MSB. The input routine for the keyboard CALL address is in bytes 3 and 4 with byte 5 being the identifying "K".

Stream 2 (S) prints to the top screen only and handles the commands PRINT and LIST. It only has an output assignment with an error routine in the input address should you try to use it for that.

Stream 3 (P) handles LLIST and LPRINT and only has an output.

Switching Streams: Everything can be changed by specifying a different stream. Make sure your 2040 printer is attached and on. If not, turn the computer off and attached it.

Now try: PRINT #3;"Hello"

It went to the printer, not the TV, right? That's because you told it to PRINT using a stream designating a printer channel (P). Without that #3 after the PRINT, it would have used Stream #2 even if you didn't tell it to. This is called using a "default" value.

Try: LPRINT #2;"Hello"

A printer command got shunted to the TV because you selected Stream number 2 which points to an "S" channel.

We can open a new stream by doing: OPEN #5, "P"

If we follow with: PRINT #5; "Hello", we get it to the printer simply because we designated stream 5 to point to a "P" channel.

Now close the stream by: CLOSE #5: PRINT #5; "Hello"

We got an error because we just closed Stream 5. The first test showed us that we had indeed OPENed a new stream. This one shows us that we again CLOSEd (erased) it.

With streams everything need not always be what it seems. The following commands are identical:

```
LPRINT = PRINT # 3        PRINT = LPRINT # 2
LLIST  = LIST # 3         LIST  = LLIST # 2
```

Don't try LIST and LLIST unless you have a test program entered to list.

You can permanently change a stream to whatever you want by re-defining a preset stream with a different letter (channel). OPEN #2, "P" now makes Stream 2 a printer stream. All LIST and PRINT commands which default to Stream 2 would now act like LLIST and LPRINT. It's going to stay that way until you set it back with: OPEN #2, "S" or CLOSE #2. Streams 0 to 3 won't stay closed however, as the 2068 again "defaults" the stream back to their original settings.

You can even open your own stream using a different letter to point to as yet an unwritten channel having that letter designation. Designing a channel is a little beyond your ability at the present time.

The computer keeps track of the present stream number in STRMNM 23755 (STReaM NuMber).

The Channel Lookup Table can be changed from 26688 to another address by changing CHANS 23631-23632 (CHANnelS).

Even the Bank Channel Lookup Table can be changed by changing CURCBN 23743 (CURrent Channel Bank Number)...this is done when the computer is operating from a cartridge.

OPERATING SYSTEM VARIABLES AND FLAGS

The 2068 has to use some System Variables to keep track of where it is:

PPC 23621-23622 (Present Program Counter) The line number being executed--not the address of the line number.

SUBPPC 23623 (SUB Present Program Counter) Since we can use ":" to put many statements in the same line, the computer has to keep track of which one it's doing at the present time.

NEWPPC 23618-23619 (NEW Present Program Counter) The line to be jumped to on a GOTO or a GOSUB.

NSPPC 23620 (New Sub Present Program Counter) The subline to be jumped to on a GOTO or a GOSUB.

NXTLIN 23637-23638 (NeXT LINe) Not the number of the next line but the memory address of the start of the next line. The only way your computer knows where that line starts is to add the line length of its present line to its present position (MSB of Line Length). It won't know what that line number is until it comes to it at which time it will update PPC and SUBPPC. The computer has to know where this line starts in the case of a "false" IF statement which causes it to skip the remaining portion of its present line and jump to the start of the next line.

OLD PPC 23662-23663 (OLD Present Program Counter) Return line for a GOSUB or CONTINUE.

OSPPC 23664 (Old Sub Present Program Counter) Return subline for a GOSUB or CONTINUE.

CONTINUE is a type of error which will continue the program if you have not used an ON ERROR statement. Scroll does the same thing. But, the computer has to know where to continue or come-back to after a scroll.

DATADD 23639-23640 (DATa ADDress) The address of the last byte used so far in a DATA statement. There is a special routine that hunts through your Basic program from the start for a DATA line as soon as it hits the first READ statement. Once it has found a DATA line it uses as much of it as it needs and stores the address of the last byte it has used here. At the next READ it continues with reading the rest of the data from that line. Should it hit an ENTER character, it hunts for the next DATA line and uses as much of that as it needs. Each time updating the address of the last byte used to DATADD. Of course, RESTORE either resets the address back to 0, the start of PROG if no argument was given, or the argument line.

There is a similar hunting routine for FN which also starts at the beginning of the Basic program looking for the DEF FN that goes with the arguments of the FN found. Once it has found the right definition, it reads the function and resets. This is the reason why DEF FN statements should occur early in a program so the routine doesn't have to search the entire program.

Data statements, on the other hand, are subject to change from one program run to another. The easiest way to effect these DATA line changes is to use high line numbers and MERGE the next set of data lines with the present program. Using identical line numbers erases the old DATA in favor of the new. Using the same RESTORE statement just before the first READ even presets the data search line so it doesn't have to look through the whole

program.

VARIABLE STORAGE and SEARCH

Kindly turn to page 256 of your User's Manual. One of the big problems in writing machine code is to interface with all the variables from the Basic part of the program. One technique used is to POKE the necessary variables into set data positions in the machine code area. However, it would be easier sometimes to search the VARS TABLE itself for the desired variable. Thus, we have to know how the 2068 stores variables.

All Sinclair based computers store variables in the same manner although the same number may not mean the same variable in the 2068 as it does in the 1000/1500/Z81 machines as they don't use ASCII coded letters and numbers as the 2068 does.

The type of variable, i.e., single character number, multiple character number, array number, string, string array or FOR can be determined by looking at the 3 high bits of the first character. A small "z" on the 2068 is code 122 which already uses Bits 5 and 6. How can they be used for something else? Every small letter uses Bits 5 and 6 so, once again, are not really needed. Unfortunately, using Bit 5 and 6 to designate the type of variable leads to your computer being unable to differentiate between an "A" and an "a"--Caps and small. Since Bit 6 is 64 and Bit 5 is 32, we have to subtract 96 (60H) from each first letter.

Going through the types of variables and their 3 high identifier bits, we get the following table:

	Bit 7	Bit 6	Bit 5
Single byte name	0	1	1
Multiple byte name	1	0	1
Number array	1	0	0
FOR	1	1	1
String	0	1	0
String array	1	1	0
unused sequence	0	0	1

We note that Bit 6 is used for all Single letter names. Bit 5 is for single numbers only. Bit 7 is used for complex variables, i.e., long names, arrays and FOR.

The number variables are always followed by just the 5 bytes of the floating point number (without the 14 slug designator). In the case of long named variables and strings where an indefinate number of extra bytes must be used, the end byte of such a name has 128 added to it to indicate the last byte. Should you ever PEEK the TOKEN SPELL TABLE (addresses 152-550) you will see the same sort of termination indicator used there.

Since strings are different lengths, two bytes are needed to indicate their length. Arrays need to know the number of dimensions as well, so that info is also included before the actual array starts. It's these variable lengths that lets the 2068 skip over variables when searching through the table much like it could skip to the start of the next line when reading a Basic program. See page 257 of your User's Manual to see how arrays are arranged.

Hunting for variables: Whenever you use RUN or CLEAR you "dump" (clear out) the old variable table and start a new one at VARS 23627-23628. Unlike some computers, the 2068 must have all variables "initialized", i.e., set to some value...it does not assume a zero if it can't find it. If you set a variable or change it either directly or with a calculation, the first thing the computer does is to try to find if it has already been used by looking for it in the VARS. It's looking for a matching first byte. If a match is found, the address is put in DEST 23629-23630 (DESTination) and corresponding value changed accordingly. If the variable is not found, the DEST is the end of the VARS. Room is made and the name and value inserted there. Dimensioning a variable or string array automatically kills the old array if any, recovering the space and putting the new dimensioned array at the end of the VARS. Your 2068 is one of only a few computers that allows redimensioning an array...it's the easiest way to clear an array and start over which can be very handy at times.

Double storage: Let A = 23456 in your Basic program has a slug of 6 bytes. When the program runs another 6 bytes are used in VARS for the same number. This seems like a waste of space. Preslugging the numbers at a time when the computer hasn't got much to do anyway as that slow human presses those keys can speed up the running of a program by 1/3rd. Checking for syntax at the same time also saves running time--and a lot of harrassed nerves. Using the Z80 chip rather than a 6500 series or 8080 saves more time. Tokenizing commands also speeds things up although most computers do this anyway even if you have to type in the whole word. No wonder Sinclair programs run 4 to 5 times faster than Apple programs (Unless you are running CP/M, in which case you are using a Z80 CPU).

If you are really pressed for space there are several ways to save some. This very seldom seems to be a problem on the 2068 but was with a 16k RAM pack on a TS1000.

1. Enter all your most used numbers as double letter variables in direct command mode. But then NEVER use RUN. Start your program with a GOTO statement. Suppose that you have a subprogram you call all the time at line 9000. Do a LET KZ = 9000 in command mode. Then when you call your routine you GOSUB KZ. This uses 7 bytes in VARS but only 2 in the program vs 10 if you used GOSUB 9000. Each additional time you used that GOSUB

you will now be saving 8 bytes. But you don't get something for nothing as your program runs a bit slower since it has to look up all these numbers. Therefore, putting them first in the VARS table is import- ant as you get a little speed back. You also lose something in program readability but letting k0 = 0, k1 = 1, k2 = 2...ka = 10...kf = 15, kg = 16, kh = 17 etc. isn't that hard to read and look at all those AT, TAB, INK, TO etc. times one uses small numbers. Each time after the first saves 5 bytes.

2. Use of VAL "9000" makes your program more readable and saves 3 bytes per use as the number is not slugged but VAL and the 2 " take 3 of the 6 slug bytes. It slows running time as the strings have to be stripped of quotes and slugged when used.

3. Use multistatement lines--each line saved is 4 bytes saved. (2 for the line number, two for the line length and 1 for enter less 1 for the ":")

4. Store numbers in string arrays, especially as codes if your numbers happen to be integers under 255.

5. Use advanced logic statements rather than page after page of individual IF statements. This includes the use of calculated GOTOs and GOSUBs--another thing your computer allows which others don't.

FLAGS

THE CONCEPT: Rather than having to know an address or a number, sometimes the computer just has to know if one or another situation exists. Like, should it use the upper or lower screen, is it checking syntax or running a program, is OVER on or off, or is INVERSE on or off. All these are one or the other situations so the information can be stored in a single bit. Since the CPU can be asked about the status of any bit anywhere, we don't have to use a different byte for each of these on/off or one/another situations but can put as many as 8 in the same byte.

The System Variable Table contains 6 of these flag bytes. To write effective machine code we need to know what is in these flags. You will note that all these addresses have an X in front of the note indicating "don't change unless you know what you are doing".

Here is the complete list courtesy the TIMEX 2068 Technical Manual. (by Bit number)

	IF ON	IF OFF
23611 FLAGS		
7	Need Interrupt	Check syntax
6	Number	String
5	Keyhit	No keyhit
4	Token/slug	Regular character
3	L mode at cursor	K mode at cursor
2	L mode at character	K mode at character
1	To printer	To screen
0	Suppress space	Don't suppress space

23612 TV FLAGS

- 7-6 not used
- 5 Clear screen when key pressed
- 4 Auto list
- 3 Echo input from keyboard
- 2 not used
- 1 Output line for edit or number for string
- 0 Use lower screen — Use upper screen

23658 FLAGS 2

- 7-6 not used
- 5 Delete key repeat
- 4 Retype possible after syntax error
- 3 Caps lock on
- 2 Inside string when doing keyboard LIST CHAR
- 1 Printer buffer not empty
- 0 Automatic listing on screen

23665 FLAG X

- 7 Line — String line
- 6 Need number
- 5 Need input
- 4-3-2 not used
- 1 Variable not — Variable found
- 0 Flexible length needed

23697 Print FLAG

- 7 Paper complement of ink permanent
- 6 Paper complement of ink temporary
- 5 Ink complement of paper permanent
- 4 Ink complement of paper temporary
- 3 Invert (INVerse) permanent
- 2 Invert (INVerse) temporary
- 1 OVER (XOR) permanent
- 0 OVER (XOR) temporary

23617 MODE

- 1 G mode
- 0 E mode — K or L mode

CHAPTER 5

BEEP and SOUND

THE BEEP COMAND

The simple Beep circuit is contained mostly in the SCLD chip which is why the Spectrum also has BEEP. The Spectrum does not have the more sophisticated AY-3 8912 Sound chip which Timex added in an attempt at music. The tiny speaker under your 2068 doesn't do it justice. But since Beep also comes out the MIC socket a line can be run to a more suitable amplifier-speaker system. It not only improves quality but also volume control.

The BEEP command is port addressed to port 254. If a write to this port sets Bit 4, the internal speaker is activated. If Bit 3 is set, the signal goes to the MIC port for recording or amplification. Take care, port 254 Bits 0-1-2 contain the active Border color. These bits have to be kept to a valid color number (0 is a black border) as an OUT (254) command is given.

The easiest noise to create is the keyboard click which is activated by setting PIP (23609) to the length of the click desired (in 1/60th of a second mode--not like BEEP from Basic which uses seconds).

The Basic command of BEEP has two arguments. The first is duration in seconds, the 2nd is the note where middle C is zero and each note on the piano, moving up and using all the half steps is a number higher. Notes below middle C are counted step or half step minus from middle C. Every 12 notes is an octave. Thus, 12 is one octave above middle C, -12 is one octave below middle C.

Translating this command to machine code is extremely difficult. The routine in ROM makes extensive use of the floating point calculatar to calculate the proper frequency of impulses to send to the speaker. By comparison, duration is relatively easy. Another precaution, if you don't prevent maskable interrupts with a DI, which incidentally also prevents updating of the screen, your notes are going to "warble" or have a vibrato to them. if you want pure notes stick to calling the ROM routine.

You have no control over the exact pitch of the note you create nor the loudness of the note, nor the tremolo, or anything else --just rough pitch and duration.

Fully exploring the BEEP command can take a lifetime of study as the command has been used to make the computer talk. Such appli-

cations are beyond the scope of this book.

A SIMPLE EXPERIMENT FROM BASIC

Enter and run the following program:

```
 5  FOR x = 1 TO 500
10  OUT 254, 7: OUT 254, 23
15  NEXT x
20  OUT 254, 7: OUT 254, 23: GOTO 20
```

You will have to STOP the program with a CAPS SHIFTED BREAK.

You got two sounds right? What we just did is toggle Port 254 by turning the speaker switch on and off many times with the statements: OUT 254, 7: OUT 254, 23. We told you we would have to keep a valid border color so we chose white (7). The first OUT keeps the border color but turns off everything else. Now if we subtract 7 from the value we sent in the 2nd OUT, 23, we get 16 --exactly what we need to turn on BIT 4 which tells the computer to turn on the speaker.

Now we cycled the on/off to get the frequency. In lines 5 to 15 we did it with a FOR-NEXT loop. In line 20 with a GOTO. Since the first tone was lower than the second, we can say that our computer executes a FOR-NEXT slower than a GOTO loop...for this program. GOTOs execute slower and slower as the program gets longer and longer. The FOR-NEXT loop cycled about 95 times a second with the GOTO at about 140 times a second.

Now add Line 1 BORDER 5: BEEP 2.5, -18: BEEP 2.5, -15 and run the program again. Compare the first pair of sounds with those of the second pair. The BEEP sounds will be cleaner purer tones. The loop sounds will have a definate "warble" or "vibrato" to them, due as we pointed out above to the interrupts to reread the display file and check for a keyboard input. Also, did you see the border change from green to white? You may want to play with the pitch of the BEEP upping or lowering them a tone or two to see if you get a better match of the two sets of sounds. Your computer may be running at a slightly different frequency than mine.

Middle C, designated by a 0 in BEEP, has a frequency of 260/second. One octave below that (at -12) would be "C below middle C" with a frequency of half middle C or 130/second. 3 steps below that at -15 is A at about 110/second. 3 notes below that at -18 is F# at about 92. What we have just done is very crudely timed our FOR-NEXT and GOTO loops.

If you don't believe me that the length of the program changes the speed of the GOTO loop, RUN the program again to attune your ear to the last frequency. Now delete all but line 20 and RUN the program again. You can already detect a pitch change.

SIMULATED SOUNDS FROM MACHINE CODE

If you read from the beginning of the book to here you should know how to load the following machine code into your computer and run it. I have given you a decimal and a hex loader program earlier. Note that I haven't given you any addresses. Since the program is short it is written address independent and can be put anywhere. You choose the address. I don't expect the novice students to understand this program at the present time but they should recognize what part of the program gets POKEd above RAM-top and what is assembly mnemonics.

```
62,5                    LD A, 5 Border Cyan
14,254                  LD C, 254 Set Port
38,0                    LD H, 0  Set wait/frequency
22,255                  LD D, 255  Set loop count
68          Again       LD B, H
203,231                 SET 4, A  On
237,121                 OUT (C), A
16,254      1st wait    DJNZ, 1st wait
68                      LD B, H
203,167                 RES 4, A OFF
237,121                 OUT (C), A
16,254      2nd wait    DJNZ, 2ND Wait
203,231                 SET 4, A  ON
237,121                 OUT (C), A
16,254      3rd wait    DJNZ, 3rd wait
203,167                 RES 4, A  OFF
237,121                 OUT(C), A
16,254      4th wait    DJNZ, 4th wait
36                      INC  H
21                      DEC D
32,226                  JR NZ, AGAIN
201                     RET
```

Running the above program will give you a sliding scale downwards. The frequency of the tone is set by the wait loops and although there is one cycle at the start of a very long wait (the 1st time through B = 0, so DEC B sets it to 255 and you have to wait the entire DEC back to 0) the next wait is the shortest as B = 1. Only one cycle at each frequency is used.

Obviously to do sustained tone one replaces INC H with a nop. To set the tone, experiment with LD H, n with different values. The higher the value the lower the sound.

To increase the length of the note, use DE as a counter instead of just D. This will require the resetting of A at the start of each loop as you will need to use the A register to check to see if DE is zero.

LOAD, SAVE and BAUD RATES

Since we have mentioned saving the border color when using BEEP

you of course have noticed how the border blinks red and green while it's waiting for a Lead-In signal from your cassette player while LOADing a program. Once it has found it, the border shifts to a red-green montage of stripes. While actually loading the few bytes of header it barely blinks on yellow and blue bands as it does while loading the program. If you havn't figured it out by now, Port 254 which contains the border color in Bits 0 to 2 also uses Bit 6 to read or write to the cassette. Sinclair, unlike other computers, uses these facts to give its users a visual verification that the program is loading or saving properly. The Baud Rate (bits per second sent or received) is too high (1200 baud) on the 2068 to give you discrete bands of bytes as on the ZX-81 or the TS-1000 (300 baud). Remember the screen refresh rate changes the way things are seen on the screen edges.

What is actually sent to your tape recorder is two different pulses of sound. A pulse at a frequency of 1020 Hz (Hertz = cycles/second) for a 1 and a pulse one octave higher at 2040 Hz for a 0. At a switch rate of 1200 baud, that's only 1 cycle for a 1 and 2 cycles for a 0. Since both these are 2 to 3 octaves above middle C (260 Hz), you are asked to turn the treble on the recorder all the way up. Should you ever play a computer tape through the recorder speaker you get that high pitched screach. Not only is it high but it's loud to give a strong signal. Turn the volume down or the dog runs for cover--at least mine does.

What really slows down SAVE, LOAD and VERIFY to a cassette are these pulses of sound. Internally, and to disk drives, a single electrical pulse, not a series of pulses at a set frequency, serves as a bit on or off. Hence the baud rates of the disk drives and internal transfers to memory can occur much faster.

The other thing that slows down cassette data transfer is that 1 bit must be sent at a time serially, one after the other. Disk drives and Dot Matrix Printers use a Centronics "parallel" port of 8 data lines sending a whole byte at a time down 8 different lines. Generally on board RAM stores a set amount of data before processing it--this is called storing it in a buffer. A printer that prints one line left to right and the next right to left has to have all the pixels for a line stored before starting to print a line. A disk drive for a 5.25 inch diameter disk operating at 300 rpm is equivalent to a tape running at 65 inches/sec (outside track) compared to 3 inches/sec for a recorder, so can record or read (still serially, one bit after the other on the disk surface) faster than a tape recorder especially when it just has to send a pulse down 1 line 1/8th of the time. The pulses can really be 8 times longer than if they had to go down a single line.

Disk drive baud rates are limited by the way they operate. Normally a drive spins a disk at 300 or 360 rpm. That's 5 or 6 turns per second. It has to read or write a track in that amount of time. Since it has to read the data that fast, it also has to

SIMULATED SOUNDS FROM MACHINE CODE

If you read from the beginning of the book to here you should know how to load the following machine code into your computer and run it. I have given you a decimal and a hex loader program earlier. Note that I haven't given you any addresses. Since the program is short it is written address independent and can be put anywhere. You choose the address. I don't expect the novice students to understand this program at the present time but they should recognize what part of the program gets POKEd above RAM-top and what is assembly mnemonics.

```
62,5                    LD A, 5 Border Cyan
14,254                  LD C, 254 Set Port
38,0                    LD H, 0  Set wait/frequency
22,255                  LD D, 255  Set loop count
68          Again       LD B, H
203,231                 SET 4, A  On
237,121                 OUT (C), A
16,254      1st wait    DJNZ, 1st wait
68                      LD B, H
203,167                 RES 4, A OFF
237,121                 OUT (C), A
16,254      2nd wait    DJNZ, 2ND Wait
203,231                 SET 4, A  ON
237,121                 OUT (C), A
16,254      3rd wait    DJNZ, 3rd wait
203,167                 RES 4, A  OFF
237,121                 OUT(C), A
16,254      4th wait    DJNZ, 4th wait
36                      INC  H
21                      DEC D
32,226                  JR NZ, AGAIN
201                     RET
```

Running the above program will give you a sliding scale downwards. The frequency of the tone is set by the wait loops and although there is one cycle at the start of a very long wait (the 1st time through B = 0, so DEC B sets it to 255 and you have to wait the entire DEC back to 0) the next wait is the shortest as B = 1. Only one cycle at each frequency is used.

Obviously to do sustained tone one replaces INC H with a nop. To set the tone, experiment with LD H, n with different values. The higher the value the lower the sound.

To increase the length of the note, use DE as a counter instead of just D. This will require the resetting of A at the start of each loop as you will need to use the A register to check to see if DE is zero.

LOAD, SAVE and BAUD RATES

Since we have mentioned saving the border color when using BEEP

you of course have noticed how the border blinks red and green while it's waiting for a Lead-In signal from your cassette player while LOADing a program. Once it has found it, the border shifts to a red-green montage of stripes. While actually loading the few bytes of header it barely blinks on yellow and blue bands as it does while loading the program. If you havn't figured it out by now, Port 254 which contains the border color in Bits 0 to 2 also uses Bit 6 to read or write to the cassette. Sinclair, unlike other computers, uses these facts to give its users a visual verification that the program is loading or saving properly. The Baud Rate (bits per second sent or received) is too high (1200 baud) on the 2068 to give you discrete bands of bytes as on the ZX-81 or the TS-1000 (300 baud). Remember the screen refresh rate changes the way things are seen on the screen edges.

What is actually sent to your tape recorder is two different pulses of sound. A pulse at a frequency of 1020 Hz (Hertz = cycles/second) for a 1 and a pulse one octave higher at 2040 Hz for a 0. At a switch rate of 1200 baud, that's only 1 cycle for a 1 and 2 cycles for a 0. Since both these are 2 to 3 octaves above middle C (260 Hz), you are asked to turn the treble on the recorder all the way up. Should you ever play a computer tape through the recorder speaker you get that high pitched screach. Not only is it high but it's loud to give a strong signal. Turn the volume down or the dog runs for cover--at least mine does.

What really slows down SAVE, LOAD and VERIFY to a cassette are these pulses of sound. Internally, and to disk drives, a single electrical pulse, not a series of pulses at a set frequency, serves as a bit on or off. Hence the baud rates of the disk drives and internal transfers to memory can occur much faster.

The other thing that slows down cassette data transfer is that 1 bit must be sent at a time serially, one after the other. Disk drives and Dot Matrix Printers use a Centronics "parallel" port of 8 data lines sending a whole byte at a time down 8 different lines. Generally on board RAM stores a set amount of data before processing it--this is called storing it in a buffer. A printer that prints one line left to right and the next right to left has to have all the pixels for a line stored before starting to print a line. A disk drive for a 5.25 inch diameter disk operating at 300 rpm is equivalent to a tape running at 65 inches/sec (outside track) compared to 3 inches/sec for a recorder, so can record or read (still serially, one bit after the other on the disk surface) faster than a tape recorder especially when it just has to send a pulse down 1 line 1/8th of the time. The pulses can really be 8 times longer than if they had to go down a single line.

Disk drive baud rates are limited by the way they operate. Normally a drive spins a disk at 300 or 360 rpm. That's 5 or 6 turns per second. It has to read or write a track in that amount of time. Since it has to read the data that fast, it also has to

handle the data that fast. The baud rate will now depend upon how many bits are written on a track. This depends upon how fast the read/write head can respond to signals. Generally a single density disk uses a bit time of 8 microseconds--that's 8 millionths of a second or 125,000 bits per second. Double density packing puts twice as many bits on a track and thus has to operate at a pheneminal 250,000 bits per second and a read/write time of only 4 microseconds.

Both dual and Quad density disks use dual packing of bits in a track. Quad density is achieved by having 80 tracks/side rather than 40.

There is no way to send 250,000 bits/sec down a single data line using a clock cycle of 3.528 mega cycles. It requires a send loop of only 13.14 T states. Using 8 data lines, as we do in parallel interface port, we can up that time to 105 T states. Even more T states are used in the interface and the disk because they generally have their own clock operating at 8 megacycles or higher. It should be pointed out that the read/write rate is NOT the effective rate as disks have a lot of overhead bytes to read and write as well. Our effective baud rate thus is less than 250,000 bits/second which would be a phenomenal 31250 bytes/second.

Modems are devices that allow one computer to talk to another over a phone line. That should be quite fast. But, a phone line is only a single data line--a series port if you please. A pure bit rate, i.e., without multiplexing onto a carrier wave, of 300 baud is already a high pitched sound which is generally about the limit that an ordinary phone line can handle without garbling. Mainframe computers talking to each other use multiplexing on special data phone lines to achieve higher baud rates. 1200 baud is about as fast as most personal computer modems can handle.

SOUND COMMAND

The sound chip has 14 internal registers assignable to 3 differant channels of sound with or without noise added. They are listed in the table on the next page. Despite having 14 registers to control sound, we still can only write 3 part harmony. With only one envelope for timbre we are stuck with a maximum of two sounds, that of the envelope and that without the envelope--hardly enough to write a symphony. About the best we can expect is to write for one instrument. With this limited ability of the envelope and the small speaker, it's still going to sound like computer sound--not organ, grand piano, mandolin or even banjo. The sound chip does much better with sound effects than with music.

SOUND CHIP REGISTERS

REG	CHAN	CONTROL	BIT 7 6 5 4 3 2 1 0
0	A	Tone control	Fine Tune----------
1	A		Coarse tune----
2	B	Tone control	Fine Tune----------
3	B		Coarse tune----
4	C	Tone control	Fine Tune----------
5	C		Coarse tune----
6	Noise Period		Coarse tune only---
7	Enable		joystick N O I S E T O N E
			C B A C B A
8	A	Loudness	ENV -----0-15----------
9	B	loudness	ENV -----0-15----------
10	C	loudness	ENV -----0-15----------
11	Envelope period		Fine tune----------
12			Coarse tune----------
13	Envelope shape		CONT ATT ALT HOLD
14	Joystick In Register		
15	In register not used		

We have to give a few definitions and define a few terms before we can use the above table effectively.

Registers 0 to 5: The sound chip receives the clock signal of 3.528 megahertz but divides it by 32 to give an effective frequency of 110,250 Hz for generating tones. We are used to thinking of the pitch (tone) of a note as a frequency, f, (middle C, C4 is 261.626 Hz). To get the period, p, of the note we have to take the reciprocal, i.e., p = 1/f. To get the tone period required by registers 0 to 5, we multiply p by the frequency of the sound chip, 110,250. (This is the step by step way of saying: tone period = 110,250/f.) These numbers can range from 14 to NO MORE THAN 4025--no bigger than we can hold in 12 bits, 4 in the high or coarse tune, and 8 in the fine tune. We, of course, have two bytes for each of our 3 sound channels. These values are already figured out for you for all the notes in your User's Manual (page 187ff). However, they are not exact due to rounding errors. The table starting on the next page is more accurate.

Register 6: Noise period is limited to values from 0 to 31 which creates a high pitched frequency used with RANDOMISE to create "white" noise. Higher values of noise produce lower sounds. Your User's Manual lists 3 short programs for some sounds, which is just a start. You're on your own with experimentation as far as creating that special effect for your super game.

Register 7: Nothing happens until you tell the sound chip what registers you want to use and that includes the joysticks. As your User's Manual says "subtract from 63 and use that number". This register is an ACTIVE when LOW type...a "0" must be in the bit indicated to activate the register. Mixing of noise with a tone is allowed but not recommended if you want to play music.

REGISTER VALUES FOR NOTES OF THE MUSICAL SCALE

NOTE	IDEAL FREQ	C	F	ACTUAL FREQ
A0	27.500	15	169	27.506
A#0	29.135	14	200	29.136
B0	30.868	13	244	30.865
C1	32.703	13	43	32.705
C#1	34.648	12	110	34.648
D1	36.708	11	187	36.713
D#1	38.891	11	19	38.889
E1	41.203	10	116	41.200
F1	43.654	9	222	43.646
F#1	46.249	9	80	46.246
G1	48.999	8	202	49.000
G#1	51.913	8	76	51.907
A1	55.000	7	213	54.998
A#1	58.270	7	100	58.272
B1	61.735	6	250	61.730
C2	65.406	6	150	65.391
C#2	69.296	6	55	69.296
D2	73.416	5	222	73.402
D#2	77.782	5	137	77.805
E2	82.407	5	58	82.399
F2	87.307	4	239	87.292
F#2	92.499	4	168	92.492
G2	97.999	4	101	98.000
G#2	103.826	4	38	103.814
A2	110.000	3	234	110.030
A#2	116.614	3	178	116.543
B2	123.471	3	125	123.460
C3	130.813	3	75	130.783
C#3	138.591	3	28	138.505
D3	146.832	2	239	146.804
D#3	155.563	2	197	155.501
E3	164.814	2	157	164.798
F3	174.614	2	119	174.723
F#3	184.997	2	84	184.983
G3	195.998	2	51	195.826
G#3	207.652	2	19	207.627
A3	220.000	1	245	220.060
A#3	233.082	1	217	233.087
B3	246.942	1	190	247.197
C4	261.626	1	165	261.876
C#4	277.183	1	142	277.010
D4	293.665	1	119	294.000
D#4	311.127	1	98	311.441
E4	329.628	1	78	330.090
F4	349.228	1	60	348.892
F#4	369.994	1	42	369.966
G4	391.995	1	25	392.349
G#4	415.305	1	9	416.038
A4	440.000	0	251	439.243
A#4	466.164	0	237	465.190
B4	493.883	0	223	494.395
C5	523.251	0	211	522.512
C#5	554.365	0	199	554.020
D5	587.330	0	188	586.436
D#5	622.254	0	177	622.881
E5	659.255	0	167	660.180
F5	698.456	0	158	697.785
F#5	739.989	0	149	739.933
G5	783.991	0	141	781.915
G#5	830.609	0	133	828.947
A5	880.000	0	125	882.000
A#5	932.328	0	118	934.322
B5	987.767	0	112	984.375
C6	1046.502	0	105	1050.000
C#6	1108.731	0	99	1113.636
D6	1174.659	0	94	1172.872
D#6	1244.508	0	89	1238.764
E6	1318.510	0	84	1312.500
F6	1396.913	0	79	1395.570
F#6	1497.978	0	74	1489.865
G6	1567.982	0	70	1575.000
G#6	1661.219	0	66	1670.455
A6	1760.000	0	63	1750.000
A#6	1864.655	0	59	1868.644
B6	1975.533	0	56	1968.750
C7	2093.005	0	53	2080.189
C#7	2217.461	0	50	2205.000
D7	2349.318	0	47	2345.745
D#7	2489.016	0	44	2505.682
E7	2637.021	0	42	2625.000
F7	2793.826	0	39	2826.923
F#7	2959.956	0	37	2979.730
G7	3135.964	0	35	2150.000
G#7	3322.438	0	33	3340.909
A7	3520.000	0	31	3556.452
A#7	3729.310	0	30	3675.000
B7	3951.067	0	28	3937.500
C8	4186.009	0	26	4240.385
C#8	4434.922	0	25	4410.000
D8	4698.637	0	23	4793.478
D#8	4978.032	0	22	5011.364
E8	5274.041	0	21	5250.000
F8	5587.652	0	20	5512.500
F#8	5919.911	0	19	5802.632
G8	6271.927	0	18	6125.000
G#8	6644.876	0	17	6485.294
A8	7040.000	0	16	6890.625
A#8	7458.621	0	15	7350.000
B8	7902.133	0	14	7875.000
C9	8372.018	0	13	8480.769

All notes of the scale can not be played beyond this point.

ADDITIONAL REGISTER VALUES FOR NOTES OF THE MUSICAL SCALE

NOTE	IDEAL FREQ	C	F	ACTUAL FREQ	
D9	9397.274	0	12	9187.500	Unless you are a dog you
D#9	9956.062	0	11	10022.727	can't hear notes higher
F9	11175.304	0	10	11025.000	than F10.
G9	12543.854	0	9	12250.000	
A9	14080.000	0	8	13781.000	
B9	15804.266	0	7	15750.000	
D10	18794.548	0	6	18375.000	
F10	22350.608	0	5	22050.000	

Be sure to turn the sound registers off by writing 63 to register 7 when you are done.

Registers 8-9-10: Loudness for channels A, B, and C respectively if you use numbers from 0 (very soft) to 15 (loud). Adding 16, turning on Bit 4, turns this maximum loudness over to the shape of the envelope which will then control its variation. Not using the envelope gives you a note of steady loudness.

Registers 11-12: Envelope Period (E.P.). The length of the envelope is based on the frequency of 6890.625 (110,250/16) and can be calculated as we did above for the tone (pitch) as long as we remember that it has to be per second--not per minute.

The use of these registers can best be explained by an example. If you strike a note on a piano and hold down the key, the sound will last for 10 to 15 seconds before it has faded away to silence. Generally a pianist doesn't wait for the sound to fade that far before hitting other notes. To keep the melody note going without having to keep a finger on the note, the pianist uses the SUSTAIN pedal which lifts the dampers on all the strings thus giving rise to harmonics. Releasing the sustain pedal drops the dampers and kills all sounds instantly (unless the key is still being pressed).

We thus have to contend with two types of timing. That which we described above is called phrasing. The second type, tempo, can be handled with PAUSE. Phrasing for a piano is a long slowly decreasing volume sound. But phrasing can also be used for a crescendo as well as a decrescendo, or it can be used as a combination of the two as well. Wind instruments can increase or decrease the loudness of a note over a period of as much as a minute or as they say, "as long as the lungs hold out". The longest "hold" on the 2068 is 9.51 seconds (65535/6890.625).

PAUSE is used to control tempo timing. Even if an envelope is used for a channel, new notes can be written to it. The envelope continues its function unabated with the new note(s). If we recall, PAUSE 1 waits 1/60th of a second. PAUSE 60 waits for 1 second. PAUSE 3600 waits for 1 minute. Since tempo is expressed in beats/minute, taking 3600/TEMPO gives the PAUSE value needed.

Below are some of the commonly used tempos:

	Beats/min
PRESTO	168-208
ALLEGRO	120-168
MODERATO	108-120
ANDANTE	76-108
ADAGIO	66-76
LARGHETTO	60-66
LARGO	40-60

Sometimes notes are played on the "off beat" or even faster including dotted notes at 1.5 times normal length. Adjusting of pause with: PAUSE = DUR*TEMPO where DUR is the duration of the note in terms of the beat note, i.e., 1/2, 1.5 or some other fraction or multiple is the easiest way to make this adjustment.

Register 13: Envelope shape is just 4 bits on or off and is NOT what was hoped for. You do not have control over the length of the attack, the length of the hold, or the length of the decay, nor can you do anything too much about vibrato or tremelo.

The diagrams on page 193 of your User's Manual also are a bit confusing. Looking at the very bottom of the diagram gives us a clue as to what is happening. We see the letters EP (Envelope Period) marked out. Each of the diagrams printed above it uses this same length envelope period. All the way across is 10 envelope periods, not 1.

About all we can do with the shape of the envelope is designate which shape it should be--attack or decay, whether it should be a one shot deal or just be a series of one type or another.

The 4 bits of the resister are:

BIT	VALUE	COMMAND
0	1	Hold
1	2	Alternate
2	4	Attack
3	8	Continue

Bit 2 is the easiest. When on it means start with an attack--increase the volume from low to high. When off, decay is the mode selected by default.

Bit 1 is next easiest--Aternate. Depending upon Bit 2, alternate attack and decay--Crescendo if both on (Value of 6). Decrescendo if both on and Bit 0 is on as well (Value of 7).

Bit 0--Hold off makes the envelope one shot except that when Bit 1 (Alternate) is on, the volume ends by jumping to what the value would have been at the end of the alternate period.

Bit 3--Continue repeats the whole process. With Bit 1 on with alternations of attack and decay.

All Bits off is a decay only and then off at the end of the envelope period.

Under NO CONDITIONS can we get an attack, then hold, then decay in one envelope period.

AN EXAMPLE

Most books and your User's Manual are a bit sketchy of exactly how to convert a piece of sheet music to sound on the 2068 so we will go through an actual example, first in Basic and then we will talk about converting it to machine code.

The music I have chosen is given on the next page. It is something you should recognize should you enter and run the program (another case of modern music plagiarizing the classics). We will just do the first 8 measures. By coincidence, it's 3 part harmony which we can handle. In cases where we have more parts we would have to simplify the music by dropping a part. In lots of cases the bass plays an octave so the obvious simplification is to drop the high note of the bass octave. We get some effect of this note anyhow from harmonics the computer generates.

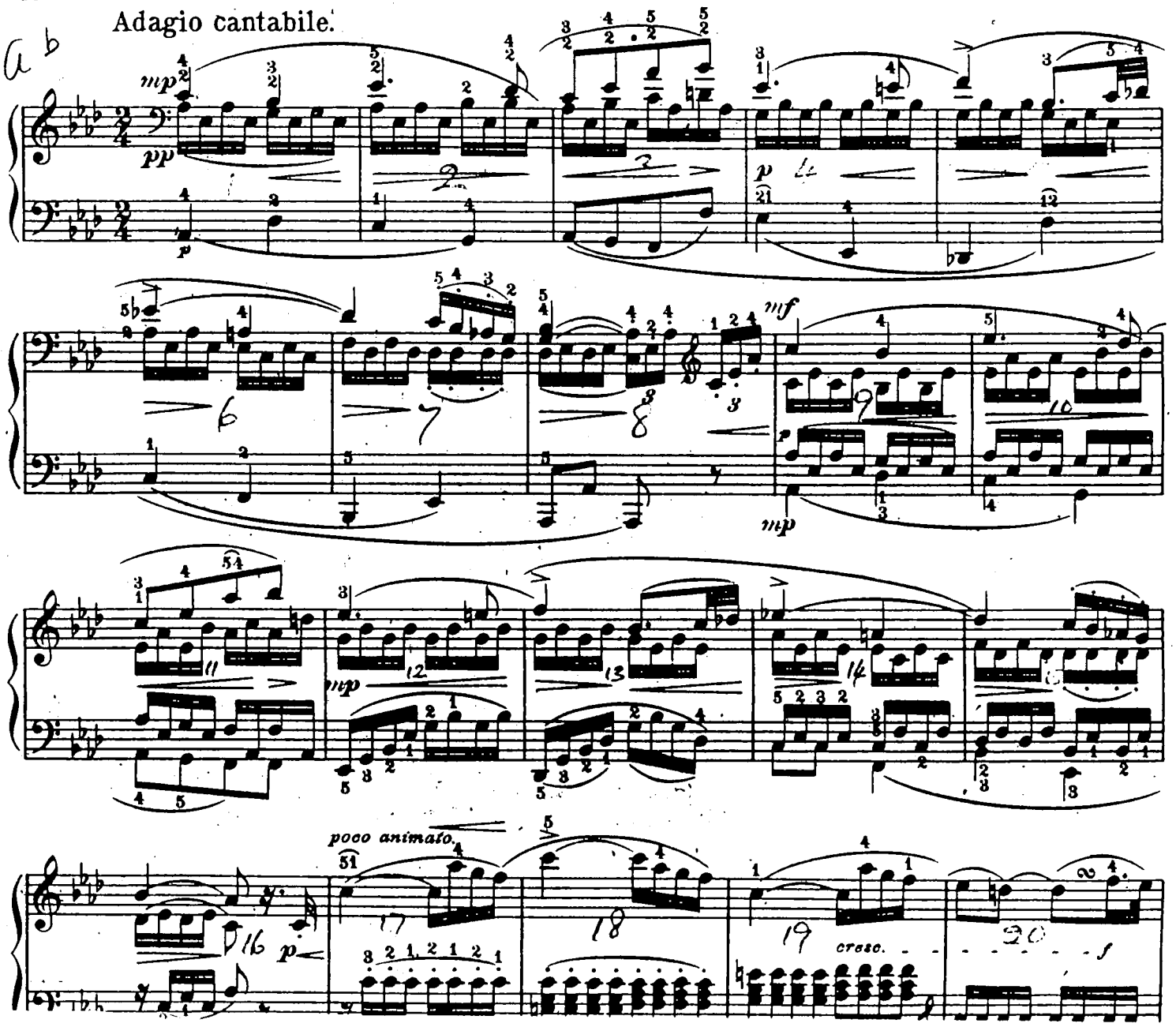
We note that the lowest note we want to play is in measure 8, a low Ab (the small b behind the A is the closest I can manage to a flat symbol). Ab is also equal to G#. This note is not written in the music with a flat sign in front of it because the key signature, at the start of each staff, says to use 4 flats. Thus, all B's, E's, A's and D's are flatted unless written otherwise (these changes are called accidentals as in measures 3, 4, 5, 6, and 7. We also note that the key signature says both staffs of music are written in the BASE CLEF (Those backward C's) rather than normal notation which starts in Measure 8.

