PEEK N' POKE

FRED CORNETT
MANAGING EDITOR
THE CURSOR GROUP

Written by Brett Bilbrey

PEEK n' POKE

Written by: Brett Bilbrey
~~FRED CORNETT~~
The Cursor Group

There has been a great deal of confusion as to exactly what PEEK n' POKE are and what they do.

Your computer consists of many thousands of simple circuits that are capable of registering their state (or condition). That state can only be one of two conditions: ON or OFF. The computer is capable of checking the status of any individual bit of memory location. A byte is one memory location. A byte is composed of 8 bits. The computer uses symbol substitution to communicate with you. If any particular bit is turned 'ON' it substitutes a "1", if it is 'OFF', a Ø is substituted. A program is a set of instructions that is placed in memory by a set pattern of one's and zero's. This is the language your computer understands. It is called "BINARY".

If we had to input our programs in one's and zero's, it would be very difficult and take forever. So, there is an intermediate language called "Assembly Language". Each type of CPU (Central Processing Unit) chip speaks a different language. Our CPU chip is a Z80. The instructions we give our Z80 are called "Mnemonics". These mnemnonics are an abreviation of assembly language. Assembly language exists only for the convenience of the programer, who can look at a program on paper and understand the logic flow.

For a computer to understand assembly language, it must have a program called an assembler. Our computer does not. So for us to force an assembly language program into our computer, we must buy a book that translates Z80 mnemonics into "OP Code".

An "OP Code" is our instruction which has been coded into a hexidecimal number (Base 16). Decimal numbers are Base 10, binary is Base 2, and OP Code (Hex) is Base 16. A book of Z80 OP Codes may be purchased in most any computer store.

Hold on, you aren't done yet!! Remember, our computer does not have an assembler, so it can't understand our "OP Code" instructions (Hex) until we convert them to decimal.

The average computer newcomer is at this moment muttering "To heck with it!!" Why Bother you ask? As you know, "BALLY BASIC" is somewhat limited, machine language is not!!! Stick with us, we have one more major problem to solve.

OK, we've written an assembly language program, converted that to mnemonics and OP Code. So how in the world do we get it into the computer? Well, it's been hard up till now, but the good person who wrote 'Tiny Basic' (Mr. Jay Fenton) must have liked us, because he gave us three very powerful commands: PEEK, POKE, and CALL. (Be thankful for these commands, many manufacturers don't give them to their users).

"PEEK" is a command that lets you look at any specific memory address and find out what is stored there. In our February 1980 issue, page 14, we printed a 'Bally Memory Map'; use this map to find memory locations. NOTE: 'PEEK' accesses memory 2 bytes at a time. Memory is normally accessed one byte at a time, but Bally pulled some dirty tricks which has had some good results and some bad results. For example: key in the following one line program

10 A=26 "DO NOT HIT RUN"

BALLY TEXT AREA: -24576 TO -22777

Lets use the 'PEEK' command to see how the program is stored.

The area where your Basic Program is stored is called the "Bally Text Area" and starts at -24576 and decrements by two's to -22777. Hence, the beginning of our program should be at location -24576. INPUT the following one line program without a line # , and press "GO":

PRINT %(-24576)

Your computer has printed "10". Remember, all line numbers occupy 2 bytes regardless of size of line #. When we use 'PEEK', we are addressing 2 bytes at a time.

Lets try finding the rest of the program. INPUT the following one line program without a line #.

```
PRINT %(-24574)
```

Your computer has printed "15681".

What in the world is that??? We told you that we are 'PEEKING' 2 bytes at a time, so now we must separate the 2 bytes from each other.

When the computer stored your program initially, it pulled a dirty trick. It multiplied the second characters ASCII Code time 256 and added the first characters ASCII Code to the total and 'POKED' it in together.

Referring to the ASCII Conversion Chart in our June issue, page 39, ASCII Code for "A" is 65; the ASCII Code for "=" is 61. Lets see if this works out: 256 x 61= 15616 + 65= 15681 , which is what our computer printed out when we told it to "PRINT %(-24574). You can check this on your computer by keying in the following without a line #:

```
X=%(-24574)÷256;TV=RM;TV=X
```

RM is a variable that contains the remainder, if any, of any division problem.

The computer will now print our "A=". So far, we have found that location -24576 contained the line number (10) and location -24574 has stored "A=".

Using the same procedure, lets look at the next location. Key in the following one line program without a line # and press "GO":

```
PRINT %(-24572)
```

Your computer will print "13874". Lets convert that by our division method:

```
X=%(-24572)÷256;PRINT RM,X
```

The computer will print "50 54". These are the next two characters stored using ASCII Codes. Using our ASCII Chart, page 39, of the June issue: '50'=2, '54'=6. So, now we have found "10 A=26". I'll bet you think we are done, we aren't!! What is the last thing we do on a line? We hit "GO". This is always stored at the end of each line. Lets look at the next location -24570. Key in the following one line program without a line # and press "GO":

PRINT %(-24570)

Your computer will print "13". Checking our ASCII Chart we find '13=GO'. Now we have the complete program: 10 A=26.

We are now in a better position of understanding how many bytes our programs use:

LINE #	occupies 2 bytes	'2'	occupies 1 byte
'A'	occupies 1 byte	'6'	occupies 1 byte
'='	occupies 1 byte	'GO'	occupies 1 byte

TOTAL : 7 bytes

If you have not "RUN" the program and you key in "PRINT A" and press "GO", the computer will print "Ø". Now, RUN the program. Key in PRINT A and press "GO" and the computer will print "26".

Lets look again at the memory map on page 14 of the February issue. We find that variable locations begin at 20078, and since we know that all variables use 2 bytes, we are able to ascertain the memory locations for all of the variables: A=20078 - add two bytes, B=20080 - add two bytes, and C=20082, etc....

