Peek N' Poke Manual
Version 1.0 - Released Oct 26, 2000
Version 1.21 - February 15, 2002

Important Note:

Note, where necessary:
    '/' replaces a 'division' sign
    'x' replaces a multiplication sign

This manual was originally written for the first version of Bally BASIC (the
cartridge that requires the joystick port cables). However, almost everything
in here is still applicable to the second version of Bally BASIC (the cartridge
with the built-in microphone jack-- also referred to as Astro BASIC). The
onboard calculator routine ($) referred to on page 12, has been removed from
AstroBasic. Ignore any reference to this original Bally BASIC command. Also,
Critter is written for use with the 300 baud BASIC.

Lines wrapping?
   This text was written to display properly using a mono-spaced font and at
least 80 columns. To avoid wrap-around and make the appearance of this file
readable try this:
        - Change the font to a mono-spaced font (Courier New)
        - Reduce the font-size of the listing
        - Reduce the left and right margins

PEEK N'POKE

FRED CORNETT
MANAGING EDITOR
THE CURSOR GROUP

AstroBasic, please contact me so that I can make note of it.

Page 0

This page intentionally left blank for double-sided printing purposes.

PEEK n' POKE

Written by: FRED CORNETT
The Cursor Group

     There has been a great deal of confusion as to exactly what PEEK n' POKE
around 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 0 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 mnemonics are an abbreviation of assembly
language. Assembly language exists only for the convenience of the programmer,
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 hexadecimal
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.

Let's 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". Let's 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. Let's 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 '0'. Now, RUN the program. Key in "PRINT A" and press "GO" and the computer will print '26'.

Let's 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 10 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. Let's eliminate the middleman 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? Let's figure it out. We are
going to store six 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).

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

     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 agitation 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 became 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=Dxl00+C
30 PRINT #1,T,"+$",T/100,".",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 Dx100 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 have been one string
variable (@(1)) or one memory location.

    To get the amount back out, we divide the total "T" by 100, 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. Let's play some dirty tricks of our own.
Key in the following program:

```
10 INPUT D,C
20 IF D>326D=D-326;T=Dx100+C;T=(-T);GOTO 40
30 T=Dx100+C
40 PRINT T
50 IF T#ABS(T)T=ABS(T);D=T/100+326;C=RM;GOTO 700
60 D=T/100;C=RM
70 PRINT #l,"$",D,".",C
```

   Now we can handle any amount LOWER than $653.68. I don't find this much 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. Let's go through that program step-by-step:

   LINE 10: If dollar amount (D) is larger than 326, subtract 326
             from dollar (D) amount. Total (T) equals dollar (D)
             amount multiplied by 100 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 30: If dollar amount is less than $327 total, (T) equals
             equals dollar amount (D) times 100 plus cent amount (C).

This ends the INPUT portion of the program.

   LINE 50: If the total (T) is a negative (-) number, change it to
             positive (+). Dollar amount (D) will equal total (T)
             divided by 100 and restore (add) 326 to it. (This is
             how we get back the 326 we subtracted in line 20). Cents
             amount becomes the remainder (RM) left over after
             dividing by 100. Skips line 60.

LINE 60:   If total (T) was not negative (meaning over $326
           originally), dollar amount (D) equals total (T) divided
           by 100. Cent amount (C) equals remainder (RM) left
           over after division.

LINE 70:   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".

    Let's 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 1, 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.
    Let's say we want to store 10 check amounts. INPUT the following changes to
the preceding program:

```
2 .123456789012345678901 2
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 WISH  TO SEE?(1-10) "N
45 T=%(Nx2+(-24574))
70 PRINT #1,"CHECK #",N,"=$",D,".",C
80 FOR Z=lTO 2000;NEXT Z;GOTO 42
```