The high note occurs in Measure 3, Bb. Therefore, we will be going from a low Ab1(G#1) through C2, C3, and C4 (middle C), up to Bb4(A#4) just short of C5. This is a range of 39 notes. Since each note will take 2 bytes, the easiest way to do this in Basic is with a numeric array.

```
10 DIM n(78)
15 FOR x = 1 TO 78
20 READ a
25 LET n(x) = a
30 NEXT x
35 DATA 8,76,7,213,7,100,6,250,6,150,6,55,5,222,5,137,5,58,
4,239,4,168,4,101,4,38,3,234,3,178,3,125,3,75,3,28,2,239,2,
197,2,157,2,119,2,84,2,51,2,19,1,245,1,217,1,190,1,165,1,14
2,1,119,1,98,1,78,1,60,1,42,1,25,1,9,0,251,0,237
```

The data, of course, is just taken from the note table given

Adagio cantabile.



earlier starting at G#1.

We can now call our notes by numbers (n) and use n*2 and n*2-1 for the fine register and the coarse register respectively of whatever tone channel we specify. While we are about it, we may just as well assign our 3 channels. Channel A will handle the melody, Channel B the accompaniment and Channel C the bass.

```
39--------Bb4
     37   Ab4
36--------G4
     34   F4
32--------Eb4
     30   Db4
  29----  C4
     27   Bb3
25--------Ab3
     24   G3
22--------F3
     20   Eb3
18--------Db3
     17   C3
15--------Bb2
     13   Ab2
12--------G2
     10   F2
   8----  Eb2
      6   Db2
   5----  C2
      3   Bb1
   1----  Ab1
```

We set up the table at the right by counting up and down from middle C (note 29). The longer lines in this table indicate the staff lines of Treble clef (upper) and the Bass clef. Only 3 lines of the Treble clef are shown as we won't be needing the rest. A note, of course, can be ON the line or BETWEEN the lines. After we do our normal scale, we add the flats to change it to the key we are using--each A, B, D, and E flatted. We finally end by adding the octave we are in. Starting at the bottom, we then add the numbers, skipping those that will only be used for accidentals as we pointed out. Using this it is quite easy to assign the correct numbers to the notes. For example, the first 3 notes of the first measure which will all be played together would be 29, 25 and 13 for Channels A, B, and C respectively.

TEMPO AND NOTE LENGTH

We now go on to note length. We see that the key signature assigns a 2/4 meter--2 quarter notes per measure. We also look above the top clef and see "Adagio Cantabile". Adgio is the tempo, Cantabile means, in a singing manner"--a subject we will handle in our discussion of the sound envelope. Going back to page 75, we see that Adagio means 66-76 beats/minute. However, we note that the accompaniment is written in 1/8th notes and this beat continues throughout the piece. Well, 1/8th notes are half as long as 1/4th notes so we can play two for every 1/4th note. Since our tempo was 66-76 quarter notes per minute, we can also say it's 132-152 8th notes per minute. Converting to 1/60th of a second using the value of 144 gives us, 3600/144 = 25, a nice integer value.

We also assign note lengths of 1/8th = 1, 1/16th = .5, 1/4th = 2 and triplet 1/8th = .67. We abbreviate these notes as E, S, Q, and TE respectively. Our PAUSE will be DUR*TEM. DUR will have the letter values above with TEM being 25.

How do we keep a quarter note in the A and C channel and write eighth notes to the B channel? We just don't change the notes until we want to. If we are going to read notes from DATA lines, this presents a problem with the read statement...unless we put dummy values in the DATA statement. Zero is a good dummy value. However, we have then several SOUND statements to use. If you

are typing this into your computer please use the same line numbers. We will fill in the missing lines later.

```
60 IF b AND NOT a AND NOT c THEN SOUND 2,n(b*2);3,n(2*b-1)
65 IF NOT a AND b AND c THEN SOUND 2,n(b*2);3,n(b*2-1);4,n(
c*2);5,n(c*2-1)
70 IF a AND b AND NOT c THEN SOUND 0,n(a*2);1,n(a*2-1);2,n(
b*2);3,n(b*2-1)
75 IF a AND NOT b AND NOT c THEN SOUND 0,n(a*2);1,n(a*2-1)
80 IF a AND b AND c THEN SOUND 0,n(a*2);1,n(a*2-1);2,n(b*2)
;3,n(b*2-1);4,n(c*2);5,n(c*2-1)
85 PAUSE DUR*TEM
```

We enclose these statements in a FOR/NEXT loop:

```
50 FOR x = 1 TO 68
55 READ DUR,a,b,c
90 NEXT x
```

We also have to add:

```
5 LET Q = 2: LET E = 1: LET S = .5: LET TE = .67: LET TEM
= 25.
```

We of course can write music without the use of the envelope. In fact, we will do that and add the envelope later. We have to decide on the volume of each channel. Of course, we will be using pure tones. We want the melody to be a little louder than the bass and the beat so how about:

```
45 SOUND 7,56;8,12;9,9;10,10
```

The enable register, 7, is set with pure sound for all 3 channels. Channel A has loudness 12 for the melody. Channel B, 9 for the harmony with Channel C a 10, slightly louder for the bass. We have to be careful with these values as too large a value can overdrive the tiny speaker. If that should happen, just do a BEEP to toggle it loose. If it persists, lower the loudness values 1 each.

Of course we should turn our program off when we are done and maybe get ready to replay it with:

```
95 SOUND 7,63;8,0;9,0;10,0: RESTORE 100
```

Our DATA lines will look like this. Every measure has its own DATA line so finding errors is simplified. Check your DATA lines. They should have a letter(s) and 3 number sequences.

```
110 DATA E,29,25,13,E,0,20,0,E,0,25,0,E,0,20,0,
    E,27,24,18,E,0,20,0,E,0,24,0,E,0,20,0
120 DATA E,32,25,17,E,0,20,0,E,0,25,0,E,0,20,0,
    E,0,27,12,E,0,20,0,E,30,27,0,E,0,20,0
130 DATA E,29,25,13,E,0,20,0,E,32,27,12,E,0,20,0
```

```
     E,37,29,1,E,0,25,0,E,39,31,22,E,0,25,0
140 DATA E,32,24,20,E,0,27,0,E,0,24,0,E,0,27,0
     E,0,24,8,E,0,27,0,E,33,24,0,E,0,27,0
150 DATA E,34,24,6,E,0,27,0,E,0,24,0,E,0,27,0
     E,27,24,18,E,0,24,0,E,0,20,0,S,29,20,0,S,30,0,0
160 DATA E,32,25,17,E,0,20,0,E,0,25,0,E,0,20,0
     E,26,20,10,E,0,17,0,E,0,20,0,E,0,17,0
170 DATA E,30,22,9,E,0,18,0,E,0,22,0,E,0,18,0
     E,29,18,8,E,27,18,0,E,25,18,9,E,24,18,0
180 DATA E,27,22,1,E,0,24,0,E,0,22,13,E,0,24,0,TE,25,17,1,
     TE,0,20,0,TE,0,25,0,TE,0,29,0,TE,0,32,0,TE,0,37,0
```

Okay, RUN the program. Not bad for a first try. But, let's admit it, a bit computerish. Since we are not using an envelope the sounds came on and stayed on at the same volume for the full length of the note. We did accomplish one thing, the melody notes were long while the accompaniment notes were short. This sound some people have labeled "organ" or sustained music--the note plays at the set volume until we release the key. But even here, there is no, what is called, "voicing" to the notes.

USING THE ENVELOPE

This is where technique ends and art begins. It's going to take a lot of experimentation to get it just right. But you might as well be warned that you are very limited in what you can a- chieve. I'm sure that another whole book could be written on the subject of sound effects and music.

Enabling the envelope is done per channel by adding 16 to the loudness registers (8,9 and 10) for the sounds you want control- ed by the envelope period and shape. Also be forewarned that strange things happen if you put two different channels under the envelope at the same time--they have a tendency to block each other if the notes are too close together. Obviously the same envelope must be used for all channels under its control.

A few guidelines. Start by working with the coarse (12) regis- ter first. Plunking sounds like guitars and banjos use rela- tively short (low numbers) periods with register 13 at 2. The sound decays logarithmically right from the start as a pluck- ed string does.

Piano and snare drums use medium periods which can be chopped by rewriting register 13 when a new note is written to the channel. Unfortunately, all notes controlled by the envelope start over. In our music above, putting Channels B and C under the envelope control replays the channel C note again. Putting Channal A un- der envelope control would be disaster as the melody notes would replay at every 1/8th note interval.

Bowed strings and wind instruments are primarily fast attack with hold and continue on (short period with hold). Such sounds can be gotten with channel 13 at 13.

VIBRATO

Vibrato is caused by the slow variation of the pitch of the sound by a few cycles per second as the note is being played. This can easily be done by a violinist by vibrating his string fingers on the fret board. The changing pressure of the finger on the string results in a small shortening and lengthening of the string to give a small change in pitch to a sustained note. On the 2068 this means we have to rewrite the fine tune register with first a higher note, then the note, then a lower note and then back to the note. For the A register it would be:

```
SOUND 0, n(a*2)+1
PAUSE 1
SOUND 0, n(a*2)
PAUSE 1
SOUND 0, n(a*2)-1
PAUSE 1
SOUND 0, n(a*2)
PAUSE 1
```

We have to repeat this sequence until we have sustained the note as long as we like. We can also add registers 2 and 4 for channels B and C to these SOUND statements if we want everything in vibrato.

TREMOLO

Tremolo is obtained by changing the volume of the note through a slightly changing cycle similar to what we did above with vibrato. We could add this to the sound commands in the above program by changing registers 8, 9 and 10 for Channels A, B and C respectively. Change these also only by 1.

With both vibrato and tremolo we have to rework the PAUSE statement into a FOR/NEXT loop with the same number of pause counts. Our original program used a PAUSE = DUR*TEM where TEM was 25. Since this value is not divisible by 4, let's use 24 which is and gives us a loop of 6. Further difficulties can be encountered with many fractional values of DUR. This can be solved by using:

```
FOR y = 1 TO (DUR*TEM)/4
PROGRAM
NEXT y
```

Why does it work? Simply that the second argument of FOR, that expression, is rounded to an INTEGER.

We have another problem however. Our data read in a zero for a value whenever we didn't want to rewrite the note. We somehow have to maintain the value of the fine tune as we sustain a note. We leave it up to the student as to how this can be a-

chieved. HINT: It's not in the DATA statement.

ENHANCING YOUR PROGRAM

We have just done a pure sound program. The big advantage of using a SOUND command is that once set up it keeps working while the computer can do something else. In our program we just made it wait with a PAUSE statement which was timed to the right interval for the note length. Doing something else while a note is playing takes exquisite timing on the part of the programmer as you have to get back and send the next SOUND command at approximately the right interval of time. Theoretically, our program ran a bit slower than the PAUSE statement allowed as we did not account for the time it took to execute the FOR/NEXT loop and the IF/THEN statements. These things may have to be adjusted for in fast music. But since all tempos have a range, sticking close to the mid or upper portion of that range insures a good tempo.

MACHINE CODE SOUND

Writing this simple program for only 8 measures of music took a lot of memory. If you don't believe me do a PRINT FREE and subtract that number from 38652. The reason is all the numbers, each of which takes an additional 6 bytes for its slug. However, note one thing. All the numbers in all our DATA lines were from 0 to 255--just a nice size to fit into individual bytes. Setting up two files, a note tuning file and a note sequence file and POKEing the data into these files and then calling it by PEEKs will save a great deal of memory.

You are now half way to a full code program. To write a SOUND command you send the register number OUT Port 245 (F5H) and then send the value you want written to that register OUT Port 246 (F6H). As long as PORT 245 contains the right register, another OUT 246 will send the same register a new value.

That is the easy part. The hard part is timing everything which has to be done with counting loops. A typical one is:

```
      LD BC, COUNT
TIME  DEC BC          6
      LD A, B         4
      OR C            4
      JRNZ, TIME     12
```

Note the numbers to right of the instructions. This is how many T states or clock cycles it takes to complete each instruction. Once through the loop takes 26 clock cycles. At 3.528 million cycles per second, it can do this loop 135,692 times a second (if no timeout is taken to handle interrupts). Loading BC with its maximum value of 65535 only holds things up for less than half a second. In our program we used a TEMpo value of 25. (25/60th seconds) and this could just barely be achieved with a value of 56538 in BC for the duration of our 1/8th notes. Even a

quarter note hold would require more than a simple timing loop.

Well, not quite. One can pad the timing loop with harmless extra instructions like LD A, A (4), nop (2) or a meaningless CALL (17) to an address that has nothing but a RET (10) in it. The numbers in () are of course the number of T states chewed up. Note that the CALL and RET consume 27 thus effectively doubling the time of the count loop (27 + 26). Any address already containing a RET will do for a call address. One still has to have a program to read the values of notes and write the registers but with these hints, the advanced student should be able to write a program. The beginning student must read on for another two chapter before attempting it.

JOYSTICKS

Sound register 14 sends or receives from whatever is connected to the joystick ports which need not necessarilly be a joystick. To enable register 14, Bit 6 of register 7 must be reset (0) for an IN, set (1) for an OUT. During all the discussion of SOUND that 63 value to register 7 reset the joystick at the same time so we had the joystick on to receive a signal all this while... we want an IN as we can only read joysticks.

Reading the joystick in machine code can be done with:

```
LD A, 7
OUT (245), A    Set register 7 to read
XOR A           Set A to zero
OUT (246), A    Send zero to register 7
LD A, 64
OUT (245), A
LD A, 1 = Left, 2= Right  3= Both
IN (246), A
```

Register A will now contain the results of register 14 (in low active format, i.e., 0 if contact closed). The bits are:

```
  7     6  5  4      3      2     1    0
button  not used  right  left  down  up
```

Generally RLA is used to CLEAR the carry flag to check for a button press while RRA is used for reading the various directions again with a clear of the carry flag. Note that some joysticks will allow diagonal directions to be read by resetting 2 bits. Others will not.

We apologize to the beginning student for this bit of assembly language without explanations. You won't recognize some of these commands. Read on and once you know the commands come back and reread sections of the book with more understanding. It's one of the problems in writing a book and trying to do things logically.

Sound register 15 exists and can be set to read or write as described above for register 14 using Bit 7 of the enable register. Unfortunately, nothing is attached to it. Any ideas hardware hackers?

CHAPTER 6

THE CENTRAL PROCESSING UNIT (CPU)

TYPE: Z80A 8 bit DATA BUS/16 bit ADDRESS BUS
OPERATING FREQUENCY: 3.528 megaHz
PHYSICAL SIZE: Dual INLINE 0.514x2.100x0.213
CONTACTS: 40
ACTUAL SIZE OF CHIP: 0.200x0.200
MANUFACTURER: ZILOG INC., 10460 Bubb Rd., Cupertino, CA 95014
2nd SOURCE: Mostec INC., 1215 W. Crosby Rd., Carrolton, TX 75006

Looking at the specifications we notice that the three main functions of the plastic case are:

1. Protect the silicon chip
2. Provide heat sinking for the chip
3. Provide a means of connecting with the outside world--going from 40 connections in less than an inch of peripheral space to 4 inches of connections in 2 rows.

What are these connections? For once they are not listed in the User's Manual, but they are given in the Technical Manual or The T/S 2068 Intermediate Advanced Guide.

From what we already know, there must be 16 address lines usually labeled A0 to A15, and 8 data lines usually labeled D0 to D7. We need power, labeled +5V and a ground, labeled GRD. That's 26.

We also need a WRite line and a ReaD line along with a MREQ (Memory request) and of course RFSH (Refresh) to refresh memory.

Everything runs by the clock signal of 3.528 megaHz supplied by an outside oscillator, and comes into the CPU at the contact labeled with the Greek letter Phi (looks like a O with a CAP I through it). As we found out one clock cycle is called a T state. Generally it takes 4 T states to just read an instruction, and another 3 T states to read memory or write to it. In the case of a port input or output further delays can be encountered with a WAIT T states--that is what the WAIT line is all about. Generally we don't worry about T states when writing machine code, but timing must be considered when using inputs and outputs to or from peripherals faster or slower than the CPU. Since we are at it, IORQ is Input/Output ReQuest.

M1 is the line used to say Read An Instruction--It is active when the CPU is reading the first of a new set of instrutions, or in the case of an extended instruction, the 2nd, and if necessary the 3rd instruction as well but not data bytes. M1 stands

for MODE 1--Read an Instruction.

INT (INTerrupt) is an interrupt from a device. It is honored at the end of the execution of the present instruction. It is ignored if Disable Interrupt is in force.

Another type of interrupt is the BUSRQ (BUS ReQuest). The device needs a BUS as well. Telling a device it has the bus is done with BUSAK (BUS AcKnowledge).

NMI is the Non-Maskable Interrupt--always honored at the end of the execution of the present statement even if DI is active.

RESET is the hardware equivalent of RANDOMIZE USER 0. It is the first thing done by the CPU upon receiving current.

HALT causes the computer to do nops (no operations) until it gets an interrupt. If you counted, that's a total of 40 lines.

A word about line notation. Most of these lines are written overlined--they have a line over the symbols or mnemonic. This line means "active when low". That means they have +5 volts on them when inactive, dropping to zero when active. Overlining symbols has the mathematical terminology of NOT.

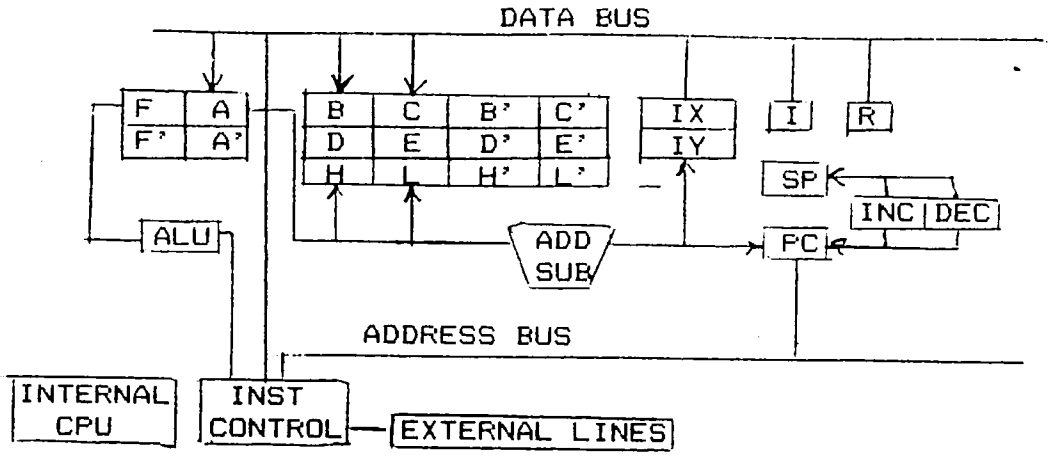
Please note: There are no instructions in the machine code set that tell the CPU to turn off or on the voltages on these various lines. The CPU automatically switches them at the right times internally as needed to execute the instructions. This is what the control portion of the CPU is all about. Similarly, as it reads an instruction, the control unit knows exactly how many bytes follow with direct data for that instruction before the start of the next instruction.

Should you pry the plastic case off the top of a CPU (definately NOT recommended except in the case of "blown" CPU that is beyond repair), you will find a spiderweb of fine gold wires leading from the edges of the centrally placed silicon chip to the various terminals. These wires are welded in place with the aid of a low powered microscope as they are too tiny to be done for very long without this magnification aid. The tininess of everything makes them impossible to repair without special equipment.

THE CPU--INTERNAL ORGANIZATION

Use the diagram on the following page when reading this discussion. The CPU has its own internal memory storage units. To differentiate them from external memory we call them registers. You have already met some of them. We have 8 main registers. F, the flag register, can't be used for storing numbers but just keeps track of the results of various math and logic operations. A, the accumulator, is the main arithmetic and only logic register in the CPU. It can use instructions that no other register has. It accumulates the result of various operations. General purpose

REGISTERS OF THE CPU



registers are B, C, D, E, H, and L. These six registers can be used individually or in the pairs BC, DE, and HL in which cases they generally are designating an address. IX and IY are special purpose address registers. IX and IY have the ability to get or send data to an offset + or - 127 from where they are set to. In the 2068, IY is always pointing to 23610--right in the middle of the System Variable Table. The beginning machine code writer should not mess with these two registers until he/she knows what he/she is doing as they have to be reset before coming back to Basic.

The Prime Registers: AF, BC, DE, and HL have prime registers--an alternate set where data can also be temporarily stored. You can't operate on the data in the prime registers until you switch it back to the regular registers at which time the regular registers get put in the primes. AF and AF' can be exchanged separately from the rest which must be done en mass--BC, DE, and HL exchanged with BC', DE' and HL' respectively as a group.

The Program Counter: (PC) Since there are no line numbers in machine code and each instruction follows the next, this register keeps track of the address of the next byte of information to be used or interpreted. It is added to or subtracted from in the case of JUMP RELATIVEs and changed to the address specified in CALLs and JUMPs. These instractions are the machine code equialents of GOTOs and GOSUBs.

The Stack Pointer: (SP) Keeps track of the address of the last number stored on the machine stack. We talked about this back in Chapter 2...go back and review its operation if necessary.

Two more registers complete the set. I is the interrupt mode register and is generally set to address 0063...this is the low part of the address--generally called a VECTOR. To it is added the high byte address at the time of the interrupt which may be

zero. The result is a jump to the interrupt routine at that address. The final register is the R (Refresh register) which is another vector address to which is added the top byte to give the address of the next section of RAM to be refreshed.

Also on the diagram is a space called the ALU (Arithmetic Logic Unit) which handles all the OR, AND, XOR, CP and CPL oeratiions done on the accumulator (A) register. It also sets the various flags in the F register which also includes overflow and underflow conditions encountered in add or subtract.

The flags in the flag register are: Carry, negative, parity/overflow, half carry, zero and sign. Each occupies a bit of the F register being "on" with a "1" and "off" with a "0". We can test for the state of all except negative and half carry at any time. These last two are used by DAA (Decimal Adjust Accumulator) instructions only. Flag conditions can be used to make conditional CALLs, JUMPs and RETurns.

We see two little boxes called INST which is internal logic to read the instructions and the CONTROL unit which turns on or off all those lines like RD, WR, IORQ etc. that we talked about earlier in this chapter. The CPU itself has to handle the decoding of the instructions, so naturally has a built in ROM for this interpretation...this ROM is not the same structure as the external ROM or RAM and can't be rewritten.

OUR FIRST MACHINE CODE PROGRAM

We have seen machine code programs earlier but this is the first one we are going to be explaining line for line. Okay, now that we know what the internal registers are, the mnemonics of Appendix B, which I told you to make a copy of, are starting to make sense. We use these registers to do our calculations and manipulations. We may use them singly or in pairs as we desire. Although we haven't discussed the complete set of instructions as yet, we are far enough along to understand this program.

The first thing we have to decide is where we are going to store it. Since it's going to be a short program, let's choose to start at the address 65000. Therefore, as we start to enter this program, we will do, in Basic, CLEAR 64999. This will set RAMTOP at 64999 and prevent any Basic from ever overrunning it. We will set up a loop to POKE in the various numbers starting at 65000. Then when we are ready to run it, we will use RANDOMIZE USER 65000 from Basic.

It is going to be easier to give you the whole program and then explain it line by line. So here is the complete program. Please refer to it as we explain it.

ADDRESS	CODE	LABEL	INSTRUCTION	COMMENT
65000	62,16		LD A, 16	ATTR byte
65002	33,0,88		LD HL, 22528	ATTR addr

```
65005   6,64                LD B, 64        Counter
65007   119         LOOP    LD (HL), A
65008   35                  INC HL
65009   5                   DEC B
65010   200                 RET Z           DONE-back to Basic
65011   24,250              JR LOOP         If not, loop.
```

Before explaining the program, let me explain that like Basic, machine code has three main portions. First we set things up, then we do the calculations and end with the printout of the results.

Each machine code line can consist of 5 parts but always has 3. The essential 3 parts are the Address, Code and Instruction. The Label is to put a label on a line while the comment is to assist us in remembering what that line does--memory does get a bit foggy after several years. In this particular program we are going to change the PAPER color to RED and the INK to BLACK with no FLASH or BRIGHT. From our discussion of the ATTRibute byte we recall tht paper color has to be multiplied by 8. Looking at the keyboard, we see RED = 2 so 8x2 = 16. Black is 0, so we add nothing for the INK color. Our attribute value is 16. We choose the A register to hold it. We pick this register as it is the register that lets us load it to an address as we do in line 4. Other registers don't have this instruction. Therefore, we write on our paper under address 65000 and under instruction:

LD A, 16

(Load A with 16--all load instructions are always load the first with the second, or READ "with" at the "comma" as we say). Under comments we write ATTR Byte to remind us that A holds the attr byte value.

Okay, but we haven't coded the line as yet. Although Appendix B is good for decoding code back to mnemonics, it's lousy for coding a program. Later on we will give you another format to do that. Suffice it for now that if you look long enough you will find the instruction LD A, n at code 62.

N is not a register is it? N is the symbol the table use to designate a number. A single N is a single byte number, a double N a two byte number. All direct load instructions require that the next byte after the instruction hold the value we want for n, or two bytes if nn is needed. Therefore, we can code our first line with 62,16. This used 2 bytes so the next available byte is address 65002 as we start our 2nd line.

Next we have to answer the question, where are we going to load these attributes? Obviously the Attribute file, but where is that file? You now are beginning to see how important those first chapters of this book are. If we don't remember, which is going to be 90% of the time, we have to look it up. Where do we look? How about a memory map? If you modified your map like I

told you to do, its start is listed as 22528. If you didn't modify your map you have to consult Chapter 3...believe me that 95% of what you will ever need to know and perhaps a lot of things you will never use are in those chapters. Now, do we want the starting address of the Attribute file? Yes, we want to change the top two rows. We have a choice of registers to use to hold this address and the one we choose is purely up to the programmer. However, there are certain advantages to use the B and C registers as counters so let's avoid them to hold our address. The DE and HL register pairs also are ideal for use as address pointers so let's pick HL. We want the instruction:

LD HL, nn.

NN will be our 22528. We write it as LD HL, 22528 and put the comment ATTR ADDR behind it. This is instruction # 33 so under the CODE column we write "33,". We now have to convert 22528 to low and high byte format. Taking out or handy calculater we divide 22528 by 256 and get 88.00000. Our high byte is 88, and because the answer came out even our low byte is 0. Remembering that it's LOW BYTE FIRST we write a "0,88" in back of our 33,. Since this instruction took 3 bytes the next byte for line 3 is 65005.

To continue we have to know some more information. We want to set up a FOR/NEXT loop to write 16 to the top two lines of screen attributes. How are they arranged in the file? If you don't remember, where do you find it? Chapter 3 again. Okay, there is one per character and they go across the screen for the first line and then the second, the third etc. We thus find that we can do all of them in a row without anything fancy. How many are we going to do? Well, there are 32 per line and we want 2 lines which is 64. Time to set up our counter. We are used to counting up from zero to 64, but in machine code it is easier to count down as it is easier to detect a zero than any other value. Let's use B as our counter and load it with 64. We want the instruction:

LD B, n

which is instruction #6. We write under instruction, LD B, 64 and write under comment, counter. Under code we write 6,64. We have thus started our FOR/NEXT loop with the statement FOR B = 64 TO 1. We can't use a STEP in machine code but will show you how that works later on. This used 2 address bytes so the next address for line 4 is 65007.

We have now finished setting up, or initializing our program. We have no calculations or processing to do so we move on to printing out our results. At this point A contains the attribute we wanted printed to address HL. In our mnemonics, () around registers or our "nn" means "memory location pointed to by". So we want:

LD (HL), A

There are no instructions like LD (xx), Y , where Y is any register other then A. This is why we chose A in line 1.

Now another important point. After this is done, register A will still hold 16, it is not erased. HL also still contains 22528. But memory location 22528 will now hold 16 as well. This is fine as we want to write 16 to address 22529 next. So all we have to do is get HL to 22529. How do we do that? Remember INCrement? We just do:

INC HL

It's instruction # 35. That goes in the CODE area. We update our address to 65009 for the start of the next line.

We now have to take care of our counter with a STEP -1. Remember DECrement?

DEC B

It's instruction #5. Everytime we do an increment or a decrement of a single register, the CPU looks at the value of that register after doing it and if "0" sets the ZERO FLAG in F. We can use this fact to get out of our FOR/NEXT loop by writing:

RET Z

RETurn if zero, else continue with the next line. Where does it return to? Well, we are NOT in a subroutine, so we go back to Basic. RET Z is instruction # 200. We write the comment, Done--back to Basic and update the address for the next line.

Suppose we are not done. What do we tell the computer? We really want to loop back and do another LD(HL), A. This is the machine code equivalent of NEXT B. We do a GOTO in the form of a JUMP but it is not a JUMP to an address where we give it an addreses, but a JUMP RELATIVE from where we are. We want the instruction:

JR, Dis

Dis is a displacement. It needs a signed number as it could be positive which would mean forward, or a negative number meaning backward. We want backward--to that LD (HL), A. To clarify to ourselves exactly where, we use a label. So at this point we label the line with LD (HL), A with the word "loop" and write behind JR, the same word "loop". Since "Dis" is a number just like n, it uses a byte in back of the code for JR, dis. which is code # 24.

What is the right number for the displacement? After reading the line but before executing it, the CPU increments the Program Counter to the start of the next instruction. Thus, the first byte of the next instruction becomes byte "0". The address holding our displacement will then be -1 which in signed binary is 255. That 24, will be 254. The 200 in the next line is 253. Continuing to count backwards we arrive at 250 when we hit the 119

of the LD(HL), A instruction. This is our displacement. Reread this paragraph again as it is very very important that you understand displacement.

What would our little program look like in Basic?

```
10 LET A = 16
15 LET HL = 22528
20 FOR B = 64 TO 1 STEP -1
25 POKE HL, A
30 LET HL = HL + 1
35 NEXT B
```

How many bytes does this Basic program take? Count 2 for each line number, add another 2 for line length, and another one for the enter code at the end of the line. Don't forget to add 6 extra bytes for the slug of each number. I got 105 bytes. Our machine code took 13. That's 8 times shorter.

Want to make a bet as to which runs faster? Enter it and then add the loader program below:

```
110 CLEAR 64999
115 FOR B = 65000 TO 65012
120 READ X
125 POKE B,X
130 NEXT B
135 DATA 62,16,33,0,88,6,64,119,35,5,200,24,250
```

Also enter the following lines:

```
40 PAUSE 0
45 RANDOMIZE USR 65000
59 STOP
```

Also change line 10 to LET A = 32 so that the Basic will give you Green paper and the machine code red. Now do a GOTO 100 and get the code entered. Then do a RUN and compare the speed of writing the attributes. Time the speed from the time you hit the enter key--PAUSE 0 requires that. Don't blink as the machine code gets it done between one screen refresh and the next.

Saving our program: Okay, so it's not that great a program to save but at least you should know how to save code. You do:

```
SAVE "RED" CODE 65000, 14 for tape
MOVE "RED.BIN", 65000, 14 for disk (AERCO system).
```

FOR HARDWARE HACKERS ONLY

You don't have to know the inner workings of the CPU, what lines are turned on and off to make a program run or to write programs. But, if you are curious as to what all is happening read on.

We will start with the Basic Line "RANDOMIZE USR 65000". The token USR is interpreted by the code in ROM as "GOSUB to the address which follows". PPC and SPPC at this point contain the present line number and subline number. The Basic program counter, which keeps track of the Basic address, by this time has stepped through the slug to get the number and now holds the address of the following ":" or ENTER. It increments once to get the address of either a line number or a token and puts this in OSPPC. The computer itself is interpreting USER somewhere in ROM and Pushes an address to the stack then loads the program counter with the number. The next instruction it will read is at the address you gave it. By now you should know how the machine stack works with a PUSH and a POP. If not go back to Chapter 2 and review it. Whenever a pair of registers is PUSHed to the stack, the STACK POINTER gets decremented two spots to keep track of the address of the last number PUSHed. We now have gone from maxhine code that causes your computer to operate like Basic to your machine code. It's all machine code whether yours or ROM. It should now become obvious to you that another ROM written with machine code could interpret PASCAL, or COBOL or C or any other language one knows of or wishes to invent.

We have skipped a few details in the above discription as far as hardware operations are concerned--we haven't told you which lines went on and off. We will do that from here on out. It's time to get our first instruction from memory. Use the diagram on the next page to help you through the various cycles.

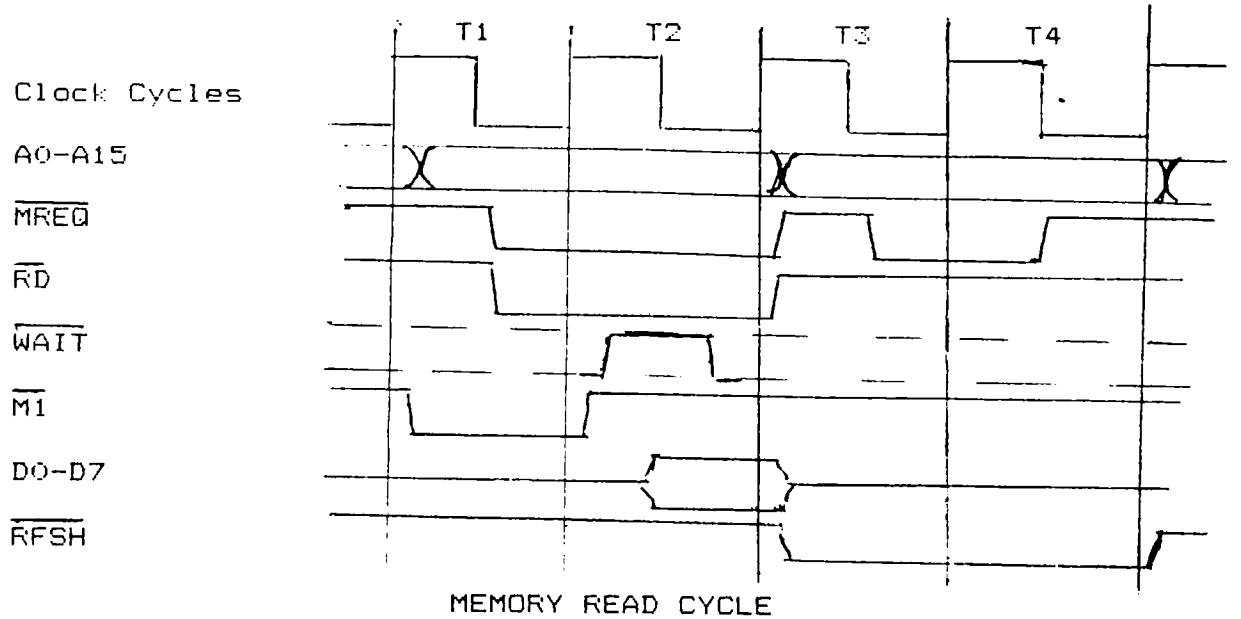
GETTING AN INSTRUCTION (M1 CYCLE)

For the instruction LD A, 16. During the first clock cycle, the M1 line, normally high, drops low and the value of the Program Counter is put on the address bus. During the last half of clock cycle 1, MREQ and RD, also normally high, drop low, signaling a read of memory. The WAIT line is sampled for a signal. Should there be a signal, the M1 cycle is extended for more clock cycles. If no WAIT is encounted, the memory cell being addressed will be putting its value on the data bus--that is all 8 memory cells, a bit each on each different line during cycle 2. In the CPU, the Instruction Register receives the data byte. During the next two clock cycles the instruction decoder inside the CPU will be interpreting the instruction and deciding what to do next. For our reading of address 65000, it's trying to figure out what to do with 62. The Program Counter is incremented to 65001.

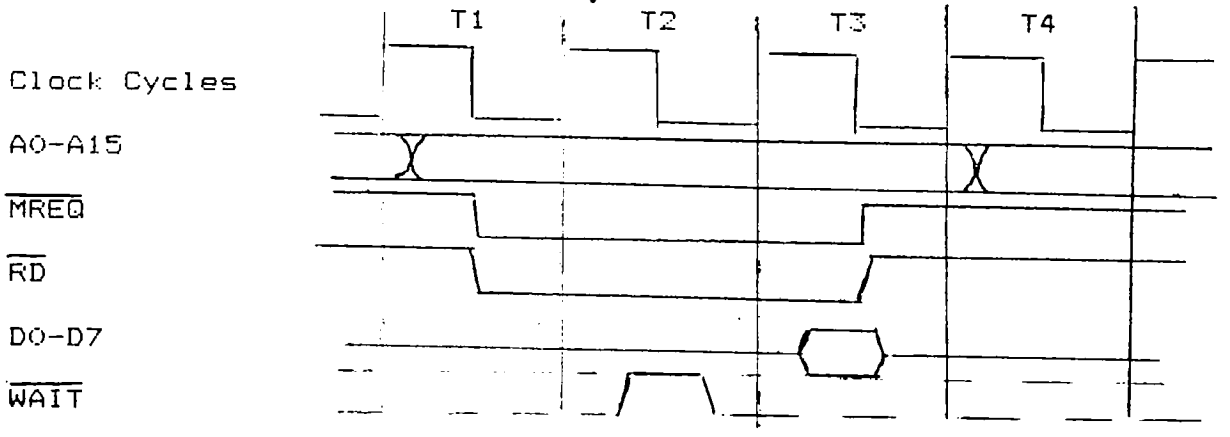
MEMORY REFRESH (Cycle 3 and 4 of M1)

While the instruction decoder is doing its work, the CPU is busy refreshing some memory by putting the address on the address bus, droping MREQ again with RFSH. At the end of cycle 4, M1 finally goes high signaling the end of the Instruction Read Cycle. It took 4 clock cycles.

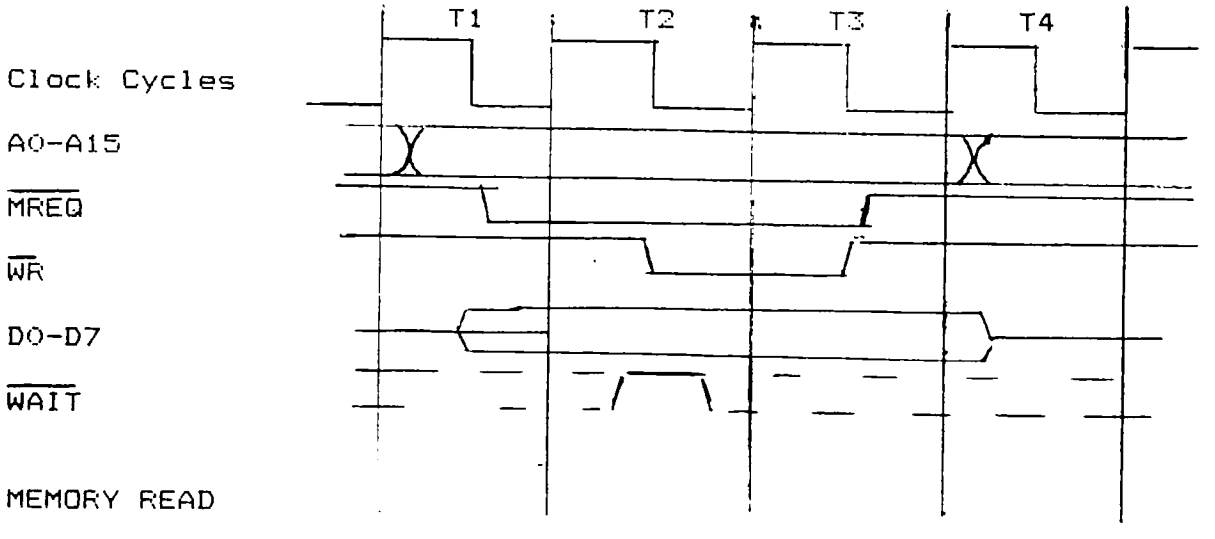
M1 CYCLE (GET OPERATION CODE)



MEMORY READ CYCLE



MEMORY WRITE CYCLE



MEMORY READ

By this time the instruction decoder has decoded that 62 into LD A, N. It notes that it has to READ the next byte of memory to get N so the first two cycles of M1 are repeated except that M1 doesn't drop low this time. WAIT again is sampled for delays if any. In cycle 3 of memory read, the value of 16 from address 65001 gets written to A and the program counter is again incremented. No memory refresh this time. Memory Read takes 3 clock cycles. We are done with the first line of instruction.