Changing a variable is pretty easy, as in the case of our LINE 1Ø program. When you key in A=26 (and hit RUN) the computer goes to location 20078 and places "26" in that location. The "BASIC" is acting as an interpreter between us and the computer. Lets eliminate the middle man and do it ourselves! Key in this:

%(20078)=40 Press "GO"

Now key in "PRINT A", press "GO". The computer will print out "40". "%(20078)=40" is a "POKE Command". It allows us to "POKE" our own information directly into a computer location without using the "BASIC" interpreter. Do you remember the Music in our March issue, or "Connect Four" in the August issue? Both of those programs made heavy use of "PEEK n' POKE".

OK fine, this is interesting and cute, but why use it? Lets look at the following example. Key in the following one line program:

10 A=22104;B=6109;C-19867;D=4107;E=9987;F=-31063

Hit "GO" and key in PRINT SZ. This took up 49 bytes. All we wanted to do is store 6 numbers. Since we only have 1800 bytes, we should use as economical a method as possible. The cheapest way of doing this is by using a "REM Statement".

10.1234567890123

Why did we add 14 numbers after the period? Lets figure it out. We are going to store 6 large numbers. Each of these numbers will require a 2 byte location for storage. In the previously explained process, we learned that the computer stores 2 different ASCII Codes in one 2 byte location, so we multiply 2 x 6 and get 12 ASCII characters. Now, we cannot 'POKE' into the line number location, which is -24576, or the location where the period (.) is -24574. Even though the period occupies only one byte, we have to fill in that location with another ASCII character (which we won't use) -- then reserve 12 bytes (for 6 numbers).

VALUE	LINE # (don't POKE)	REM (don't POKE)	VALUE 22104	VALUE 6109	VALUE 19867	VALUE -4107	VALUE 9987	VALUE -31063	"GO" is stored here
	10	.1	23	45	67	89	01	23	
MEMORY LOCATION	-24576	-24574	-24572	-24570	-24568	-24566	-24564	-24562	-24560

INPUT the following line: 10 .1234567890123

Now INPUT the following program without a line number:

FOR A=-24572TO-24562STEP 2;INPUT %(A);NEXT A

The computer will print "%(A)" and wait! It is waiting for you to INPUT the 6 values, pressing "GO" after each one: 22104, 6109, 19867, -4107, 9987, -31063. After you have done this, key in "LIST". The computer will print:

10 .1XV???M???'??

The reason the computer printed garbage in LINE 10 is; it is dividing by 256 and printing the result of an ASCII Code. Whenever an ASCII Code is not a standard ASCII Character, our computer prints "?".

To make sure we really have "POKED" those 6 values into LINE 10, lets "PEEK" at them. INPUT this one line program without a line # and hit "GO":

FOR A=-24572TO-24562STEP 2;PRINT %(A);NEXT A

The computer will print:

22104 6109 19867 -4107 9987 -31063

Remember when we used the old method of storing this info: A=22104; B=6109; etc? When we keyed in "PRINT SZ" it printed "1751", which means the old way took up 49 bytes. NOW key in "PRINT SZ". The computer will print "1783". A savings of 32 bytes (and we are only using 6 numbers). Think of the saving if you had 100 numbers!!!

I generally find myself in a high state of aggitation with people who gripe a lot about things they can't change. As an American, I have found that the best way to beat the system is not to become an anarchist and destroy it, but to work within the system and get it to work for you!! This can be beautifully applied to the "BALLY". An example:

Many of us have been frustrated at the 'apparent' inability of our "BALLY"

to store decimal formatted numbers (i.e., $127.10) in one variable or location. We have learned in this tutorial that 2 ASCII Codes can be stored in one 2 byte location and yet be separated by multiplying one of them by 256 (x256) and adding the other to the total.

We have one limitation: ALL VALUES MUST FALL BETWEEN 32767 and -32767. Suppose we are trying to put a checkbook program together and we want to store dollars and cents as one value in string variables.

Using the logic that Jay Fenton used in storing two characters in one location, lets pull our own dirty trick. INPUT the following program:

```
10 INPUT D,C
20 T=Dx1ØØ+C
30 PRINT #1,T,"+$",T÷1ØØ,".",RM
```

Now RUN this program. When the computer prints "D" INPUT 127; when the computer prints "C" INPUT 10. The computer will now print:

```
12710=$127.10
```

What we are doing is this: D=Dollars, C=Cents. We are multiplying Dx1ØØ and adding "C" to it to create "T" (for Total). What we have done is store Dollars and Cents in one variable, which could just as easily been one string variable (@(1)) or one memory location.

To get the amount back out, we divide the total "T" by 1ØØ, which will give us Dollars and we get the Cents amount from the variable "RM". Easy isn't it?

Remember the limitation we spoke of? We cannot use any values larger than 32767, which in our case boils down to $327.67. That wouldn't have been much of a drawback in 1955 but it sure is now. Lets play some dirty tricks of our own. Key in the following program:

```
10 INPUT D,C
20 IF D>326D=D-326;T=Dx1ØØ+C;T=(-T);GOTO 4Ø
30 T=Dx1ØØ+C
40 PRINT T
50 IF T#ABS(T)T=ABS(T);D=T÷1ØØ+326;C=RM;GOTO 7Ø
60 D=T÷1ØØ;C=RM
70 PRINT #1,"$",D,".",C
```

Now we can handle any amount LOWER than $653.68. I don't find this much of of a limitation; if we were writing a checkbook program, I doubt if many of us would write more than one or two checks exceeding $653.67 per month. Whenever we need to write an amount for more than that, just break it into two checks instead of one. Lets go through that program step-by-step:

LINE 1Ø: If dollar amount (D) is larger than 326, subtract 326 from dollar (D) amount. Total (T) equals dollar (D) amount multiplied by 1ØØ and add cent (C) amount. Set a flag to notify computer that this amount is more than $326 by making the total (T) negative (-T). Skip line 30.

LINE 3Ø: If dollar amount is less than $327 total, (T) equals dollar amount (D) times 1ØØ plus cent amount (C).

This ends the INPUT portion of the program.

LINE 5Ø: If the total (T) is a negative (-) number, change it to positive (+). Dollar amount (D) will equal total (T) divided by 1ØØ and restore (add) 326 to it. (This is how we get back the 326 we subtacted in line 20). Cents amount becomes the remainder (RM) left over after dividing by 1ØØ. Skips line 60.

LINE 6Ø: If total (T) was not negative (meaning over $326 orginally), dollar amount (D) equals total (T) divided by 1ØØ. Cent amount (C) equals remainder (RM) left over after division.

LINE 7Ø: The "#1" used in this print statement is a tab function telling the computer how many spaces to place between the separate types of print. For 3 spaces we would have used "#3".

Lets look at alternate ways of storing our check information. Since this is a "PEEK n' POKE MANUAL, lets do just that!

Remember the earlier exercise which uses a 'REM' Statement? Since our BASIC does not allow 'DATA' Statements, we have to create our own. This is what we did with our "Three Voice Music Assembler", VOL I, Issue #3 (March), and also the "Connect Four", VOL II, Issue #1 (August). This is a far more stable way to store data than in the string arrays. Also, it is very easy to store on tape.

Lets say we want to store 10 check amounts! INPUT the following changes to the preceeding program:

```
 2 .123456789Ø123456789Ø12
 5 CLEAR;FOR P=-23572TO-24554STEP 2
40 %(P)=T;NEXT P
42 FOR P=-24572TO-24554STEP 2
45 T=%(P)
80 NEXT P
```

Now you can store 10 check in line #8. RUN the program and input 10 different dollar and cent values. If you want to reprint the values of these checks after the program has been run, key in GOTO 42 and hit "GO".

If we wanted to view the checks individually, we would make the following changes:

```
42 CLEAR;INPUT "WHAT CHECK # DO YOU WISHÇÇTO SEE?
   (1-10)"N
45 T=%(Nx2+(-24574))
70 PRINT #1,"CHECK #",N,"=$",D,".",C
80 FOR Z=1TO 2ØØØ;NEXT Z;GOTO 42
```

Now, to view a particular check, you input the number between 1 and 10. The key here is is LINE 45, which is doing the following:

Whenever you want to add to a negative number you decrement it: TOTAL = (-24574) PLUS 2 TIMES the number of the check. If we wanted Check #2 its memory location would be: -24574 + 2xN or -24570. Our first memory location that we can use on Line #5 is -24572, the second would be -24570. See, it all checks out. (LINE 80 is merely a timing loop).

Now, lets add a few lines that will give us a total of all the checks:

```
42 X=Ø;Y=Ø;FOR P=-24572TO-24554STEP 2
45 T=%(P)
70 X=X+D;Y=Y+C;PRINT #1,"$",D,".",C
80 NEXT P
90 IF Y>99X=X+Y÷1ØØ;Y=RM
100 PRINT #1,"TOTAL=$",X,".",Y
```

Since the computer is not aware of the differance between dollars and cents we must supply a way of adding these items seperately. We also need a way of correcting the dollar and cent totals if the cent total exceeds 99.

LINE 42: Let X=Total Dollar Counter; Let Y=Total Cent Counter

LINE 70: Add the individual check amounts (D and C) to the

total counters (X and Y)

LINE 90: If total cents (Y) exceeds 99, divide total cents (Y) by 100 and add this to total dollar counter (X). Let total cents (Y) be the remainder (RM) of that division.

You could also establish 'Payee Codes' and store the 'Payee Names' in a REM Line. Also, the complete date could be stored in one location: 12/31/1980 would be 12310 (using only the last digit of the year). REMEMBER: get as much mileage as possible out of each memory location.

EXAMPLE: We would need to know if an individual check has been reconciled with our statement. The easiest way to handle that would be to make the date a negative number if the check has been reconciled.

I would also make the payee code # location also contain the tax code. All of this would not use too much memory:

AMOUNT = 2 bytes

DATE/RECONCILE = 2 bytes

PAYEE CODE/TAX STATUS = 2 bytes

That's only a total of 6 bytes per check.

MACHINE LANGUAGE

This manual is not intended as a course in Z80 Machine Language or Assembly Language. To write a decent machnine language course would occupy the better part of a year, and subsequently because of its limited appeal to the average Bally user be priced totally beyond the value of the information contained.

There are numerous excellent paperback books written on that subject. Cursor does recommend the following books which may be purchased locally at a computer store.

> Z-80 and 8080 ASSEMBLY LANGUAGE PROGRAMMING by Kathe Spracklen, published by Hayden (15BN Ø-81Ø4-5167-Ø); Library of Congress Catalog Card Number 79-65355 for approximately $8.95.
>
> Z80 INSTRUCTION HANDBOOK by Nat Wadsworth, published by SCELBI Publications for approximately $4.95.

We strongly suggest you purchase both of these books! If they are not available locally, contact: OPAMP/TECHNICAL, 1033 North Sycamore Ave., Hollywood, CA 90038. Telephone: (213)-464-4322 for mail order service.

The purpose of this portion of the manual is too acquaint you with the methods required by the Bally for you to utilize your own machine language routines. To utilize user supplied routines you must adhere to several rigid rules, the most important of which is:

A machine language program that is used in conjunction with the Bally Basic Cartridge must be stored in a location that will not interfere with the Basic Text! If we were to store a machine language program in the text area (-24576 thru -22777), the Basic Language would attempt to interpret our machine code as BASIC, thereby driving you crazy with "What and How" messages. Therefore, you must find locations that will not interfere.

The easiest to use are the "Tape Input Buffer" (20002 - 20049) and the "Line Input Buffer" (20180 - 20283). There is also an 18 byte area (20144 - 20161) that can be utilized as long as you are not using the OnBoard Calculator routine ($).