Instruction 2, (LD HL,22528). By now it should be clear that it will be one M1 cycles followed by 2 Memory Read cycles to get the data to L and H respectively. Another part of memory will be refreshed again during the last half of the M1 cycle.

Instruction 3, (LD B, 64). This is a duplicate of the first instruction with the only difference being a shunt of the data to register B.

Instruction 4, (LD (HL), A. This requires a WRITE TO MEMORY Cycle after the M1 cycle. The cycle is similar to the read memory cycle except that the WR line goes low. It also takes 3 clock cycles. Instead of the value of the program counter going onto the address bus, the value of HL is put there and the data bus gets the value of A, with the memory cells being set to write.

Instruction 5, (DEC B). Only requires an M1 cycle.

Instruction 6, (RET Z). Also only requires an M1 cycle if false and the program goes on with the next statement. If the Z Flag is set, the value of the last 2 bytes in the machine code stack are put into the program counter and the stack pointer is incremented twice. Since the program counter is now pointing back to the Basic ROM we are again back in Basic.

Instruction 7, (JR, 250) will require an M1 and a Read Memory Cycle. In our case the 250 is already in 2's complemented form so it's merely added to the low register of the program counter and any carry ignored and not put in the high byte. Thus the next instruction gotten will be LD (HL), A again.

I/O TIMING CYCLE

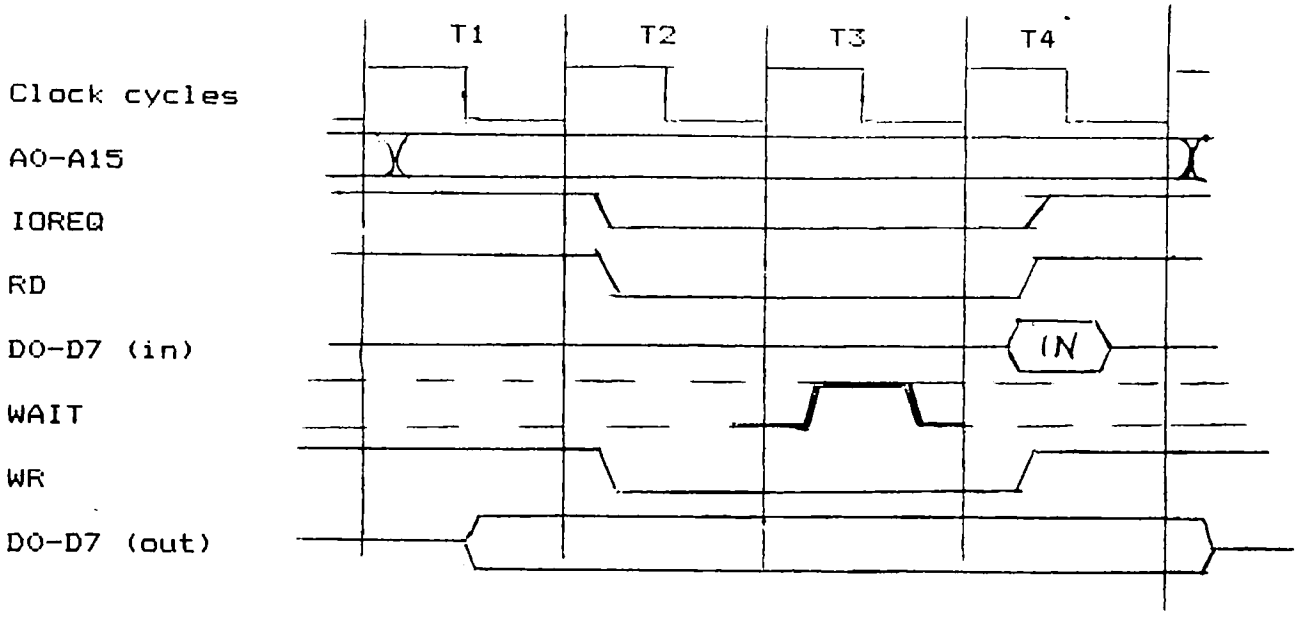
Although our program doesn't use it, readers may find the I/O timing cycle of use. We have included a WAIT cycle which is quite prevalent when inputting or outputting to peripherals. It is on the next page.

There are several more types of cycles like:

- Interrupt (INT)
- BUS Request
- Interrupt (NMI)
- Halt

They are beyond the scope of this book.

I/O TIMING CYCLE



COMPARING THE Z80 WITH THE 8080 CPU AND THE 6502 CPU

You might think that the Z80 is quite limited in what it can do with its few registers. How would you like to do with less? The Z80 was developed from the 8080 CPU. The difference is that the 8080 has no indexing registers (no IX and IY) and no prime registers and no extension instructions (all the CB and ED instructions don't exist. Without IX and IY there is no need for FD and DD sublists).

That translates to no multiply and divide instructions except in the A register with RRCA, RLCA and RLA. There is nothing for the rest of the registers so values have to be constantly be shifted into and out of the A register. Also no bit testing, resetting and setting. And no superpowerful instructions like LDIR, CPIR, LDDR, INIR, OTIR etc. In addition, some of the commands have to be changed so all the JR conditionals don't exist including DJNZ. There are less than 256 instructions compared to 800+ for the Z80. Okay, the assembly language is more difficult to remember at the start but writing in it becomes much more effective. Actually, the Z80 assembly language is easier to learn as the mnemonics are in English, not computerese.

6502 vs. Z80

If you are crying over only 7 registers to work with, try working with only A, X and Y and again less than 256 instructions. The code goes on ad infinitum as values have to be stored and retrieved a lot more. In addition, the 6502 clock is down in the kiloHz region so it is no wonder that the 2068 is 4 to 10 times

faster then the Apple II or the Atari or the Commodore (64k models)--except when they are using CP/M. Well, in CP/M you insert a Z80 CPU board and run the computer with that. My question is, why didn't they design the computer with the Z80 in the first place?

THE FUTURE

The future belongs to the 16 bit processor. Only the reduction in memory costs will say when this will happen but 16 bit 68000 processors are already available (FAT MAC, SINCLAIR QL, etc.) The programs that these machines run are quite complex and don't leave much chance for the beginner to really write a program that can take advantage of the machine's ability. The future is also aimed at the user and not the programmer. Users have to live with their programs. With the 2068, I feel that I can change things to the way I want them which is the computer living with me, not me living with the computer. I get quite irritated with user types who say, "But you can't do it that way on the computer". Computers also will become even more user friendly. In fact, that is one of the complaints about the MAC. It is so user friendly that there is very little room to do anything useful. Voice recognition of commands really is going to gobble up memory fast.

Learning a second assembly language is going to be easier than the first time as many instructions do carry over from CPU to CPU even if the mnemonics do not.

CHAPTER 7

MACHINE CODE--ASSEMBLY LANGUAGE

Think of assembly as a bunch of routines put together much as one does subroutines in Basic. In your study of assembly you will be making a collection of these routines which will be used again and again. If you don't have a notebook listing these routines, preferrably in corrected, debugged form, you will have to reinvent them every time you need them. This is as good a time as any to start. This should be a different notebook than the one you use to work out routines in. Looseleaf notebooks are excellent for this purpose as they allow you to rearrange things and insert better routines for older ones as you perfect them.

We write our programs in the mnemonics of assembly language which then have to be converted to the number sequences of machine code which is what is entered into computer memory. The experienced writer will already be using an assembler program such as "Hot Z" to do all the machine coding for them. For the novice, I strongly recommend doing a few programs by hand. In this way we don't have to worry about all the ins and outs of using a complex program like Hot Z. It's a good program but can be frustrating the first few times that you try to write a program with it. We will work with just pencil and paper. I do recommend a pencil with a good eraser. The beginner makes a lot of mistakes. The terms ASSEMBLY LANGUAGE and MACHINE CODE are sometimes used interchangably by authors when in reality each has its own meaning.

Machine code, the number sequences, can be written in Hexadecimal (Hex) or in decimal. One has to be specific about which is being used as 10h is different from 10d. Sometimes people write the numbers of assembly language in decimal with the machine code in Hex. Addresses also can be written in Hex or decimal. Reading Hex addresses is a real pain. One can choose to write in any convention one desires. But be consistent. If everything on the assembly side is in decimal, keep it that way throughout. Don't go mixing the hex with the decimal. We are going to use decimal throughout--address, machine code and assembly numbers.

OTHER CONVENTIONS USED

We have already met some of these but here they all are again:

1. Read the word "with" everytime you see a comma. LD A, B is Load register A with register B.

2. () are read as "the contents of the address". LD A, (HL) is

Load A with the contents of the address in HL.

3. N and NN denote single byte and double byte (word) size numbers.

4. d and dis is a signed single byte displacement number, 0 to 127 being positive or forward and 255 to 128 negative or backwards.

5. Prefixes CB(203), ED(237), DD(221), FD(253). Referring to the table in Appendix B of the User's Manual we have 3 columns of mnemonics. The first column is used when none of the above prefixes is used. To get to the second column one must use the prefix CB followed by the instruction number. ED gets you to the 3rd column.

   There are two more prefixes not listed in the table. FD and DD. These prefixes change otherwise HL instrutions to IY and IX respectively. For example, Instruction 35 normally is INC HL. With a FD(253) prefix it reads INC IY. With a DD(221) prefix it reads INC IX.

6. Conditional commands use the abbreviations C for carry, NC for no carry, Z for zero, NZ for non zero, M for minus, P for positive, PE for parity even and PO for parity odd.

THE FLAG REGISTER

The F register is the flag register and holds 6 flags. It could hold 8 but we only need 6. We can never directly change the value of the flag register by LOADing in a different value. We can change some flags by doing certain operations.

CARRY (Bit 0) All ADD, SUB, ADC, SBC, and CP instructions will change the carry flag if the results goes through zero (overflow or underflow) being set if reset and reset if set. All AND, OR and XOR instructions will RESET the carry flag. Rotation instructions will rotate the end bits into and out of the carry flag reseting or setting it accordingly. CCF (Compliment Carry Flag) will always change the carry flag to its opposite setting. SCF (Set Carry Flag) will always set the carry flag to 1. There is no reset carry flag but AND A does it nicely for us.

ZERO (Bit 6) is set if the results of an operation like ADD, SUB, ADC, SBC, INC, DEC, CP, AND, OR and XOR is exactly zero when using the A register. SBC and ADC change the zero flag when using HL. INC and DEC of a double register will not set the zero flag when the value reaches zero. Rotation and Bit testing also affect the zero flag. AND A will CLEAR the carry flag as mentioned above but also clears the zero flag. XOR will clear A and thus SET the zero flag. No load instructions affect the zero flag except LD A, I and LD A, R.

SIGN (Bit 7) shows if a result is negative or positive with respect to 2's complement arithmetic. In other words, a copy of Bit 7 of a single register or Bit 15 of a double register. Thus all ADD, INC, SUB, DEC, SBC, CP, AND, OR, and XOR with single register and ADC and SBC with double registers change the sign flag. Rotation will also affect the sign flag. Only LD A, R and LD A, I change the sign flag. Block searching instruction use the sign flag. There is no way to deliberately set or reset the sign flag except using the scheme above--seting or resetting Bit 7 of a register.

OVERFLOW/PARITY (Bit 2) is a dual purpose flag. The overflow part tests whether the result of an operation in 2's complemented arithmetic is correct or not--not as in carry above where insufficient space or in sign above where it would indicate a negative number (bit 7 being on).

Consider adding 20 to 113, the results, 133 would be correct using unsigned numbers but incorrect using 2's complimented signed numbers as it would indicate a negative number with bit 7 being on, -125.

Parity counts the number of 1 bits and SETS the Parity flag if the number is even. All AND, OR and XOR are tested for Parity.

All ADD, ADC, SUB, SBC, and CP are tested for OVERFLOW.

Block instruction use the parity/overflow flag. There are no instructions for explicitly handling the overflow/parity flag.

HALF CARRY (Bit 4) and NEGATIVE (Bit 1) cannot be tested for and are only used by the DAA operation internally.

Bits 3 and 5 are not used.

THE Z80 ASSEMBLY INSTRUCTION SET

LOAD INSTRUCTIONS

Some texts on assembly go through the full set of different ways to load a register like immediate, direct, indirect, implied and extended. The reason for this is that mnemonics of the 6500 and 8080 CPU's were of the nature: LD A, Immediate. What the writer is trying to do is show the relationship between that mnemonic and the Z80 equivalent, LD A, n.

The Z80 mnemonics are much more direct. We don't have to know what type of load we are using to use it, we just do it. This is

why I consider the mnemonics for the 6500 and 8080 CPU's as written in computerese instead of English. They are just gosh awful as you are required to know all this extraneous junk. The future bodes ill also as the 68000 processor use the same obtuse set of mnemonics. Originally they wanted no mnemonic longer than 3 letters but with all the extra commands needed for a 16 bit processor what was already a strained restriction really gets bent so that "any resemblence to what the mnemonic says and what really happens is purely accidental."

The Load commands can be used to transfer one register value to another as in: LD A, C (Load A with C) or its reverse: LD C, A. What should be remembered is that after A is loaded with the value in C, both A and C contain the same number. The number is not erased from a register until it is overwritten with another number. Once in a while one encounters nonsense instructions like LD A, A which really does nothing except waste time.

One can also load a register directly with a number. LD A, n means load A with the number n. In the code, this number n must immediately follow the instruction number. LD A, 6 codes to 62,6.

Loading to and from memory is achieved with: LD (nn), A and LD A, (nn) respectively. These two instruction are equivalent to POKE nn, A and LET A = PEEK nn. In both these cases, nn again must follow the code number in LSB/MSB format.

We can also load the contents of A to and from memory by using a register pair as a pointer. LD (HL), A and LD A, (HL) load A to and from the address held in HL. Similar instructions exist for the BC and DE pairs using A. All registers can be loaded to or from memory using HL as a pointer, but only A with BC or DE.

IX and IY can also be used to load A to and from memory but require an offset. These mnemonics are written LD (IY+d), A and LD A, (IY+d) for load to and read from memory position IY+d respectively. The 2068 sets IY to the value 23610, an address in the middle of the Systems Variables. Then, with the offset, d, set at 10, the value of address 23620 will be read or rewritten. With a negative number like 245, the address 23600 will be of concern. One can move 127 spaces either side of the address in IX or IY. BUT, a word of caution: IY must be reset to the value of 23610 before comimg back to Basic. Futher care must be taken when using IY and then calling ROM routines as these routines and the subroutines they may call may need IY set back to 23610 to check a system variable. IX, on the other hand, is used by the Bank Switching, Function dispatcher and the floating point calculator routines. It, however, is usually reset before use.

In the code, remember that all IY and IX instructions must be preceded by the Prefix FD or DD. This is followed by the instruction number which is ALWAYS followed by the displacement (if dealing with memory locations) in the 3rd position and any

other codes following that. Also note that you can use double prefixes like FD,CB and FD,ED.

We can also load a double register with a number as in LD BC, nn. In the code, NN must follow the instruction number in LSB/MSB format. Do not confuse the above instruction with LD BC, (nn) as this instruction loads C with the contents of address nn and loads B with the contents of address nn+1. LD (nn), BC puts C in address nn and B in address nn+1.

If you look at the double register instructions allowed, you will note that there is no LD BC, HL or any other load one double register with another. You have to do it one at a time.

CODE FOR LOAD of a SINGLE REGISTER

with	A	B	C	D	E	H	L	(HL)	N	(IY*) (IX*)	nn	(BC)	(DE)
LD A	127	120	121	122	123	124	125	126	62n	126d	55nn	10	26
B	71	64	65	66	67	68	69	70	6n	70d			
C	79	72	73	74	75	76	77	78	14n	78d			
D	87	80	81	82	83	84	85	86	22n	86d			
E	95	88	89	90	91	92	93	94	30n	94d			
H	103	96	94	98	99	100	101	102	38n	102d			
L	111	104	105	106	107	108	109	110	46n	110d			
(HL)	119	112	113	114	115	116	117	---	54n				
(IX*)/ (IY*)	119d	112d	113d	114d	115d	116d	117d		54dn				
nn	50nn												
(BC)	2												
(DE)	18												

*prefix all IX with 221
all IY with 253
d = displacement is 3rd byte
n and nn = reminder next byte(s) must be number(s).

CODE FOR LOAD of DOUBLE REGISTERS

with	nn	(nn)	BC	DE	HL,IX*,IY*	SP
LD (nn)	--	--	237 67,nn	237,83,nn	34,nn or 237,115,nn	
LD BC	1nn	237,75,nn	--	--	237,99,nn	
DE	17nn	237,91,nn	--	--	--	
IY*,HL,IX*	33nn	42nn or 237,106,nn	--	--	--	
SP	49nn	237,123,nn	--	--	249	

SPECIAL LOAD INSTRUCTIONS

LD A, I	237,87	LD I, A	237,71
LD A, R	237,95	LD R, A	237,79

Kindly note above that some double register instructions can be coded two different ways--with or without using a prefix. The STACK POINTER is considered a double register. The novice assembly code writer will not use the special load instructions as they deal with changing the interrupt and refresh registers...a topic much too complicated to be discussed here. We only list these here for completeness. LD A, I and LD A, R are also the only two LD instructions that change any flags.

BLOCK MOVE INSTRUCTIONS

LDI 237,160 LDD 237,168 LDIR 237,176 LDDR 237,184

These four instructions move one or more consecutive bytes from one memory location sequence to another. We could write the following program to transfer a string of consecutive bytes from one spot to another.

```
      LD DE, destination address
      LD HL, source address
      LD BC, number of bytes to transfer
again LD A, (HL)
      LD (DE), A
      INC DE
      INC HL
      DEC BC
      LD A, B
      OR C
      JRNZ, again
```

OR, we could use the instruction LDIR to take the place of the whole "again" loop. LDIR (LOAD, INCrement and REPEAT) has to be set up in the DE, HL and BC registers as noted above and does all the instructions in the "again" loop WITHOUT the use of the A register.

LDI is the same as LDIR but only moves one byte. Thus it can do without the BC counter. It has limited usefulness.

The above routine works with transfers of bytes that don't overlap addresses. When we just want to move things up a few spaces to make room, we have to start at the back of the string of bytes and move forward with a DEC of HL and DE so that we don't erase what we havn't yet moved. LDDR (LOAD, DECrement and REPEAT) is the same as LDIR in the setup of DE, HL and BC but uses DEC HL and DEC DE instead of the INCrements. Of course, LDD is the one shot equivalent of LDI.

JUMPS, JUMP RELATIVES, CALLS and RETURNS

JR (Jump Relative) and JP (Jump) are the assembly equivalents of the Basic GOTO statement. They can be asolute (without conditions) or conditional (based on the condition of a flag in the F register. Thus, conditional jumps are the equivalent of IF-THEN GOTO. We thus read, JP Z, nn as: If Zero (flag set) then Jump to address "nn". Again, JP instructions have to be followed by 2 address bytes in LSB/MSB format.

There are 3 absolute jumps that use an address contained in a register to jump to:

JP (HL) JP (IX) JP (IY)

Don't be confused with these statements. The jump is to the address of HL, NOT the address contained by the memory address HL is pointing to.

JR's are limited to a jump that can be written in one byte, i.e., +/-128, and is relative to where we are now--an offset or displacement if you please. It uses 2's complimented numbers, i.e., 128-255 are negative or backward jumps as used in loops or Next statements, 0-127 are forward jumps.

Students have a great deal of difficulty calculating these displacement values and even though we did it in the last chapter we are going to give you another example. A typical counting loop (sometimes used to do nothing else but waste time like waiting for that slow human to get the finger off the key) consists of:

```
1,y,x           LD BC, value ( x = high byte, y = low byte)
11      Again   DEC BC
120             LD A, B
177             OR C
32,____         JR NZ, Again.
(0),(1)
```

We have to fill in the missing value to get us back to "Again". As your CPU reads the full instruction JR NZ, dis, the Program Counter has advanced to the first address of the next instruction whatever that may be. Instead of the code of the instruction which would normally be written there, I have written a (0) to indicate where a displacement of "0" would get us. Similarily, the position of the next byte is written as a (1) to show where a 1 would get us.

Well, if those bytes are 0 and 1, then the displacement byte has the value of 255 which is -1. The "32" has the position of 254 (-2), the 177 the position of 253, etc. All we have to do is keep counting backwards until we get to that 11 at 251 and write that number in the blank. For forward, we have to remember the (0) position by starting to count with it rather than a "1". Or as they say "THROUGH ZERO".

DJNZ is a very special JR instruction which reads: DEC B and Jump Relative if NOT ZERO. It's always register B that gets DECremented and it's always IF NOT ZERO. It's never the double register BC.

Because it is easier to detect zero (the zero flag goes up) all loops in machine code COUNT DOWN rather than up.

CALL and RETurn are the assembly equivalents of GOSUB and RETURN respectively. All CALLs have to have an "nn" type address, there is no CALL (HL) or any other double register address. Like JP

and JR, CALL and RET can be conditional which then makes them equivalent of IF-THEN GOSUB or IF-THEN RETURN. There are always two considerations to be made when using a CALL:

1. Do we have to save any values to continue the rqutine we are in when we get back from the subroutine? We have to save these values unless we know beyond any shadow of a doubt that the subroutine and the other subroutines it might call don't use the register that holds the value we must save.

2. Do we need any values we have already put on the stack in the subroutine?

The first consideration is quite obvious. However the ROM routines sometimes get so convoluted and involved that the begining student may have some difficulty following them through to all their different ramifications.

The second is not quite so obvious. The reason for the second is that when we do a CALL the first thing the CPU does is PUSH the present address of the next statement onto the machine stack to use as a RETurn address, thus effectively burying any values you may need. This PUSHed value must be retained at all cost or when the computer reads that RETurn statement it's going to POP off the next values from the stack and go to that address--and get hopelessly lost.

This brings up another consideration. Before you tell the CPU to RETurn from a subroutine you had better have POPed off all values you PUSHed when doing the routine or the CPU also will use the numbers of the forgotten PUSH as an address with the same disastrous results. ALWAYS MAKE SURE THE PUSHES EQUAL THE POPS.

CODE for JUMPS, CALLS and RETurns

	ABS	Z	NZ	C	NC	M	P	PO	PE
JP	195nn	202nn	194nn	218nn	210nn	250nn	242nn	226nn	234nn
JR	24d	40d	32d	56d	48d				
DJNZ	16d								
CALL	205nn	204nn	196nn	220nn	212nn	252nn	244nn	228nn	234nn
RET	201	200	192	216	208	248	240	224	232

JP(HL) 233 JP(IX) 221,233 JP(IY) 253,233

Note that you can't use the sign or parity flags for conditional jumps.

CONVERTING SPECTRUM PROGRAMS TO THE 2068

Now that we know what the GOTO and GOSUB statements are, we can

relocate a program to a different area. Or, if we have a "Spectrum ROM to 2068 ROM" conversion table we can convert Spectrum programs to the 2068. These tables have been published. They are also contained in expanded form in the "2068 ROM DISASSEMBLY MANUSCRIPT" already discussed.

You really have no need for a Spectrum emulator or Spectrum ROM anymore. Who, in their right mind, would want to convert from a computer that has sound, joysticks, bank switching and 4 screen mode capabilities to one without these? Just because the program you want isn't written for the 2068 but the Spectrum doesn't mean you convert your computer to a limited Spectrum. Instead you convert the Spectrum to the 2068. You already know enough code to do it.

In just a Spectrum to 2068 conversion, we don't have to worry about the Basic part of the program except for USR CALLs to the ROM. These and the machine code have to be changed. We can't read code so a disassembler like "HOT Z", which does assembly and disassembly, can help a lot. It's best to get a hard copy so you can really look at it rather than change it from the screen. What we are looking for are all calls to ROM (below 16384). The two bytes in back of these calls are the address in the Spectrum Rom and must be changed to the same spot in the 2068 ROM. Check the JP's as well to make sure they don't jump to ROM and then check for JP (HL), JP (IX) and JP (IY) for the same type of jump to the ROM. Generally JP's are not to ROM. That's it. Save your new code to a new tape or disk before running as you might just crash the first time through because you missed something or got it wrong. After it runs successfully where it is you may consider changing its location.

MOVING CODE TO A DIFFERENT LOCATION

FIRST we have to disassemble the machine code back to mnemonics. We can use the same copy we made above if we can still read it. Anyway we need a hard copy of the mnemonics. In cases like this a disassembler that does a printout saves a lot of time.

SECOND, decide where you want to move it to and calculate your offset, that is, how much your addresses are going to change. If you aren't cramped for space, it's easier if this number is divisible by 256 which means that you just have to change the high byte of an address by a certain amount.

THIRD, mark your printout of the disassembly for all LD R, (address), and LD (address), R statements (R is a single or double register) that are pointing to addresses within the code program. You will be happy to find out that the Spectrum and the 2068 use exactly the same addresses for the screen and the system variables except the 2068 has a longer table with extra values added to the end.

FOURTH, check only the JP, JP(HL), JP(IX) and JP(IY) and CALL statements for internal jumps and calls and change these with the offset calculated above. You can ignore calls to ROM if it already is a 2068 program. Otherwise, they must be changed as discussed above.

Changing JP(HL)/(IY)/(IX) is a bit of a problem as you have to go back up through the program and see how they calculate the value in those registers. Somewhere they do an offset add or offset load and the change is made there.

FIFTH, don't forget to change your RAND USR statement in Basic.

SIXTH, all POKEs to and PEEKs from the code areas must be changed in the Basic portion of the program to correspond to the new addresses.

SEVENTH, write yourself an LDIR program as given on page 104 to move your code to the new address. Make sure you place your move code in a spot where it won't get overwritten by your new code.

SAVING REGISTERS--EX, EXX, PUSH AND POP, DI and EI.

Seven registers sometimes are not enough so we have to store a value somewhere while we ae using a register for something else. This is especially true with the A register, but sometimes addresses held in register pairs must be saved as well. Where you save the values depends upon how soon you will need it back and how often the value will be needed.

First let's discuss A. Since this register is necessary for all math and logic one wants to get values in and out fast. The easiest is just LD the value to another unused register such as B, C, D, E, H, or L if one is available and not going to be used.

If all the registers are in use or will be used, we can do EX AF, AF' and save our value in A' (sorry but F is also saved in F' whether we like it or not). Doing another EX AF, AF' gets our value back to A and also saves in AF' another value of A.

The other 3 register pairs have to be saved as one and cannot be saved separately. EXX (217) exchanges HL with HL', DE with DE', and BC with BC'. That's H with H', L with L' etc. We can't exchange BC, DE or HL with their primes by itself.

Note Well: since HL is the only double register pair we can add to or sub- tract from, there is one more exchange, EX HL, DE which just swaps the present pairs of values--HL to DE, DE to HL. There is no EX BC, HL or EX BC, DE.

Other EX mnemonics are EX (SP), HL and of course the extended IY

and IX registers. The use of EX (SP), HL exchanges HL with the last two bytes pushed on the stack--a rapid way to change pointers or counters. PUSH BC; EX (SP), HL; POP BC is a fast way to exchange HL and BC without aid of a 5th register.

A word of caution about using prime registers. Don't have values stored there and then Call a floating point routine as they will be gone. I learned code on the Z81 (T/S1000) machine where the prime registers are used to refresh the screen so use of the primes generally created a crash. The 2068 is quite a bit more tolerant but don't say I didn't warn you.

One way out of this dilemma is to prevent maskable interrupts by using DI (Disable interrupts) 243. The use of this instruction does two things while in effect. It prevents reading the keyboard and it doesn't allow for updating of the screen. You can change the whole Display File but it won't appear on the screen until you again Enable Interrupts with EI (251). You never, never, never want to come out of machine code without being sure you have the interrupts enabled. The result is not a crash, but its equivalent--the keyboard is dead and so is the entry of any more commands--you might as well pull the plug.

Rather than go on at this point with listing more ways of storing values of registers let's stop for a minute and look at an interesting use of the EX AF, AF' instruction--a visual use.

This time around I'm going to give you the assembly mnemonics and let you code the program and enter it starting at 65000.

```
        DI
        LD DE, 22528 (first attr)
        LD BC, 767   counter
        LD A, (DE)   read first attr
        EX AF, AF'   save attr
Loop    INC DE       move to next position
        LD A, (DE)   read present attr
        EX AF, AF'   save present attr/get last attr
        LD (DE), A   put in file
        DEC BC       counter update
        LD A, B      test counter for zero
        OR C
        JR NZ, Loop
        EX AF, AF'   get last attr
        LD (attr 1), A  put in posn 1
        EI
        RET
```

The Basic that goes along with this program is:

```
 5 BORDER 5: GOSUB 100: CLS
10 FOR X = 22528 to 23298
15 LET Y = (INT (RND*158)): REM Random INK, PAPER,
BRIGHT and a few flash--but not all.
```

```
 20 POKE X, Y
 25 NEXT X
 30 PAUSE 0: REM Stop for a look at your random paper
 screen--notice that we have put attrbutes all the way
 down into the 2 bottom lines. Hit a key to continue.
 35 FOR X = 1 TO 768: REM can be what you want, but
 768 is once around.
 40 RANDOMize USR 65000
 45 PAUSE 30: REM a delay or it goes too fast
 50 NEXT X
 55 STOP
100 FOR X = 65000 TO 65023
105 READ Y
110 POKE X, Y
115 NEXT X
120 RETURN
125 DATA 243,17,0,88,1,255,2,26,8,19,26,8,18,11,120,
    177,32,247,8,50,0,88,251,201
```

Did you get 24 bytes of code and end at address 65023? Do your numbers agree with the numbers in the DATA line of the Basic program? Did you get your JR NZ displacement right? What number did you use for ATTR 1? If you got it all right, good for you. You know how to assemble code.

Did you run the program? A nice display and all done in attributes only. Did you notice that as soon as the program had to print the program complete message the bottom 2 lines reverted to border color.

Notes on the assembly program: The A register is very busy. Read all the notes written for the various lines of assembly. Note how the AF' register when loaded back contains the value we want to "print" for the next attribute.

The big question is why did we do the loop only 767 times instead of 768? The reason is that we don't handle the first attribute first but only get its value to put it into attribute 2. When the value of the last attribute becomes available we use that to finish the screen by putting it in the Attr 1 space.

What the program actually does is scrolls the attributes one position throughout the entire table. We use the RANDOMIZE USER 65000 inside a Basic loop together with a pause to slow things down. The loop as written will scroll one byte through all 768 positions and ends up with what we started with.

GOING FURTHER

What we have now is the basic core of our program. We can now easily build other things into it. Let's start by adding PAUSE to the assembly code. PAUSE is equivalent to a wait loop as we have already discussed in Chapter 5 page 82. We make the com-

puter waste time by doing nothing but count down to zero. Typical is:

```
      LD BC, 20000
Wait  DEC BC          6
      LD A, B         4
      OR C            4
      JR NZ, Wait     12
```

The higher the value in BC the longer the wait. Where do we put this loop? How about right after EI? Just before the return. Now we can take PAUSE out of our Basic program. Playing around with different values in BC will give you an indication of just how fast the 2068 can count.

How about also doing the FOR-NEXT loop that surrounds the RANDOMIZE USR statement? How do we do it? We need another counter. Let's use BC again. This presents a minor problem. We have to save the value of BC while we are using the other. Put the following 2 lines at the start of the program:

```
      LD BC, 768  loop counter
Times PUSH BC     Save counter
```

We insert the following just before the RETurn statement:

```
POP BC      Get loop counter
DEC BC      Update counter
LD A, B     Test for zero
OR C
JR NZ, Times
```

Now we can also take the loop from around the RANDOMIZE USR line and just leave that. We leave it up to the student to code in these extra changes and add them to the correct positions in the DATA line. Make sure you extend the FOR-NEXT loop in the Loader routine to READ and POKE the extra bytes. Notice that we now have used BC as counters in 3 different situations.

Want to go further? Just to check that it's the attributes, put some printing on the screen and then call the scroll attributes routine. The letters stay in position although the use of flash for some of the attributes causes them to change colors. Delete all printing and run the program again. Now, try a screen copy to the printer. Suprised? Nothing to print.

Okay, I told you the program would be visual. Don't you think it rates at least a "jump off a high cliff with a beautiful soar and maybe a spiral or two down to the bottom"? If you don't know what I'm talking about, read page 1.

TIMING

We forgot to explain the numbers after our timing loop. They are

the number of clock cycles, at 3,528,000/sec that it takes to execute the instruction. They are also called T states. Mostec has worked out exactly how long the computer takes to execute any instruction. Because of the way the CPU works this is alway a full number of cycles. To determine exactly how long a wait loop takes, we add up all the T states for each instruction. These are listed in Appendix A. Notice that the JR instruction has two times given. One is for when it has to make a jump (always the longer) and the other is when it ignores the instruction--it still takes time to read it, even if it doesn't act on it. For the wait loop above this comes to a total of 26 T states each time through the loop. For our loop, that's an elapsed time of 26/3,528,000 seconds. Or to put it a different way, it will loop through the loop 135,692.3 times a second. With a value of 20,000 in BC, this loop only waits for 0.147 seconds. Since PAUSE 1 is 0.0166 seconds, our loop is equivalent to PAUSE 8.8 (not counting setup time for the computer to look up and interpret PAUSE).

MORE WAYS TO SAVE REGISTERS

We got a bit ahead of our story in that last program but a very safe way to save a register pair is to PUSH it on to the machine stack. We can't push a single register, it alway must be a pair. As mentioned earlier we have a problem of logistics to consider using this method. The first arises with multiple pushes to the stack. We bury our value as they are POPed off in reverse order --what was PUSHed last is POPed first. If we PUSH BC and then PUSH DE and now want to POP BC we must first POP DE. If we use POP BC without first using POP DE we get the value that was in DE into BC. The computer doesn't keep track of which pair was pushed or poped when--that is up to the programer.

We discussed the machine stack back in Chapter 2 when we were talking about where things were stored. If you recall, the stack builds from the top down. Thus when a value is PUSHed to the stack, it's always 2 bytes long. The stack pointer which keeps track of where the end of the stack should be, is always automatically decremented twice. When something is POPed, the values aren't really erased but only the pointer is incremented twice. Therefore, doing 2 DEC SP is equivalent to a rePUSH of a value once on the stack back on the stack in the same position. Similarily, doing a double INC SP POPs a value to nowhere.

The second problem is with CALLs. The CPU pushes the RETurn address onto the stack, then jumps to the routine. When it gets the RETurn instruction, it POPs the next values off the stack and returns to that address. You had better have POPed everything you PUSHed since you did the CALL. The return address also gets in the way of using other saved value from a previous routine as well.

Numbers used in numerous routines and that may get buried on the machine stack are best stored in a memory location--similar to

what the computer does with its System Variables. Many programs start with a series of addresses used for this purposes. One of the problems of a disassembler is that it never knows when it is disassembling instructions and when it's disassembling data. The result is that sometimes you get some really weird code that doesn't make sense. Most disassemblers have a routine that let's you read data directly as ASCII symbols. That doesn't help much when it comes to a series of numbers. It is up to you to decide when code doesn't make sense if it's ASCII or numeric data. Numeric data can be 1 to 5 bytes long (floating point numbers are 5 byte). Programs that use this method of storing numeric data generally are full of LD (address), register and LD register, (address) mnemonics.

CODE for PUSH, POP, EXCHANCE and DI/EI

	PUSH	POP					
AF	245	241	EX AF, AF'	8	EX (SP), IX	221,227	DI 243
BC	197	193	EXX	217	EX (SP), IY	253,227	EI 251
DE	213	209	EX DE, HL	235	EX DE, IX	221,235	
HL	229	225	EX (SP), HL	227	EX DE, IX	253,235	

SIMPLE ARITHMETIC AND LOGIC

INC and DEC

Only these two instructions can be used on all single registers and register pairs. We have already met them--good old INC and DEC (Add 1 to a value or subtract one from a value). Double registers work by only INC and DEC the low byte using the high byte to store overflow or borrow from. Decrementing a single register that is at zero puts it at 255 with the carry flag set. Incrementing 255 puts it to zero with the zero flag set. All flags are affected by INC and DEC, however, DEC and INC of double registers DO NOT automatically set the zero flag or the carry flag. This is why we can do a JR immediately after a DEC of a single register but must test a double register for zero with LD A, B and OR C. If A is zero after these two operations, the zero flag is set. Of course similar instructions can be used to check zero of other register pairs. The Parity flag goes EVEN when a double register is at zero. Unfortunately this flag can't be used in conditional Jump Relatives.

ADD, ADC, SUB and SBC

All other math and logic operations can only be done with the A, HL, IX and IY registers. The double registers are limited. Your User's Manual uses two varitions of mnemonics:

ADD A, C and SBC A, C represent the first kind.
SUB B and OR B represent the second kind.

The first set we have no trouble with, ADD to A the value of C is quite explicit. We know what is happening and we know that the answer ends up in A. With SUB B we know we should be subtracting B, or is it subtract something from B? Well, it's always happening to register A. So it's SUB A, B., OR to A the value in B. The answer is always in A.

ADD means add directly and don't worry about the carry flag. ADC means add and if the carry flag is set add that too. Similarily, with SUB and SBC except that SBC subtracts the value of the carry flag. There is no SUB instruction with double registers, only SBC.

The reason for ADC and SBC is that zero is counted as a number as the register rolls over zero (like the milage counter on your car rolling through 100,000). For example, 255 + 1000 should give us 1255. 1000 is 3 in the high byte and 232 in the low byte. Adding 255 and 232 gives us 231 with an overflow. The carry flag is on at this point. Now, if we add the high bytes using ADD we get 0 + 3 = 3 in the high byte which is wrong--it should be 4. But by doing an ADC instead we get 3 + 0 + carry for a 4. 1024 + 231 is 1255. We do have to take the precaution of setting carry to zero before using the ADC or SBC instructions or we will get a wrong answer...that means everytime we add or subtract double registers. How do we reset the carry flag? By doing AND A. CCF is complement, not clear, the carry flag.

The same thing applies to double registers with the answers always ending up in HL, IX or IY. We do not have SUB just SBC with double registers.

We will save all the instructions used for multiply and divide for the next section. A simple multiply by 2 can be achieved with ADD A, A. Doing it twice is equivalent to multiplying by 4. Three times is times 8, etc. Similarly ADD HL, HL can be used for the HL double register multiplications of powers of 2.

LOGIC

CP r and CP n. CP (Compare), compares the contents of the register (r) or the number (n) to the contents of the A register by doing a MENTAL subtraction. Neither A nor the register change. The answer is stored nowhere. Only the flags are set or reset depending upon what the results would have been. Thus if A = R, the zero flag would be set. If A < R the carry flag would have been set and if A > R the carry flag would be reset. By testing the flags we can tell if A was greater than, equal to, or less than the r (or n). Here is where all the conditional jumps, jump relatives, calls and returns can really come into play. Doing a compare and then a conditional branch we have just executed the equivalent of a Basic IF-THEN statement.

CPI 237,167 CPIR 237,177. CPD 237,169 CPDR 237,185

These look similar to LDI, LDIR, LDD and LDDR and they are. In reality it's CP (HL) followed by INC (if I) or DEC HL (if D), DEC BC, and Repeat (if R). However, 2 conditions STOP the repeat: when A = (HL) or when BC reaches 0. One can look through a string of data, or the variable table looking for a matching byte to A. Obviously A must be set up with the value we are looking for. HL must be set at the starting address (in the case of INC, at the start, in the case of DEC at the end), and BC must have the length of the list. If a match is found, HL will contain the address of the matching byte. If no match is found, BC = 0.

AND, OR and XOR

The first two of these ARE NOT THE SAME AS their Basic equivalents. These are the Boolean operations. They do a bit by bit comparison of the binary number in the A register with the binary number in a register or a number and leave the answer in A. It is best to write out the binary number in A with the other number (or the register number) below it when working these out.

AND. If the number has this bit set, don't change the bit in A, else reset it to zero. Or to put it another way, save only those bits of A whose bits I have "on" in the number--a MASK if you please. AND 255 thus would not change A. AND 0 sets A to zero as it saves nothing. AND A does nothing but turn on the zero flag. A few examples:

```
A = 159    10011111    A = 255    11111111    A = 231    11100111
AND 14     00001110    AND 24     00011000    AND 24     00011000
results    00001110               00011000               00000000
```

OR. Try to add the number bit to A but don't do a carry, else leave it alone. Thus if a bit in A is already on, it stays on and nothing happens. If the bit in A is off, it is turned on only if the number has it on. Thus OR 255 sets A to 255. OR 0 doesn't change A.

```
A = 159    10011111    A = 1    00000001    A = 200    11001000
OR 13      00001101    OR 31    00011111    OR 244     11110100
results    10011111             00011111               11111100
```

XOR Exclusive OR). Flip those bits of A I tell you to, else leave them alone. XOR A makes A = 0 and sets the zero flag. XOR 255 flips every bit in A and proves to be very useful for flipping all the bits in a pixel byte when doing inverse printing to the screen...every bit that was off is on and every bit that was on is off.

```
A = 159    10011111    A = 0    00000000    A = 255    11111111
XOR 200    11001000    XOR 255  11111111    XOR 84     01010101
results    01010111             11111111               10101010
```

To repeat, ALL LOGIC AFFECT FLAGS if necessaary. The answer is

always in A.

OTHER SIMPLE MATH OPERATIONS

The discussion of these had to be delayed until the logic operators were discussed.