Slightly more difficult to use would be the "Screen Memory Area" (16384 - 20479). The problem with this area is that graphics and text concurrently occupy

the locations. Remember the "CRITTER" program in the October 1980 issue of Cursor? When you ran that program you had an area at the bottom of the screen that was twinkeling. That was where the Machine Language program was stored (19584 thru 19880). This works great, but you cannot put any graphics or text in the same area. That is why you had to keep the Cursor from scrolling down to that location. If you were to scroll down there, it would destroy the machine language.

If you check most of the machine language programs we have printed, you will see that we stored them in the Line Input Buffer (20180 thru 20283): February 1980 issue of CURSOR page 10, Line 1Ø1Ø "M=20180"; March 1980 issue page 22, Line 50 "M=20180"; October 1980 issue page 71, Line 20 "A=20180". In each case the starting location is 20180.

When doing Machine Language programming, we cannot stress enough the importance of a "TI PROGRAMMER CALCULATOR". It isn't cheap ($59.00) but it will save hundreds of hours of work. Most of the time spent in programming is changing from Binary to Hexadecimal and Decimal and back again. By purchasing this calculator, which is designed exclusively for machine language programmers, you eliminate all that work.

The binary form of number representation is the basis of computer operations. It requires the use of only two digits: Ø and 1. These two digits are represented by voltages in the computer, a low voltage (Ø) and a hight voltage (1). The following is a representation of the numbers Ø through 5 written in binary form: Ø=Ø, 1=1, 10=2, 11=3, 100=4, 101=5. Notice how rapidly the numbers get very long. Let's tackle some larger numbers.

In the Decimal System (base 10) each digit represents a power of 10. For example:

$$423 = 4 \times 100 \quad \text{or} \quad 4 \times 10^2$$
$$+2 \times 10 \quad \text{or} \quad 2 \times 10^1$$
$$+3 \times 1 \quad \text{or} \quad 3 \times 10^0$$

Any number raised to the Ø power equals 1, i.e., $2^0=1$, or $16^0=1$, etc.

In the Binary System (base 2), each digit represents a power of 2. For example:

1101 =	1 x 2^3	or	1 x 8
	+ 1 x 2^2	or	1 x 4
	+ 0 x 2^1	or	0 x 2
	+ 1 x 2^0	or	1 x 1
BINARY	1101	=	13 DECIMAL

So, 1101 in binary is the equivalent of 13 in decimal. Conversion between the two number systems can be done using these rules, but for our purposes, we would normally convert binary to hexadecimal.

SMALL NUMBER CONVERSION TABLE

DECIMAL	BINARY	HEXADECIMAL
Ø	Ø	Ø
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

In the hexadecimal (base 16) number system, there are 16 different digits. The digits Ø thru 9 are borrowed from the decimal system and letters of the alaphabet fill in the other six. In the hexadecimal system, each digit represents a power of 16. For example:

$$
\begin{array}{lll}
1B3 = 1 \times 16^2 & \text{or} & 1 \times 256 \\
\quad + B \times 16^1 & \text{or} & 11 \times 16 \\
\quad + 3 \times 16^0 & \text{or} & 3 \times 1 \\
\hline
\text{HEX} \quad 1B3 & = & 435 \quad \text{DECIMAL}
\end{array}
$$

POWERS OF 16 CHART

$$
\begin{aligned}
16^0 &= 1 \\
16^1 &= 16 \\
16^2 &= 256 \\
16^3 &= 4{,}096 \\
16^4 &= 65{,}536
\end{aligned}
$$

Lets convert 2AF3 to decimal. 2AF3 is composed of four numbers, so we count down four places on the Powers of 16 Chart for our first number:

$$
\begin{array}{lll}
2 \times 16^3 = 2 \times 4096 & \text{or} & 8192 \\
+ A \times 16^2 = 10 \times 256 & \text{or} & 2560 \\
+ F \times 16^1 = 15 \times 16 & \text{or} & 240 \\
+ 3 \times 16^0 = 3 \times 1 & \text{or} & 3 \\
\hline
2AF3 \quad \text{HEX} = & & 10995 \quad \text{DECIMAL}
\end{array}
$$

If our hexadecimal number had been 2AF we would count down three places on the powers of 16 chart for our first number:

$$
\begin{array}{lll}
2 \times 16^2 = 2 \times 256 & \text{or} & 512 \\
+ A \times 16^1 = 10 \times 16 & \text{or} & 160 \\
+ F \times 16^0 = 15 \times 1 & \text{or} & 15 \\
\hline
2AF \quad \text{HEX} = & & 687 \quad \text{DECIMAL}
\end{array}
$$

To convert decimal to hexadecimal, we divide the decimal number by the largest power of 16 equivalent that will fit. Example: lets convert 10999 to Hex

```
          2
4096 ) 10,995
       8,192
       2,903 remainder
```

Our first Hex number is "2". Now divide the remainder by the next lower power of 16 equivalent:

```
         10
256 ) 2803
       256
        243 remainder
```

Our second Hex number is "A". Looking back on our conversion chart we find 10 = A. Now we divide the remainder by the next lower power of 16 equivalent.

```
         15
16 ) 243
       16
       83
       80
        3 remainder
```

Our third Hex number is "F". Looking back on our conversion chart we find 15 = F. Now, our final remainder becomes our fourth Hex number "3".

10995 DECIMAL = 2AF3 HEXADECIMAL

All of this has a tendency to overwhelm the beginner, but the concepts are easily grasped if you'll stick with it, the rewards are tremendous!

There is one additional stickler. Our BASIC cannot handle any decimal number larger than 32,767. But it can handle the same number in negative form. In other words, it can handle the decimal range of -32767 thru 32767 which gives us the full range of 65,536. If you are going to use a hex number larger than 7FFF it will when converted to decimal be larger than 32767. We must therefore convert that number to a negative number that will be accepted by our computer. The easiest way to do that is with the following program:

HEX TO DECIMAL CONVERTER

```
 9 PRINT "HEX # ?",
1Ø FOR A=1TO 4
2Ø @(A)=KP
3Ø IF @(A)>47IF @(A)<58TV=@(A);@(A)=@(A)-
   48;NEXT A
35 IF @(A)>64IF @(A)<71TV=@(A);@(A)=@(A)-
   55;NEXT A
37 IF @(1)<16GOTO 6Ø
4Ø GOTO 2Ø
6Ø B=4Ø96; T=0;FOR A=1TO 4
65 IF A=1IF @(A)>7GOSUB 1ØØ;NEXT A
7Ø IF A=1IF @(A)<8@(1)=@(1)xB;T=T+@(1);NE
   XT A
75 B=B÷16;@(A)=@(A)xB;T=T+@(A)
8Ø NEXT A;GOTO 15Ø
1ØØ T=-32767;IF @(A)=8RETURN
11Ø T=T+((@(A)-8)x4096)-1;RETURN
15Ø PRINT ;PRINT #1,"DEC. EQUIV=",T
16Ø GOTO 9
```

REMEMBER TO REVERSE HEX PAIR ORDER PRIOR TO INPUT.

Why On-Board ROM Sub-routines?

We can fairly simply write small machine language programs that will do anything you want but, like any type of program, they take up memory space. We are very fortunate that a manual exists explaining machine language routines that are built into our unit already. These routines, when properly utilized, require us merely to call them. Example: Page 6, Bally On-Board ROM Sub-routines, Subroutine #48:

48 026A E,D,C,B,L,H SCROLL
Block moves. Moves C bytes from (HL+DE) to (HL). Increments HL by DE and repeats B times.

Lets look at the first line: '48' is the sub-routine number. '026A' is the hexadecimal address of this subroutine (618 decimal). 'E,D,C,B,L,H' are the Z80 registers that must be utilized and the order in which they must be loaded. Looking at this again we will be using registers DE, C, B, HL.

Whenever we use a machine language routine, we must use a "CALL" to tell the computer where to go and also to notify it that the information will not be in BASIC. Sometimes, when using a CALL, the computer can't find its way back, and

your keyboard will lock up and do funny things. For this reason, it usually wise to "SAVE THE BASIC POINTER" as the very first thing, and RETURN TO BASIC as the last.

Lets put together a machine language program using the scroll subroutine #48. First we will write the program in machine language (OP Code).

OP CODE	ASSEMBLY LANGUAGE	COMMENTS
D5	PUSHD	Save Basic Pointer
FF	RST 56	On Board Subroutine Notification
31		Subroutine 48 + 1 Converted to Hex

Lets look again at the subroutine: it is saying that HL must be loaded with the screen address of the first line we want to scroll. Please refer to the DMA GRAPHICS article on page 25 of the April/May issue for an explanation of screen address locations. I selected 18424 as the location I wanted. Onward..... Our subroutine says it increments HL by DE. That means the increment is to be stored in DE. Well, we want to scroll in one line increments, and referring back to the April/May article we know that one full line on our TV is 40 bytes. Therefore, we know that we want 18424 in HL and 40 in DE.

Looking back at our subroutine it says "Moves C bytes from (HL + DE) to HL". Therefore, 'C' would have to contain the number of bytes on a line you want to move. Lets move half a line. Half of 40 is 20. Therefore 'C' must contain 20.

Now for the last leg. Refer back to the subroutine: "Repeat B times". Therefore, 'B' would be the total number of lines we want to move upward. In this case lets move 20 lines.

OK! HL = 18424; DE = 40; C =20; B = 20. Now we have to convert these decimal values to hexadecimal:

HL = 47F8; DE = 28; C = 14; B = 14

Several more items of necessary information before we proceed.

HL and DE are register pairs. A 'register pair' is a combination of two registers. HL = H and L; DE = D and L. The purpose is to allow larger number

handling capability. Referring back to the subroutine, it gave us the register load order (sequence): E,D,C,B,L,H

This means we must split apart DE and HL. Lets load DE first. Remember, DE=28. Each single register can handle a maximum of 2 hex numbers. In the case of DE it already is two numbers so lets precede with ØØ. DE=ØØ28. Therefore, E=28, D=ØØ. Getting back to our program.

D5 Save Basic Pointer
FF Call Subroutine number
31 48
28 E Register
ØØ D Register
14 C Register
14 B Register

Next comes HL. The subroutine tells us to load HL backwards as we loaded DE, so, HL=47F8 becomes F8 47.

F8 L Register
47 H Register
D1 POP DE; Put Basic Pointer Back
C9 Return; Go Back to Basic

Next, we must convert these Hex pairs to Hex bytes (4 at a time) and then convert Hex bytes to Decimal by using the Hex to Decimal Converter program. to get Hex bytes we must first reverse their order:

HEX PAIRS	HEX BYTES	DECIMAL
D5 FF	FFD5	-43
31 28	2831	10289
00 14	1400	5120
14 F8	F814	-2028
47 D1	D147	-11961
C9 00	00C9	201

Notice the "ØØ" in the last line. We had to add zeroes so it would fill in the space. ØØ in assembly language is known as "NOP", which means 'NO OP'; in other words - nothing.

We converted the Hex to Decimal because our computer can't understand the Hex Code. Ah, someday.....

In previous pages we discussed the locations we can store a machine language program. Now, we will discuss how to get it in there.

We will store our program in the Line Input Buffer starting at 2Ø18Ø. Look at our complete program now:

```
1Ø1Ø M=20180;B=M;C=1Ø9Ø
1Ø2Ø L=-43;GOSUB C
1Ø3Ø L=10289;GOSUB C
1Ø4Ø L=5120;GOSUB C
1Ø5Ø L=-2028;GOSUB C
1Ø6Ø L=-11961;GOSUB C
1Ø7Ø L=201;GOSUB C
1Ø8Ø FOR A=1to14;CALL B;NEXT A;STOP
1Ø9Ø %(M)=L;M=M+2;RETURN
```

Lets run through this program the same way the computer will:

Nothing happens until we get to 1Ø2Ø, we GOSUB C, which is Line 1Ø9Ø.