CPL (Complement). First, don't confuse CPL (47) with CP L (compare L) (189). CPL is actually XOR 255, flipping all the bits in the A register. If it is followed by INC A, we have done what is called a 2's complement of a number, which in layman's language means change the sign of the number. For example, we know that -1 is 255. Thus by doing an XOR 255 (or a CPL) we get 0. Then INC A gives us 1. We thus have converted -1 to +1. One can only complement to the A register.

NEG (negate). Means change the sign of a number, It is equivalent to XOR 255 and INC A all wrapped up in one instruction. It is also equivalent to a 2's complement. As in Algebra where minus a minus number is a positive number, so it is with the computer.

One last comment. A single byte number and its complement add up to 255. A single byte number and its 2's complement add up to 256. CPL only work on A, not HL. If you need to complement a double byte number you have to move it into A a byte at a time, do a CPL, and move it back. Then only INC the low byte. Double byte numbers and their complements always add up to 65535. A double byte number and its 2's complement add up to 65536.

For the student. What is the 2's complement of zero? Well, XOR 255 makes it 255 and INC A brings it back to "0". Okay, what is the 2's complement of 128? XOR 255 changes it to 127 and INC brings it right back to 128. Since 128 has bit 7 set it should be a negative number. But since its 2's complement is also 128, we have to say that it is undefined.

CHECKING BITS. BIT, SET and RESET. (All these instructions require a 203 (CB) prefix.)

Since we are playing around with the bits inside a byte, let's discuss the BIT checking operations of the Z80. We can ask the computer to give us the status of any bit we want to, either in a register or in the memory address pointed to by the HL register pair as in BIT n, (HL). n designates the Bit number 0 to 7. If the bit is 1 the zero flag is turned off, if "0" the zero flag is on. If we preface with the IX or IY preface we can also use the addresses in these registers with a displacement to ask about a memory bit pointed to by (IX+d) and (IY+d).

We can also change a BIT anywhere. SET n, r or SET n, (HL) will turn on that particular bit (n) whereever it is. RESET, working exactly the same way, of course, turns off that Bit.

CODES for MATH, LoGIC and BIT OPERATIONS

	A	B	C	D	E	H	L	(HL)	n	(IX*)/(IY*)
ADD	135	128	129	130	131	132	133	134	198n	134d
ADC	143	136	137	138	139	140	141	142	206n	142d
SUB	151	144	145	146	147	148	149	150	214n	150d
SBC	159	152	153	154	155	156	157	158	222n	158d
AND	167	160	161	162	163	164	165	166	230n	166d
XOR	175	168	169	170	171	172	173	174	238n	174d
OR	183	176	177	178	179	180	181	182	246n	182d
CP	191	184	185	186	187	188	189	190	254n	190d
INC	60	4	12	20	28	36	44	52		52d
DEC	61	5	13	21	29	37	45	53		53d
CPL	47									
NEG	237,68									

	BC	DE	HL	SP	IX*/IY*
ADD HL	9	25	41	57	
ADC HL	237,74	237,90	237,106	237,122	
SBC HL	237,66	237,82	237,98	237,114	
ADD IY*	9	25	--	57	
ADD IX*	9	25	--	57	
INC	3	19	35	51	35
DEC	11	27	43	59	43

CPD	237,169
CPDR	237,185
CPI	237,161
CPIR	237,177

PREFACE WITH 203 (CB)

		A	B	C	D	E	H	L	(HL)	(IX+d)*/(IY+d)*
BIT	0	70	64	65	66	67	68	69	70	d70
	1	79	72	73	74	75	76	77	78	d78
	2	87	80	81	82	83	84	85	86	d86
	3	95	88	89	90	91	92	93	94	d94
	4	103	96	97	98	99	100	101	102	d102
	5	111	104	105	106	107	108	109	110	d110
	6	119	112	113	114	115	116	117	118	d118
	7	127	120	121	122	123	124	125	126	d126

		A	B	C	D	E	H	L	(HL)	(IX+d)*/(IY+d)*
RES	0	135	128	129	130	131	132	133	134	d134
	1	143	136	137	138	139	140	141	142	d142
	2	151	144	145	146	147	148	149	150	d150
	3	159	152	153	154	155	156	157	158	d158
	4	167	160	161	162	163	164	165	166	d166
	5	175	168	169	170	171	172	173	174	d174
	6	183	176	177	178	179	180	181	182	d182
	7	191	184	185	186	187	188	189	190	d190

		A	B	C	D	E	H	L	(HL)	(IX+d)*/(IY+d)*
SET	0	199	192	193	194	195	196	197	198	d198
	1	207	200	201	202	203	204	205	206	d206
	2	215	208	209	210	211	212	213	214	d214
	3	223	216	217	218	219	220	221	222	d222
	4	231	224	225	226	227	228	229	230	d230
	5	239	232	233	234	235	236	237	238	d238
	6	247	240	241	242	243	244	245	246	d246
	7	255	248	249	250	251	252	253	254	d254

MULTIPLY AND DIVIDE--ROTATE AND SHIFT

Suppose we have the number 32 (only Bit 5 on) and want to multiply it by 2. The obvious answer 64 requires that Bit 6 be turned on and bit 5 turned off. Similarily, if we wanted to divide 32 by 2, the answer 16 requires that Bit 5 again be turned off and Bit 4 be turned on. In both cases, an operation that shifted or rotated everything one bit to the left or one bit to the right would be nice. That is exactly what the rotate and shift instructions do. However, we have one further consideration. What do we do with the bit that is pushed off the byte either right or left? This will depend upon what we want to do with that bit. Hence several different rotates and shifts.

The easiest way to describe what is happening is to draw you a picture of the various commmands.

RLC (rotate left circular)

RRC (rotate right circular)

RL (rotate left)

RR (rotate right)

SLA (shift left arithmetic)

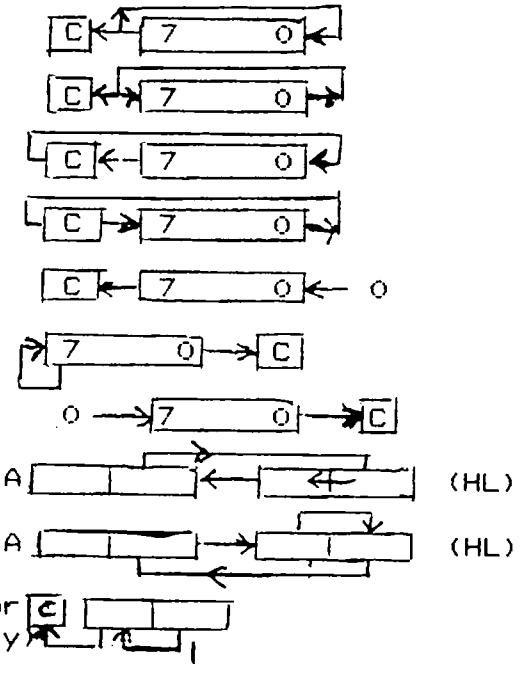
SRA (shift right arithmetic)

SRL (shift left logic)

RLD (rotate left digit) (by nybble)

RRD (rotate right digit) (by nybble)

DAA (decimal adjust accumulator (shift out 10 if necessary)



The last 3 instructions are used for handling Binary Coded Decimal packed number. That is, each decimal number occupies a 4 bit nybble. The DAA instruction works something like this:

```
          26      0010  0110
        + 35      0011  0101
Binary add        0101  1011  -- This nybble is 11
    DAA shift 10  ____10____
corrected answer  0110  0001   Correct answer 61
```

MULTIPLYING

By a power of 2. For a single byte number is quite easy.

```
LD L, number
LD H, 0
```

```
        AND A    clear carry
        RL L  X2
        RL H  overflow to H
        RL L  X4
        RL H  overflow to H + X2 of H
```

Notice how if the L register overflows the bit goes to the carry which then gets loaded into the H register on the RL H instruction. The next time around anything already in H automatically gets rotated Left effectively multiplying it by 2 as Bit 0 again gets any input from the carry flag. In these instances, the carry flag is acting as the 9th bit of a register.

Division by powers of 2 of course would start with the number in H and use alternating RR H and RR L's. Any number in L is of course a fraction.

Multiplying and dividing double byte numbers of 2 can be handled in the same manner as long as we don't get any overflow of the registers either way, or, in the case of division, we are not interested in the fraction. If overflow should occur we have to make sure the carry flag is reset before continuing. If we must save the overflow, we have to use a third register to handle it.

RR and RL ruin the number in the register. If we use a counter and do things exactly 8 times, as will become necessary when we do not necessarily want to multiply by a power of 2, we can use RLC and RRC and rotate the number back into the register for use in the next operation. This would require our answer to be kept in other registers.

MULTIPLICATION AND DIVISION BY ANY NUMBER

To multiply or divide by any number we have to do things differently. For example 47 x 7 = 329. Our answer is going to require two registers. Let's use HL for that.

```
        47 is 00101111 binary
         7 is 00000111 binary
```

B will be our counter, set to 8. DE will hold our 47. A is our multiplier of 7.

```
        LD B, 8   counter
        LD HL, 0  clear our answer
        LD DE, 47 load our multiplicand
Next dig. AND A     clear carry flag
        RRC A     next digit of multiplier to carry
        JR NC, update
        ADD HL, DE
update  AND A
        RL E
        RL D
        DJNZ, next digit (binary)
```

With our example, the first time we are going to add DE to HL so:

```
              HL =    00101111
the 2nd time add      01011110
so HL becomes         10001101
the 3rd time add    + 10111100
so HL becomes       1 01001001   this is 329.
```

The 4th, 5th, 6th and 7th times through we are rotating zeros into the carry so nothing more is being added to HL.

DIVISION--SUCCESSIVE SUBTRACTIONS GIVING INTEGER VALUE ONLY

This method of division is not very efficient but is simple and easy to understand. It stops when it has computed the full number leaving HL with the remainder. It has to go through the loop as many times as the integer value of the answer:

```
        LD BC, 0 clear answer
        LD HL, dividend (number to be divided)
        LD DE, divisor
Loop    AND A  clear carry flag
        SBC HL, DE  try to subtract
        JR C, rem
        INC BC
        JR loop
rem     ADD HL, DE
        RET
```

A more complex but efficient division.

```
        LD HL, dividend (number to be divided)
        LD D, divisor
        LD IX, 0      clear answer
        LD A, L       shift dividend to A and L
        LD L, H
        LD H, 0       clear remainder register
        LD B, 16      set counter
Loop    ADD HL, HL    shift L until you can subtract
        RL A
        JR NC, Shift
        INC L
Shift   ADD IX, IX
        INC IX        assume can subtract
        OR A, A       clear carry
        SBC HL, DE    do subtract
        JR NC, next   Could subtract so loop
        ADD HL, DE    got negative # so add back and
        DEC IX        adjust answer down
Next    DJNZ, Loo     do 16 times
```

These are relatively short numbers. The routine is quite simple. But this last gives you some idea of what must be done with

longer binary numbers. We leave it up to the student to devise more advanced routines.

FLOATING POINT

The above discussion was done using integer numbers, but the routines can be used for decimal numbers (to an extent). We have assumed that the decimal point was always at the end of a byte, but it doesn't have to be. We can put it anywhere in our binary number as long as we keep track of where it is. This is really the function of the exponent byte of a Floating point number.

Suppose that we put our answer in memory and now are back in Basic. Reading out our 2 byte answer would be done with:

```
PRINT PEEK X +256*PEEK (X+1)
```

If we had deliberately shifted our binary decimal point one place left, i.e., between Bit 0 and Bit 1 of our low byte, our readback statement would change to:

```
PRINT (PEEK X)/2 +128*PEEK (X+1)
```

Everything gets divided by 2. For every position left we move our decimal point it's another power of 2 to divide by. Thus 4 binary positions left is:

```
PRINT (PEEK X)/16 +(256/16)*PEEK (X+1)
```

A whole byte of binary decimal left is:

```
PRINT (PEEK X)/256 + PEEK (X+1)
```

This is a simple way to get a fraction in the our divide routines above.

Well, how about shifts right? Multiply instead of divide. For 1 place right it is:

```
PRINT 2*PEEK X +512*PEEK (X+1)
```

Please note that in all of this we are talking about the placement of the decimal point, the marker between the integer and the fraction, in a binary string of numbers. Review binary numbers as we discussed them in Chapter 1 if you are not entirely clear on this point. Since our computer is going to take these fractions and print them in decimal, one should take with a grain of salt the accuracy of all those decimal positions.

Another word of caution: Don't divide by zero. A routine to trap out a zero divisor must be included in the above routines if you don't know what your divisor is going to be.

CODES FOR SHIFT, ROTATE and BCD ARITHMETIC

All have 203 (CB) prefixes except where noted

	A	B	C	D	E	H	L	(HL)	(IY*)/(IX*)	Alternate method without prefix
RLC	7	0	1	2	3	4	5	6	6d	
RRC	15	8	9	10	11	12	13	14	14d	
RL	23	16	17	18	19	20	21	22	22d	RLC A 7
RR	31	24	25	26	27	28	29	30	30d	RRC A 15
SLA	39	32	33	34	35	36	37	38	38d	RL A 23
SRA	47	40	41	42	43	44	45	46	46d	RR A 31
SRL	63	56	57	58	59	60	61	62	63d	

RLD 237,111
RRD 237,103
DAA 39 (no prefix)

IN/OUT

As we pointed out earlier, every perpheral on the computer including anything we add must be addressed through ports. This includes the keyboard, sound chip, joysticks, 2040 printer, dot matrix printer, modem, disk drives, cassette recorder, screen or monitor, extra memory, cartridge programs, microdrives and whatever else you may wish to add.

The 2068 can use as many as 256 (0 to 255) different ports. Every time we use an IN or an OUT we must specify the port. Only IN A, (n) and OUT (n), A need the port number immediately after the instruction number (direct addressing). All the rest of the IN/ OUt's require that the port be held in register C.

The port number is put on the 8 low address lines, should there be a value in B, this is put on the eight top address lines. Thus the 2068 can really address 65536 different ports. Almost always B is "0" as 256 ports are more than adequate to handle everything we may ever need.

Obviously one has to know what is attached to what port before writing any code for it. In the first part of this book we gave you all the necessary port assignments for the normal equipment you may attach. Attaching 3rd party equipment requires the firmware program written to interface that piece of equipment to the 2068. Disassembling programs of this nature is going to reveal a lot of confusing instructions which seem to have no purpose (to the novice). They do serve a purpose however in that sending things out and getting things in have to be timed and what you are really looking at are pseudo timing loops.

INI	237,162	IND	237,170	INIR	237,178	INDR	237,186
OUTI	237,163	OUTD	237,171	OTIR	237,179	OTDR	237,187

Again we have a set of series transfer of bytes out the same port or in the same port. These are really IN (HL),(C) and OUT (C), (HL) instructions. The setup is as:

```
LD HL, address to be loaded in or out of.
LD C,  port
LD B,  count
```

Because of 1 register count, one INIR will only work for 256 bytes before it has to be repeated.

CODES for IN/OUT COMMANDS

Preface with 237 (ED)

	A	B	C	D	E	H	L	(HL)
IN r, (C)	120	64	72	80	88	96	104	112
OUT (C), r	121	65	73	81	89	97	105	113

IN A, (n) 219,n (no preface)
OUT (n), A 211,n (no preface)

There is an error in your User's Manual:

Change 237,112 to IN (HL), (C)
Change 237,113 to OUT (HL), (C)

RESTARTS (RST)

We have 8 of them which are really CALLs to the addresses 0, 8, 16, 24, 32, 40, 48, and 56. For the 2068, these are special routines which can be very useful.

RST 0 (PLUGIN). This is the first instruction that your computer uses. Hence it is equivalent to restarting the whole computer setup without turning the computer off and then back on. It is equivalent to RANDOMIZE USER 0 which we have already talked about.

RST 8 (Error). Your computer uses this restart to print errors at the bottom of the screen. Using RST 8 (207) must be followed by a DATA byte indicating the desired error. This number is always one less than the error number. Thus 255 will give Error 0 which is OK. A "0" will give error 1, NEXT without FOR, etc. Error A is 10, B is 11, G is 16, H is 17 etc.

RST 16 (Print A Character). Prints the ASCII symbol for the the code carried in register A to the present screen position. Be sure to initialize the screen after doing a CLS with a "PRINT ," before going into code and calling your routine. It can be used to give print commands such as AT (27) followed by line and column numbers, TAB (23) followed by a column number, INK (16) and

PAPER (17), need a color number while FLASH (18), BRIGHT (19), INVERSE (20), and OVER (21) need the usual 1 for on and 0 for off. PRINT comma (6) works like TAB 16 or TAB 0 while ENTER (13) is the equivalent to NEWLINE, the print apostrophe. Don't use the cursor controls as they only operate in LIST mode which is hardly where one would be while running code.

RST 16 can be used in a loop as follows: We use a number like 24 which isn't used for anything, to end the loop so we don't even have to use a counter.

```
      LD DE, Data base
Loop  LD A, (DE)
      CP 24
      RET Z
      RST 16
      INC DE
      JR Loop
```

Data base of course is the address of where you have put what you want to print. This routine will work with all graphics and TOKEN prints as well.

RST 24 (Get Character). These two restarts are not to useful
RST 32 (Next Character). as they work on the line being edited or the line being executed and depend upon having the address in 23645 CHAR ADDR.

RST 40 (Do Floating Point Calculation). This is a topic we will cover in the next chapter. It needs a whole chapter to itself.

RST 48 (Make BC spaces). Is used to insert a new line into a program or insert a variable into the variable table. Both these require everything above them in memory to be moved up to make space. It is also used to insert a character into a line that you are editing (i.e., either inside the line or at the end as you are entering it--in these cases BC = 1.

RST 56 (Maskable interrupt routine). In reality it is update screen and scan keyboard for a new input. It can't be called with DI in effect and will do an EI before returning. If you want to use this to read the keyboard, you will find your answer in LAST K. If you have not used DI, the maskable interrupt will automatically do it for you every 1/60th of a second anyway.

CODES for RESTARTS

RST 0	199	Plug in
RST 8	207	Error
RST 16	215	Print a Character
RST 24	223	Get a character
RST 32	231	Next character
RST 40	239	Do Floating Point calculation

RST 48 247 Make BC spaces
RST 56 255 Maskable interrupt. Update screen, Scan Keyboard.

MISCELLANEOUS INSTRUCTIONS

nop (0) no operation. A nice instruction in case you goof and have to change your program and all of a sudden have too many bytes--just fill in with zeros. Also can be used to pad timing loops to waste more time. Since CLEAR sets all memory locations to zero, memory is actually full of these instructions until something else is POKEd there.

HALT (118) This stops the CPU until it receives an interrupt from a peripheral device. This is used to synchronize the CPU with the peripheral. Don't use it unless you understand interrupts or you have just stopped your computer with no way to start it again--unless you turn it off and back on.

IM0/1/2 Interrupt modes 0, 1 and 2. IM0 is the default mode compatible with 8080 processors. The interrupting device must give the CPU an instruction code during the interrupt acknowledge time.

IM1 causes restart to address 56 (38h). Generally the 2068 is set to this mode.

IM2 causes restart to the address given by the interrupting device (low byte) with the I register giving the high byte.

CODES FOR MISCELLANEOUS INSTRUCTIONS

IM0	237,70	nop 0
IM1	237,86	halt 118
IM2	237,94	

EXTRA INSTRUCTIONS

WARNING: Although these codes exist, they are not checked out by the manufacturer and may cause your particular CPU to lock up and malfunction. They must be checked out before being used.

The missing CB instructions 48-55

These do the following operations SL and INC. The easiest abbreviation is SLL. It is the missing operation in the shift and rotate instructions.

SLL A	203,55
SLL B	203,48
SLL C	203,49
SLL D	203,50

SLL E	203,51
SLL H	203,52
SLL L	203,53
SLL (HL)	203,54

The other extra instructions deal with the IY and IX registers needing DD(221) or FD(253) prefixes to convert to IX and IY respectively instructions normally dealing with H (which converts to the high register of IX of IY) and L (which converts to the low register of IX and IY).

CODES for extra IX and IY Instructions

Prefix 221(DD) for IX, 253(FD) for IY

with	A	B	C	D	E	n	(IY-IY) L(IX-IX)	(IY-IY) H(IX-IX)
LD H(IX/IY)	103	96	97	98	99	38	101	---
LD L(IX/IY)	111	104	105	106	107	46	---	108

	H(IX/IY)	L(IX/IY)
LD A	124	125
LD B	68	69
LD C	76	77
LD D	84	85
LD E	92	93
ADD	132	133
ADC	140	141
SUB	148	149
SBC	156	157
CP	188	189
AND	164	165
OR	180	181
XOR	172	173
INC	36	44
DEC	37	45

Additional NEG	Additional RET N
237,76	237,85
237,84	237,93
237,92	237,101
237,100	237,109
237,108	237,117
237,116	237,125
237,124	

Unused ED instructions are nop.

CHAPTER 8

THE FLOATING POINT CALCULATOR

A full explaination of what actually goes on in the floating point calculator is quite complex. A full description of all the possible modes and how they are implemented and the full workings of each instruction could be a book in itself.

The floating point calculator always uses the SLUG notation for numbers. We talked about slugs in Chapter 1 so a review at this time would be in order.

A NOTE ABOUT PRECISION

The slug is always 5 bytes long. The first byte is always the EXPONENT followed by 4 bytes of the 32 most significant binary bits of the number (padded out with zeros if necessary). The 4 byte binary numbers are called the mantissa, a word mathematics majors will remember from their study of logarithms. In logs, the mantissa was a decimal. The 2068 uses a binary mantissa of 4 bytes that is not a fraction. We can't mix the mantissa with the exponent as in logarithms.

Some books on floating point are going to call the use of a single binary byte numbers single precision, double byte operations double precision, three byte long numbers as triple precision and four byte numbers as quad precision. In this sense of the word "precision", our 2068 already uses quad precision. This type of notation has nothing to do with the number of decimal digits one can get from such numbers.

Decimal precision is the ability to give numbers in decimal notation to so many places accurately. The decimal 0.100000000 is accurate to the 9th decimal. If your computer represents that same decimal as 0.09999999957 it also is accurate to 9 places. IF we force rounding of the fraction beyond the 9th place, we get the same exact number. The 2068 rounds the fraction exceeding its precision to maintain the maximun amount it is capable of. This is the true definition of precision.

Some computers define single precision numbers as so many places. IBM calls its single precision as 6 numbers long, its double precision as 16 numbers long. The 2068 has no double standard as everything is 9 place precision. It takes a little over 3 bytes to signify each unit of 10 in binary. Visualize that 7 is 111 binary--3 bits, with 100 as 01100100--8 bits, and 1000 as 11 11101000--10 bits. 10 bits binary divided by 3 decimal bits = 3.33 bits binary per decimal place. 32 binary bits

would then give us 32/3.33 which is 9 but not 10 bit decimal accuracy. The student will note that this is the area where the computer shifts over and starts printing numbers in scientific notation. With this information one can also calcuclate about how many binary bits are necessary for 16 place accuracy 16x3.33 = 53.28 bits. Rounding up to full bytes (56 bits) requires a 7 byte mantissa.

Many beginning students in assembly language realize this limitation of their computers and immediately want to jump into routines that can do calculations and manipulations on numbers with more accuracy. They think that the process is very simple or that maybe somebody has written a routine that can already do this for them. As far as I know, nobody has done this for either the ZX81/TS1000 or the 2068. The present floating point routines occupy from 12377 to 15496 in ROM as they are. Writing a double precision floating point calculator is really quite simple. All you you have to do is add a few more bytes to the mantissa of the numbers and if you really want to handle big and tiny number use two bytes for the eponent. Then you have to understand all the various functions of the floating point calculator (not only add, subtract, multiply and divide but all trigonometric functions, draw, circle, log, exp, square root etc.) and write a new routine for those. Some of these use something called Chebyshev polynomials to generate the numbers--there are no trig tables or log tables in the ROM. And one more thing, translate your numbers from binary back to decimal when you are done and print them. You still may want to use scientific notation as well so it gets complex.

As I say, "Let's learn to walk before we fly" and just learn how to do some simple routines with the floating point calculator from machine code and save the rewriting for a later time.

THE TECHNIQUE

The f.p. calculator has its own stack, similar to the machine stack we are already familiar with. Only this stack is 5 bytes wide so it can handle full numbers and it builds from the low addresses up. It does not hang like the machine stack. This stack is located at the very top of the Basic program with its bottom starting address held by the system variable STK BOT (23651-23652) and its top by STK END (23653-23654). They are the same address when the stack is empty. There is no pointer to these positions like we had for the machine stack with the STACK POINTER. We have to provide our own and generally use HL.

In addition, the f.p. calculator has its own memory store where it can temporarily store up to 6 numbers. This storage is also in the system variables called MEMORY BOT (23698). Notice that it is 30 bytes long--just long enough to hold 6 five byte long numbers.

Operations of the f.p. calculator consists of loading the stack

with the correct slugs in the right order and then using RST 40 to tell it what to do with the numbers. The f.p. calculator use a notation common to FORTH in that it operates on the top number or the top two numbers on the stack only and moves down as these numbers are replaced by the partial answers. Another name for this type of operation is called REVERSE POLISH--give the computer the two numbers, then tell it what to do with them. Hewlette-Packard had a few hand held calculators that used this type of notation.

Machine code numbers following a RST 40 instruction are NOT interpreted using the normal set of mnemonics but with the set given below. Note that this set is specific for the 2068 and cannot be translated directly back to the TS1000 which has a similar but slightly different list. The use of instruction 56 (38H) END F.P. tells the computer that the calculation is done and go back to regular mnemonics. At this point your answer is on the top of the f.p. calculator stack. The following is the list of f.p. calculator commands. The symbol # means number.

DEC	HEX	OPERATION	DESCRIPTION
00	00	Jump-true	JR if preceding operation true. Dis = 0 byte
01	01	Exchange	Exchange the top two #'s on the stack.
02	02	Delete	Delete top # on stack.
03	03	Subtract	Subtract top # from 2nd. Leave answer in 2nd #. Delete top #.
04	04	Multiply	Multiply 2nd # by top #. Leave answer in 2nd #. Delete top #.
05	05	Divide	Divide top # into 2nd #. Leave answer in 2nd #. Delete top #.
06	06	TO THE	Raise 2nd # to power of top #. Delete top #. Leave answer in 2nd #. Corrupts B & M0-M3.
07	07	OR(x or y)	Leave x if y = 0, else leave a 1.
08	08	AND(x or y)	Leave x if y <> 0, else leave a 1.
09	09	X <= Y	Leave 1 if true, else 0 for false. B=code.
10	0A	X >= Y	Leave 1 if true, else 0 for false. B=code.
11	0B	X <> Y	Leave 1 if true, else 0 for false. B=code.
12	0C	X > Y	Leave 1 if true, else 0 for false. B=code.
13	0D	X < Y	Leave 1 if true, else 0 for false. B=code.
14	0E	X = Y	Leave 1 if true, else 0 for false. B=code.
15	0F	ADD	Add top # to 2nd #. Leave answer in 2nd #. Delete top #.
16	10	X$ AND Y	Gives X$ if Y = 0 else gives "". B=code.
17	11	X$ <= Y	Leave 1 if true, else 0 for false. B=code.
18	12	X$ >= Y	Leave 1 if true, else 0 for false. B=code.
19	13	X$ <> Y	Leave 1 if true, else 0 for false. B=code.
20	14	X$ > Y	Leave 1 if true, else 0 for false. B=code.
21	15	X$ < Y	Leave 1 if true, else 0 for false. B=code.
22	16	X$ = Y$	Leave 1 if true, else 0 for false. B=code.
23	17	X$ + Y$	Concatenate string. Add Y$ to end of X$.
24	18	VAL$	Replace top of stack with VAL$. B=code

DEC	HEX	OPERATION	DESCRIPTION
25	19	USR$	Replace top of stack with USR of string item.
26	1A	READ-IN	Read INKEY$ from channel specified.
27	1B	NEG	Negate top value of stack.
28	1C	CODE	Replace top of stack with CODE of string.
29	1D	VAL(b=code)	Replace top of stack with VAL of string.
30	1E	LEN	Replace top of stack with LEN of string.
31	1F	SIN	Replace top of stack with SINe. B,M0-M2 bad.
32	20	COS	Replace top of stack with COSin. B,M0-M2 bad
33	21	TAN	Replace top of stack with TANgent B,M0-M2 bd
34	22	ASN	Replace top of stack with ASN of value (in radians). B,M0-M2 corrupted.
35	23	ACS	Replace top of stack with ACN of value (in radians). B,M0-M2 corrupted.
36	24	ATN	Replace top of stack with ATN of value (in radians). B,M0-M2 corrupted.
37	25	LN	Replace top of stack with LN. B, M0-M2 bad
38	26	EXP	Replace top of stack with EXP. B,M0-M3 bad
39	27	INT	Replace top of stack with INT. M0 corrupted
40	28	SQR	Replace top of stack with SQR. B,M0-m3 bad
41	29	SGN	Replace top of stack with Sign of value.
42	2A	ABS	Replace top of stack with ABS value.
43	2B	PEEK	Replace top of stack with PEEKed value.
44	2C	IN	Replace top of stack with IN (port) value.
45	2D	USR	Replace top of stack with USR value (INT).
46	2E	STR$	Replace top of stack with STR$. M0-M5 bad
47	2F	CHR$	Replace top of stack with CHR$ of value.
48	30	NOT	Leave 1 (true) if zero, else 0 for false.
49	31	Duplicate	Make duplicate of stack top on stack top.
50	32	X mod Y	Relace 2 top values with INT(X/Y) on top and remainder below. Y is on top to start. M0 bad
51	33	Jump	Unconditional jump relative. Dis=byte 0.
52	34	STK DATA	Stack number which follows.
53	35	DJNZ	as in assembly (B register).
54	36	X < 0	Leave 1 if true, else 0 for false.
55	37	X > 0	Leave 1 if true, else 0 for false.
56	38	END f.p.	RETURN to normal machine code.
57	39	GET OPER	Convert a function operand to a value M0 bad
58	3A	TRUNCATE	Replace top of stack with truncation (to 0).
59	3B	SINGLE CAL.	Perform single calculation (code in B)
60	3C	E convert	Convert a number of #Em to top of stack
61	3D	Restack	Restack a number.
	86,88,8C series		Series generator for trig fct. etc. B,M0-M2
	A0-A4 Stk		A0 = STK 0, A1 = STK 1, A2 = STK 1/2, A3 = STK PI/2, A4 = STK 10.

C0-C5 STK MEM Stack from top of stack to memory 0 to 5 respectively.
E0-E5 GET MEM Put on top of stack MEM 0 to 5.

Some of these routines need a bit more explanation.

Routine handling instructions. You will notice the inclusions that really don't do any calculations but merely aid the programer in writing a routine. These are:

Jump if true--true is 1 from stack not the zero flag.
(Jump & jump if T calculate dis from dis byte)
Delete (top number only)
STK to MEM (C0 series) does not clear number from stack.
GET from MEM (E0 series) doesn't clear memory.
Duplicate top number again
DJNZ needs number in B register
Exchange top number with 2nd number.
END F.P.

Also notice the ability to stack often used constants, 0, 1, 1/2, PI/2 and 10 with the A0-A5 series. If other constants are needed they can be added at the right time with STK DATA followed by your 5 byte number.

The STK series (86,88,8C) is used by the floating point calculator to calculate LN, EXP and the trigonometic functions by use of the Chebyshev polynomials. For an explanation of their use see Logan, "The Complete Timex TS1000/Sinclair ZX81 ROM Disassembly" Appendix.

RESTACK # can only be used under the following conditions. HL must be pointing at the sign of the number which is held low/high in the next 2 bytes of memory.

Logic functions. You will notice the inclusion of all the logical operations. Thus a 1 or a 0 are always left on the top of the stack to do with what you want.

The rest of the functions are the calculating kind and should be quite obvious as to what they are doing.

X mod Y is little understood but really means divide X by Y but stop at the integer value, don't go into decimals. Leave the remainder. Thus using it and not having a zero remainder immediately tells one X is not divisible by Y. The integer is on the top of the stack with the remainder beneath it.

A word of caution about using RST 40. It uses ALL, and I do mean all the registers including all the primes. Anything that you have to save must be pushed before you do a RST 40 or it's lost.

LOADING AND UNLOADING THE STACK

Getting numbers to and from the stack can best be done using the routines already available in the ROM. There are quite a few because of all the different ways of handling numbers.

INTEGERS:

To Load: 12518 STACK A (single byte unsigned)
12521 STACK BC (double byte unsigned)

To get back: 12640 F.P. to BC
12691 F.P. to A

Your integer number starts in A or BC and comes back in A or BC. If the number is too big you get an error message.

ALREADY SLUGGED NUMBERS:

Always uses the format AEDCB
To load: 11892 PUT AEDCB to stack
To get back: 12207 Stack fetch to AEDCB

You of course have to put the number into AEDCB format and it's still in AEDCB when it comes back. To save it to memory or get it from memory write the following subroutines.

```
MEM to AEDCB: LD HL, last byte of # before calling.
              LD B, (HL)
              DEC HL
              LD C, (HL)
              DEC HL
              LD D, (HL)
              DEC HL
              LD E, (HL)
              DEC HL
              LD A, (HL)
              RET

AEDCB to MEM: LD HL, exponent byte address before calling.
              LD (HL), A
              INC HL
              LD (HL), E
              INC HL
              LD (HL), D
              INC HL
              LD (HL), C
              INC HL
              LD (HL), B
              RET
```

Note that they are written in opposite form, one forward, one reverse. Thus, should you get a number from memory and load it into the stack all you have to do is PUSH HL to save the address so you just POP HL and call the return back to memory, thus storing your answer where the number originally was.

DECIMAL NUMBERS

The problem with decimal numbers, numbers with decimal points in them, is that they have to be slugged first. You have two alter-
natives: either slug them yourself, or have the 2068 do that for you.

Having the 2068 do it for you requires the use of the routine at 12406 Decimal to F.P. Since this routine uses RST 24 and RST 32 we have to set CHAR ADDR (23645-23646) to the address of the first number of our number. This number must be in ASCII code but may be in regular or scientific (E) format. Make sure the byte after the end of the number is a non-integer ASCII code like a space. Our number ends up right on the top of the stack.

Binary numbers from ASCII code can use the same routine with a call to 12377 after A is loaded with the token for BIN (196). This routine runs right into the decimal to floating point rou-
tine.

Getting slugs back into decimal is easy if you want them to the screen. It's not so easy if you just want to store them as the routine uses RST 16, PRINT A CHAR. It is called at 12705. Remem-
ber to do "PRINT ," before calling the routine. This routine also uses MEM STK locations so any numbers you had stored in these locations are overwritten.

PUTTING IT ALL TOGETHER--A F.P. EXAMPLE

Now that we understand something about the operation of the f. p. calculator, let's do an expression. How about:

X = (-B + (B^2 -4*A*C)^(1/2))/(2A)

If you remember your algebra, it's one of the solutions to a quadratic equation of the form:

A*X^2 + B*X + C = 0

Let's use: 2X^2 + 3X -65 = 0

Then, A = 2, B = 3 and C = -65

We have to calculate the expression B^2 - 4*A*C first so let's put B on the stack first, followed by C and then A on top.

```
62,3          LD A, 3      B = 3
205,230,48    CALL STACK A (12518)
62,65         LD A, 65     C = 65
205,230,48    CALL STACK A
62,2          LD A, 2      A = 2
205,230,48    CALL STACK A
239           RST 40       Do f.p. calc.
```

```
49              Duplicate  A to top of stack
15              Add        2A
196             STK MEM 4  Save 2A
49              Duplicate  2A to top of stack
15              Add        4A
1               Exchange   C to top
27              Negate     C= - 65
4               Multiply   4AC
1               Exchange   B on top
197             STK MEM 5  Save B
49              Duplicate  B to top of stack
4               Multiply   BxB
1               Exchange   4AC on top
3               SUB        BxB -4AC
40              SQR *      (BxB -4AC)^(1/2)
229             GET MEM 5  B on top
27              NEG        -B
15              ADD        -B +(BxB -4AC)^(1/2)
228             GET MEM 4  2A on top
5               Divide     (-B +(BxB -4AC)^(1/2))/(2A)
56              END f.p. calc
205,161,49      CALL Print f.p. (12705)
201             RET
```

* Will get invalid argument if you try to take the SQR of a negative number.

Notice how we generated a 2A by duplicate and add and, since we need it later we saved it to the stack. 4A was gotten by another duplicate and add. B squared was gotten by duplicate and multiply. Also notice how both the subtrahend (the number to be subtracted) and the divisor (the number that we are going to divide by) have to be on the top of the stack with the other number directly below it.

A few more precautions. Notice how STACK A only stacks an unsigned number. Doing LD A, 191 does not stack a -65. The same applies to STACK BC.

Also, saving numbers to STK MEM doesn't always guarantee that they will be there when you want them as something else you may be doing may require a STK MEM location. It is better to use them from the top down, i.e., high locations first.

Troubleshooting floating point routines. It is best to go through your routine and check what is on the top of the stack, or if what is on the stack is correct. If you are using a DATA statement to enter your code, it is quite simple to add the 56 for END F.P. and follow that with 205,161,49,201 Print f.p. and RET and move it along as you check out the routine step by step. Especially check GET MEM to make sure it's still the same number you did with STK MEM.

We are still handicapped with not being able to enter fractional

numbers directly as this point, but read on.

DIGGING DEEPER

A lot more is happening than really meets the eye when we use the floating point calculator. The routines are not the relatively simple ones for add, subtract multiply and divide of the preceding chapter. The complexity results because of the use of an exponent byte and the normalization of numbers.

Normalizing numbers.

You ask what is normalization? It's another mathematical term meaning putting in a standardized form. Normalization is one of the things the 2068 does when it slugs a number. Since you may wish to use numbers in a form ready for the f.p. calulator let's look at the process.

The process consists of 3 parts:

1. Writing the binary form for a number using the 32 most significant bits. For conversion of fractions to binary see Chapter 1.

2. Shifting the decimal point of the binary number full left by adjusting the exponent.

3. Adjusting the first bit of the mantissa for the sign of the number.

For an example, lets use: 65535:

STEP 1. Binary notation.

1111 1111 1111 1111. 0000 0000 0000 0000

We could use:

0000 0000 0000 0000 1111 1111 1111 1111.

but the instructions say, "32 MOST significant bits" so the 2nd version is out.

STEP 2. Shift decimal.

Notice where the decimal point is. At this point our exponent is still 128. We have to move it 16 places left. That means add. 128 + 16 = 144. Our number now is:

144 .1111 1111 1111 1111 0000 0000 0000 0000

STEP 3. Put In Sign.

If our number were negative we would be done. It's posi-

tive. Zero the first byte of the mantissa. The final form is:

144 .0111 1111 1111 1111 0000 0000 0000 0000

Our 5 bytes become:

65535 = 144,127,255,0,0. Also note: - 65535 = 144,255,255,0,0.

Kindly note that the same 32 bits of numbers can represent a whole set of 256 different numbers depending upon what the exponent is. For our number some of these are:

EXP	NUMBER	EXP	NUMBER	EXP	NUMBER
144	65535.00000	138	1023.984375	145	131.070
143	32767.50000	137	512.9921875	146	262,140
142	16383.75000	136	255.99609375	147	524,280
141	8191.87500	135	127.998046875	148	1,048,560
140	4095.93750	134	63.9990234375	149	2,097,120
139	2047.96875	133	31.99951171875	150	4,194,240

Note that we have given the EXACT values of the numbers to as many decimal places as it took. As we point out in Chapter 1, the computer couldn't give you 31.995117900 exactly. It's accuracy goes awry about the 9th decimal number.

Floating Point Additions of Exponentiated Numbers.

Trying to follow through the ROM routines for just Add, Subtract, Multiply or Divide can be a bit harrowing since the routines take the slugged numbers and put them in registers. This makes their manipulation easier and faster but it also limits the length of the mantissa that can be handled. Imagine if you will, having to have to handle two numbers as in multiply or divide. That takes 10 registers! Once you remove the sign of the number, you need another byte to hold that so you really need 12 registers for the two numbers. Then, you can't just put one number into say BCDEHL and the other into B'C'D'E'H'L' as there would be no way to add L to L' or subtract L' from L. The numbers are split up as H'B'C'BC and L'D'E'DE. A routine to shift the first number left one bit then would read something like:

```
RL C
RL B
EXX
RL C
RL B
EXX
```

Just keeping track of which set of registers is in use and being worked on is an effort. This makes understanding what is going on difficult. What follows is a simplification of the process--just explaining the reasoning without getting into the actual

HOW it's done.

Limited Precision. Because of the use of registers for manipulation of the numbers, a person desiring to write a higher precision routine literally must start over with a new way of doing things and use HL and DE to point at storage bytes. There are no more registers to handle any longer mantissas. By working "in place", a routine could be written for as long a mantissa as desired.

Let's see what really goes on when we try to add two numbers given in the SLUG form, i.e., with an exponent to keep track of the binary decimal point. Suppose we want to add:

```
65,535.000000   144   0111 1111  1111 1111  0000 0000  0000 0000
 1,023.984375   138   0111 1111  1111 1111  0000 0000  0000 0000
```

Our slugs would look as given for both numbers.

The first thing we have to do is get the correct mantissas and store the signs in a register. Well, 2 registers, one for each number. So our slugs become:

```
144   1111 1111  1111 1111  0000 0000  0000 0000
138   1111 1111  1111 1111  0000 0000  0000 0000
```

We can't add the mantissas yet as the exponents are different. We have to get the smaller (lower exponent number) to the same exponent by rotating right the mantissa the correct number of bits. In our case it's 6.

```
144      1111 1111  1111 1111  0000 0000  0000 0000
144      0000 0011  1111 1111  1111 1100  0000 0000

Adding:  144 (1) 0000 0011  1111 1110  1111 1100  0000 0000
```

We got an overflow which means that that first (1) is really in the carry flag at this point. We have to shift our whole mantissa right one bit to accomodate it and at the same time up our exponent:

```
145  1000 0001  1111 1111  0111 1110  0000 0000
```

And finally put the sign back:

```
145  0000 0001  1111 1111  0111 1110  0000 0000
```

We leave it to the student to verify that this number is really 66558.983475.

Floating Point Subtraction of Exponented Numbers:

Let's use the same two numbers. In algebra we learned that to

subtract we change the sign and add the two numbers. In machine code tht means we NEGate the number. Negate you will recall is CPL (complement) and INC. Let's start with the shifted number 1023.984375 already corrected.

```
          144  0000 0011  1111 1111. 1111 1100  0000 000Q
CPL            1111 1100  0000 0000. 0000 0011  1111 1111
INC            1111 1100  0000 0000. 0000 0100  0000 0000
```

Adding the two numbers:

```
          144      1111 1111  1111 1111  0000 0000  0000 0000
          144    + 1111 1100  0000 0000  0000 0100  0000 0000
          144 (1)  1111 1011  1111 1111  0000 0100  0000 0000
```

Notice how the INC of the number after the CPL was done at the end of the low bit. Now, let's look at our final number. Since we were adding a negative number we EXPECT an overflow so we don't adjust the final number but merely forget about the bit in the carry flag (See above for add). If we did not get a carry we would have to adjust everything LEFT a bit and adjust the exponent downward.

Let's see, the first two bytes are 65535-1024 = 64511. The 3rd byte adds 0.015625, the 4th nothing. 65411.015625 = 65535 - 1023.984375.

Multiplying Two Slugged Numbers

We again start by adjusting the sign bit of the mantissas for both numbers. Since we are only interested in the 32 most significant bits of the mantissa we don't care if we rotate the low part of our number off the low end but we do care if we lose bits off the high side. We will thus start by multiplying from left to right. One number called the multiplicand (MPC) will start out unchanged and will be added to the answer if the left most remaining digit of the multiplier (MX) is a 1. After that the MPC will be rotated right one bit for the next add. If no carry we will skip adding to the answer accumulator. We have to allow for an accumulator adjust routine just in case we overflow our answer mantissa on the high (left) side.

For our numbers let's chose:

```
MX multiplier     1100 1100  1100 1100  0000 0000  0000 0000
MPX multiplicand  1111 1111  1111 1111  0000 0000  0000 0000
```

Instead of giving you the shifted left multiplier each time we will just indicate it by MX=1 or MX=0. Just count off the bits from the left as you go through the following routine.

```
Starting answer   0000 0000  0000 0000  0000 0000  0000 0000
Starting MPC      1111 1111  1111 1111  0000 0000  0000 0000
```

```
1st MX=1  ANS =    1111 1111  1111 1111  0000 0000  0000 0000
Shift right MPC    0111 1111  1111 1111  1000 0000  0000 0000
2nd MX=1  ANS = c  0111 1111  1111 1110  1000 0000  0000 0000
Adjust answer      1011 1111  1111 1111  0100 0000  0000 0000
```

The answer adjust is a SR and an INC in the 1 bit Also we have to INC the EXP. This requires a double shift right of the MPC, one for the shifted answer and a regular shift.

```
Double SR MPC      0001 1111  1111 1111  1110 0000  0000 0000
3rd MX=0 SR MPC    0000 1111  1111 1111  1111 0000  0000 0000
4th MX=0 SR MPC    0000 0111  1111 1111  1111 1000  0000 0000
5th MX=1 OLD ANS   1011 1111  1111 1111  0100 0000  0000 0000
         NEW ANS   1100 0111  1111 1111  1011 1000  0000 0000
         SR MPC    0000 0011  1111 1111  1111 1100  0000 0000
6th MX=1 NEW ANS   1100 1011  1111 1111  1011 0100  0000 0000
         SR MPC    0000 0001  1111 1111  1111 1110  0000 0000
7th MX=0 SR MPC    0000 0000  1111 1111  1111 1111  0000 0000
8th MX=0 SR MPC    0000 0000  0111 1111  1111 1111  1000 0000
9th MX=1 OLD ANS   1100 1011  1111 1111  1011 0100  0000 0000
         NEW ANS   1100 1100  0111 1111  1011 0011  1000 0000
         SR MPC    0000 0000  0011 1111  1111 1111  1100 0000
10th MX=1 N. ANS   1100 1100  1011 1111  1011 0011  0100 0000
         SR MPC    0000 0000  0001 1111  1111 1111  1110 0000
11th MX=0 SR MPC   0000 0000  0000 1111  1111 1111  1111 0000
12th MX=0 SR MPC   0000 0000  0000 0111  1111 1111  1111 1000
13th MX=1 O. ANS   1100 1100  1011 1111  1011 0011  0100 0000
          N. ANS   1100 1100  1100 0111  1011 0011  0011 1000
          SR MPC   0000 0000  0000 0011  1111 1111  1111 1100
14th MX=1 N. ANS   1100 1100  1100 1011  1011 0011  0011 0100
```

The routine goes on with more SR MPC and in another 3, we start losing the least significant digits off the bottom end. However, if we examine our MX we have just passed the last 1 and so there will be no more additions to the answer. It is complete. All that remains is for us to calculate the exponent of the number.

You noticed that we didn't start by designating the exponent. In multiplication, it makes no difference what they are until we come to the answer adjustment routine. We start out the answer with the same EXP as the MPC. But how do we know where to INC-rement the answer accumulator when we do an answer adjust? Well, if 1 has an EXP of 129 we just adjust the EXP - 128 place from the left in the mantissa. Okay, our multiplicand has to have the starting EXP of 128 + 16 = 144 as we incremented the 16th place. That's good old 65535 again. We also incremented this exponent, so at this point it's 149 to take care of the shift. The final exponent of the answer now has to be adjusted for the EXP of the Multiplier (MX). This adjustment is simply + (EXP MX - 129).

Of course we still have to adjust our number by adjusting the first bit of the mantissa for the sign of the number following the usual algebraic notations.

Dividing Slugged Numbers.

The routine again starts by adjusting the first bit of the mantissa for the signs. It does not use negated numbers and adding to do subtractions. This time the number being divided will be rotated left while the divisor remains stationary and is subtracted from it. If subtraction results in an overflow, indicating a negative number, the divisor is added back. The answer is rotated left and incremented if subtraction was possible. No increment takes place if subtraction was not possible. Since rotation left of the remainder may result in an overflow, a test of the higest bit is first made to make sure it can be done.

For our example, let's do 65535/25 = 2621.4.

```
65535 =  1111 1111  1111 1111  0000 0000  0000 0000
SUB 25   1100 1000  0000 0000  0000 0000  0000 0000
REM      0011 0111  1111 1111  0000 0000  0000 0000
INC ANS and RL= 1(0). The (0) indicates the next place which may
or may not be INCremented before the next rotation.
RL REM   0110 1111  1111 1110  0000 0000  0000 0000
SUB      1100 1      not possible
ANS =    10(0)
RL REM   1101 1111  1111 1100  0000 0000  0000 0000
SUB      1100 1
REM      0001 0111  1111 1100  0000 0000  0000 0000
ANS      101(0)
RL REM   0010 1111  1111 1000  0000 0000  0000 0000
SUB      1100 1      not possible
ANS      1010(0)
RL REM   0101 1111  1111 0000  0000 0000  0000 0000
SUB      1100 1      not possible
ANS      1010 0(0)
RL REM   1011 1111  1110 0000  0000 0000  0000 0000
SUB      1100 1      not possible
ANS      1010 00(0)
At this point the test of the high bit will indicate that RL REM
is not possible. We must instead RR the Divisor. We also increment the EXP of the answer.
   REM   1011 1111  1110 0000  0000 0000  0000 0000
RR & SUB 0110 01
REM      0101 1011  1110 0000  0000 0000  0000 0000
ANS      1010 001(0)
RL REM   1011 0111  1100 0000  0000 0000  0000 0000
SUB      0110 01
REM      0101 0011  1100 0000  0000 0000  0000 0000
ANS      1010 0011 (0)
RL REM   1010 0111  1000 0000  0000 0000  0000 0000
SUB      0110 01
REM      0100 0011  1000 0000  0000 0000  0000 0000
ANS      1010 0011  1(0)
RL REM   1000 0111  0000 0000  0000 0000  0000 0000
SUB      0110 01
REM      0010 0011  0000 0000  0000 0000  0000 0000
ANS      1010 0011  11(0)
RL REM   0100 0110  0000 0000. 0000 0000  0000 0000
```

```
SUB       0110 01      not possible
ANS       1010 0011  110(0)
RL REM    1000 1100  0000 0000
SUB       0110 01_____________
REM       0010 1000  0000 0000  X
ANS       1010 0011  1101 (0)
RL REM    0101 0000  0000 0000
SUB       0110 01      not possible
ANS       1010 0011  1101 0(0)
RL REM    1010 0000  0000 0000
SUB       0110 01_____________
REM       0011 1100  0000 0000  X
ANS       1010 0011  1101 01(0)
RL REM    0111 1000  0000 0000
SUB       0110 01_____________
REM       0001 0100  0000 0000  X
ANS       1010 0011  1101 011(0)
RL REM    0010 1000  0000 0000
SUB       0110 01      not possible
ANS       1010 0011  1101 0110 (0)
RL REM    0101 0000  0000 0000
SUB       0110 01      not possible
ANS       1010 0011  1101 0110 0(0)
RL REm    1010 0000  0000 0000
SUB       0110 01_____________
REM       0011 1100  0000 0000  X
```

Notice that the remainders keep alternating between 101 and 1111 at the lines we have marked with an X. Our number will never come out even. We however can write the remaining digits of our answer as alter-sequences of 0110.

FINAL ANSWER 1010 0011 1101 0110 0110 0110 0110 0110

Calculating our answer exponent. We started with 144 for our number 65535 which we assign to the answer. In the course of the operations, it got incremented to 145. We now subtract off the EXP of the divisor and add the zero point. 25 would have an EXP of 133. Thus, 145 - 133 + 128 = 140 or 12 spaces right. Our number without exponent is:

1010 0011 1101. 0110 0110 0110 0110 0110

Consulting our table in Chapter 1 we get:

```
    1   .25
    4   .125
    8   .015 625
   16   .007 812 5
   32   .000 976 562 5
  512   .000 488 281 25
 2048   .000 061 035 156 25
        .000 030 517 578 125
        .000 003 814 697 265 625
____    .000_001_907_348_632_812_5
2621    .399 999 618 530 273 437 5
```

Rounding: 2621.400000

You can see from this that our answer is accurate to 10 significant numbers as advertised.

PRINT A F.P NUMBER/DECIMAL TO F.P.

We did the translation above by hand. How does the computer take a floating point binary number and convert it to decimal or take a decimal number and make it a floating point binary number?

The routine for decimal to floating point works something like:

Stack a zero.
Check for Digit, if it is, stack it.
Exchange top two numbers.
Stack 10 and multiply--effectively multiplying everything previously on stact by 10.
Add the new number to the 10X previous number.
Loop back and get the next number.
If you get a non-decimal, check for "." and "E" else go to end routine.
If "." store the exponent and number at this point. Start a new number by stacking a 1--there may be several zeros following the decimal point and this will keep track of them. Get the next code and proceed as above.
If "E" start a 3rd number. The next symbol will be the sign so save that. Then proceed as above with digits. At end add to EXP of integer if sign was "+" or subtract if "-". Go to end routine.
END Routine. Adjust fractional number to EXP 129 and subtract the 1. Get the integer number and add the two with the exponents as calculated.

The PRINT F.P. NUMBER routine located at 12705 is 422 bytes long and is quite complex as it has to handle a lot of different numbers. It converts a floating point binary number to decimal and ends by printing it to the screen all in one. It works something like this (simplified).

Check sign of number--jump to positive or negative routine.
Duplicate number.
Take integer and save it.
Subtract integer from fraction and save fraction.
Handle the integer as follows.
RL digits and move into A with ADC A, A effectively doubling the previous number already in A.
Do a DAA if necessary and load A into (HL).

The routine is as follows:

Shift all Binary digits left.

```
Loop LD A, (HL)
     ADC A, A
     DAA
     LD (HL), A
     DEC HL
     DEC C
     JR NZ, Loop
```

Let's follow a typical number through this routine and see how the decimal numbers get generated. Why not pick good old 65535. It's 16 consecutive 1's if you remember. It will require 16 rotations of the number each one of which will cause a carry. We have already cleared MEM to hold our number and HL is pointing at what would normally be the exponent byte. Watch the numbers that appear in the memory locations. This is the one and only time the 2068 uses BCD notation. The DAA operation is ommited if it accomplishes no change.

	MEM 1	#/#		MEM 2	#/#		MEM 3	
1st ADC	0000 0001	0/1						
2nd ADC	0000 0011	0/3						
3rd ADC	0000 0111	0/7						
4th ADC	0000 1111	0/15						
DAA	0001 0101	1/5						
5th ADC	0010 1011	2/11						
DAA	0011 0001	3/1						
6th ADC	0110 0011	6/3						
7th ADC	1100 0111	12/7						
DAA	0010 0111	c2/7	ADC	0000 0001	0/1			
8th ADC	0100 1111	4/15	ADC	0000 0010	0/2			
DAA	0101 0101	5/5						
9th ADC	1010 1011	10/11						
DAA	0001 0001	c1/1	ADC	0000 0101	0/5			
10th ADC	0010 0011	2/3	ADC	0000 1010	0/10			
			DAA	0001 0000	1/0			
11th ADC	0100 0111	4/7	ADC	0010 0000	2/0			
12th ADC	1000 1111	8/15	ADC	0100 0000	4/0			
DAA	1001 0101	9/5						
13th ADC	10010 1011	18/11						
DAA	1001 0001	c9/1	ADC	1000 0001	8/1			
14th ADC	10010 0011	18/3						
DAA	1000 0011	c8/3	ADC	10000 0011	16/3		MEM 3	
			DAA	0110 0011	c6/3	ADC	0000 0001	0/1
15th ADC	10000 0111	16/7						
DAA	0110 0111	c6/7	ADC	1100 0111	12/7			
			DAA	0010 0111	c2/7	ADC	0000 0011	0/3
16th ADC	1100 1111	12/15						
DAA	0011 0101	c3/5	ADC	0100 1111	4/15			
			DAA	0101 0101	5/5	ADC	0000 0110	0/6

The final registers have the numbers, 3/5 5/5 0/6 in them reading nybblewise. If we rearrange the bytes reading right to left we get 065535.

Converting Binary Fractions To Decimal Fractions.

We now can get back from STK MEM any fraction and work on that. The fractional conversion uses a different scheme of multiplying the fraction by 10. The integer part of the number then turns out to be the next digit of the fraction. The multiplying of the number of course starts with the least significant byte of the mantissa. Any overflow is carried over to the next byte and added after this byte is multiplied by 10 with any overflow again being carried over to the next most significant byte. The final carryover is the digit we wish to print. This is all done by an ingenious routine called CA = 10*A + C.

An example is .1111 1111 0000 0000 0000 0000 0000 0000. From our table in Chapter 1, we know the answer should be .99609375. Starting with the least significant bytes we note that there will never be a carryover from lower bytes so that our process becomes simplified...all we have to do is multiply the number by 10 until we have zero remainder.

Start		.1111 1111	
x10	1001	.1111 0110	Integer 1001 = 9. Drop integer & repeat.
x10	1001	.1001 1100	Integer 1001 = 9.
x10	0110	.0001 1000	Integer 0110 = 6.
x10	0000	.1111 0000	Integer 0000 = 0.
x10	1001	.0011 0000	Integer 1001 = 9.
x10	0011	.1100 0000	Integer 0011 = 3.
x10	0111	.1000 0000	Integer 0111 = 7.
x10	0101	.0000 0000	Integer 0101 = 5.

Our number is complete.

Rounding And Using Scientific Notation

The routine is far from done as it probably has decoded too many digits and so must proceed with rounding procedures. The use of scientific notation also still has to be decided upon. This means changing the position of the decimal point in the number to the 2nd printed figure. It is a little beyond the scope of this book to handle these subjects here. We also leave it to the student on how to get from BCD numbers (one number per nybble) to ASCII codes a single digit to a byte. The routine for doing this is really inserted between digitizing the integer part of the number and digitizing the fractional part if you're trying to translate the ROM.

GRAPHICS--PLOT and DRAW

Plot is the basis for all draw routines. One tells the computer to Plot a point and then draw a line, or an arc, to the next point. Draw takes the two end coordinates and calculates the next point to be plotted. Plot takes these positions and converts them to the correct screen bit to be turned on. It is in-

teresting to see how the coordinates X and Y are converted into a screen position. The following routine, given by Timex, uses the FULL screen including the bottom 2 lines not normally available in Basic.

For setup Y is held in B, X in C. The scheme for the bytes is as follows:

```
                Y                                   X
 7  6  5  4  3  2  1  0     7  6  5  4  3  2  1  0
Screen Line #   Scan row      Column   #   Bit  #
block  in block in line         0-31         0-7
0,1,2    0-7     0-7
```

Bits 2-0 of Y become Bits 2-0 of H. Bits 7 and 6 of Y must be moved down to 3 and 4 of H. Bit 7 of H must be 0 with Bit 6 = 1. Bits 5-3 of Y become Bits 7-5 of L while Bits 7-3 of X become Bits 5-0 of L. Bits 1-3 of X are the pixel position to be turned on reading left to right and must be converted to right to left Bit number notation.

```
PLOT X,Y  PUSH BC   Save coordinates for further calculations
          LD A, 192     Test B
          SUB B
          JR C, Error   Y too big
          LD B, A       B now inverted, TOP = 0
          AND 192       Save only screen block
          RRA           Rotate down to positions 3 and 4
          RRA
          RRA
          LD H, A       Save block
          LD A, B       Reload B
          AND 7         Save only scan row
          OR H          ADD in screen block
          OR 64         Add in D File start
          LD H, A       Save in H
          LD A, C       Get X
          RLCA          Rotate around partly
          RLCA
          RLCA
          AND 199       Save only bits 7,6,2,1,0
          LD L, A       Save partly shifted data
          LD A, B       Get line # from Y
          AND 56        only save line #
          OR L          Add in L
          RLCA          Finish rotation
          RLCA
          LD L, A       L done
          LD A, C       Work on pixel #
          AND 7         Save only pixel #
          LD B, A       Save in B, # = 0-7
          INC B         # now 1-8
          LD A, 1
Loop      RRCA
          DJNZ, Loop
```

```
            OR (HL)       Add in screen byte
            LD (HL), A    Print to screen

                Handle attribute here

            POP BC
            RET
      Error RST 8
            10      INTEGER out of range
```

DRAW

Drawing a straight line must only increment X and/or Y by one or the line will not be continuous. Partial increments of X or Y must be rounded and either the whole position or an increment addition to the present increment must be made. The MEM STK keeps these numbers between successive plots.

For example: PLOT 0,0: DRAW 200,50. Calculates the slope of the line to be (50-0)/(200-0) =.25. It's going to take 200 PLOTS to do it for the 200 positions of X to be navigated. Y is going to move up only .25 of a pixel for each X plotted. In Basic it would be:

```
Let Slope = (Y1 - Y0)/(X1 - X0)
FOR X = 1 TO 200
LET Y = INT (Slope*X)
PLOT X,Y
NEXT X
```

Imagine how difficult plotting an arc or an ellipse must be.

CHAPTER 9

PERIPHERALS

DOT MATRIX PRINTERS

To operate a dot matrix printer you not only need the printer but an interface and a firmware program called the Print Driver Program. The interface has to be compatible to the 2068, i.e., match the circuit board lines that come out the back side of the computer. These are different on the 2068 than the Spectrum and the "Silver Avenger". The other end has to plug into your printer. Since Timex never got around to supplying us with a standard dot matrix printer much less an interface, 3rd party venders have stepped in to fill the void. The two favorite interfaces are the AERCO and the TASMAN. Both are parallel interfaces. One could use a serial interface but that slows things down tremendously.

Printers are pretty well standardized. Generally any Printer Driver Program can be modified to run any printer although driver prograns may not work with a different interface. Timing for some printers has had to be modified inside the printer driver program to get them to run. Modifications to operate a particular dot matrix printer are done in Basic and POKEd into the code portion of the program which then can be saved and used "as is" each time it is needed without the Basic.

One thing that the printer driver program does is change the channel address jumped to by the LPRINT and LLIST commands to call the dot matrix driver routine instead of the 2040 printer routine. Once modified LPRINT and LLIST automatically go to the dot matrix printer.

Since the COPY command does not use a channel, using copy still sends the signals to the 2040 printer. To COPY to a Dot Matrix Printer requires the use of a RANDOMIZE USR call to that portion of your driver program that handles it. (We can't change the COPY routine as it is in unchangable ROM unless we go through some hardware changes.) Since COPY will be sending pixel lines rather than ASCII code to the printer, the printer must be converted to accepting this type of signal by sending it some escape code commands.

ESCAPE CODES

When we first turn on our printer, its internal ROM sets itself up with certain values for the margins, font, line spacing and the like. These preset values are default values. In other

words, the ones it will use if you don't send it any commands to change them. They are generally called escape codes because the majority of them all start with the character #27. In fact, all the codes below 32 are considered command controls for the printer. ASCII codes start at 32 and run to 127 only. All the ESCape codes are given in one of the appendixes in the back of your printer manual. #27 is labeled ESCape. You will notice that certain of the numbers below 32 have other names as well. Of particular note is #13 better known as linefeed. Like machine code instructions, the printer ROM knows when it receives a certain code exactly how many following bytes go with that code and will interpret the next numbers it receives as fulfilling those requirements. Since we can't just press a key that will put code #27 inside a string we are forced to do it with CHR$ 27. Some numbers which follow an ESC (27) are codes for symbols or letters and can be sent as a quote letter, symbol or number instead of CHR$. We can also use an OUT statement to send codes to the printer if we know what port number the interface and the printer are using. The AERCO and TASMAN interfaces use port 127. Several OUT statements, one after the other can also be used to change the configuration of a printer. I have to give you a word of caution here. Some printers are extremely slow in accepting ESCape codes. When they are busy processing a code they send a busy code back saying don't send any more numbers yet. This must be detected by the Printer Driver Program and obeyed or the next numbers will go unheeded. Since you are not using the Printer Driver Program when sending OUT statements directly, you either have to detect these signals with an IN (port), A and IF A <> 0 THEN GOTO (the same line like when using INKEY$), or wait a sufficiently long time to assure that the printer is free before sending the next code number (such as a PAUSE).

DOT MATRIX PIXELS

We learned in Basic that a character or graphic is made up of 8 bytes of 8 pixels each. Actually, in the case of capital letters, there is a blank pixel all the way around the letter. Pixel lines 1 and 8 are blank and so are the 1st and last pixels in the middle 6 bytes. So in reality, we only use a 6x6 matrix to draw the letter. Most lower case letters follow the same format except for j, q, p, g, and y which go below the line with what are called descenders as as such use byte 8 for that purpose. Dot matrix printers use various formats, one of the common ones being a 9x9. This difference in the number of pixels that form a char- acter is no problem in printing the letters as the printer has its own pixel tables for chacharacters--plural as most have several standard fonts already in their ROM. Since we just send the ASCII letter codes, not the pixels, the printer just looks up the right pixel codes like your computer does and uses that matrix. The printer also prints in a different manner by doing all the leftmost pixels of a character at a time, then the 2nd leftmost etc.--the pixels are vertical not horizontal.

The problem comes with COPY where the pixels are sent. The line

spacing has to be changed as well as the mat. Only the top dot is used to print the pixel line sent. The paper moves up only a fraction of a character line to print the next row of pixels, etc. The printer just reads the whole row of pixels one at a time.

Designing Your Own Dot Matrix Characters

Before jumping off and designing your own characters consult the appendixes in your printer manual as you might just find what you are looking for in characters 160 to 255. How you get your printer to print them will depend upon your Print Driver Program. Using CHR$ 180 with some will get you the word TAN which may mean that you may have to resort to the "OUT (port), code" mode of operation discussed above.

Most printers allow for the designing of your own font or character. Of course, it must be designed within the limits of the printer character space. Once designed, the printer must be configured to accept the new font characters and the pixel bytes loaded.

Generally you will only be redesigning a few characters so the first thing you want to do is download (that's the term for getting the entire pixel table the printer uses into its RAM where it can be changed.) The entire set is downloaded and then just the ones you want changed are changed. You have to give up the printing of a standard ASCII character for each new character you want. Without downloading a standard set, your printer is going to print blanks for every character you havn't entered. In this way you can mix your own characters with a standard set.

Designing your own character set is similar but different than designing characters for the screen as you did with UDG characters. First, you can't use those characters directly unless you copy the screen in its entirety--this of course leaves you with a format only 32 characters wide which probably is not what you want on a full sized piece of paper. Since printers vary in how they will use the pixel data you have to consult your Printer's Manual for the required design. Generally, they are going to use pixels in vertical columns rather than in horizontal rows. In addition, your printer may not allow the same print dot to be on in two consecutive columns. There are additional instructions for descenders as well. (Generally a downshift of the entire character by two dot lines.)

Once having your vertical pixel bytes designed and counted you have to send them to the printer as an ESC code. Be sure you have the correct number of bytes required. You have to also include the number of the character you want to change. Typical is:

ESC/42/1/#/d/t1/t2/t3/t4/t5/t6/t7/t8/t9

ESC/42/1 sets up the printer to receive the new pixel data. The

is the ASCII code for the symbol you are going to replace with yours. d is the descender on or off (shift down two dot rows or not) byte. t1-t9 are the vertical data bytes. You have to repeat the whole sequence for each character you are going to want to change.

Finally, you have to enable the now changed set with: ESC/$/1. You turn it off with: ESC/$/0, and go back to the default set.

DIP SWITCHES

Because different Printer Driver Programs operate differently, it may be necessary to reset some of the DIP switches on your printer when you move from one driver program to another. Most printers have 2 sets of these switches, one is usually on the backside of the printer while the second set is inside and not readily available. If your printer isn't behaving correctly, check the setting of these switches. Of particular note is the switch that controls printing whenever a CR (Carriage return) is sent or when the printer's buffer is full. Since most word processors have their own on-board printer driver routines, it's vital for these programs as you can't very readily change the code that easily and must work with the driver given in the program.

WORD PROCESSORS

TASWORD II by Tasman Software, 17 Hartley Crescent, Leed LS6 2LL
M-SCRIPT by Micro-Systems Inc. Distributed by Zebra Systems, NY

Many 2068 owners use their machines for word processors, a situation that wasn't very feasible with the T/S1000 becuase of it's poor keyboard. A better keyboard on the 2068 would make word processing even easier as punctuation isn't easy with the "as received" keyboard--that symbol shift has got to go. The following paragraphs are not a tutorial on how to use word processors but just compare features on the two most popular.

Once you get used to a word processor you will never use a typewriter again. It is the only way to produce perfect copy everytime--if you proofread carefully enough. At least you have the option of correcting your spelling, composition and grammar before you print the copy.

Tasword uses a 32 column screen but shoves 2 characters into each normal character space to achieve a 64 character line. It has its own 4 bit wide pixel character table to do this. (After all, it is an adaptation from the Spectrum which only has one screen mode.) As a result, the characters on the screen are a bit hard to read. The pixels are wide and not reduced in width as they are in the M-Script screen which is a true 64 character screen using both Display Files. M-Script is easier to read and results in less eye strain. The new modified FAT M-Script, a modification you can do yourself, is even better.

In addition, M-Script has a command called "window" which allows you to use lines up to 132 characters across as would be needed with a 15 inch wide dot matrix printer. Only 64 characters are displayed on the screen at a time but as you type in more of a line the screen scrolls left giving you the last 64 characters entered on that line at all times. When you reach the end of the line, it jumps back all the way to the start of the next line and gives you the beginnings of previous lines already entered. In addition it has a keyboard buffer which allows you to keep right on typing while it is taking time to switch lines and do all its other work and then processing the keyboard entries. This is very evident if you are doing line scrolls with cap shifted 6 and enter a whole string of them.

Both programs allow for a whole passel of editing functions such as deleting, inserting, moving blocks, duplicating blocks, searching for strings, merging files and the like. One difference is in the way they handle margins. On Tasword, the left margin and the right margin can be indented giving a shorter line length and spacing it on the screen accordingly. When it goes to the printer it will remain centered. In M-Script there is no left margin set for the screen. Using "window" will shorten your lines but they are all left justified and right unjustified. You don't see the actual composition on the screen that you get on the printer. M-Script has the advantage of using printer commands right in the text to allow for italicizing a word in a line or underlining one. Using center on Taswide centers it within the Text of the screen, in M-Script it doesn't but will on the printer. Tasword is limited to 64 characters to the printer line, M-Script can be any width.

Both use a word entry process known as wrap around. If there is not enough room on the end of the line for the whole word, it is deleted from that line and put on the next line...automatically. One doesn't have to use a carriage return as with a typewriter until the end of a paragraph. Tasword just jumps to the end of a line while M-Script places a "\" at the end of the line and then jumps to the next line. The "\" is not printed, but it makes a difference in how each program uses its file space. Neither pro- gram has a glossary to check word spelling or hyphenation. You have to go back and hyphenate words yourself or put up with long gaps between words on a line. Any guesses as to the longest one syllable word in the English language? Try through.

Both use a file to store the characters that will be sent to the printer but they do it differently. In Tasword, each line consumes 64 bytes of file whether it's a blank line or not. In other words, each line is padded with spaces. This limits Tasword to exactly 300 lines of text including all empty lines. M-Script, on the other hand, starts with a slightly smaller file but doesn't pad lines. A blank line, as used between paragraphs, is just represented by the "\" and uses one byte of file. Par-

tial lines just consume what is written on them plus the "\" character. Thus M-Script can have as many lines as it takes to completely fill its file. Dispite a smaller file, M-Script can hold a longer document than Tasword.

Dot matrix printers generally have several type fonts available such as pica, elite and italics at the very minimum. These can be done in normal, condensed and expanded mode, with or without underline, and in either Bold (that is what double strike and emphasized is all about) or normal strength printing. Extra features are superscripts, subscripts and special characters. You can have a real brawl using all of these and they work from your word processor as well.

Tasword uses the graphics character set to put them into a program. Generally the same number in inverse turns off what you turned on. Only one problem with this is that you are limited to 8 on-off sets. Tasword, as is, does not have any means of printing multipage documents so the first thing that has to be done is take two graphic codes and redefine them for page length and page skip--the number of lines to skip after printing the first page before starting to print the 2nd page so you get over the "fold" in the paper and can leave some margin on the bottom and top of each page. Although the 16 graphics can be changed you are limited to no more than 16 codes.

M-Script has a lot more commands given right into the print format including page numbering, page length and page skip, right and/or left justification. Instead of graphics, M-Script has a "define print statement" that can be used to define up to 10 different print codes at once and then call them up by number as you need them. Nothing prevents you from redefining them half way through a document thus giving one an unlimited number of commands available. Underline, **BOLD** and Tab are already included. One can also include REM statements to remind the operator of anything needed or defined. The use of headers (as is used for printing this book) allows one to automatically number pages--and skip numbering as on the first page of each chapter. A different header for even and odd pages keeps the page numbers on the outside margins. Use of a header spacing automatically gives one the empty lines between the header and the text. Footers may also be used. Forcing of new pages is also allowed.

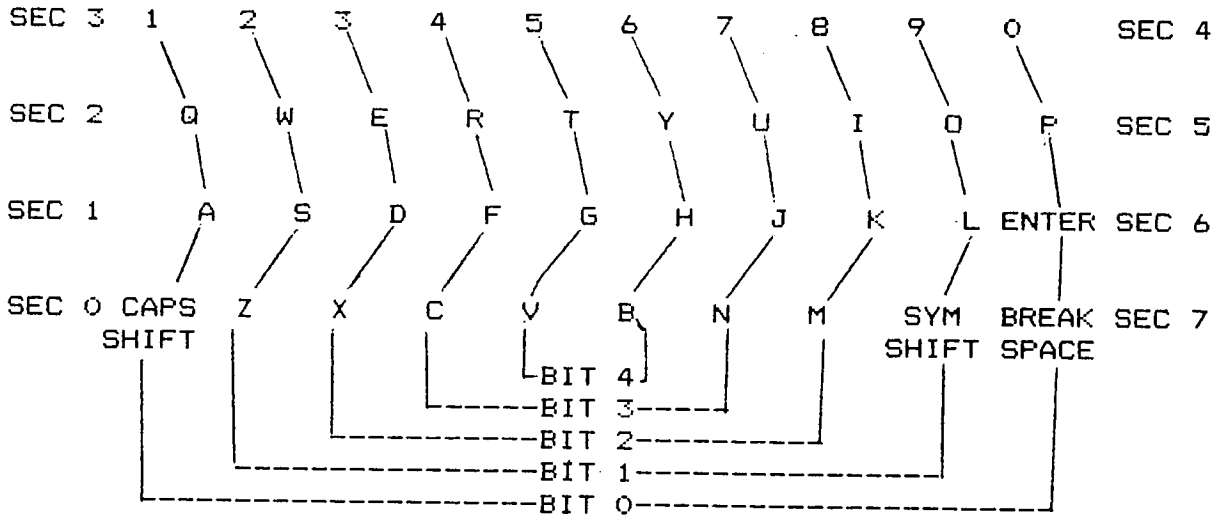
Word Processor Limitations

Because word processors do so much and have redefined most of the characters on the keyboard to handle other fuctions, some cannot handle the simple graphics or special characters. They are generally limited to the ASCII set of 96 codes from 32 to 127. If you want to use any of the other characters your printer has (160-255) you have to download them into the ASCII set as already mentioned. Of course, doing this loses you the normal ASCII character that would be printed with that code.

Please note: In the whole discussion of Printer Driver Programs and Word Processors we have not done a single listing of code. These programs are copyrighted and the property of the writers. Listing their code would be an infringement of that copyright. You may look at it on your own but I can't publish it. Most of it is going to be beyond the ability of the novice machine code student.

THE KEYBOARD

The keyboard is the most versatile means for your computer to obtain information. It is possible to query the entire keyboard or just a section of it to see if a key has been or is being depressed.



The keyboard is sectioned off into 8 sections of a half line each as in the above diagram. It's these sections that can be queried individually. When depressing a key two contacts are closed, one for the half line and a second for the keyboard bit. Both are returned in the format of active bit LOW. Thus no key will return an FFh, a section 0 key FEh, a section 1 key FDh, etc. to Section 7 at 7Fh (127).

Since the keyboard is accessed through port FEh (254), and the port number is always put on the 8 low address lines, the section byte of the keyboard then is put on the 8 high address lines with the keyboard bits going onto the 5 low data lines. Setting B to 0, and C to 254 and then doing an IN (C), A will get you the section number in B and the Bit number in A. Thus, the computer has all the information which, together with mode (K, L, C, E, or G) that it needs to calculate the correct code (ASCII token, graphic or UDG) for whatever key is being pressed.

The Sinclair line of computers (ZX81, TS1000, TS1500 and TS2068) work a bit differently than other micros in that they don't wait for an interrupt from the keyboard but use the automatic 1/60th of a second maskable interrupt to check the keyboard. The 2068 has the additional feature of being able to sense that a key is still being held down and will count for a certain fraction of a second before it starts repeating that key code as another key press. It also waits a different length of time after the first repeat to do a second repeat of the same key.

KEYBOARD ROUTINES

A_Completely_Dead_Keyboard.

Since the keyboard is scanned every 1/60th of a second with a maskable interrupt (INT) routine, disabling this maskable interrupt with the instruction DI kills the keyboard. Doing a RST 56 is equivalent to calling the interrupt routine and will scan the keyboard even with DI being on, AND since there is an EI statement in the end of that routine will enable them from then on. If you don't do a RST 56, make sure you do an EI before coming back to Basic or you are in trouble.

A temporary stop of your program can be achieved with HALT. With the next interrupt (if DI not on, within 1/60th of a second) the program continues. Without INTerrupts enabled you have stopped without a way to restart.

You can also disable the keyboard by setting Bit 6 of Port 255. This is done by:

```
IN A, (255)
SET 6, A
OUT (255),A
```

Reset with RES 6, A instead of SET 6, A.

Caps_Lock

Caps lock can be set with setting Bit 3 of FLAGS 2 (23658). It can be done directly if in normal video mode even from Basic. But in any other video mode it must be done with:

```
LD A, (23658)
OR 8
LD (23658), A
```

To unlock, use AND 247 rather than OR 8.

Reading_The_Whole_Keyboard.

If every 1/60th of a second isn't fast enough for your fast action game, you can force a keyboard read by RST 56. This checks

to see if a key is being depressed and if not just returns. If a key is depressed, it goes through the whole keyboard read routine in ROM, sets Keyhit (Bit 3 of FLAGS, 23611) and puts the code in in LAST K (23560). The code will depend upon what MODE the computer is in at the time (K, L, C, E or G). Mode can be set by altering Bits 0 and 1 of MODE (23617). If you are just interested in the CAPS mode value of keys (no tokens, UDG or lower case), this code can be found at 23556. This method is not the fastest available. It also has the problem of giving you the last key pressed without bothering to reset. To do it properly, one should RESET KEYHIT and then call RST 56, then check for a hit and recycle until you get one if you like.

Of course one can call the keyboard routines directly with CALL 737. The keyboard routines are quite complex and consist of the following routines.

```
688  Keyboard scan
737  Update keyboard
822  Repeat key
860  K Base
881  Character Code
```

The 2068 Technical Manual gives a full flow diagram of these routines. It is beyond the scope of this book to discuss them here.

Calling 737, Update keyboard, will call all the other routines needed and our answer will be in LAST K. We, however, have to have a way of determining if a new key has been pressed by resetting the flag KEYHIT. If a key is being pressed, Keyhit will be set and the ASCII code or Token code will be in Last K. Be sure to Push any registers you have to save before making the keyboard call as the routine uses all the registers. If you Push registers make sure you Pop them after the call.

```
      RES 5, (IY + 1) Flags-Keyhit
Again CALL 737
      BIT 5, (IY + 1)
      JR NZ, Again  Loop waits for hit
      LD A, (23560)  Lask K
```

Testing_Last_K

You may be interested in a whole keyboard or just a few keys. The easiest way to test for just a few keys is to test for a direct match. From the above, Lask K is already in A so:

```
CP #
JR Z, that # routine
CP #2
JR Z, #2 routine
etc.
```

Of course, calling the Basic routines to decode the keyboard is no faster than Basic. But just remember that the keyboard is checked every 1/60th second to see if a key is being pressed and then does the routine to read it. That would be equivalent to just checking it once in the first program above without cycling to wait for a press of a key.

Fast Action--Just Activating Part Of The Keyboard.

If you are just interested in a few keys from the keybaord, you can do a direct decode using the scheme we talked about earlier-
sections and bits. This is what one would use for a keyboard acting as a joystick. The keyboard is read through Port 254. We must remember that our input will be in ACTIVE HIGH format.

```
LD BC,  C=254, B = Section #
IN (C), A
CPL     Convert to active high
BIT X, A    X = Bit #
JR Z, Bit X routine
BIT Y, A    Y = Bit #2
JR Z, Bit Y routine
```

0	254(FE)
1	253(FD)
2	251(FB)
3	247(F7)
4	239(EF)
5	223(DF)
6	191(7F)
7	127(7F)

You can examine several sections of the keyboard in the same call merely by putting two bits low in B (Notice how the section numbers given above are really that Bit # low). But be warned, the answer in A also comes back mixed and you have to have some way of decoding them. For example, turning on section 2 and 5 by putting 219 into B, will let you read QWERT and YUIOP only. But Q and P are both going to reset Bit 0, W and O are both going to reset Bit 1 etc. Doing the CPL turns the Bits from 0's into 1's (it isn't necessary but makes it easier thinking about it). If you have to somehow differentiate between "W" and "O" this method won't do it.

This method of scanning the keyboard doesn't wait until the key is debounced so generally you can repeat it for each of the keys that you wish to check. An obvious application is to make four keys into a keyboard type joystick. The big trouble with most fast action games is that they don't ask for keyboard inputs often enough to move that gun to the right spot or shoot bullets often or fast enough. Remember the screen is wider than it is high.

Break?