1Ø9Ø says:

Poke Location M with the Value of L, then increment M by 2 and return. This goes on through line #1Ø7Ø thusly:

```
%(20180)=-43
%(20182)=10289
%(20184)=5120
%(20186)=-2028
%(20188)=-11961
%(20190)=-201
```

Then we go to Line 1Ø8Ø. In this case, we want to CALL this Subroutine 14 times and then STOP. Notice we are using the Variable B, which gives the beginning location of OUR Subroutine. 'M' wouldn't help us at all because we were incrementing 'M'.

If you are still somewhat confused don't feel alone. The more you re-read, and work with the examples we print in our issues, the clearer it will become.

INTERRUPT HANDLING

This is a somewhat more complex area to handle. If you are a total beginner this section may appear to be gobbledy-gook. However, if you buy the books we have recommended and do the exercises in our issues and manuals, you will come to an understanding. Computer Programming is like any other endeavor, *you must learn to walk before you run.*

OUTPUT PORT D (HEX) (13 DECIMAL) INTERRUPT FEEDBACK:

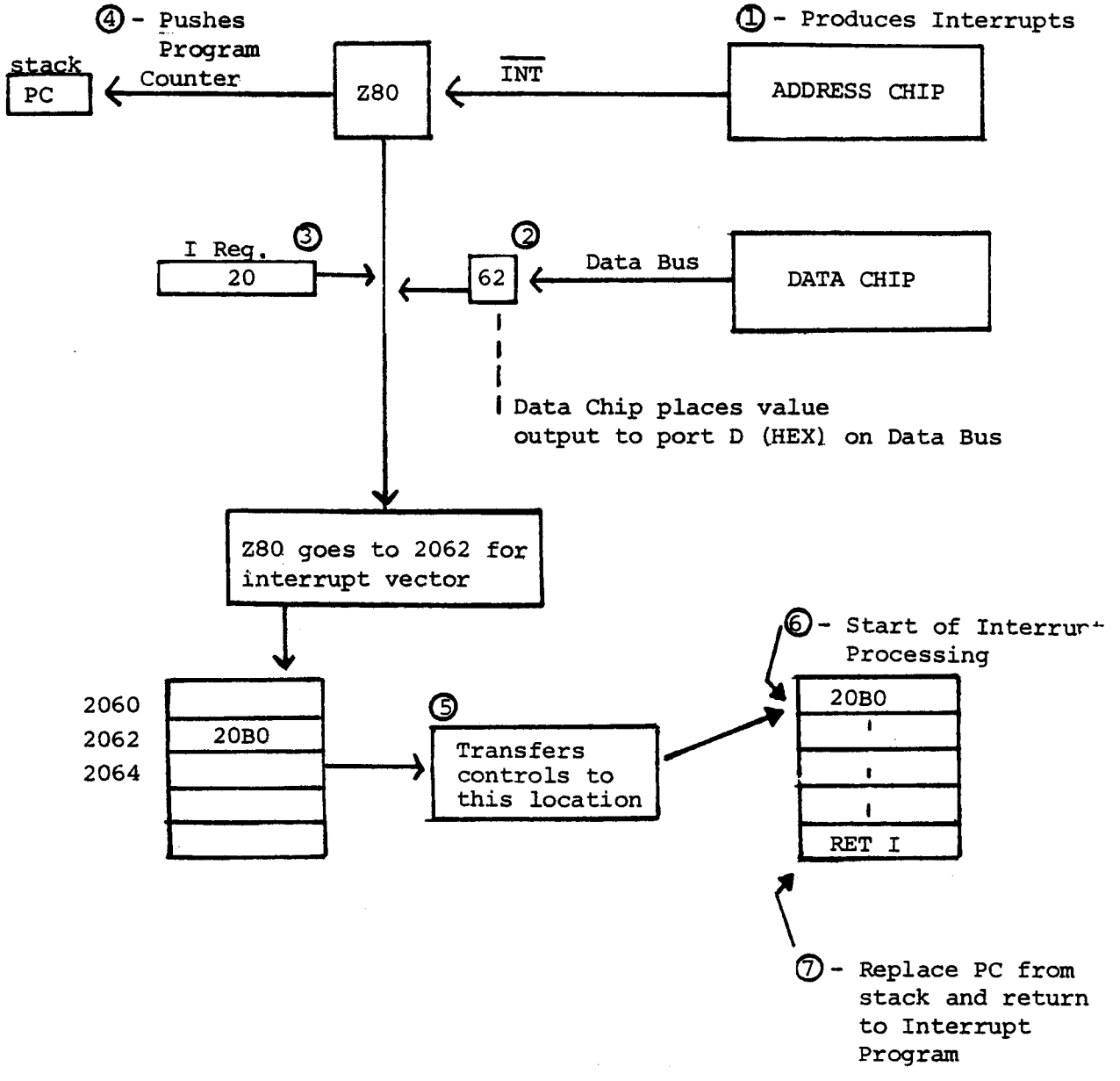
This Port works with the IM2 instructions to place on the data bus at each interrupt, the data outted to this port.

In most Z80 applications IM2 is used to determine where to go for a device interrupt. The Bally uses it to generate a location to go to at each screen interrupt.

When the Z80 receives an $\overline{\text{INT}}$ from the Address Chip, it looks to the 'I' register for the high order byte (or page) and to the data bus for the low order byte, of the address for the interrupt vector. This interrupt vector points to the interrupt processing routine. Only the upper 4 bits are used in responding to a Light Pen interrupt.

EXAMPLE IM2 INTERRUPT. This example is what the BASIC really does. BASIC's interrupt routine is at 2ØBØ (HEX).

I M 2 INTERRUPT



OUTPUT PORT E (HEX) (14 DECIMAL) INTERRUPT MODE:

The value output to this port determines what type of interrupt is to occur. There are two types of interrupts: Screen Interrupts and Light Pen Interrupts.

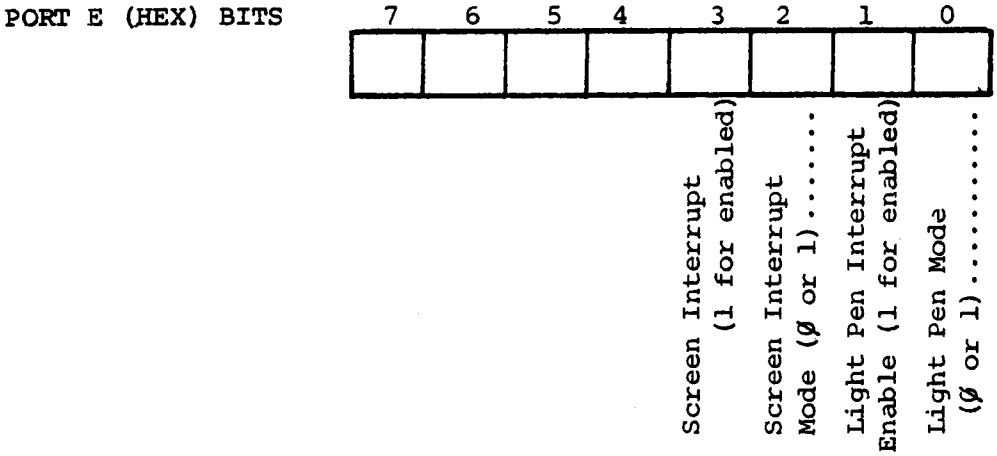
The Screen Interrupt is used to synchronize the software with the video display. The Screen Interrupt is the $\overline{INT}$ Signal sent to the Z80. The Screen Interrupt occurs when the video system completes scanning the line in the interrupt line register (output port F (HEX). This interrupt can be used for timing since each line is scanned 60 times a second.

By writing your own interrupt routines and using the 'I' register and output port D (HEX) to point to it you can put up to 256 different colors on the screen by changing the color registers each interrupt.

The Light Pen Interrupt occurs when the Light Pen Interrupt mode is set and the light pen is triggered and the video scan crosses the point on the screen where the light pen is.

There are two modes for both the Screen Interrupt and Light Pen. In mode Ø the custom chips will continue to try to interrupt the Z80 until it finally acknowledges the interrupt. In Mode '1' the custom chips give up if the Z80 does not acknowledge it by the next instruction. Both interrupts can occur if both are set but the screen interrupt has priority.

INTERRUPT CONTROL BITS



You can see from this (refer to INTERRUPT CONTROL BITS chart page 23) that if you want just a Screen Interrupt in Mode Ø (must interrupt) without the Light Pen, you would just set bit 3 and the decimal value would be 8. This is why 8 is always output to Port E in the games and in BASIC!!

OUTPUT PORT F (HEX) (15 DECIMAL) INTERRUPT LINE:

The value output to this port determines when a screen interrupt ($\overline{INT}$ to Z80) occurs. In our low resolution system only bits 1 - 7 are used with bit Ø set to zero. In low resolution there are 102 lines of 40 bytes with 16 bytes left over. Since the custom chips were designed to operate in a high resolution mode they scan 204 lines. This means that for every line of low resolution that is scanned 2 lines of high resolution were scanned. Since the reference for Port F (HEX) is for high resolution, we have to multiply the number of lines in low resolution by two for the value we output to the port. This is why bit zero is set to zero and we only use bits 1 - 7.

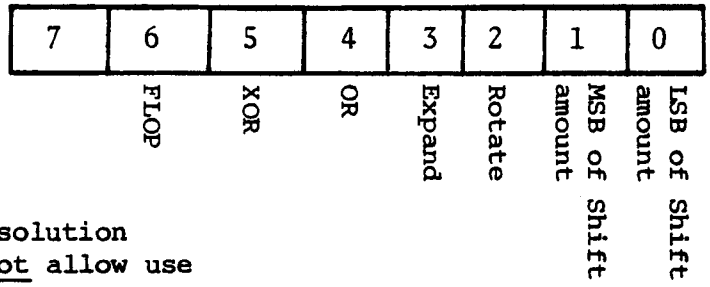
When the custom chip have finished scanning the number of lines output to Port F (HEX) a screen interrupt is generated. Each line is scanned 60 times a second and there are 256 lines per frame so 15,360 lines are scanned per second. If you divide output Port F (HEX) by 15,360 you will get the time in seconds between interrupts. EXAMPLE:

PORT F = 200 lines $200/\overline{15360}$ = .013 seconds between interrupts or 13 milliseconds

OUTPUT PORT C (HEX) (12 DECIMAL) THE MAGIC REGISTER:

When a On-Board WRITE Routine that calls for a MAGIC RESIGER value, this means that it is to modify the data before placing it in memory. This is valid only if the 'write' is from Ø to 16K. What happens is, if you write to a location between Ø and 16K. In our low resolution system this only works from Ø to 4K since we only have 4K of memory.

FUNCTIONS SET BY MAGIC REGISTER BITS



"NOTE": Low Resolution does not allow use of Rotate.