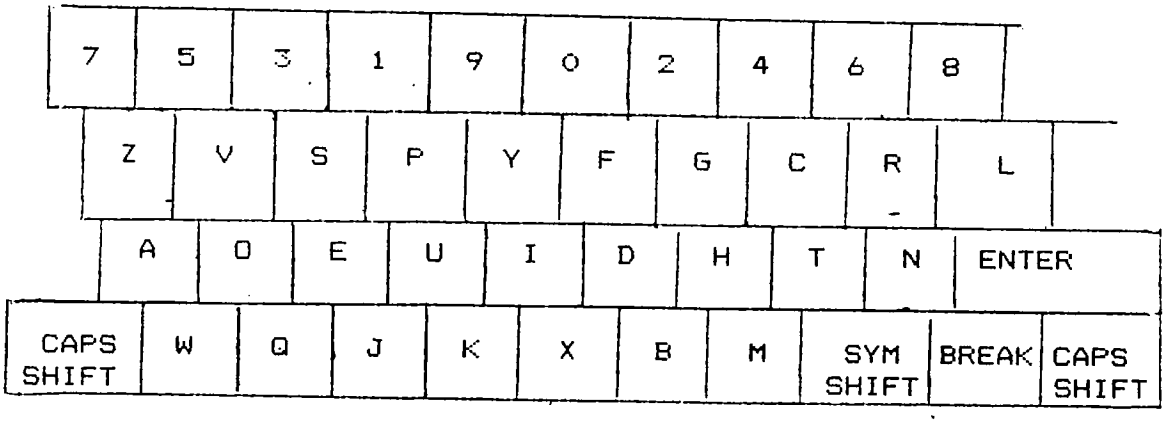
Sometimes one just wants to see if the BREAK key (Break and Caps shift) has been pressed. Just make sure the carry flag is set and then CALL 8201 (2009H). The carry flag will be cleared if Break was pressed.

Input

Reading in a sequence of keys in a routine (like a name) one after the other can be done by storing the LAST K's in a space and going back to get the next until a certain key like ENTER is pressed. However, your computer is so fsst compared to your fingers that you must put in a wait loop so you have time to get your fingers off the keys or you get a whole string of the same letter.

REDEFINING THE KEYBOARD

We can only redefine part of the keyboard from Basic--the UDG's. These require the graphics mode and are limited to 21 in number. Hardly enough for a total redefine of the entire keyboard. What some people would like to do is convert their entire keyboard from "QWERTY" to another like say "DVORAK". A diagram of the Dvorak keyboard is given below.



We had to make some compromises with this layout because the 2068 keyboard doesn't have enough keys. S is supposed to be where the ENTER key is, W where the symbol shift key is with Z at the right CAPS SHIFT. We only have 26 letter keys so all the punctuation is still in the symbol shift mode. W is put where the ":" should be, S where the "." should be, V where the "," should be and Z where the "?" should be.

The way the 2068 assigns codes to keys is through what are called "lookup tables". There is a different table for each mode K, L, C, E and E-Symbol shifted. Each key has a number offset that is added to the table base to give the desired address in the table. That address contains the code sent to Last K. These tables are in ROM and can't be changed.

However, certain programs have their own lookup tables. One such is M-Script. Since these tables are in RAM we can change them. For M-Script the two tables for CAPS and lower case letters are at 42647 and 42583 respectively. Load M-Script as usual and when

the screen comes up asking for color changes, etc. just do a BREAK and enter and run the following program. Use GOTO 600 to get it into the code area.

```
600 RESTORE 600
605 FOR X = 42583 TO 42622
610 READ Y: POKE X, Y: NEXT X
615 FOR X = 42647 TO 42685
620 READ Y: POKE X, Y: NEXT X
625 DATA 119,113,106,107,97,111,101,117,105,122,118,
  115,112,55,53,51,49,57,56,54,52,50,48,108,114,99,1
  03,102,13,110,116,104,100,32,0,109,98,120
630 DATA 87,81,74,75,65,79,69,85,73,90,86,83,80,89,1
  29,130,131,132,133,8,137,136,135,76,82,86,71,70,13
  ,78,84,72,68,128,0,77,78,88
635 STOP
```

Once having run the program, GOTO 9000 and save on a new tape. As long as you stay in Basic, your keyboard is normal. Once you are in the program it's changed, and that includes all letter commands for M-Script. If this is your first time with "Dvorak" I suggest you tape a copy of the keyboard to the upper portion of the computer. The only key that stays the same is A. Have fun.

GRAPHIC PIXEL GENERATION

We told you earlier that the graphic symbols were not included in the pixel character table but that they were done dynamically from the graphic symbol codes themselves. If you have ever bothered to look at these graphic symbols they are composed of four quadrants. Let's label them as:

#2	#1
#8	#4

Let's look at the graphic numbers by subtracting 128 from each and seeing what is left. For this, use Appendix B page 242 of the User's Manual.

```
128 = 0  Space, all empty
129 = 1  #1 space on
130 = 2  #2 space on
131 = 3  #1 and #2 spaces on
132 = 4  #4 space on
133 = 5  #4 and #1 spaces on
134 = 6  #4 and #2 spaces on
135 = 7  #4, #2 and #1 spaces on
136 = 8  #8 space on
137 = 9  #8 and #1 spaces on
138 = 10 #8 and #2 spaces on
139 = 11 #8, #2 and #1 spaces on, etc.
```

The routine goes a follows. Assume CHAR CODE in A to start.

```
            LD B, A
        LD D, 2  Count
Next 4  RR B     Rotate 1st bit to carry
        SBC A, A  If Bit = 1, A = 255, else A = 0
        AND 15    Save low nybble
        LD C, A   Save in C
        RR B      Next bit to carry
        SBC A, A  As above 255 or 0
        AND 240   Save high nybble
        OR C      Add in low mybble
        LD C, 4   count
Again   LD (HL), A  To Display file
        INC H
        DEC C
        JR NZ, Again  4 times
        DEC D
        JR NZ, Next 4  do lower quadrants.
        RET
```

This takes 28 bytes. 16 graphic characters times 8 bytes each would take 128 bytes.

MICRODRIVES

The microdrive, originally developed in England for the Srectrum
and since adopted for the 2068, is nothing but a fast, miniaturized, expensive tape recorder. It uses microtapes of a continuous nature which have an inherant failure problem.

The problem with any continuous tape is, and always has been, that there is too much tension at the "crossover point"--a necessary evil of the system. The excessive tension pulls on the tape and in a short time stretches the outside edges causing them to ripple thus aggravating the problem. The fast speed required for the high baud rate only adds to the problem. In addition, the tape has to rub over itself causing abrasion of the metallic oxide surface and the plastic tape. For the uninitiated, a grinding belt consists of a metallic oxide glued to a cloth belt. Fine iron and chrome oxides on a belt are called polishing belts. They still abrade away material but more slowly.

How bad is this problem? In case you haven't heard, many programs that have been "protected" to prevent copies from being made have been returned to the manufacturer because of malfunction in as little as several months time. So many, in fact, that some manufacturers have refused to support the Spectrum and the QL with any more new products. I don't know what it is, but the British have an aversion to disk drives that borders on being

paranoiac, doing everything and anything to avoid them. The Ferranti chip on the microdrive also has a tendency to malfunction permanently.

Having discussed the major drawback of the microdrive, let's look at the advantages. We get a fast baud rate. The fast baud rate is achieved by doing away with recording sounds for bits as was discussed earlier and just recording pips as is done on disk drives. This achieves closer packing of bits and bytes. A speeded up tape also helps. Loading times for a 32k program including a search for the program are down in the seconds range rather than minutes as with a normal tape recorder. The QL and the 2068 microdive cannot be interchanged.

Economics: You pay $150 for the microdrive interface and a drive. Each tape costs $4.50 vs. $1.50 for a normal tape. An AERCO disk drive and interface is going to cost you $300 but will use disks at $1.00 each. Each disk will handle 32 programs or 395k vs. 85k for each microdrive tape. In addition you will have an additional 64k of memory and an RBG interface ready to accept the cable to your RBG monitor when you get one. Jerry at AERCO is busy with the CP/M DOS system which will allow you to run all those programs on your 2068 as well. Your $300 for the disk drive isn't all just for the interface and the drive. For $50 extra you can get 256k extra memory right on your interface.

MODEMS

Communications with another computer over the phone lines may be the wave of the future but not yet--unless AT&T kills it. In addition to the cost of long distance phone lines, AT&T has threatened to hit Modem Users with a special service fee.

What one really does with a modem is use it much like a dot matrix printer--a printer to another computer. With the printer you sent it print commands to tell it how you wanted something printed and then send it what you wanted printed. A Printer Driver Firmware Program did it for you. Modems work the same way with an interface and a Modem Driver Program in the computer telling it first how to send it and then what to send over a phone line.

Because phone lines are voice channels, it is necessary for the computer to use tones when talking to each other. Because of this, the transmission rate is, of necessity, low--only 300 baud (bits/sec). You are back to the good old TS1000 LOAD/SAVE rates. A 16k program takes over 7 minutes to transmit. Some modems have various baud rates that can be selected. Other things that must be the same for both computers are the word (really byte) size (5, 6, 7 or 8 bits), parity (even, odd or none), and the number of stop bits used (1 or 2).

We have to explain a few terms here.

PARITY: When transmitting data an extra bit is used to make the parity even or odd. The computer counts the number of 1's in a byte and then sets the parity bit to make it even or odd depending upon what is required. In this way the receiving computer can tell if a byte got garbled in transmission. This used to be the standard of transmission between a computer and a printer or other peripheral in olden days when tube noise was a problem.

WORD SIZE: Because baud rates are so low, one wants to use the lowest possible byte length. This, however, depends upon what you are sending. If it's a message all in ASCII code, 7 bits are enough. If it's code or a 2068 program, all 8 bits are necessary. If it's digitized data numbers, only 4 or 5 bytes are needed.

STOP BITS: A blank bit or two at the end of each byte is needed to keep my computer in "sync" with yours which may be operating at a slightly different frequency than mine. This empty space allows both computers to wait until the start of the next byte.

What this all amounts to is that a normal 8 bit byte grows to 9 with the addition of the parity bit and then to 10 with the extra Stop bit. Our transmission rate is now down to 30 bytes per second. If word length is only 4 bits it can be as high as 50 bytes/sec, or 66% faster.

Connecting Up

You just can't plug your modem interface into the back of your computer, connect it to the modem, connect the modem to the phone line, dial a number and expect things to work. You have to know ahead of time what settings to use for a particular service. If you don't know, you have to make a manual phone call and find out. You may wish to talk to the host operator anyway.

Modems attach to phone lines in two ways, direct wiring and acoustically (that the one with the cradle for the receiver). You have to hang up the phone on the wirein type as it's the same type of screech as the tape recorders. Also stray noises are picked up by the mouthpiece and may garble your data.

DUPLEXING. Once hooked together there are various routines that can be used to communicate, person to person, with the operator at the other end of the phone line through the keyboard. Computers use what is called duplexing. This, in reality, sends back to the sending computer the byte that was just received for a match. In HALF duplexing, only what the other computer sends back to you is displayed on your screen. In FULL duplex, both what you typed and what the other computer sent back are displayed one symbol after the other.

USE OF A BUFFER. Typing in messages is slower than using pretyp-

ed messages and programs. Like word processors, most of the computer's memory is not used so can store what is sent for processing at a later time. The buffer is generally made as big as pos- sible to handle as long a program as desired (within limits). Later processing may mean sending it to a printer . or making a tape out of it. It is called "downloading".

That's okay when receiving material. When you are sending, you must first "uplaod" what you want to send from a tape into your buffer, then tell your computer to send it. Note that I haven't said to disk when saving a buffer. The reason is that many modem driver programs were written before disk drives and havn't been converted as yet. Adding disk drive saving of programs will require a "patch" to the modem driver program. Generally this is at the expense of some buffer space. Writing such a patch is beyond the ability of a beginning machine code programmer.

A 2068 computer talking to another 2068 computer and exchanging programs is fine. A 2068 talking to an IBM or Apple or any other computer is not so fine unless the program being transmitted to you is either a Spectrum or a 2068 program or code for that program. Converting even a Basic program from another computer to your 2068 is not an easy task. Be advised that although you have to type out all commands for those computers they have strange abbreviations and may "tokenize" them in storage. Each line would have to be edited to get it into 2068 format. Getting a printout and then reentering is the easiest.

BULLETIN BOARDS.

Some modems allow autodialing and autoanswering. Unfortunately, with autoanswering the computer doesn't know if it's another computer calling or a human who wants to talk to you, not your computer. Using the computer to answer with a blast of its carrier tone can cause one to lose many friends quite rapidly. However, since the computer doesn't answer until the 3rd ring picking up the phone before that will avoid this. What you do when you are not there ready to answer the phone is a problem the firmware writers don't tackle. Auto dialing is only possible with touch tone phones and requires a storage area with the number and other pertinent data for making a connection. MTERM provides for up to 10 such phone numbers.

Some modems have selectable baud rates...300 bits/sec is too slow for mainframes to talk to each other. Rates as high as 9000 baud can be used. To use these higher rates one uses what is called multiplexing the signal onto a carrier signal. This uses special phone lines requiring special installation which must be rented from the phone company. This is obviously not for the average person.

DISK DRIVES

There are disk drives and there are disk drives. Most of the time a disk recorded with one system will not read on another. There are many variations that disk recording has gone through during the years and I'm sure more are still to come. What used to take a 12 inch disk to record can now be recorded on a 3.5 inch disk. Since home or personnel computer arrived on the scene relatively recently, most of them are coupled with at most 8 inch disks drives with 5.25 and 3.5 inch drives being more common. Which disk drive you chose will depend upon what most of the programs you want to use are written on.

Because Timex never got around to providing a disk drive for the 2068, 3rd party venders stepped in and filled the gap. At this writing, there are 3 different venders. In alphabetical order they are: AERCO, RAMEX and ZEBRA. Belatedly, the Portugese "Silver Avenger" is arriving on the scene with its 2068/Spectrum ROMs and a Spectrum back bus and supposedly a 3 inch disk. Yep, not 3.5 but 3 inch--again nonstandard. Since the interface for this drive will be attaching to a Spectrum back bus, it's not going to fit the 2068.

USE OF SINGLE SIDED DISKS

Although the 8 inch floppy preceded it, the 5.25 inch floppy is what most 2068 users will purchase so we will concentrate on that disk. In reality it is a thin piece of plastic coated on both sides with an iron oxide-chromium oxide magnetizable material and then packaged in a permanent paper dust jacket. Because of the close packing of the data and the fragile nature of the very thin magnetic coating, never touch the disk with your fingers through the windows on both sides of the disk. Inspection of the disk through these windows will show you that over half of the disk surface is empty and only a band 0.833 inches wide is actually used. In manufacture all disks are made single or dual density double sided. In inspecting disks a few from a lot are tested and if flaws are detected the whole lot is labeled as single sided. The 2nd side may be perfectly good, but only one side is guaranteed good.

The 2nd side may be used at your risk. A mere formatting of the disk will not indicate that it is good or bad. You only find out when you MOVE a program into a track and then reload it into the computer which is a bit too late. There is no verify for disks. One should always make two copies of a disk. Since the side guaranteed is the side with the label and since the catalogue track limits how many programs may be stored on a disk (32 with AERCO), you may only be using side 1. Two copies should be sufficient. Once you get to side 2 (less than 200k remaining), you are playing Russian roulette with a 2 chambered gun. If you don't like 50-50 odds, make more copies, 3 to 4. At least one of them should hit tracks that are perfectly good.

CARE OF FLOPPIES

Never write on a label already on a disk with a ball point pen. Use only the slightest pressure with a felt tipped pen, or better still, write out another label and paste it over the old one. As long as we are on the subject of disk care, never turn off power to the disk drive with a disk still in the system. Never turn on the power to the disk drives with the cardboard inserts still in place--it has a tendency to ruin heads. Never store disks near or on top of a TV or monitor as the degaussing of the picture tube that takes place when you turn them off creates a strong magnetic field capable of messing up disks. Large speakers can do the same. Never put disks on top of the drive itself as motors starting and stopping spell disaster. Always store your spare set some place different then your working set. Always use the jackets for the disks...there is nothing that messes them up faster than spilled coffee or soda. AND don't smoke when operating the drives.

DISC DENSITY

Depending upon the drive there are 35, 40 or 80 tracks per side of a disk. Unlike a phonograph record where a track is really a continuous spiral, these tracks are perfect circles with the end of a track running right back into the beginning of the same track. With this sort of system one needs to use some sort of track marker to mark the beginning/end of the track. Tracks are also divided into sectors, 10 sectors/track being common. We all know that you cannot use a new disk directly but first must do what is called "formatting" a disk. Formatting puts a magnetic marker at the start/end of each track. To do this, the drive uses the hard sector hole to mark the start of each track around the disk. This hole is also used to strobe the disk to see if the drive is rotating and if a disk is present. The signal indicates the start of sector 1 and the end of sector 10. Nine more sector start/end singals are written on each track and the disk stepped through all tracks. In single density format each sector will have 256 bytes (2048 bits). In dual density, each sector will contain 512 bytes (4096 bits). Quad density is achieved not with another doubling of the density of the bits in a sector but by doubling the number of tracks on a disk--usually 80 per side.

THE 5.25 INCH FLOPPY

Tracks are packed radially at 48/inch (single or dual density) or 96/inch for quad density. In either case we end up with a band 0.833 inches wide. With a 1 1/8 inch hole and evenly spacing the empty part of the disk on each side of the bands gives us an inside diameter of 2.3 inches (circumference 7.22 inches) and an outside track diameter of 4.0 (12.57 circumference). At 48 tracks per inch radially, each track has to be something less than 0.021 inches wide (the track separation distance) as there must be some space between tracks to prevent cross talk from one track to another. In quad density, the tracks have to be half

that wide.

Using the outside track (12.57 inches around) with a dual density track of 10 sectors of 512 bytes gives us 512x10x8 bits per track. Not counting space used by the sector end markers this works out to a 0.00031 inches per bit or, to put it another way, 3258 bits/inch. For comparison, a cassette tape recorder at 3 inches/sec and a baud rate of 1200 gives a bit width of 0.0025 or 400 bits/inch--an order of magnitude wider. Using the smaller inside track makes things even tighter.

At 5125 bytes per track and 40 tracks per side, that's 205,800 bytes per side at dual density. This is unformatted capacity. Depending upon the format, 18 to 33 percent of the capacity of the disk are chewed up with overhead housekeeping data--one of which is a full track devoted to the directory. Additionally each track has to have a few bytes devoted to track number and what sector to go to next. With most systems handling double sided disks and up to 4 drives, double density-double sided systems of 4 drives could have a whopping 1.64 megabytes of storage. Quad density systems a whopping 3.28 megabytes.

With that much space available, disk drives are somewhat wasteful of space. Some drives insist upon using a minimum of 1 track per program. Since a track has an effective storage of 5120 bytes, a tiny program of Basic, calling a wee program of machine code (separate save = another track) and a screen (another save of 2 tracks as over 5120 bytes) can chew up 4 tracks and be almost empty.

A big advantage of the disk drive is that it can read a track as many times as it needs to. In a tape recorder it goes by only once. Asking a disk to load a program into the computer first has the disk consult the directory for a name match which then gives it a track number. The head placement worm gear grinds out the correct number of turns to the track position while the computer waits a second for this to take place. The head is put into the read mode and starts looking for the track start signal. If it can't find it, it will stop and increment the head half a step up and then half a step down in an attempt to find the track. If not found, the appropiate error-DISK NOT READABLE will appear on the screen. If the signal is found, it waits another revolution of the disk before reading the other operating data. Again another rotation before it starts sending the actual program bytes--including a parity byte at the end of the track. NOTE: Some disk drives will get this done in 2 rotations rather than 3. The computer is also calculating parity of the incoming bytes and compares its calculated value with the last byte--if they don't agree another load is tried immediately.

DOS--DISK OPERATING SYSTEM

Disk drives just don't plug into the back of your 2068 computer through an interface and run. First of all, the ROM doesn't sup-

port the commands CAT, FORMAT, ERASE and MOVE--where have you read that term before? These commands have to be handled by the DOS--Disk Operating System. Also more instructions have to be written for turning on drive motors, finding the right track or an empty track if MOVE (the disk equivalent of SAVE) is used, and a dozen other things that a disk controller needs to function. If you are fortunate enough to have an AERCO system with CP/M not only are you using 5.25 inch disks but also 8 inch which needs a different set of operatives to work. In fact the Aerco system will read single, double and quad density disks in 3 different formats. This takes a full 8k operating system. Other systems get by with 4k but may be limited to reading and writing disk that are of only their same format.

This system has to be loaded somewhere and may take the top 4-8k of your memory--unless you have an Aerco system in which case it's in Chunk 1 of Bank 0. If you don't remember where that is, it's called the DOCK bank. Being there, it doesn't use any RAM. That extra 64k gets put to use as soon as you turn on your computer and have your disk drive interface attached. Although on the back of the computer and not under the front cover, it is wired to act like it's under the front cover. Thus, as soon as you turn your computer on, it senses a cartridge present and calls the EPROM on the interface to make all the corrections to the Bank Switching routines which at this time have already been moved to Chunk 3 of RAM and then sets itself up in Chunk 1 of the DOCK Bank. It then checks to see if the disk you are using has a BOOT program on it and automatically loads and runs that. The Boot program is short and its only function is to load another program.

Okay, you want to use your dot matrix printer as well so you have piggybacked your printer interface to the back of your disk interface. Having a Boot program ask you if you need a Printer Driver Program and if the answer is "Y" automatically loading the version of it you need is really nice. You also are assured of not putting it someplace where it might overwrite some of the DOS program. The Aerco system uses bytes 23856-23863 in the unused reserved System Variables Table.

There is one disadvantage to this setup in that when running a program Chunk 1 of Home ROM must be used thus disabling Chunk 1 of the Dock Bank. When you want the DOS again you may have to first enable the DOCK Bank with OUT 244, 1--you will find out what that means in the next chapter. That is a small price to pay for a full use of Home RAM and Disk drives.

What all happens when you MOVE (save) a program? If you want a blow by blow discription, Aerco has copies of its entire DOS disassembly available for $20. The MOVE command is similar to the SAVE command except your name MUST be followed by a 3 symbol extension. The word must is emphasized as it MUST be used. No LINE token, just a comma and the line number. The MOVE command automatically gets the Drives going. The first thing it checks

is to see if you have specified a drive in your name--it has to be a, b, c, or d followed by a ":". If not, it uses the last drive you told it to use. The 2nd thing it checks is to see if a disk is turning. The 3rd thing it checks is to see if the disk is write protected and if it is, asks if it can overwrite. The 4th thing it checks is to see if the name and extension you gave it match what is already on the disk--after all, you may be re-loading a program. If this is the case it will check the program length and see if it has enough room in the same number of tracks or it is the last loaded program. In either of these cases it will overwrite the old program. If a new program, it will next check to see if there are enough empty tracks to hold the program. If enough memory remains, it puts the name padded with blanks if necessary into the directory together with track number, moves the head to that track and writes in the program. All this in the course of a few seconds. This description is of necessity brief and only gives some of the highlights of what all is going on-the whole disassembly goes on for some 50 odd pages and makes for some great bedtime reading. You say you don't read code at bedtime? Well, that's your problem.

In the case of ERASE, the program is erased out of the directory and as such, the tracks becomes available for the next program MOVEd onto the disk that can fit the size.

Because we have to use extensions, the same name with a differ-ent extension is considered a different program. With 4 differ-ent extensions, .bas, .bin, .scr, and .dat, everything can go under the same name.

DOS code is very involved and not a project for the novice ma-chine code programmer.

CHAPTER 10

ADVANCED CONCEPTS--I/O PORTING and BANK SWITCHING

In Basic, we didn't worry too much about how doing a PRINT got a signal to the screen or how the keyboard gave the right code for a number, letter or token, or as far as that goes why pressing Caps Shift and Break at the same time stopped some programs. We also didn't care how the cassette tape recorder interfaced to the computer other than MIC goes with MIC and EAR goes with EAR. On the ZX81 or TS1000, machine code was put in the first statement of the program as a REM line so there were no machine code saves and loads. The 2068 changed that with its machine code saves, loads and merges of programs, codes and screens. The BEEP command seemed easy enough but using SOUND commands seemed unduly complicated. How the JOYSTICK worked through the sound chip was a mystery we never did really understand but the commands were easy enough.

When we got our 2040 printer we just plugged that into the back bus and all of a sudden could do LPRINT, LLIST and COPY without really worrying about buffers, ports and interfacing. Things got a little more complicated when we bought a Dot Matrix Printer. All of a sudden we had to have a printer interface but we also had to load a program called the Printer Driver. Setting up the printer driver was easy enough but sending some of the other codes for changing type faces, margins and something called linefeed was a bit difficult with word processors like Tasword II especially when we wanted to do something the program wasn't set up to do. However, once done, life again became simple and even more enjoyable. Life became even easier with the addition of a disk drive or microdrive.

The fact is that all the complicated work of switching was done for us by the firmware that came with the equipment. Since these programs were in machine code we didn't bother to look at them. If we had, we would have found the code to be a lot more difficult to understand than code not having all those INs and OUTs. Even with the aid of a port map we have difficulty understanding how the same port can be used for several devices, not at the same time, but one after the other.

The truth of the matter is that the screen, keybord, sound and joysticks, cassette MIC and EAR, printers, disk drives, modem and anything else we may wish to add are all addressed through ports and ports only. This includes the topic of bank switching which is also done through ports.

You have also probably heard of parallel, serial, centronics,

232C and IEEE ports, just to mention a few. These are names of different ways and standardizations of the use of a port or ports.

What handes all this port switching and porting?

THE SCLD

SCLD stands for "Standard Cell Logic Device". It is sometimes considered the workhorse of the computer. It handles:

1. Bank Switching--the Horizontal Select register.
2. Z-80 clock.
3. Video Display.
   Display Timing.
   DMA Display File access
   Attribute control
4. Interrupt generator.
5. BEEP output.
6. Sound chip control, joystick reading.
7. Cassette I/O.
8. Any other porting to other peripheral device.

This chip is manufactured exclusively for Sinclair--well, at least partly. What the chip manufacturer does is make a chip of standard groupings of gates, flip-flops, inverters, buffers, internal memory, etc. and leaves them unconnected. Then, when a computer manufacturer like Sinclair finally finalizes its design, the wiring diagram of the SCLD is designed and the wiring added to the chips. All micros use the same standard cell logic chip (or the equivalent chips) but with different internal wiring. Thus Apples handle switching differently from IBMs which is still different from Timex etc. If you open your 2068 (not recommended as it voids your warranty), you will find the SCLD chip bristling with contacts on all sides--68 to be exact. You can't miss it.

There is nothing magic inside the SCLD. Its role could be undertaken by a collection of logic chips just as well, but its use greatly compacts things. One reason why the 2068 circuit board looks relatively simple and uncluttered is the fact that the SCLD chip replaces 15 to 20 normal ICs. The SCLD is attached to both the data and address buses, and it also receives most of the Z80's control signals. Using this information, it generates more specific signals of its own which control the whole computer. Because of its small size, not all the components can be located inside the SCLD so it is surrounded with the more bulky components it needs. For an example, the Z80 clock signal requires a quartz crystal and a trimmer to get it exactly the right frequency. These are too bulky to include inside the package. Other connections include the EAR and MIC sockets for cassette operations, the output to activate the sound chip, extension memory banks and keyboard. Additionally, the SCLD reads the Display File along with the Attribute File in memory and gener-

ates the signal for the screen.

Since the SCLD works independently of the CPU to generate the screen signal, it must read memory at the same time the CPU may need it. To facilitate this, chips known as multiplexers are used (74LS157). They have two inputs for each bit of the address bus, one from the Z80 and one from the SCLD. A control signal from the SCLD determines which of these addresses is passed onto the RAM chips.

The sound chip, (AY-3 8912) also connects to the data bus, but since this chip is port addressed, access to its registers is via the SCLD. The sound chip also interfaces to the joysticks.

Signals for or from anything else that you have to address through a port also goes through the SCLD.

BANK SWITCHING--AN OVERVIEW

Bank switching of memory is done in chunks of 8k at a time using a BYTE that has the correct BITS zero for all chunks to be enabled for that bank (active low format). The HOME bank is the bank of default and will have all those chunks enabled that are not enabled elsewhere.

What we are allowed to enable will depend upon the type of program we are writing. Although plugin cartridges containing programs have to be ROMs or EPROMs we can also use the DOCK bank as a RAM. It can plug in through the front door or on the back bus. In the case of a RAM we have an extended bank of memory in the DOCK that we can read or write to.

Since memory, either RAM or ROM is enabled in chunks, we don't have to enable all of a bank at one time if we don't want to. In the case of a CARTRIDGE written in Basic (with or without some machine code), we have to keep CHUNK 3 in HOME RAM because it contains the necessary routines for bank switching and the machine stack. Also, since we are using the ROM routines to interpret our Basic, we need CHUNKS 0 and 1 with CHUNK 2 for our screen display. (Having just read about the SCLD we note tht even in an LROS program we have to supply the same type of pixel mapped display for the SCLD to interpret to generate a signal for the screen). AROS thus is limited to the use of the top 4 chunks of memory only. LROS probably is restricted from using CHUNKS 2 and 3 with part of chunk 0 being needed for interrupt handling routines.

With both AROS and LROS we need some RAM somewhere to write to while our program is running to store variables in. In the case of AROS, the DOCK bank is just activated long enough to write the next program line into the ARSBUF and then the HOME bank is completely activated again to RUN the line. A somewhat similar situation, without deactivation of the complete LROS BANK must be provided in LROS situations.

Since we can have as many banks of memory as we want, (or our power supply can handle on refresh), but since only 8 chunks can be active at any one time, it's the chunks that are in control, not the banks. Granted, we can designate the active chunk to be from whatever bank we want whenever we desire, but it's still only one Chunk 0, one chunk 1, etc.

If we are in Basic we have to call the changes through what is called the Horizontal Select Register (addressed through port 244) located in the SCLD chip. Since this register is port addressed the only way we can send it information is with an OUT statement. We receive information back with an IN statement. This does not mean we can get along without the bank switching routines in Chunk 3 as they are used by the ROM to find out which chunk a certain address is in. Remember that all banks have the same 0 to 65535 address numbers so we have to know which, say, address 12345 we want, if chunk 1 is active in bank 0, it's bank 0 we want, not bank 255. The horizontal select register finds which chunk is active for us in what bank and turns that bank on for us. This is what OUT 244, 1 does for us when we wanted to use our Aerco disk drives back in Chpater 9--it found out that we specified Chunk 1 in Bank 0 and so gave us that...the DOS program.

Writing in machine code we can of course call the fuctions of the dispatcher directly. However we have to know how to set them up on the machine stack.

THE FUNCTION DISPATCHER

RAM RESIDENT CODE (25088-26688)

Better known as the Function Dispatcher and the Bank Switching routines. As we said, it is a disgrace to Timex as it is full of errors. DO NOT USE it until you run the following program to make the necessary corrections. The Function Dispatcher is the first of several routines that permanently reside in the top part of Chunk 0 of the Extended ROM. Upon start up the computer tranfers these routines to chunk 3, addresses 25088 to 26646 of the Home RAM. While it was in ROM we couldn't do much with it unless we decided to write a new Extended ROM chip. Once it's in RAM we can change it. This program must be run while in Video Mode 0 (the normal one screen mode), while these routines are in their low addresses. ALSO NOTE: If you have an Aerco Disk Drive don't bother entering and running this program as the corrections have already been made for you in the disk startup program.

```
65000  F3              DI
65001  21,30,FE        LD HL, DATA B  Fix GET STATUS
```

```
65004  11,0A,64          LD DE, 25610
65007  01,09,00          LD BC, 9        For 9 bytes
65010  ED,B0             LDIR
65012  11,30,64          LD DE, 25648
65015  01,1A,00          LD BC, 25       For 25 bytes
65018  ED,B0             LDIR
65020  3E,D5             LD A, 213       Fix PUT WORD
65022  32,40,63          LD (25408), A
65025  11,50,63          LD DE, 25424
65028  01,06,00          LD BC, 6        For 6 bytes
65031  ED,B0             LDIR
65033  3E,09             LD A, 9         Fix CALL BANK
65035  32,10,66          LD (26128), A
65038  AF                XOR A           Fix BANK ENABLE
65039  32,99,64          LD (25753), A   and RESTORE STATUS
65042  3E,F3             LD A, 243
65044  32,9D,64          LD (25757), A
65047  32,4A,65          LD (25930), A
65050  3E,FB             LD A, 253
65052  32,1C,65          LD (25884), A
65055  32,70,65          LD (25968), A
65058  FB                EI
65059  C9                RET
65060  28,24,FE,FF,28,37,A7,28,27,       DATA B
65069  0E,FF,DB,FF,E6,80,28,12,18,
       08,0E,FF,DB,FF,E6,80,20,08,
       DB,F4,2F,18,02,DB,F4,4F,
65095  C1,D1,73,23,72,2B
```

The errors you have just corrected stay corrected no matter if you are in single screen mode or double screen mode. The trouble comes in the introduction of new errors each and every time you transfer the code from 25088 to 63936 or back down again. The following POKEs correct the 3 errors introduced on switching program locations:

DOWN	UP
POKE 25870,205	POKE 64718,205
POKE 25871,92	POKE 64719,28
POKE 25872,99	POKE 64720,251
POKE 25878,205	POKE 64726,205
POKE 25879,92	POKE 64727,28
POKE 25880,99	POKE 64728,251
POKE 26447,205	POKE 65295,205
POKE 26448,232	POKE 65296,168
POKE 26449,102	POKE 65297,254

Why do these routines have to be moved when in dual screen modes? If you consult your Memory Map of the home bank and look at the right (dual screen) map, you will note that Display File 2 occupies the space where the Machine Stack and the Ram Resident Code normally reside. If you also look at the top part of the map, you will see that these two have been moved up to the very top of RAM with the User Defined Graphics (UDG) being moved

just below them. Additionally, the remaining routines (Machine Code Variables, ARSBUF and CHANS) are moved up a bit as well. The start of the Basic program is now at 31510 rather than 26710.

AROS and BANK SWITCHING

It was Timex's great idea to build a computer with the ability to addresss more than 64k of RAM (although not more than 64k at any one time) and still use the ROM to operate plugin cartridges. To do this it developed a system of activating chunks of 8k RAM or ROM at a time. Since AROS cartridges can be written in Basic, Chunks 0 and 1 which contain the main ROM, Chunk 2 which contains the Display File and Chunk 3 which has the Bank Switching can't be used leaving only the top 4 chunks (32768 and above) for use. The AROS routine does not support having the Bank Switching Routines in high memory so you are restricted to only the single screen mode. That means programs no longer than 32k. Since the cartridge by necessity has to be ROM memory (non-volatile) more memory has to be made available in the home bank to allow for the VARiable table. What really happens is that the AROS chunks are only activated to get a Basic line and copy it to the ARSBUF, along with a data line if a READ statement. Them memory is switched back to the home bank where address 32553 is designated as the start of the VARS table and the line is executed. The process then repeats. Should you switch to machine code by using a USER function, this code must be limited to the SAME Chunk. If you cross a chunk boundary, the computer will think that the code continues in the next chunk of the home RAM. Machine code routines are the only time the computer stays in the dock bank for any length of time.

CARTRIDGE INITIALIZATION

When you plug in a cartridge, your computer should be off. As you turn it on, it begins to set itself up and one of the things it does is check for a cartridge. It checks both Chunk 0 and 4 of the dock for an LROS or an AROS cartridge respectively. The first 8 bytes of Chunk 4 must contain the following information.

32768 Language type 1 = Basic (and code)
2 = code only
32769 Cartridge type 2 = AROS
32770/1 Starting address of Basic (LSB/MSB)
32772 Memory Chunks active (low) by Bit
00001111 = Chunks 4 to 7
32773 Auto start 0= no, 1 = yes.
32774/5 Number of bytes of RAM to be reserved for machine code (LSB/MSB). This saves space in the Home RAM which must first be written there.

ERRORS IN AROS ROUTINE

The following ommissions and malfunctions have been found:

USR function sometimes gives wrong address.

If the variable specified in a FOR statement is already beyond its final limit the program may not find the line after the next statement. This can be avoided merely by setting the variable within the FOR limits.

The overhead bytes for an LROS cartridge are:

0 Not used
1 Cartridge type 1 = LROS
2/3 Starting address (LSB/MSB) to be jumped to after initialization is complete.
4 Memory Chunks active (low) by bit as in AROS.

In addition, Interrupt mode 1 is specified by the computer for LROS cartridges. Instructions written at 56(38H) must handle these interrupts. Instructions at 102(66H) must handle non-maskable interrupts. Should you want to use RST 40 (floating point calculator) it must be enabled with:

```
LD HL, 23698 (5C92H) MEM BOT
LD (23698), HL
```

Should you use an extended video mode, the routine doesn't check to see if there is sufficient memory nor is RAMtop set.

Should Chunk 3 be specified as active for LROS, when the memory chunk specification is written to port 244 (F4H) by the Bank Enable code, execution will continue from that point in Chunk 3 of the Dock Bank with the stack pointer addressing ROM.

Since you are in complete control with LROS, what you do is your own business. However, remember that somewhere you have to put your VARS file and other things that constantly change--they certainly cannot be located in non-volatle Cartridge memory.

If you want to do AROS from your LROS cartridge you also have to tell your computer to check for an AROS application as once it finds an LROS it will not continue to check for an AROS. Also note that Chunk 0 of the Dock and the EXROM are mutually exclusive. If in LROS you cannot get to EXROM.

LROS Display File. Since the SCLD chip still is active you still must use a Display File just as you do if no cartridge were present--or a double screen mode if you like. A relocatable Mode 0 (normal 32 column screen) routine for printing to the screen, including all screen tokens is given in Appendix B. I use this routine as a class exercise to introduce students to the operation and use of the various registers and a hands on study of a machine code routine.

CARTRIDGE SETUP

Several things happen when the computer has sensed the presence of a cartridge, one of the first of which is to set up the Cartridge Flag (23750). Setting Bit 7 tells the computer from now on that a cartridge is present. Many of the routines have a cartridge check written into them which checks this flag and then moves accordingly. Bits 3 and 4 contain flags for Next Line and Data Line, Bit 1 a flag for use of upper or lower screen. Also addresses:

23751/2	Store the current data line address if any.
23753/4	Store the length of the data line.
23755	The stream number in use.
23748/9	The address of the next program line.

Further information is stored in the System Configuration Table which is pointed to by Addresses 23740/1.

The full set of routines in the RAM Resident Code are:

NAME	ADDRESS	(IN HEX)
Function Dispatcher	6200	F9C0
INT (for RST 38H)	62AE	FA6E
INT NMI	6307	FAC7
GET WORD	6316	FAD6
PUT WORD	633B	FAFB
WRITE BANK STATUS REG	635C	FB1C
READ BANK STATUS REG	63AD	FB6D
GET BANK STATUS	6405	FBC5
GET CHUNK	644D	FC0D
GET BANK NUMBER	645E	FC1E
BANK ENABLE	6499	FC59
SAVE STATUS	651E	FCDE
RESTORE STATUS	654A	FD0A
GOTO BANK	6572	FD32
CALL BANK	65D0	FD90
XFER BANK	6722	FEE2
GO EX	6815	FFD5

The Function Dispatcher and Bank Switching Routines cannot be used from Basic since it cannot be set up in Basic. It can only be used from machine code. Looking at some of the routines in the Bank Switching Routines we get some idea of what it all can do. The CALL address is listed along with the routine. Both low and high locations are given.

GET WORD--Returns in HL the word from the address in HL in the Bank specified in B.

PUT WORD--Writes the word in DE to the address in HL in the Bank specified by B.

GET BANK STATUS--Returns in A the bank number controlling the address in HL. Literally answers the

question of what Bank hold the active chunk of this address.

BANK ENABLE--With Bank in B and Memory Selection (Chunk # low) in C, enables that bank through the horizontal select register.

GOTO BANK--Push Target Address, then Bank #/memory select on to the stack. Then call GOTO BANK. Works like Basic GOTO without a return to call bank.

CALL BANK--As above but with a return to CALLing BANK and ADDRESS. Set up with target address, then Bank #/ Chunk # low, then # of parameters to be passed IN, then number of parameters to be passed OUT. Parameters IN and OUT are 2 bytes each--use zero if none. Then CALL CALL BANK.

XFER BYTES--Copies n byte(s) from specified source to specified destination in ascending or descending order. Can be in different banks but cannot cross chunk boundaries. Push as follows:

Source Bank/Destination Bank
Source Address
Destination Address
Length (2 bytes)
0/ Direction 0 = ascending, 255 = descending.

FUNCTION DISPATCHER--Call Basic routines from Home ROM by use of a Service Code Number.

1. Set up memory and stacks (machine and calculator as if invoking service directly.
2. Push parameters for Dispatcher on stack.
3. Set up registers as if invoking directly especially the Interpret Flag (Bit 7 of FLAGS, 0 = syntax check, 1 = execute.)
4. Use Parameters IN and OUT if using CALL. Don't use if Jump.
5. Set Bit 15 if a GOTO. Don't if CALL. Other byte is service code.
6. Make CALL or GOTO BANK.
6. Pop off only the number of parameters of returned bytes.

From this discription, one can see that using the function dispatcher requires extensive knowledge of the routines being called. This gets compounded when dealing with the floating point calculator.

Here is the list of Service Codes. Some with an asterick are supposed to work but have been found to have problems.

CODE	SERVICE	SETUP and RETURN
00*	SAVE data	
01*	LOAD data	
02*	READ BIT	
03*	READ EDGE	
04*	SAVE, LOAD, VERIFY, MERGE	
05*	LOAD	
06*	MERGE	
07*	SAVE	
08*	CHANGE VIDEO	
09*	WRITE BORDER COLOR	
10-13	Not assigned or assignment unknown.	
14	GET STATUS	Returns Chunk in C for Bank in B.
15	GET BANK NUMBER	Returns Bank in A for address in HL.
16*	BANK ENABLE	See routine page 177.
17*	GOTO BANK	See routine page 177.
18*	CALL BANK	See routine page 177.
19*	XFER BYTES	See routine page 177.
20-24	Not assigned or assignment unknown.	
25	UPDATE KEYBOARD	Answer in Last K.
26	PARP	Generates DE+1 cycles of tone 8N+236 to 8N+246 T states long. HL = N.
27	BEEP COMMAND	Process parameters on calculator stack. Exit via PARP.
28	K DUMP (COPY)	Dumps Display File 1 to printer.
29	SEND TV	Character code in A to screen/printer.
30	SET AT	B = Line #, C = Column #. Sets AT.
31	ATTR BYTE	Address in HL. Sets byte Attr T, Mask T and P flag.

32	R ATTS	Permanent ATTR INFO to TEMP ATTR VARiable.
33	CLLHS	CLEAR lower screen in Display File 1 only.
34	CLS	CLEAR entire screen. Display File 1 only.
35	DUMP PR	PRINT/Clear entire buffer.
36	PR SCAN	Pixel Address in HL. Number of scans remaining in B (1-8) SENDs 32 bytes of pixels to printer.
37	DESLUG	Address in HL. Remove number slugs from Edit line buffer.
38	K NEW	NEW command.
39	INITialize	A = 0 for power on, FF for NEW. DE = Max RAM address.
40	IN CHannel	Input CHARacter to A from currently selected Channel. NC of no input.
41	SELECT	Stream # in A. Selects channel.
42	INSERT BC Bytes	Insert BC bytes before byte whose address is in HL. Copies up all from HL to STKEND. Updates all affected system variables. Returns BC = 0; DE = address of last byte inserted space; HL address of byte before spaces.
43	RESET	Reset calculator stack. STKEND = STKBOT. MEM=MEMBOT.
44	CLOSE # command	Channel # on calculator stack.
45	CL CHANnel	BC = value from streams. Closes that channel.
46	OPEN	Channel and device spec on calculator stack.
47	OPen CHANnel	DE = pointer into STRMS based on Channel #. Device # on calculator stack.
48*	CAT command	Needs Disk Drive ROM.
49*	ERASE command	Needs Disk Drive ROM.

50*	FORMAT command	Needs Disk Drive ROM.
51*	MOVE command	Needs Disk Drive ROM.
52	FLASH A	Flash CHAR in A to lower screen (cursor flash).
53	FIND Line	Number in HL. Returns address in HL with Z if found. Else next line number address (or VARS if none higher).
54	SUB LINe	D = statement #, E = 0 (or keyword with D = 0) HL = line. Returns HL and CH ADD pointing at address one before keyword if Dth statement found and Z set. If match on E, HL and CH ADD pointing to address of E with NZ, NC set and D decremented by number of statements looked at. If no match on E, returns NZ, C and HL and CH ADD at end-of-line byte.
55	RECord LENgth	HL at address of line, string or numeric variable or array. Returns LEN in BC. DE = HL + BC = next line or variable.
56	DELete RECord	HL = address of record start. BC = length of record. Deletes from VARS or Program (if line). Recovers spaces and adjusts all affected system variables.
57	PUT BC	BC = number in binary. Converts to ASCII code and outputs to currently active channel. If less than zero, prints 0.
58	SYNTAX	Check syntax of command or program line in E Line. ERR 255 if none, else ERR # - 1.
59	EXECUTE	Executes commands from E LINE (Direct mode).
60*	FOR	FOR command--complex use of calculator stack.
61	STOP command	RESTART 8 with error # 9.
62*	NEXT command	Complex use of calculator stack.
63*	READ command	Complex use of calculator stack.
64*	DATA command	Complex use of calculator stack.
65	RESTore BC	BC = Line number to restore to.

66	RANDomize command	Set SEED for RAND number generator. Parameters on calculator stack loaded to SEED. If zero, FRAMES loaded to seed.
67	CONTinue	OLDPPC and OSPPC to NEWPPC and NSPPC. Startup at NEWPPC and NSPPC.
68	JUMP(GOTO)	Line # in calculator stack loaded to NEWPPC with NSPPC set to zero. Just sets up GOTO, not runs.
69	FIX U1	Number on calculator stack to A. RST 8 with ERR B for too big a number. Number is unsigned.
70	FIX U	Floating point number on calculator stack to BC (unsigned). ERR B if out of range.
71	CLEAR command	Parameter on calculator stack to BC for use in CLR BC.
72	CLR BC	BC = New RAMTOP. Deletes VARS, CLEARS screen and calculator stack as in Basic CLEAR.
73	GOSUB command	With parameters of GOSUB on calculator stack, puts present line # (LSB/MSB) on machine stack followed by statement number. Then CALLS JUMP(68) to process GOSUB parameters and returns. Does not do GOSUB--just sets up.
74	CHecK SiZe	Checks to see if BC + 80 bytes left between STKEND and RAMTOP. (The 80 is "left over" from T/S 1000 code where machine stack was at top of RAM.) ERR 4 if not enough room. Used when adding a new variable to VARS table or a new line to program or bringing down an EDIT Line.
75	RETURN command	Retrieves GOSUB block from machine stack and loads data to NEWPPC and NSPPC and returns. ERR 7 if line number MSB = 62 (end of stack marker).
76	PAUSE command	Parameters on calculator stack to BC. Then waits BC frames or key pressed. (Uses HALT so interrupts must be enabled.)
77	BREAK?	Reads BREAK key. NC if pressed and ON

		ERROR not active.
78*	DEF	Define function uses complex calculator stack operations.
79	K LPRint	Selects Channel 3 and processes items in LPRINT statement or output via WRCH.
80	K PRINT	Selects Channel 2 and processes items in PRINT statement for output via WRCH.
81	Print SEQuence	Address in CH ADD. Code used by K LPR and K PRIT to process output data and controls in Basic statement.
82*	INPUT command	Selects Channel 1 and processes I/O for KEYboard/lower screen using a buffer at WORKSP for input.
83	Input SEQuence	Line Address in CH ADD. Code used by INPUT to process input items and controls in Basic statement.
84	NOT KB?	Returns Z if current channel is keyboard/lower screen (device specification K).
85	COLOR	D = 0-9 Color desired. Set C for INK. NC for PAPER. Adjusts ATTR-T, MASK-T and P FLAG for color change. ERR K if D invalid.
86	HIFLASH	D = 0, 1 or 8. Set Carry for flash. NC for Bright. Adjusts ATTR-T, MASK-T and P FLAG for FLASH or BRIGHT. ERR K if D invalid.
87	SCRaMBLe	B = Y, C = X. Returns to HL the Display File 1 address with A holding the pixel of the address--reading left to right. ERR B if Y > 175.
88	PLOT command	Process X/Y parameters on calculator stack to BC. Ready for plotting pixel via PLOT BC.
89	PLOT BC	B = Y, C = X. P FLAG set for INVERSE and OVER. Sets pixel, updates attribute and sets COORD = BC.
90	GET XY	Top # on calculator stack will go to B, sign in D. Next # on calculator stack to C, Sign in E (+1 or -1).

91*	CIRCLE command	Calculates successive plot positions from parameters in Basic statement. Uses complex calculator operations.
92*	DRAW command	Calculates successive plot positions from parameters in Basic statement. Uses complex calculator operations.
93*	DRAW Line	Parameters (X,Y) on calculator stack. Plots straight line from current position (COORD) to X,Y. Uses complex calculator operations.
94*	EXPRessioN	CH ADD = Line address. Evaluates expression in Basic line putting value on calculator stack.
95	Find SCReeN	Line and column positions on calculator stack. Will try to find ASCII character. BC = 0 if not found, 1 if found. DE = address of code in Character table if found.
96	Find ATTRibute	X,Y on calculator stack. ATTR in A for that pixel position.
97	RND function	Puts pseudo-random number on calculator stack using SEED. ERROR IN THIS CALL WILL CAUSE MALFUNCTION. DO NOT USE.
98	Find PI	PI to calculator stack.
99	Find INKeY	Scans keyboard and puts CHAR code in WORKSP if detected. Pushes registers AEDCB onto calculator stack. BC = 0 if no input, 1 if CHAR. DE = address of CHAR code byte.
100*	FIND Number	Find Variable. CH ADD = address of VARiable. Searches VARS table for match. Flags Bit 6 = 1 set for numeric, 0 for string. Also used to find parameters for User Defined Functions.
101	PuSH STRing	Clears Bit 6 of FLAGS. Pushes registers AEDCB on calculator stack adjusting STKNXT upwards. DE = address of string; BC = LEN of string.
102	PUT AEDCB	As with Push String but no change to Bit 6 of FLAGS.
103*	LET command	Processes existing or creates new vari-

ables. Uses complex calculator stack operations.

104 POP STRing — Pops STKNEXT-1 to STKNEXT-5 to registers BCDEA, clearing calculator stack down to STKNEXT-5.

105* DIM statement — Creats numeric or string arrays or resets old arrays. Complex calculator stack operations.

106 STack Unsigned Number — CH ADD = address of first number (which is in A, as a number, decimal point token or binary token). Processes number to floating point number on calculator stack (converts ASCII string of numbers into the numbers of a slug. Does not do slug insertion).

107 STacK A — A = unsigned integer of 1 byte. Loads A to C, sets B to 0 and executes STK BC. Leaves number on calculator stack.

108 STacK BC — BC = unsigned 2 byte number. Places on top of calculator stack.

109 IN INTeger — CH ADD = address of 1st number. A holds 1st number as ASCII code. Converts string of numbers into an unsigned floating point number on calculator stack. Terminates when non-digit found.

110 FP2BC — Pops top of calculator stack to INT in BC, rounded to nearest number. NZ if value negative. C if number greater than 65535.

111 FP2A — Pops top of calculator stack to INT in A (rounded to nearest number). NZ if value negative, C if number greater than 255.

112 OUTPUT — Number on top of stack converted to ASCII code and written via WRCH to current channel output.

The following routines all use the floatng point calulator. Full explanations are too complex to detail fully.

113 SUBtract — Subtracts top FP number (DE) from 2nd from top (HL) leaving answer on stack top.

114	ADD	Adds two top numbers on calculator stack leaving answer on top of stack.
115	MULTiply	Integer multiply (HL)*(DE). Carry set if overflow.
116	TIMES	Floating Point multiply two top numbers on the calculator stack.
117	DIVIDE	Floating point divides top number on calculator stack into 2nd number on stack leaving answer on stack.
118	TRUNCate	Truncate top FP number to zero leaving integer.
119	FLOAT	HL = address of start of slug. Puts on stack.
120	INTeger DIVide	Replaces top 2 numbers (X,Y) on calculator stack with X Mod Y and INT (X/Y). DE and HL point to stack addresses.
121	INTeger	Top of calculator stack replaced with INT value. HL set to this address.
122	EXPonent	Top number x, on calculator stack replaced with EXP (x). HL = address of stack pointer.
123	LN	Top number on calculator stack replaced with natural logarithm. HL points to it.
124	ANGLE	Top number on calculator stack replaced by results of the calculation of Y from SIN # = SIN (PI/2*Y).
125	COSine	Top number on calculator stack replaced by its COSINE.
126	SINe	Top number on calculator stack replaced by its SINE.
127	TANgent	Top number on calculator stack replaced by its TANGENT.
128	ATN	Top number on calculator stack replaced by its inverse tangent. (ARC TANGENT).
129	ARS	Top number on calculator stack replaced by its inverse sine. (ARC SINE).
130	ACS	Top number on calculator stack replaced

		by its inverse cosine. (ARC COSINE).
131	ROOT	Top number on calculator stack replaced by its square root.
132	TO THE	Y in top of calculator stack, X in 2nd position. Replaced by X^Y.
133	ReaD CHaracter	Wait for character from currently selected channel (Calls IN CH) and puts in A.
134	SEND CHaracter	A = ASCII code for Character. Writes it using current channel.
135	WRite CHaracter	A = ASCII code for Character. RESTART 16 (to screen).
136	K SCAN	Scan keyboard for key being depressed if any. State returned in A.
137	Pointer LeFT	Cursor moved 1 column left for selected device. S POSN, SPOSN L or P POSN updated (screen, lower screen or printer respectively).
138	Pointer RighT	Outputs a space to currently selected device and updates as above.
139	Pointer NewLine	End of Line. Sets current position to start of next line if screen, or outputs printer buffer if printer.
140	PUT MESsage	Outputs message to currently selected device. DE points to start of Message Table. A set for message number (0 upwards). First byte of table and last byte of each message must have Bit 7 set.
141	K CLS	CLS command. Executes both CLS and CLLHS.
142	SCRolL	Scrolls entire screen (top) up 1 line.
143	Find PoiNT	POINT function. X and Y on calculator stack. Processes to BC. Returns unsigned 0 or 1 to stack indicating state of pixel.
144	DRAW LiNe	B = Y, C = X. Else same as DRAW L(93).
145	PUT LiNe	Output line number as 4 digits to currently selected channel (right justi-

fied and spaced if necessary). HL = address of MSB of line number.

Some of these routines are not going to be too useful as they call for the interpretation of a Basic line while being in machine code. Others require entries to already be made to the calculator stack and then processing these entries. Some just set up routines, others having to be called to execute them. Many of the more complex routines result in multiple set up and use of numbers processed on the calculator stack. This is especially true of the draw graphics (line, circle, etc.) routines, where they are done a pixel at a time across the screen.

You will also note that most of the routines are pieces of Basic so your code will resemble machine code written in a Basic format. This may or may not be the fastest way to do things, only you can decide that. At least you will know some of the setup for Basic type routines and even if you don't use the function dispatcher you will know how to set up some of the registers to call the routines directly. The actual address called can be found in the "2068 ROM Manuscript" previously mentioned.

Routines 0-9 are in Extended ROM and although assigned, no data on setup is given. This is because these routines experienced some difficulties. With proper setup they should work but I'm afraid you will have to do your own "woodshedding" on these and maybe some of the others with asterisks.

HORIZONTAL SELECT REGISTER

BANK SWITCHING

Bank switching is done in chunks. Each Bank of 64k is divided into eight Chunks of 8k each--labeled from 0 to 7. The HOME bank is the bank of default--that is, its chunks are enabled if no other bank has already had that chunk enabled. The next lowest priority bank is the Dock or Cartridge bank. All other banks have higher priorities.

Switching is done with the aid of a device called the Horizontal Select Register. Whenever we use the number 244 as a port number we are using the Horizontal Select Register. Setting up this register is done through the following ports:

DKSPT 244(F4H) DocK Horizontal Select PorT. Which chunks of DOCK bank are Enabled. Chunks enabled are by Bit number-- Bit 0 "on" = Chunk 0 of dock bank enabled.

BDATPT 252(FCH) Expansion Bank DATa PorT. The data you want to store.

BCMDPT 253(FDH) Expansion Bank CoMmanD (address) PorT.

HREXPT 255(FFH) Home ROM EXtension select PorT. Setting Bit 7 gets the Extended ROM in Chunk 0 of the DOCK bank (if no other Bank has enabled Chunk 0).

There are also 5 control registers to hold data for the Horizontal Select Register. These are:

HOLD	Temporary hold register
ABN	Assigned Bank number (one per expansion port).
BNA	Bank Number accessed register.
HS	Expansion bank Horizontal Select Register (one for each expansion bank).
Status	Status Nybble.

Bit 0 Set to 0 if bank used an IRQ interrupt.
Bit 1 Not used.
Bit 2 Set to 0 if bank is responding to memory read/write.
Bit 3 Not used.

These registers cannot be addressed directly but must be addressed through ports 253 (BDATPT) and 252 (BCMDPT). They also must be used to READ the registers if that is desired. OUT obviously WRITEs, IN obviously READs.

ENABLING THE EXROM.

This is quite simple to do--all we have to do is set bit 7 in an OUT 255 command. Unfortunately, PORT 255 just doesn't control the switching in and out of the EXROM. Each bit, or a group of bits controls something. They are:

Bit 0	Enable D-File 2 (Dual screen modes).
Bit 1	Enable high color resolution mode.
Bit 2	Enable 64 column display (with Bit 0).
Bit 3, 4 and 5	Paper color for 64 column mode.
Bit 6	Disable keyboard interrupts.
Bit 7	Enable EXROM.

Looking at all of these we are in good shape if we are NOT in the Dual Screen mode. So all we have to do is OUT 255, 128 (Bit 7 on, rest off). Getting back after we are done with the EXROM is done with OUT 255, 0 (all bits off). If we happen to be in Dual Screen mode, we have to make sure that all the bits that are on stay on as we enable the EXROM and again when we Disable the EXROM. What this actually does is switch the EXROM to Chunk 0 of the Dock Bank and run it as if there. THUS EXROM and CHUNK 0 of the DOCK are mutually exclusive.

To enable:
```
IN A, (255)
SET 7, A
OUT (255), A
XOR A
OUT (244), A
```

To Disable:
```
IN A, (255)
RES 7, A
OUT (255), A
XOR A
OUT (244), A
```

PORT 254

Port 254 is quite busy as well. On the input side, Bits 0-4 are used to read the keyboard signal, with Bit 6 reading the Cassette Tape Signal. On the output side (writing) Bits 0-2 set the Border Color, Bit 3 the cassette Tape output signal, and Bit 4 toggles the speaker (BEEP).

ENABLING A BANK OF EXTENDED MEMORY

Turning on and off the EXROM was quite easy as the Horizontal Select Register didn't have to be used except to turn the HOME ROM CHUNK 0 back on. But, suppose that you have some ROM attached to a peripheral device that you want to operate in an extension memory bank. Also suppose that it's 8k long and is memory mapped from 0 to 8k addresses and you wish to put it in Bank 1. With that information, it has to be Chunk 0 of Bank 1.

We have to work with the BCMDPT and the BDATPT registers. These out commands are as follows:

OUT 253 (BCMDPT)		OUT 252 (BDATPT)
0	Write command Type I	14 Reset controller--prepare to initialize.
		13 Start Interrupt REG sequence
		11 Initalization done-move to next.
		7 Reset Interrupt Flag.
1	Write command Type II	14 Dump Hold to ABN
		13 Dump Hold to BNA
		11 Dump Hold to HS
		7 Not used.
2	Write hold low nybble.	
3	Write hold high nybble.	

There are 4 Type I commands with the numbers 14, 13, 11 and 7. There are 3 Type II commands with the same numbers (7 is not used). From the BCMDPT commands we see that we are dealing with nybbles in all these cases. In fact, the Type I and Type II commands of BDATPT are what we call ACTIVE when LOW nybbles. 7 is equivalent to Bit 3 low, 11 is Bit 2 low, 13 is Bit 1 low and 14 is Bit 0 low.

The general procedure is to get the correct numbers to the Hold register and then DUMP them to the other registers (ABN, BNA, and HS). It might take a few steps to accomplish each move.

```
OUT 253, 0     These 2 steps start initialization by reset-
OUT 252, 14     ting the controller.

OUT 253, 2     Since HOLD is now zero, all we have to do is
OUT 252, 1       write to the low nybble our BANK 1 (it will
                 also be our Chunk 0 on number).
```

```
OUT 253, 3     Just to make sure, let's write the Hi nybble
OUT 252, 0       as well.
OUT 253, 1     Move HOLD to BNA
OUT 252, 13
OUT 252, 11    HOLD to HS as well.
```

We are now into Bank 1, Chunk 0. We will stay there until we switch back with another series of OUT commands.

This routine is doing a bank switch directly. There is another way of doing it using the Bank Switching routines which is somewhat simpler.

Before leaving the topic of bank switching, we can also READ the status of the Horizontal Select Register with the following in commands.

IN 253 (BCMDPT)
- 0 read status--as per status nybble.
- 1 not used.
- 2 Read HS low nybble
- 3 Read HS high nybble

These commands also can be handled through the Bank Switching routine called GET STATUS.

ENABLING CHUNKS IN THE DOCK BANK

The DOCK (Cartridge) Bank is the Bank of Preferred RAM or ROM additions. It is BANK 0. To activate it you have to set BNA to 0 and then do an OUT 244, with the Chunks enabled in that Bank. Keep in mind AROs or LROS type cartridges--OUT 244, 15 will turn on all top 32k of the cartridge. Fortunately, when turning on your computer, one of the first things it does is check to see if you have plugged in a cartridge. If you have, it reads the first 8 bytes at the start of the 32k mark and automatically sets itself up to work from the cartridge port.

ENABLING MORE CHUNKS OF THE EXROM.

At present, only chunk 0 can be activated. This contains the EXROM. Only a hardware change will allow you to use higher chunks in Bank 254 as address lines 13, 14 and 15 are not interpreted. Another 74LS32 chip must be added to handle that. The implementation of this is available thanks to John Olliger. His article appeared in Syncware News, Vol. 2, #3,Jan-Feb 1985. An advantage to doing this is that the EXROM bank is native to the computer whereas other banks are not. The Disadvantage is that the 254 bank cannot be used at the same time as the Dock (0) bank. However, with 253 other banks avaiable one may also wonder why one should bother making a hardware change when all the others just require sofware changes.

APPENDIXES

These appendixes are designed to give you an easy amd quick reference to most of the things you will want to look up.

A detailed list of the precise effects of each Z80 instruction may be found in Chapters 6 and 7 and should be treated as an addition to these tables.

The appendixes are as follows:

APPENDIX A	TIMING TABLES
APPENDIX B	A MACHINE CODE PRINT and INPUT Routine
APPENDIX C	MACHINE CODES BY NUMBERS
APPENDIX D	MACHINE CODES BY FUNCTIONS (DEC and HEX)
APPENDIX E	ADDRESS and NUMBER CONVERSION TABLES
APPENDIX F	Bibliography and Copyright acknowledgements

APPENDIX A TIMING TABLES

Abbreviations: r = register i = Index register(IX,IY)
n = number b = bit number
ct = condition true cf = condition false

LD r, r	4	ADD r	4	OUT (n), A/IN A, (n)	11
LD r, n/(HL)	7	ADD n	7	OUT (C), r/IN r, (C)	12
LD (HL), r	7	ADD (HL)	7		
LD A, (rr)	7	ADD (i+d)	19	LDI/LDD/CPI/CPD/INI	
LD (rr), A	7			IND/OUTI/OUTD	16
LD (HL), n	10	ADC as			
LD A, (nn)	13	SUB in		LDIR/LDDR	
LD (nn), A	13	SBC the		CPIR/CPDR BC <> 0	21
LD r, (i+d)	19	AND four		INIR/INDR BC = 0	16
LD (i+d), r	19	OR cases		OTIR/OTDR	
LD (i+d), n	19	CP above			
LD A, I/R	9			nop	4
LD I/R, A	9	DEC/INC r	4	HALT	4
LD rr, nn	10	DEC/INC (HL)	11	DI/EI	4
LD HL, (nn)	13	DEC/INC (i+d)	23	IM0/IM1/IM2	8
LD rr, (nn)	20	DEC/INC rr	6		
LD i, (nn)	20	DEC/INC i	10	PUSH rr	11
LD (nn), HL	16			POP rr	10
LD (nn), i	20	ADD HL, rr	11	PUSH i	15
LD SP, HL	6	ADC HL, rr	15	POP i	14
LD SP, i	10	SBC HL, rr	15		
		ADD i, rr	15	EX DE, HL	4
JP nn	10			EX AF, AF'	4
JP c, nn	10	RLC r	8	EXX	4
JR ct, d	12	RLC (HL)	15	EX (SP), HL	19
cf, d	7	RLC (i+d)	23	EX (SP), i	23
JR d	12				
JP (HL)	4	RL as		BIT b, r	8
JP (i)	8	RR in		BIT b, (HL)	12
DJNZ B = 0	8	RRC the		BIT b, (i+d)	20
B <> 0	13	SLA three		RES/SET b, r	8
		SRA cases		RES/SET b, (HL)	15
CALL nn	17	RRL above		RES/SET b, (i+d)	23
CALL cf, nn	10				
ct, nn	17	RLCA/RLA/RRCA/RRA	4		
		RLD/RRD	18		
RST	11	DAA	4		
RET	10	CPL	4		
RET cf	5	NEG	8		
ct	11	SCF/CCF	4		
RET I/R	14				

The extra instruction set involving the High or Low bit of the Index Registers have cycle times 7 T states longer than their H and L equivalents.

APPENDIX B

A PRINT ROUTINE THAT WORKS LIKE A BASIC PRINT STATEMENT

This routine is address independent. It will print normal letters, numbers and characters as well as graphics and UDG's but not tokens. Tokens not activated will print as "?".

However, the following tokens are activated and can be used in data statements:

INK	217	Follow by color 1-7 in next byte
PAPER	218	Follow by color 1-7 in next byte
FLASH	219	Follow by 0 for off, 1 for on in next byte
BRIGHT	220	Follow by 0 for off, 1 for on in next byte
INVERSE	221	Follow by 0 for off, 1 for on in next byte
OVER	222	Follow by 0 for off, 1 for on in next byte
NEW LINE	13	Start an new line--Print apostrophe.
AT	172	Follow by line # and column # in next two bytes
TAB	173	Follow by column # in next byte
END	0	To indicate end of message.

TO SETUP: LD DE with your data base address.
LD HL with your print start address (or start your data with an AT statement).

The following Basic line:

```
PRINT AT 1,13; PAPER 6; INK 4; FLASH 1; "Hello.";
FLASH 0; TAB 7; INK 2; PAPER 5; "I'm your friendly";
PAPER 6; INK 0; '' TAB 7; "                    ";
TAB 5; " "; INVERSE 1; "TIMEX/SINCLAIR 2068"; INVERSE
0; " "; TAB 5; "                    "; '' TAB 11; INK
1; BRIGHT 1; "Computer"; '' TAB 5; "What is your
name?"; AT 13,14; OVER 1; "____ ____"; OVER 0;
BRIGHT 0
```

NOTE: Graphic A is a.

Would be encoded as: (Underlines show the print control characters encoded.)

```
64000  172,1,13,218,6,217,4,219,1,72
64010  101,108,108,111,46,219,0,173,7,217
64020  2,218,5,73,44,109,32,121,111,117
64030  114,32,102,114,105,101,110,110,100,108,121
64040  218,6,217,0,13,13,173,5,139,131
64050  131,131,131,131,131,131,131,131,131,131
64060  131,131,131,131,131,131,131,131,135,173
64070  5,138,221,1,84,73,77,69,88,47
64080  83,73,78,67,76,65,73,82n32,50
64090  48,54,56,221,0,133,173,5,142,140
64100  140,140,140,140,140,140,140,140,140,140
64110  140,140,140,140,140,140,140,140,141,13
64120  13,173,11,217,1,220,1,67,111,109
64130  112,117,116,101,114,46,13,13,173,5
```

```
64140  144,87,104,97,116,32,105,115,32,121
64150  111,117,114,32,110,97,109,101,63,114
64160  172,12,14,222,1,95,95,95,95,32
64170  95,95,95,95,222,0,222,0,0
```

The main differences are the omission of all quote marks, commas, and semi-colons.

NOTE: This routine will print to all 24 lines of the screen. However, the last 2 lines will disappear as soon as the program returns to Basic and the computer needs to print an error message on them. It will also return when out of screen rather than promp for a scroll. A CLS routine, not part of the above PRINT program is also included at address 65152. An INPUT routine to answer the question asked by the text is given at address 65184. I use this routine in my basic machine code class to familiarize students with the operation of registers and other techniques common to writing code. The explanation of what is going on and why are part of the class presentation and are not included in the program. I, therefore, leave it up to the student to supply them. Start reading at address 65000. Should you wish a tape of this program, kindly send $4 in care of SMUG at the address on the title page.

```
64724  26              INK   LD A, (DE)
64725  254,6                 CP 8
64727  48,66                 JR NC, OUT
64729  79                    LD C, A
64730  58,143,92             LD A, (23695) ATTR T
64733  230,248               AND 248
64735  177                   OR C
64736  24,54                 JR  LD ATTR 1

64738  26              PAPER LD A, (DE)
64739  254,8                 CP 8
64741  48,52                 JR NC, OUT
64743  167                   AND A
64744  23                    RLA
64745  23                    RLA
64746  23                    RLA
64747  79                    LD C, A
64748  58,143,92             LD A, (23695) ATTR T
64751  230,199               AND 199
64753  177                   OR C
64754  24,36                 JR  LD ATTR 1

64756  26              FLASH LD A, (DE)
64757  254,0                 CP 0
64759  32,7                  JR NZ, SET F
64761  58,143,92       Reset LD A, (23695) ATTR T
64764  230,127               AND 127
64766  24,24                 JR  LD ATTR 1
```

```
64768  58,143,92     SET F  LD A, (23695)  ATTR T
64771  246,128              OR 128
64773  24,17                JR  LD ATTR 1

64775  26           BRIGHT  LD A, (DE)
64776  254,0                CP 0
64778  32,7                 JR NZ, SET B
64780  58,143,92     Reset   LD A, (23695) ATTR T
64783  230,191              AND 191
64785  24,5                 JR  LD ATTR 1
64787  58,143,92     SET B   LD A, (23695)
64790  246,64               OR 64
64792  50,143,92 LD ATTR 1  LD (23695), A
64795  24,62                JR  RET NEXT CHAR

64797  167          TOKENS 3  AND A
64798  254,217              CP 217
64800  40,178               JR Z,  INK
64802  254,218              CP 218
64804  40,188               JR Z, PAPER
64806  254,219              CP 219
64808  40,202               JR Z, FLASH
64810  254,220              CP 220
64812  40,217               JR Z, BRIGHT
64814  254,221              CP 221
64816  40,2                 JR Z, INVERSE
64818  24,19                JR  OVER

64820  26           INVERSE  LD A, (DE)
64821  254,0                CP 0
64823  32,7                 JR NZ, SET I
64825  58,145,92     Reset   LD A, (23697) P(rint) FLAG
64828  230,251              AND 251
64830  24,24                JR  LD ATTR 2
64832  58,145,92     SET I   LD A, (23697)
64835  246,4                OR 4
64837  24,17                JR  LD ATTR 2

64839  26             OVER   LD A, (DE)
64840  254,0                CP 0
64842  32,7                 JR NZ, SET O
64844  58,145,92     Reset   LD A, (23697) P(rint) FLAG
64847  230,254              AND 254
64849  24,5                 JR  LD ATTR 2
64851  58,145,92     SET O   LD A, (23697)
64854  246,1                OR 1
64856  50,145,92 LD ATTR 2  LD (23697), A
64859  209   RET NEXT CHAR   POP DE
64860  19                   INC DE
64861  24,125               JR  NXT CHAR

64863  167              TAB  AND A
64864  125                   LD A, L
64865  214,32      SUB LINE  SUB 32
```

```
64867  48,252                JR NC, SUB LINE
64869  198,32                ADD A, 32
64871  79                    LD C, A     SAVE PART OF LINE
64872  209                   POP DE
64873  26                    LD A, (DE)
64874  19                    INC DE
64875  71                    LD B, A   SAVE TAB #
64876  254,32                CP 32
64878  208                   RET NC  = OUT
64879  185                   CP C
64880  125                   LD A, L
64881  56,5                  JR C,  NXT LINE
64883  145                   SUB C  SUB PART OF LINE
64884  128     NOT END OF 8  ADD A, B   ADD TAB
64885  111                   LD L, A
64886  24,112                JR  NXT CHAR
64888  145         NXT LINE  SUB C
64889  198,32                ADD 32  ADD A LINE
64891  32,247                JR  NZ, NOT END OF 8
64893  0                     nop  (my goof)
64894  128           ADJ PR  ADD A, B
64895  111                   LD L, A
64896  124                   LD A, H
64897  214,8                 ADD A, 8
64899  254,88                CP 88  END OF SCREEN?
64901  200                   RET Z     = OUT
64902  103                   LD H, A
64903  24,95                 JR  NXT CHAR

64905  167           TOKENS  AND A
64906  254,217               CP 217
64908  56,4                  JR C, TO TOKENS 2
64910  254,223               CP 223
64912  56,139                JR C, TOKENS 3
64914  254,172 TO TOKENS 2   CP 172
64916  40,8                  JR Z, AT
64918  254,173               CP 173
64920  40,197                JR Z, TAB
64922  62,63 UNPRINTABLE ?   LD A, 63
64924  24,112                JR  REG CHAR

64926  167                AT AND A
64927  209                   POP DE
64928  26                    LD A, (DE)
64929  19                    INC DE
64930  254,24                CP 24
64932  208                   RET NC  = OUT
64933  71                    LD B, A
64934  26                    LD A, (DE)
64935  19                    INC DE
64936  254,32                CP 32
64938  208                   RET NC   = OUT
64939  79                    LD C, A
64940  38,64                 LD H, 64
```

```
65942  120                    LD A, B
64943  214,8                  SUB A, 8
64945  56,10                  JR C, 1ST 8
64947  214,8                  SUB A, 8
64949  56,4                   JR C, 2ND 8
64951  203,228                SET 4, H
64953  24,4                   JR NEXT
64955  203,220        2ND 8   SET 3, H
64957  198,2          1ST 8   ADD A, 8
64959  167             NEXT   AND A
64960  23                     RLA
64961  23                     RLA
64962  23                     RLA
64963  23                     RLA
64964  23                     RLA
64965  129                    ADD A, C  ADD COLUMN AMT
64966  111                    LD L, A
64967  24,31                  JR  NXT CHAR

64969  167          NEW LINE  AND A
64970  125                    LD A, L
64971  214,32         SUB LN  SUB 32
64973  48,252                 JR NC, SUB LN
64975  198,32                 ADD A, 32
64977  79                     LD C, A
64978  125                    LD A, L
64979  145                    SUB C  SUB PART OF LINE
64980  198,32                 ADD A, 32  NEW LINE
64982  111                    LD L, A
64983  254,0                  CP 0
64985  40,3                   JR Z, ADJUST
64987  209                    POP DE
64988  24,10                  JR  NXT CHAR
64990  167            ADJUST  AND A
64991  124                    LD A, H
64992  198,8                  ADD A, 8
64994  254,88                 CP 88   END OF SCREEN?
64996  40,123                 JR Z, OUT
64998  103                    LD H, A
64999  209                    POP DE

65000  26 PRINT = NXT CHAR    LD A, (DE)
65001  167                    AND A
65002  19                     INC DE  UPDATE DATA FILE POSN
65003  213                    PUSH DE
65004  254,0                  CP 0
65006  40,113                 JR Z, OUT
65008  254,13                 CP 13
65010  40,213                 JR Z, NEW LINE
65012  254,32                 CP 32
65014  56,162                 JR C, UNPRINTABLE ?
65016  254,165                CP 165
65018  48,141                 JR NC, TOKEN
65020  254,128                CP 128
```

```
65022  56,6                    JR C,  REG CHAR F&S
65024  254,144                 CP 144
65026  56,95                   JR C, GRAPHIC
65028  214,133                 SUB 133   A UDG
65030  254,124 REG CHAR F&S    CP 124
65032  40,143                  JR Z, UNPRINTABLE ?
65034  254,126                 CP 126
65036  40,139                  JR Z, UNPRINTABLE ?

65038  22,0        REG CHAR    LD D, 0
65040  167                     AND A
65041  23                      RLA
65042  23                      RLA
65043  203,18                  RL D
65045  23                      RLA
65046  203,18                  RL D
65048  95                      LD E, A
65049  122                     LD A, D
65050  0                       nop
65051  32,4                    JR NZ, NOT UDG
65053  22,255                  LD D, 255
65055  24,3                    JR  OVER?
65057  198,60       NOT UDG    ADD A, 60
65059  87                      LD D, A

65060  58,145,92      OVER?    LD A, (23697) P  FLAG
65063  6,255                   LD B, 255
65065  31                      RRA
65066  56,1                    JR C, INVERSE?  OVER ON--B = 255
65068  4                       INC B           OFF--B = 0
65069  31           INVERSE?   RRA
65070  31                      RRA
65071  159                     SBC A,A   = 0 IF OFF, ELSE 255
65072  79                      LD C, A
65073  62,8                    LD A, 8  COUNT
65075  167                     AND A
65076  235                     EX DE, HL
65077  245         PIX LINE    PUSH AF   SAVE COUNT
65078  26                      LD A, (DE) GET SCREEN BYTE
65079  160                     AND B
65080  174                     XOR (HL) CHAR BYTE
65081  169                     XOR C   INVERT
65082  18                      LD (DE), A
65083  20                      INC D
65084  35                      INC HL
65085  241                     POP AF  GET COUNT
65086  61                      DEC A
65087  32,244                  JR NZ, PIX LINE

65089  21           DO ATTR    DEC D
65090  122                     LD A, D
65091  15                      RRCA
65092  15                      RRCA
65093  15                      RRCA
```

```
65094  230,3                 AND 3  MASK 2 LOWEST BITS
65096  246,88                OR 88
65098  103                   LD H, A
65099  107                   LD L, E
65100  58,143,92             LD A, (23695)  ATTR T
65103  119                   LD (HL), A
65104  235                   EX DE, HL

65105  35   UPDATE PR POSN   INC HL
65106  125                   LD A, L
65107  40,7                  JR Z, SKIP
65109  124                   LD A, H
65110  214,7                 SUB A, 8
65112  103                   LD H, A
65113  209          RETURN   POP DE
65114  24,140                JR  NXT CHAR
65116  124            SKIP   LD A, H
65117  254,88                CP 88    END OF SCREEN?
65119  32,248                JR NZ, RETURN
65121  209             OUT   POP DE
65122  201                   RET

65123  71          GRAPHICS  LD B, A
65124  22,2                  LD D, 2
65126  167          NEXT 4   AND A
65127  203,24                RR B
65129  159                   SBC A, A IF BIT = 1, A = 255, ELSE
65130  230,15                AND 15     MASK LOW NYBBLE     0
65132  79                    LD C, A
65133  203,24                RR B
65135  159                   SBC A, A  AS ABOVE
65136  230,240               AND 240   MASK HIGH NYBBLE
65138  177                   OR C      ADD LOW NYBBLE
65139  14,4                  LD C, 4  COUNT
65141  119           AGAIN   LD (HL), A
65142  36                    INC H
65143  13                    DEC C
65144  32,251                JR NZ, AGAIN
65146  21                    DEC D
65147  32,233                JR NZ, NEXT 4
65149  235                   EX DE, HL
65150  24, 193               JR   DO ATTR

65152  33,0,64         CLS   LD HL,  D FILE
65155  1,0,24                LD BC, 6144
65158  62,0            LOOP  LD A, 0
65160  119                   LD (HL), A
65161  35                    INC HL
65162  11                    DEC BC
65163  120                   LD A, B
65164  177                   OR C
65165  32,247                JR NZ, LOOP
65167  1,0,3         CL ATTR  LD BC, 768
65170  58,141,92             LD A, (23693) ATTR P
```

```
65173  87                      LD D, A
65174  122          LOOP 2     LD A, D
65175  119                     LD (HL), A
65176  35                      INC HL
65177  11                      DEC BC
65178  120                     LD A, B
65179  177                     OR C
65180  32,248                  JR NZ, LOOP 2
65182  201                     RET

65183  0                       nop

65184  17,0,255       INPUT    LD DE, INPUT STORE
65187  62,0                    LD A, 0  CLEAR STORE
65189  6,32                    LD B, 32 COUNT
65191  18              LOOP    LD (DE), A
65192  19                      INC DE
65193  16,252                  DJNZ, LOOP
65195  17,0,255                LD DE, INPUT STORE
65198  253,203,1,174   KEY     RES 5, (IY+1)  KEYHIT
65202  213                     PUSH DE
65203  205,225,2       WAIT    CALL 737  UPDATE KEYBOARD
65206  167                     AND A
65207  253,203,1,110           BIT 5, (IY+1) HIT?
65211  40,246                  JR Z, WAIT
65213  1,0,80                  LD BC, 20480
65216  11           DEBOUNCE   DEC BC
65217  120                     LD A, B
65218  177                     OR C
65219  32,251                  JR NZ, DEBOUNCE
65221  58,8,92                 LD A, (23560) LAST KEY
65224  167                     AND A
65225  254,13                  CP 13  ENTER?
65227  209                     POP DE
65228  200                     RET Z
65229  254,12                  CP 12  DELETE?
65231  40,23                   JR Z, DELETE
65233  254,32                  CP 32,  CHECK KEY
65235  56,217                  JR C, KEY    UNPRINTABLE
65237  254,123                 CP 123
65239  48,213                  JR NC, KEY
65241  18                      LD (DE), A  STORE LETTER
65242  19                      INC DE
65243  213           PR INP    PUSH DE
65244  17,0,255                LD DE, INPUT START
65247  33,224,80               LD HL, BOTTOM LINE
65250  205,232,253             CALL 65000   PRINT
65253  209                     POP DE
65254  24,198                  JR  KEY
65256  62,0          DELETE    LD A, 0 CLEAR HOLD FILE
65258  27                      DEC DE
65259  18                      LD (DE), A
65260  33,224,80 CL BOT LN     LD HL, 20704 CLEAR SCREEN LINE
65263  14,8                    LD C, 8  BOTTOM LINE
```

```
65265  6,31            ROW  LD B, 31
65267  119            LINE  LD (HL), A
65268  35                   INC HL
65269  16,252               DJNZ, LINE
65271  36                   INC H
65272  46,224               LD L, 224
65274  13                   DEC C
65275  32,244               JR NZ, ROW
65277  24,220               JR PR INP
65279  0
65280-65312                 DATA BASE  INPUT STORE
```

The above input only allows for a name of 32 characters. It is printed to the bottom line of the screen and allows for deletes of a wrong character. It CALLs the PRINT routine at 65000 so if relocated, that line must be changed. Now that you have your input stored it's up to you do with it what you want. In our class we do:

```
63000  243                  DI
63001  17,0,250             LD DE, DATA 1
63004  205,232,253          CALL PRINT
63007  0                    nop
63008  205,160,254          CALL INPUT
63011  17,186,250           LD DE, 64186  XFER NAME TO DATA B
63014  33,0,255             LD HL, 65280
63017  1,32,0               LD BC, 32
63020  237,176              LDIR
63022  33,224,80 CL BOT LN  LD HL, 20704
63025  62,0                 LD A, 0
63027  14,8                 LD C, 8
63029  6,31            ROW  LD B, 31
63031  119            LINE  LD (HL), A
63032  35                   INC HL
63033  16,252               DJNZ, LINE
63035  36                   INC H
63036  46,224               LD L, 224
63038  13                   DEC C
63039  32,244               JR NZ, ROW
63041  58,72,92     CL ATTR LD A, (23624)
63044  230,56               AND 56
63046  33,224,90            LD HL, LAST LINE ATTR ADDR
63049  6,32                 LD B, 32
63051  119            LOOP  LD (HL), A
63052  35                   INC HL
63053  16,252               DJNZ, LOOP
63055  17,179,250  PR NAME  LD DE, 64179
63058  205,232,253          CALL PRINT
63061  17,219,250  PR MESS  LD DE, 64219
63064  205,232,253          CALL PRINT
63067  251                  EI
63068  201                  RET
```

APPENDIX C A COMPLETE CODE TABLE

Codes with * are not verified by manufacturer.