As many as four functions can be cone at one. Order of operation is as follows:

1. Expansion
**2. Rotating and Shifting
3. Flopping
**4. OR and XOR

**NOTE: Rotate and Shift, and OR and XOR cannot be set at the same time.

INPUT PORT 8 (HEX (8 DECIMAL) INTERCEPT FEEDBACK REGISTER:

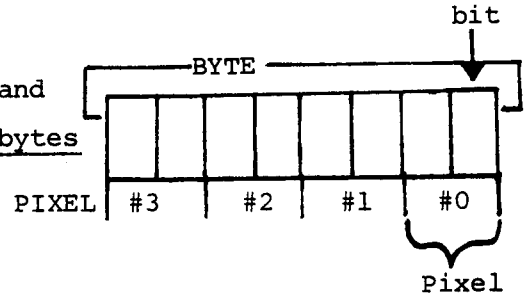
This is an Input function. By looking at this register after an OR or XOR has been performed we can determine if we have written on top of something and also where.

A '1' in the intercept register means we have written on top of something. Bits Ø - 3 give information for all OR or XOR Writes since the last input from the intercept register resets these bits. This means every time something is written into memory using an OR or XOR a check is made to see if the Write occurred over other data, if so, Port 8 (HEX) bits Ø - 3 are reset to zero.

INTERCEPT FEED BACK BITS

7	6	5	4	3	2	1	0
Same as bit 4 for #ø	Same as bit 4 for #1	Same as bit 4 for #2	Intercept is pixel #3 in last OR or XOR Write	Same as bit ø for #3	Same as bit ø for #2	Same as bit ø for #1	Intercepts in pixel #3 in an OR or XOR Write since last reset

Relation between byte, bit and pixel. NOTE: A WORD is 2 bytes



MACHINE CODE LISTING OF "CRITTER PROGRAM from page 66, October 1980 issue of CURSOR:

```
4C80    F3              DI
        D9              EXX
        3E    4C        LD A, 4C
        ED    47        LD I, A         Load I with page of interrupt vector
        3E    E0        LD A, E0
        D3    0D        OUT (OD), A     Load custom chips with line of interrupt vector
        D9              EXX
        FB              EI
        C9              RET

4CE0    E13   4C        DEFW "4CE3"     Points to interrupt routine

4CE3    CD    B0  20    CALL 20B0       Call BALLY's interrupt routine
        F3              DI
        ED  73  70  4C  LD 4C70, SP     Save SP
        31  70  4C      LD SP, 4C70     Move SP
        F5              Push AF
        C5              Push BC
        D5              Push DE
        E5              Push HL
        DD    E5        Push IX
        FD    E5        Push IY
        DB    1C        In A, (1C)      Get KN(1) Value
        32  3A  4D      LD (403A), A    Place in vector block
        FF              RST 38          On Board Call
        00              Routine 00      Start multiple Calls
        07  18  4D      M CALL (4D18)   Call V Write Routine
        3F 38 4D 20 40  VECT            Move vector (see ROM manual)
        07  18  4D      M CALL (4D18)   Call V Write Routine
        02              Routine 02      End Multiple Calls
        FD  E1          pop IY
        DD  E1          pop IX
        E1              pop HL
        D1              pop DE
        C1              pop BC
        F1              pop AF
        ED 7B 70 4C     LD SP, (4C70)   Return SP
        FB              EI
        C9              RET

4D18    1F 38 4D 24 4D  V Write
        08              M RET

4D20    00 98           DEFW  152       X Boundries
        00 40           DEFW  64        Y Boundries
        00 00           DEFW  0         (0,0) Position
        02 08           DEFW  520       2 byte, 8 line pattern size
        0A A0           DEFW -24566
        22 88           DEFW -30685
        AA AA           DEFW -21846
        2A A8           DEFW -22486
        08 20           DEFW  8200
        20 08           DEFW  2080      } PATTERN
```

```
08 20          DEFW  8200      } PATTERN
00 00          DEFW  0         }

20             DEFB
80             DEFB
00             DEFB
05             DEFB
00             DEFB
00             DEFB
00             DEFB
03             DEFB
05             DEFB
00             DEFB
00             DEFB
00             DEFB
03             DEFB
```

VECTOR BLOCK (see ROM manual page 39 - 41)

ASCII CONVERSION CHART

ASCII	CHARACTER	ASCII	CHARACTER	ASCII	CHARACTER	ASCII	CHARACTER	ASCII	CHARACTER
13	GO(Carriage Rtn)	47	/	64	@	81	Q	98	x(Multiply)
31	ERASE	48	Ø (Zero)	65	A	82	R	99	÷
32	SPACE	49	1	66	B	83	S	1Ø4	LIST
33	!	5Ø	2	67	C	84	T	1Ø5	CLEAR
34	"	51	3	68	D	85	U	1Ø6	RUN
35	#	52	4	69	E	86	V	1Ø7	NEXT
36	$	53	5	7Ø	F	87	W	1Ø8	LINE
37	%	54	6	71	G	88	X	1Ø9	IF
38	&	55	7	72	H	89	Y	11Ø	GOTO
39	' (Apostrophe)	56	8	73	I	9Ø	Z	111	GOSUB
4Ø	(	57	9	74	J	91	[	112	RETURN
41	)	58	:	75	K	92	\	113	BOX
42	*	59	;	76	L	93	]	114	FOR
43	+	6Ø	<	77	M	94	↑	115	INPUT
44	, (Comma)	61	=	78	N	95	←	116	PRINT
45	- (Dash)	62	>	79	O	96	↓	117	STEP
46	. (Period)	63	?	8Ø	P	97	→	118	RND
								119	TO

MEMORY MAP

	DECIMAL
On Board ROM	Ø - 8191
Bally Basic ROM	8192 - 12287
Screen Memory Area	16384 - 2Ø479
Bally Basic Graphics/ Program area	16384 - 19983
Bally Basic Scratchpad	2ØØØØ - 2Ø463
Tape Input Buffer	2ØØØ2 - 2ØØ49
Variables begin at	2ØØ78
Line Input Buffer (1Ø4 Characters)	2Ø18Ø - 2Ø283
Stack Area	2Ø284 - 2Ø462
Text Area	-24576 --22777
Note Lookup Table	12Ø46

Special acknowledgement to: Mr. Brett Bilbrey for his contribution of information to this manual!!!