DEC	HEX	CODE	NORMAL	AFTER CB	AFTER ED	AFTER DD	AFTER FD
0	00	----	nop	RLC B			
1	01	----	LD BC, NN	RLC C			
2	02	----	LD (BC),A	RLC D			
3	03	----	INC BC	RLC E			
4	04	----	INC B	RLC H			
5	05	----	DEC B	RLC L			
6	06	PRINT ,	LD B, N	RLC (HL)			
7	07	EDIT	RLCA	RLC A			
8	08	C LEFT	EX AF,AF'	RRC B			
9	09	C RIGHT	ADD HL,BC	RRC C		ADD IX,BC	ADD IY,DE
10	0A	C DOWN	LD A,(BC)	RRC D			
11	0B	C UP	DEC BC	RRC E			
12	0C	DELETE	INC C	RRC H			
13	0D	ENTER	DEC C	RRC L			
14	0E	SLUG	LD C, N	RRC (HL)			
15	0F	----	RRCA	RRC A			
16	10	INK CTR	DJNZ, d	RL B			
17	11	PAPER CTR	LD DE,NN	RL C			
18	12	FLASH CTR	LD (DE),A	RL D			
19	13	BRIGHT CT	INC DE	RL E			
20	14	INVERSE C	INC D	RL H			
21	15	OVER CTR	DEC D	RL L			
22	16	AT CTR	LD D, N	RL (HL)			
23	17	TAB CTR	RLA	RL A			
24	18	----	JR d	RR B			
25	19	----	ADD HL,DE	RR C		ADD IX,DE	ADD IY,DE
26	1A	----	LD A, (DE)	RR D			
27	1B	----	DEC DE	RR E			
28	1C	----	INC E	RR H			
29	1D	----	DEC E	RR L			
30	1E	----	LD E, N	RR (HL)			
31	1F	----	RRA	RR A			
32	20	SPACE	JR NZ, d	SLA B			
33	21	!	LD HL, NN	SLA C		LD IX,NN	LD IY,NN
34	22	"	LD(NN),HL	SLA D		LD(NN),IX	LD(NN),IY
35	23	#	INC HL	SLA E		INC IX	INC IY
36	24	$	INC H	SLA H		INC HIX*	INC HIY*
37	25	%	DEC H	SLA L		DEC HIX*	DEC HIY*
38	26	&	LD H, N	SLA (HL)		LD HIX,N*	LD HIY,N*
39	27	'	DAA	SLA A			
40	28	(	JR Z, d	SRA B			
41	29	)	ADD HL,HL	SRA C		ADD IX,IX	ADD IY,IY
42	2A	*	LD HL,(NN)	SRA D		LD IX,(NN)	LD IY,(NN)
43	2B	+	DEC HL	SRA E		DEC IX	DEC IY
44	2C	,	INC L	SRA H		INC LIX*	INC LIY*
45	2D	-	DEC L	SRA L		DEC LIX*	DEC LIY*
46	2E	.	LD L, N	SRA (HL)		LD LIX, N*	LD LIY, N*
47	2F	/	CPL	SRA A			
48	30	0	JR NC, d	SLL B*			

APPENDIX C CODE TABLE

DEC	HEX	CODE	NORMAL	AFTER CB	AFTER ED	AFTER DD	AFTER FD
49	31	1	LD SP,NN	SLL C*			
50	32	2	LD(NN), A	SLL D*			
51	33	3	INC SP	SLL E*			
52	34	4	INC (HL)	SLL H*		INC(IX+d)	INC(IY+d)
53	35	5	DEC (HL)	SLL L*		DEC(IX+d)	DEC(IY+d)
54	36	6	LD(HL),N	SLL (HL)*		LD(IX+d),N	LD(IY+d),N
55	37	7	SCF	SLL A*			
56	38	8	JR C, d	SRL B			
57	39	9	ADD HL,SP	SRL C		ADD IX,SP	ADD IY,SP
58	3A	:	LD A, (NN)	SRL D			
59	3B	;	DEC SP	SRL E			
60	3C	<	INC A	SRL H			
61	3D	=	DEC A	SRL L			
62	3E	>	LD A, N	SRL (HL)			
63	3F	?	CCF	SRL A			
64	40	@	LD B, B	BIT 0, B	IN B,(C)		
65	41	A	LD B, C	BIT 0, C	OUT(C),B		
66	42	B	LD B, D	BIT 0, D	SBC HL,BC		
67	43	C	LD B, E	BIT 0, E	LD(NN),BC		
68	44	D	LD B, H	BIT 0, H	NEG	LD B,HIX*	LD B,HIY*
69	45	E	LD B, L	BIT 0, L	RET N	LD B,LIX*	LD B,LIY*
70	46	F	LD B,(HL)	BIT 0,(HL)	IMO	LD B,(IX+d)	LD B,(IY+d)
71	47	G	LD B, A	BIT 0, A	LD I, A		
72	48	H	LD C, B	BIT 1, B	IN C,(C)		
73	49	I	LD C, C	BIT 1, C	OUT(C),C		
74	4A	J	LD C, D	BIT 1, D	ADC HL,BC		
75	4B	K	LD C, E	BIT 1, E	LD BC,(NN)		
76	4C	L	LD C, H	BIT 1, H	NEG*	LD C,HIX*	LD C,HIY*
77	4D	M	LD C, L	BIT 1, L	RET I	LD C,LIX*	LD C,LIY*
78	4E	N	LD C, (HL)	BIT 1,(HL)		LD C,(IX+d)	LD C,(IY+d)
79	4F	O	LD C, A	BIT 1, A	LD R, A		
80	50	P	LD D, B	BIT 2, B	IN D,(C)		
81	51	Q	LD D, C	BIT 2, C	OUT(C),D		
82	52	R	LD D, D	BIT 2, D	SBC HL,DE		
83	53	S	LD D, E	BIT 2, E	LD(NN),DE		
84	54	T	LD D, H	BIT 2, H	NEG*	LD D,HIX*	LD D,HIY*
85	55	U	LD D, L	BIT 2, L	RET N*	LD D,LIX*	LD D,LIY*
86	56	V	LD D, (HL)	BIT 2,(HL)	IM 1	LD D,(IX+d)	LD D,(IY+d)
87	57	W	LD D, A	BIT 2, A	LD A, I		
88	58	X	LD E, B	BIT 3, B	IN E,(C)		
89	59	Y	LD E, C	BIT 3, C	OUT(C),E		
90	5A	Z	LD E, D	BIT 3, D	ADC HL,DE		
91	5B	[	LD E, E	BIT 3, E	LD DE,(NN)		
92	5C	\	LD E, H	BIT 3, H	NEG*	LD E,HIX*	LD E,HIY*
93	5D	]	LD E, L	BIT 3, L	RET N*	LD E,LIX*	LD E,LIY*
94	5E	^	LD E, (HL)	BIT 3,(HL)	IM 2	LD E,(IX+d)	LD E,(IY+d)
95	5F	_	LD E, A	BIT 3, A	LD A, R		
96	60	`	LD H, B	BIT 4, B	IN H,(C)	LD HIX, B*	LD HIY, B*
97	61	a	LD H, C	BIT 4, C	OUT(C),H	LD HIX, C*	LD HIY, C*
98	62	b	LD H, D	BIT 4, D	SBC HL,HL	LD HIX, D*	LD HIY, D*
99	63	c	LD H, E	BIT 4, E	LD(NN),HL	LD HIX, E*	LD HIY, E*

APPENDIX C CODE TABLE

DEC	HEX	CODE	NORMAL	AFTER CB	AFTER ED	AFTER DD	AFTER FD
100	64	d	LD H, H	BIT 4, H	NEG*	LD HIX, H*	LD HIY, H*
101	65	e	LD H, L	BIT 4, L	RET N*	LD HIX,LIX*	LD HIY,LIY*
102	66	f	LD H, (HL)	BIT 4,(HL)		LD H,(IX+d)	LD H,(IY+d)
103	67	g	LD H, A	BIT 4, A	RR D		
104	68	h	LD L, B	BIT 5, B	IN L,(C)	LD LIX, B*	LD LIY, B*
105	69	i	LD L, C	BIT 5, C	OUT(C), L	LD LIX, C*	LD LIY, C*
106	6A	j	LD L, D	BIT 5, D	ADC HL,HL	LD LIX, D*	LD LIY, D*
107	6B	k	LD L, E	BIT 5, E	LD HL,(NN)	LD LIX, E*	LD LIY, E*
108	6C	l	LD L, H	BIT 5, H	NEG*	LD LIX, H*	LD LIY, H*
109	6D	m	LD L, L	BIT 5, L	RET N*		
110	6E	n	LD L,(HL)	BIT 5,(HL)		LD L,(IX+d)	LD L,(IY+d)
111	6F	o	LD L, A	BIT 5, A	RL D		
112	70	p	LD (HL),B	BIT 6, B	IN(HL),(C)	LD(IX+d),B	LD(IY+d),B
113	71	q	LD (HL),C	BIT 6, C	OUT(C),(HL)	LD(IX+d),C	LD(IY+d),C
114	72	r	LD (HL),D	BIT 6, D	SBC HL,SP	LD(IX+d),D	LD(IY+d),D
115	73	s	LD (HL),E	BIT 6, E	LD(NN),SP	LD(IX+d),E	LD(IY+d),E
116	74	t	LD (HL),H	BIT 6, H	NEG*	LD(IX+d),H	LD(IY+d),H
117	75	u	LD (HL),L	BIT 6, L	RET N*	LD(IX+d),L	LD(IY+d),L
118	76	v	HALT	BIT 6,(HL)			
119	77	w	LD (HL),A	BIT 6, A		LD(IX+d),A	LD(IY+d),A
120	78	x	LD A, B	BIT 7, B	IN A,(C)		
121	79	y	LD A, C	BIT 7, C	OUT(C),A		
122	7A	z	LD A, D	BIT 7, D	ADC HL,SP		
123	7B	ON ERR	LD A, E	BIT 7, E	LD SP,(NN)		
124	7C	STICK	LD A, H	BIT 7, H	NEG*	LD A,HIX*	LD A,HIY*
125	7D	SOUND	LD A, L	BIT 7, L	RET N*	LD A,LIX*	LD A,LIY*
126	7E	FREE	LD A, (HL)	BIT 7,(HL)		LD A,(IX+d)	LD A,(IY+d)
127	7F	RESET	LD A, A	BIT 7, A			
128	80		ADD A, B	RES 0, B			
129	81		ADD A, C	RES 0, C			
130	82		ADD A, D	RES 0, D			
131	83		ADD A, E	RES 0, E			
132	84		ADD A, H	RES 0, H		ADD A,HIX*	ADD A,HIY*
133	85		ADD A, L	RES 0, L		ADD A,LIX*	ADD A,LIY*
134	86		ADD A,(HL)	RES 0,(HL)		ADD A,(IX+d)	ADD A,(IY+d)
135	87		ADD A, A	RES 0, A			
136	88		ADC A, B	RES 1, B			
137	89		ADC A, C	RES 1, C			
138	8A		ADC A, D	RES 1, D			
139	8B		ADC A, E	RES 1, E			
140	8C		ADC A, H	RES 1, H		ADC A,HIX*	ADC A,HIY*
141	8D		ADC A, L	RES 1, L		ADC A,LIX*	ADC A,LIX*
142	8E		ADC A,(HL)	RES 1,(HL)		ADC A,(IX+d)	ADC A,(IY+d)
143	8F		ADC A, A	RES 1, A			
144	90	UDG A	SUB A, B	RES 2, B			
145	91	UDG B	SUB A, C	RES 2, C			
146	92	UDG C	SUB A, D	RES 2, D			
147	93	UDG D	SUB A, E	RES 2, E			
148	94	UDG E	SUB A, H	RES 2, H		SUB A,HIX*	SUB A,HIY*
149	95	UDG F	SUB A, L	RES 2, L		SUB A,LIX*	SUB A,LIY*
150	96	UDG G	SUB A,(HL)	RES 2,(HL)		SUB A,(IX+d)	SUB A,(IY+d)

APPENDIX C CODE TABLE

DEC	HEX	CODE	NORMAL	AFTER CB	AFTER ED	AFTER DD	AFTER FD
151	97	UDG H	SUB A, A	RES 2, A			
152	98	UDG I	SBC A, B	RES 3, B			
153	99	UDG J	SBC A, C	RES 3, C			
154	9A	UDG K	SBC A, D	RES 3, D			
155	9B	UDG L	SBC A, E	RES 3, E			
156	9C	UDG M	SBC A, H	RES 3, H		SBC A,HIX*	SBC A,HIY*
157	9D	UDG N	SBC A, L	RES 3, L		SBC A,LIX*	SBC A,LIY*
158	9E	UDG O	SBC A,(HL)	RES 3,(HL)		SBC A,(IX+d)	SBC A,(IY+d)
159	9F	UDG P	SBC A, A	RES 3, A			
160	A0	UDG Q	AND B	RES 4, B	LDI		
161	A1	UDG R	AND C	RES 4, C	CPI		
162	A2	UDG S	AND D	RES 4, D	INI		
163	A3	UDG T	AND E	RES 4, E	OUTI		
164	A4	UDG U	AND H	RES 4, H		AND HIX*	AND HIY*
165	A5	RND	AND L	RES 4, L		AND LIX*	AND LIY*
166	A6	INKEY$	AND (HL)	RES 4,(HL)		AND (IX+d)	AND (IY+d)
167	A7	PI	AND A	RES 4, A			
168	A8	FN	XOR B	RES 5, B	LDD		
169	A9	POINT	XOR C	RES 5, C	CPD		
170	AA	SCREEN$	XOR D	RES 5, D	IND		
171	AB	ATTR	XOR E	RES 5, E	OUTD		
172	AC	AT	XOR H	RES 5, H		XOR HIX*	XOR HIY*
173	AD	TAB	XOR L	RES 5, L		XOR LIX*	XOR LIY*
174	AE	VAL$	XOR (HL)	RES 5,(HL)		XOR (IX+d)	XOR (IY+d)
175	AF	CODE	XOR A	RES 5, A			
176	B0	VAL	OR B	RES 6, B	LDIR		
177	B1	LEN	OR C	RES 6, C	CPIR		
178	B2	SIN	OR D	RES 6, D	IRIR		
179	B3	COS	OR E	RES 6, E	OTIR		
180	B4	TAN	OR H	RES 6, H		OR HIX*	OR HIY*
181	B5	ASN	OR L	RES 6, L		OR LIX*	OR LIY*
182	B6	ACS	OR (HL)	RES 6,(HL)		OR (IX+d)	OR)IY+d)
183	B7	ATN	OR A	RES 6, A			
184	B8	LN	CP B	RES 7, B	LDDR		
185	B9	EXP	CP C	RES 7, C	CPDR		
186	BA	INT	CP D	RES 7, D	INDR		
187	BB	SQR	CP E	RES 7, E	OTDR		
188	BC	SGN	CP H	RES 7, H		CP HIX*	CP HIY*
189	BD	ABS	CP L	RES 7, L		CP LIX*	CP LIY*
190	BE	PEEK	CP (HL)	RES 7,(HL)		CP (IX+d)	CP (IY+d)
191	BF	IN	CP A	RES 7, A			
192	C0	USR	RET NZ	SET 0, B			
193	C1	STR$	POP BC	SET 0, C			
194	C2	CHR$	JP NZ, NN	SET 0, D			
195	C3	NOT	JP NN	SET 0, E			
196	C4	BIN	CALL NZ,NN	SET 0, H			
197	C5	OR	PUSH BC	SET 0, L			
198	C6	AND	ADD A, N	SET 0,(HL)			
199	C7	<=	RST 0	SET 0, A			
200	C8	>=	RET Z	SET 1, B			
201	C9	<>	RET	SET 1, C			

APPENDIX C CODE TABLE

DEC	HEX	CODE	NORMAL	AFTER CB	AFTER ED	AFTED DD	AFTER FD
202	CA	LINE	JP Z, NN	SET 1, D			
203	CB	THEN	(prefix)	SET 1, E			
204	CC	TO	CALL Z, NN	SET 1, H			
205	CD	STEP	CALL NN	SET 1, L			
206	CE	DEF FN	ADC A, N	SET 1,(HL)			
207	CF	CAT	RST 8	SET 1, A			
208	D0	FORMAT	RET NC	SET 2, B			
209	D1	MOVE	POP DE	SET 2, C			
210	D2	ERASE	JP NC, NN	SET 2, D			
211	D3	OPEN #	OUT(N), A	SET 2, E			
212	D4	CLOSE #	CALL NC,NN	SET 2, H			
213	D5	MERGE	PUSH DE	SET 2, L			
214	D6	VERIFY	SUB N	SET 2,(HL)			
215	D7	BEEP	RST 16	SET 2, A			
216	D8	CIRCLE	RET C	SET 3, B			
217	D9	INK	EXX	SET 3, C			
218	DA	PAPER	JP C, NN	SET 3, D			
219	DB	FLASH	IN A, N	SET 3, E			
220	DC	BRIGHT	CALL C, NN	SET 3, H			
221	DD	INVERSE	(IX prefix)	SET 3, L			
222	DE	OVER	SBC A, N	SET 3,(HL)			
223	DF	OUT	RST 24	SET 3, A			
224	E0	LPRINT	RET PO	SET 4, B			
225	E1	LLIST	POP HL	SET 4, C		POP IX	POP IY
226	E2	STOP	JP PO, NN	SET 4, D			
227	E3	READ	EX(SP),HL	SET 4, E		EX(SP),IX	EX(SP),IY
228	E4	DATA	CALL PO,NN	SET 4, H			
229	E5	RESTORE	PUSH HL	SET 4, L		PUSH IX	PUSH IY
230	E6	NEW	AND A	SET 4,(HL)			
231	E7	BORDER	RST 32	SET 4, A			
232	E8	CONTINUE	RET PE	SET 5, B			
233	E9	DIM	JP (HL)	SET 5, C		JP (IX)	JP (IY)
234	EA	REM	JP PE, NN	SET 5, D			
235	EB	FOR	EX DE,HL	SET 5, E		EX DE,IX	EX DE,IY
236	EC	GOTO	CALL PE,NN	SET 5, H			
237	ED	GOSUB	(prefix)	SET 5, L			
238	EE	INPUT	XOR N	SET 5,(HL)			
239	EF	LOAD	RST 40	SET 5, A			
240	F0	LIST	RET P	SET 6, B			
241	F1	LET	POP AF	SET 6, C			
242	F2	PAUSE	JP P, NN	SET 6, D			
243	F3	NEXT	DI	SET 6, E			
244	F4	POKE	CALL P,NN	SET 6, H			
245	F5	PRINT	PUSH AF	SET 6, L			
246	F6	PLOT	OR N	SET 6,(HL)			
247	F7	RUN	RST 48	SET 6, A			
248	F8	SAVE	RET M	SET 7, B			
249	F9	RANDOMIZE	LD SP,HL	SET 7, C		LD SP,IX	LD SP,IY
250	FA	IF	JP M, NN	SET 7, D			
251	FB	CLS	EI	SET 7, E			
252	FC	DRAW	CALL M,NN	SET 7, H			

APPENDIX C CODE TABLE

DEC	HEX	CODE	NORMAL	AFTER CB	AFTED ED	AFTED DD	AFTER FD
253	FD	CLEAR	(IY prefix)	SET 7, L			
254	FE	RETURN	CP N	SET 7,(HL)			
255	FF	COPY	RST 56	SET 7, A			

DOUBLE PREFIX CODES

DEC	HEX	AFTER DDCBdd	AFTER FDCBdd
6	06	RLC(IX+d)	RLC(IY+d)
14	0E	RRC(IX+d)	RRC(IY+d)
22	16	RL(IX+d)	RL(IY+d)
30	1E	RR(IX+d)	RR(IY+d)
38	26	SLA(IX+d)	SLA(IY+d)
46	2E	SRA(IX+d)	SRA(IY+d)
54	36	SLL(IX+d)	SLL(IY+d)
62	3E	SRL(IX+d)	SRL(IY+d)
70	46	BIT 0,(IX+d)	BIT 0,(IY+d)
78	4E	BIT 1,(IX+d)	BIT 1,(IY+d)
86	56	BIT 2,(IX+d)	BIT 2,(IY+d)
94	5E	BIT 3,(IX+d)	BIT 3,(IY+d)
102	66	BIT 4,(IX+d)	BIT 4,(IY+d)
110	6E	BIT 5,(IX+d)	BIT 5,(IY+d)
118	76	BIT 6,(IX+d)	BIT 6,(IY+d)
126	7E	BIT 7,(IX+d)	BIT 7,(IY+d)
134	86	RES 0,(IX+d)	RES 0,(IY+d)
142	8E	RES 1,(IX+d)	RES 1,(IY+d)
150	96	RES 2,(IX+d)	RES 2,(IY+d)
158	9E	RES 3,(IX+d)	RES 3,(IY+d)
166	A6	RES 4,(IX+d)	RES 4,(IY+d)
174	AE	RES 5,(IX+d)	RES 5,(IY+d)
182	B6	RES 6,(IX+d)	RES 6,(IY+d)
190	BE	RES 7,(IX+d)	RES 6,(IY+d)
198	C6	SET 0,(IX+d)	SET 0,(IY+d)
206	CE	SET 1,(IX+d)	SET 1,(IY+d)
214	D6	SET 2,(IX+d)	SET 2,(IY+d)
222	DE	SET 3,(IX+d)	SET 3,(IY+d)
230	E6	SET 4,(IX+d)	SET 4,(IY+d)
238	EE	SET 5,(IX+d)	SET 5,(IY+d)
246	F6	SET 6,(IX+d)	SET 6,(IY+d)
254	FE	SET 7,(IX+d)	SET 7,(IY+d)

APPENDIX D MACHINE CODES FOR ENCODING--DECIMAL

with	A	B	C	D	E	H	L	(HL)	n	(IX,IY+d)	(nn)	(BC)	(DE)
LD A	127	120	121	122	123	124	125	126	62n	126d	58nn	10	26
B	71	64	65	66	67	68	69	70	6n	70d			
C	79	72	73	74	75	76	77	78	14n	78d			
D	87	80	81	82	83	84	85	86	22n	86d			
E	95	88	89	90	91	92	93	94	30n	94d			
H	103	96	97	98	99	100	101	102	38n	102d			
L	111	104	105	106	107	108	109	110	46n	110d			
(HL)	119	112	113	114	115	116	117	---	54n				
(IX,IY+d)	119d	112d	113d	114d	115d	116d	117d		54dn				
(nn)	50												
(BC)	2												
(DE)	18												

PREFACE ALL IX with 221
ALL IY with 253

with nn		(nn)	LD (nn) with:	
LD BC	1nn	237,75nn	BC	237,67nn
DE	17nn	237,91nn	DE	237,83nn
IX,IY,HL	33nn	237,106nn or 42nn	HL/IX/IY	237,99nn or 34nn
SP	49nn	237,123nn	SP	237,115nn

```
LDI 237,160
LDD 237,168
CPI 237,161
CPD 237,169
LDIR 237,176
LDDR 237,184
CPIR 237,177
CPDR 237,185
LD R, A 237,79
LD A, I 237,87
LD A, R 237,95
LD I, A 237,71
LD SP, HL/IX/IY
        249
```

	A	B	C	D	E	H	L	(HL)	n	(IX,IY+d)	nop	0
ADD r	135	128	129	130	131	132	133	134	198n	134d	CPL	47
ADC r	143	136	137	138	139	140	141	142	206n	142d	NEG	237,68
SUB r	151	144	145	146	147	148	149	150	214n	150d	CCF	63
SBC r	159	152	153	154	155	156	157	158	222n	158d	SCF	55
AND r	167	160	161	162	163	164	165	166	230n	166d	HALT	118
XOR r	175	168	169	170	171	172	173	174	238n	174d	DI	243
OR r	183	176	177	178	179	180	181	182	246n	182d	EI	251
CP r	191	184	185	186	187	188	189	190	254n	190d	IMO	237HL) n

(IX,IY+d)	nop	0										
ADD r	135	128	129	130	131	132	133	134	198n	134d	CPL	47
ADC r	143	136	137	138	139	140	141	142	206n	142d	NEG	237,68
SUB r	151	144	145	146	147	148	149	150	214n	150d	CCF	65
SBC r	159	152	153	154	155	156	157	158	222n	158d	SCF	55
AND r	167	160	161	162	163	164	165	166	230n	166d	HALT	118
XOR r	175	168	169	170	171	172	173	174	238n	174d	DI	243
OR r	183	176	177	178	179	180	181	182	246n	182d	EI	251
CP r	191	184	185	186	187	188	189	190	254n	190d	IMO	237,70
INC	60	4	12	20	28	36	44	52		52d	IM1	237,86
DEC	61	4	13	21	29	37	45	53		53d	IM2	237,94

	BC	DE	HL	SP	IX/IY
ADD HL,rr*	9	25	41	57	
ADC HL,rr	237,74	237,90	237,106	237,122	
SBC HL,rr	237,66	237,82	237,98	237,114	
INC rr	3	19	35	51	35
DEC rr	11	27	43	59	43

* also to IX and IY.

	AF	BC	DE	HL*
PUSH	245	197	213	229
POP	241	193	209	225

```
EX AF, AF'  8
EX DE, HL/IX/IY 235
EX SP, HL/IX/IY 227
EXX             217
```

2		A	B	C	D	E	H	L	(HL)	(IX,IY+d)
0	RLC r	7	0	1	2	3	4	5	6	6d
3	RRC r	15	8	9	10	11	12	13	14	14d
P	RL r	23	16	17	18	19	20	21	22	22d
R	RR r	31	24	25	26	27	28	29	30	30d
E	SLA r	39	32	33	34	35	36	37	38	38d
F	SRA r	47	40	41	42	43	44	45	46	46d

```
RLCA  7
RRCA 15
RLA  23
RRA  31
RLD 237,111
RRD 237,103
```

```
  SLL r 55 48 49 50 51 52 53  54    54d     DAA  39
X SRL r 63 56 57 58 59 60 61  62    62d

                                                        (HL)
             Z      NZ     C      NC     M      P      PE     PO (IX,IY)
  JP 195nn 202nn 194nn 218nn 210nn 250nn 242nn 234nn 226nn 233
CALL 205nn 204nn 196nn 220nn 212nn 252nn 244nn 236nn 228nn
 RET 201   200    192    216    208    248    240    232    224
  JR  24d   40d    32d    56d    48d    DJNZ = 16d
```

APPENDIX D MACHINE CODES FOR ENCODING--DECIMAL

Preface with 237(ED)

```
             A  B  C  D  E  H   L  (HL)    No prefix
IN r, (C)  120 64 72 80 88 96 104 112   IN A, (n)  219n
OUT (C), r 121 65 73 81 89 97 195 113   OUT (n), A 211n

IND  237,170  INI  237,162  INDR 237,178  INIR 237,178
OUTD 237,171  OUTI 237,163  OTDR 237,187  OTIR 237,179
```

```
2          A   B   C   D   E   H   L  (HL)(IX,IY)
0 BIT 0   71  64  65  66  67  68  69   70    d70     RST  0 RESET 199
3     1   79  72  73  74  75  76  77   78    d78     RST  8 ERROR 207
P     2   87  80  81  82  83  84  85   86    d86     RST 16 PRINT 215
R     3   95  88  89  90  91  92  93   94    d94     RST 24 GET C 223
E     4  103  96  97  98  99 100 101  102    d102    RST 32 NXT C 231
      5  111 104 105 106 107 108 109  110    d110    RST 40 FP CL 239
      6  119 112 113 114 115 116 117  118    d118    RST 48 BC SP 247
X     7  127 120 121 122 123 124 125  126    d126    RST 56 KEYBD 255

2          A   B   C   D   E   H   L  (HL)(IX,IY+d)
0 RES 0  135 128 129 130 131 132 133  134    d134
3     1  143 136 137 138 139 140 141  142    d142
P     2  151 144 145 146 147 148 149  150    d150229 230   d230
F     5  239 232 233 234 235 236 237  238   d238
I     6  247 240 241 242 243 244 245  246   d246
X     7  255 248 249 250 251 252 253  254   d254
```

The following commands may be unsupported:

```
              A   B   C   D   E  n  LIX,LIY HIX,HIY
LD H(IX,IY) 103  96  97  98  99 38   101     ---
LD L(IX,IY) 111 104 105 106 107 39   ---     108
```

```
     H(IX,IY) L(IX,IY)          H(IX,IY) L(IX,IY)   RET N  NEG
LD A    124      125     ADD      132      133     237,85 237,76
LD B     68       69     ADC      140      141      93     84
LD C     76       77     SUB      148      149     101     92
LD D     84       85     SBC      156      157     117    100
LD E     92       93     CP       188      189     125    108
INC      36       44     AND      164      165             116
 EC      37       45     OR       180      181             124
                         XOR      172      173
```

APPENDIX D MACHINE CODE FOR ENCODING--HEX

```
    with A  B  C  D  E  H  L (HL) n (IX,IY+d)(nn)(BC)(DE)
LD   A   7F 78 79 7A 7B 7C 7D 7E 3En  7Ed    58nn 10  26
     B   47 40 41 42 43 44 45 46 06n  46d
     C   4F 48 49 4A 4B 4C 4D 4E 0En  4Ed         LDD EDA8
     D   57 50 51 52 53 54 55 56 16n  56d         LDI EDA0
     E   5F 58 59 5A 5B 5C 5D 5E 1En  5Ed         CPD EDA9
     H   67 60 61 62 63 64 65 66 26n  66d         CPI EDA1
     L   6F 68 69 6A 6B 6C 6D 6E 2En  6Ed         DIR EDB0
   (HL)  77 70 71 72 73 74 75 -- 36n             LDDR EDB8
(IX,IY+d)77d70d71d72d73d74d75d  36dn              CPIR EDB1
   (nn)  32nn                                    CPDR EDB9
   (BC)  02      PREFACE ALL IX with DD         LD I,A ED47
   (DE)  12              ALL IY with FD         LD R,A ED4F
                                                LD A,I ED57
    with nn    (nn)        LD (nn), with:      LD A,R ED5F
LD BC    01nn ED48nn       BC       ED43nn
   DE    11nn ED58nn       DE       ED53nn
IX,IY,HL 21nn ED68nn/42nn  HL/IX/IY ED63nn/22nn
   SP    31nn ED78nn          SP    ED73nn  LD SP, HL/IX/IY

      A  B  C  D  E  H  L (HL) n (IX,IY+d)  nop  00
ADD r 87 80 81 82 83 84 85 86 C6n  86d       CPL  2F
ADC r 8F 88 89 8A 8B 8C 8D 8E CEn  8Ed       NEG  ED44
SUB r 97 90 91 92 93 94 95 96 D6n  96d       CCF  3F
SBC r 9F 98 99 9A 9B 9C 9D 9E DEn  9Ed       SCF  37
AND r A7 A0 A1 A2 A3 A4 A5 A6 E6n  A6d       HALT 76
XOR r AF A8 A9 AA AB AC AD AE EEn  AEd       DI   F3
OR  r B7 B0 B1 B2 B3 B4 B5 B6 F6n  B6d       EI   FB
CP  r BF B8 B9 BA BB BC BD BE FEn  BEd       IM0  ED46
INC   3C 04 0C 14 1C 24 2C 34      34d       IM1  ED56
DEC   3D 05 0D 15 1D 25 2D 35      35d       IM2  ED5E

            BC   DE   HL   SP IX,IY       AF BC DE HL IX,IY
ADD HL,rr*  09   19   29   39          PUSH F5 C5 D5 E5  E5
ADC HL,rr  ED4A ED5A ED6A ED7A           POP F1 C1 D1 E1  E1
SBC HL,rr  ED42 ED52 ED62 ED72
INC rr      03   13   23   33          EX AF,AF' 08
DEC rr      0B   1B   2B   3B          EX DE,HL/IX/IY EB
 * also to IX and IY. LD HL,HL.       EX SP,HL/IX/IY E3
                                       EXX        D9
C       A  B  C  D  E  H  L (HL)(IX,IY+d)
B RLC r 07 00 01 02 03 04 05 06   06d       RLCA  07
  RRC r 0F 08 09 0A 0B 0C 0D 0E   0Ed       RRCA  0F
P RL  r 17 10 11 12 13 14 15 16   16d       RLA   17
R RR  r 1F 18 19 1A 1B 1C 1D 1E   1Ed       RRA   1F
E SLA r 27 20 21 22 23 24 25 26   26d       RLD  ED6F
F SRA r 2F 28 29 2A 2B 2C 2D 2E   2Ed       RRD  ED67
I SLL r 37 30 31 32 33 34 35 36   36d       DAA   27
X SLR r 3F 38 39 3A 3B 3C 3D 3E   3Ed

           Z   NZ    C   NC    M   P   PE   PO   (HL,IX,IY)
  JP C3nn CAnn C2nn DAnn D2nn FAnn F2nn EAnn E2nn    E9
CALL CDnn CCnn C4nn DCnn D4nn FCnn F0nn E8nn E0nn
 RET C9   C8   C0   D8   D0   F8   F0   E8   E0
  JR 18d  28d  20d  38d  30d    DJNZ = 10d
```

APPENDIX D MACHINE CODES FOR ENCODING--HEX

	A	B	C	D	E	H	L	(HL)		
IN r, (C)	ED78	ED40	ED48	ED50	ED58	ED60	ED68	ED70	IN A,(n)	DBn
OUT (C),r	ED79	ED41	ED49	ED51	ED59	ED61	ED69	ED71	OUT(C),A	D3n

IND	EDAA	INI	EDA2
OUTD	EDAB	OUTI	EDA3
INDR	EDBA	INIR	EDB2
OTDR	EDBB	OTIR	EDB3

E		A	B	C	D	E	H	L	(HL)	(IX,IY+d)		
D	BIT 0	47	40	41	42	43	44	45	46	d46	RST 0	RESET C7
	1	4F	48	49	4A	4B	4C	4D	4E	d4E	RST 8	ERROR CF
P	2	57	50	51	52	53	54	55	56	d56	RST 16	PRINT D7
R	3	5F	58	59	5A	5B	5C	5D	5E	d5E	RST 24	GET C DF
E	4	67	60	61	62	63	64	65	66	d66	RST 32	NXT C E7
F	5	6F	68	69	6A	6B	6C	6D	6E	d6E	RST 40	FP CL EF
I	6	77	70	71	72	73	74	75	76	d76	RST 48	BC SP F7
X	7	7F	78	79	7A	7B	7C	7D	7E	d7E	RST 56	KEYBD FF

E		A	B	C	D	E	H	L	(HL)	(IX,IY+d)
D	RES 0	87	80	81	82	83	84	85	86	d86
	1	8F	88	89	8A	8B	8C	8D	8E	d8E
P	2	97	90	91	92	93	94	95	96	d96
R	3	9F	98	99	9A	9B	9C	9D	9E	d9E
E	4	A7	A0	A1	A2	A3	A4	A5	A6	dA6
F	5	AF	A8	A9	AA	AB	AC	AD	AE	dAE
I	6	B7	B0	B1	B2	B3	B4	B5	B6	dB6
X	7	BF	B8	B9	BA	BB	BC	BD	BE	dBE

E		A	B	C	D	E	H	L	(HL)	(IX,IY+d)
D	SET 0	C7	C0	C1	C2	C3	C4	C5	C6	dC6
	1	CF	C8	C9	CA	CB	CC	CD	CE	dCE
P	2	D7	D0	D1	D2	D3	D4	D5	D6	dD6
R	3	DF	D8	D9	DA	DB	DC	DD	DE	dDE
E	4	E7	E0	E1	E2	E3	E4	E5	E6	dE6
F	5	EF	E8	E9	EA	EB	EC	ED	EE	dEE
I	6	F7	F0	F1	F2	F3	F4	F5	F6	dF6
X	7	FF	F8	F9	FA	FB	FC	FD	FE	dFE

The following commands may be unsupported:

	A	B	C	D	E	n	LIX,LIY	HIX,HIY
LD H(IX,IY)	67	60	61	62	63	26	65	--
LD L(IX,IY)	6F	68	69	6A	6B	27	--	6C

	H(IX,IY+d)	L(IX,IY+d)		H(IX,IY+d)	L(IX,IY+d)
LD A	7C	7D	ADD	84	85
LD B	44	45	ADC	8C	8D
LD C	4C	4D	SUB	94	95
LD D	54	55	SBC	9C	9D
LD E	5C	5D	AND	A4	A5
INC	24	2C	XOR	AC	AD
DEC	25	2D	OR	B4	B5
			CP	BC	BD

APPENDIX E DECIMAL/HEX CONVERSION TABLES

ADDRESS CONVERSION

HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL	HEX	DECIMAL
0		0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	1	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15
	65535		4096		255		15

NEGATIVE NUMBER CONVERSION TABLE

	0	1	2	3	4	5	6	7	8	9
0	0	255	254	253	252	251	250	249	248	247
	00	FF	FE	FD	FC	FB	FA	F9	F8	F7
10	246	245	244	243	242	241	240	239	238	237
	F6	F5	F4	F3	F2	F1	F0	EF	EE	ED
20	236	235	234	233	232	231	230	229	228	227
	EC	EB	EA	E9	E8	E7	E6	E5	E4	E3
30	226	225	224	223	222	221	220	219	218	217
	E2	E1	E0	DF	DE	DD	DC	DB	DA	D9
40	216	215	214	213	212	211	210	209	208	207
	D8	D7	D6	D5	D4	D3	D2	D1	D0	CF
50	206	205	204	203	202	201	200	199	198	197
	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5
60	196	195	194	193	192	191	190	189	188	187
	C4	C3	C2	C1	C0	BF	BE	BD	BC	BB
70	186	185	184	183	182	181	180	179	178	177
	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1
80	176	175	174	173	172	171	170	169	168	167
	B0	AF	AE	AD	AC	AB	AA	A9	A8	A7
90	166	165	164	163	162	161	160	159	158	157
	A6	A5	A4	A3	A2	A1	A0	9F	9E	9D
100	156	155	154	153	152	151	150	149	148	147
	9C	9B	9A	99	98	97	96	95	94	93
110	146	145	144	143	142	141	140	139	138	137
	92	91	90	8F	8E	8D	8C	8B	8A	89
120	136	135	134	133	132	131	130	129	128	
	88	87	86	85	84	83	82	81	80	

APPENDIX F BIBLIOGRAPHY

Baker, Toni, "Mastering Machine Code on Your ZX81", Reston Publishing Co. Inc., 11480 Sunset Hills Rd., Reston, Va. 22090 $12.95.

Carr, Joseph P., "Timex Sinclair 2068,1500 and 1000 Machine Language Programming and Interfacing", Reston Publishing Co. Inc., 11480 Sunset Hills Rd., Reston, Va. 22090.

Corcoran, V. C., and Branigin, M. H., "Timex Sinclair 2068 Personal Color Computer Technical Manual", Timex Computer Corporation, Waterbury, CT. 06720 $25.00.

Dreger, Lloyd H., "The Timex/Sinclair 2068 ROM Manuscript", S.M.U.G., Box 101, Butler, WI. 53007 $16.95.

Leventhal, Lance A., "Z80 Assembly Language Programing", Osborn/McGraw Hill, 2600 10th St., Berkley, CA. $18.95.

Logan, Dr. Ian, & O'Hara, Dr. Frank, "The Complete Timex TS1000/Sinclair ZX81 ROM Disassembly", Melbourn House Publisher.

Mazur, Jeff, "Timex Sinclair 2068 Intermediate/Advanced Guide", Howard W. Sams & Co. Inc., 4300 W 62nd St., Indianapolis, IN. 46268 $9.95.

Naylor, Jeff., and Rogers, Diane, "Inside The Timex Sinclair 2000 Computer", Sunshine Books, 12-13 Little Newport St., London WC2R 3LD $11.95.

Spracklen, Kathe, "Z-80 Assembly Language Programing", Osborn/McGraw Hill, 2600 10th St., Berkley, CA. 94710 $9.70.