Now, to view a particular check, you input the number between 1 and 10. The key here is in 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=0;Y=0;FOR P=-24572TO-24554STEP 2
45 T=%(P)
70 X=X+D;Y=Y+C;PRINT #1,"$",D,".",C
80 NEXT P
90 IF Y>99X=X+Y/100;Y=RM
100 PRINT #1,"TOTAL=$",X,".",Y
```

Since the computer is not aware of the difference between dollars and cents we must supply a way of adding these items separately. 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 number location 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 machine 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 (ISBN 0-8104-5167-0);
          Library of Congress Catalog Card Number 79-65355 for
          approximately $8.95.

          Z80 Instruction Book 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 twinkling. 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 1010 "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: 0 and 1. These two digits
are represented by voltages in the computer, a low voltage (0) and a high
voltage (1). The following is a representation of the numbers 0 through 5
written in binary form: 0=0, 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 0 power equals 1, i.e., 20=1, or 160=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
           -------     ------     -----------
              0          0            0
              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 0 through 9 are borrowed from the decimal system and letters of the
alphabet fill in the other six. In the hexadecimal system, each digit
represents a power of 16. For example:

```
                 1B3 = 1 x 16^2   or    1 x 256
                     + B x 16^1   or   11 x 16
                     + 3 x 16^0   or    3 x 1
                       ---------          -------
                 HEX   1B3       =         435 DECIMAL


                 POWERS OF 16 CHART

                 16^0  =  1
                 16^1  =  16
                 16^2  =  256
                 16^3  =  4,096
                 16^4  =  65,536
```

    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:

```
            2 x 163   =     2 x 4096  or   8192
          + A x 162   =    10 x 256   or   2560
          + F x 161   =    15 x 16    or   240
          + 3 x 160   =     3 x 1     or     3
            --------                        ----
            2AF3 HEX  =                     10995 DECIMAL
```

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

```
            2 x 162   =     2 x 256   or   512
          + A x 161   =    10 x  16   or   160
          + F x 160   =    15 x   1   or   15
            ---------                      ---
            2AF HEX   =                     687 DECIMAL
```

To convert decimal to hexadecimal, we divide the decimal number by the largest power of 16 equivalent that will fit. Example: lets convert 10995 to Hex

```
                  2
                ------
          4096/10,995
                8,192
                -----
                2,803 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 through 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 - Note: Remember to reverse hex pair
                       order prior to input.
                   9 PRINT "HEX # ?",
                  10 FOR A=1TO 4
                  20 @(A)=KP
                  30 IF @(A)>47IF @(A)<58TV=@(A);@(A)=@(A)-48;NEXT A
                  35 IF @(A)>64IF @(A)<7lTV=@(A);@(A)=@(A)-55;NEXT A
                  37 IF @(l)<16GOTO 60
                  40 GOTO 20
                  60 B=4096; T=0;FOR A=1TO 4
                  65 IF A=1IF @(A)>7GOSUB 100;NEXT A
                  70 IF A=1IF @(A)<8@(l)=@(l)xB;T=T+@(l);NEXT A
                  75 B=B/16;@(A)=@(A)xB;T=T+@(A)
                  80 NEXT A;GOTO 150
                 l00 T=-32767;IF @(A)=8RETURN
                 110 T=T+((@(A)-8)x4096)-l;RETURN
                 150 PRINT ;PRINT #1,"DEC. EQUIV=",T
                 160 GOTO 9
```

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 is usually wise to "SAVE THE BASIC POINTER" as the very first thing, and "RETURN TO BASIC" as the last.

Let's 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.

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. Let's 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 let's 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 let's proceed with 00. DE=0028. Therefore,
E=28, D=00. Getting back to our program.

```
   D5   Save Basic Pointer
   FF   Call Subroutine number
   31   48
   28   E Register
   00   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
   Dl   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
DF	FFD5	-43
FF		
31	2831	10289
28		
00	1400	5120
14		
14	F814	-2028
F8		
47	D147	-11961
Dl		
C9	00C9	201
00		

   Notice the "00" in the last line. We had to add zeroes so it would fill in
the space. 00 in assembly language is known as "NOP", which means 'No Op' (No
Operation); 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 20l80. Look
at our complete program now:

```
1010 M=20l80;B=M;C=l090
1020 L=-43;GOSUB C
l030 L=10289;GOSUB C
l040 L=5120;GOSUB C
1050 L=-2028;GOSUB C
1060 L=-11961;GOSUB C
1070 L=-201;GCSUB C
1080 FOR A=ltol4;CALL B;NEXT A;STOP
1090 %(M)=L;M=M+2;RETURN
```

   Lets run through this program the same way the computer will: Nothing
happens until we get to l020, we GOSUB C, which is Line l090. 1090 says:
   Poke Location M with the Value of L, then increment M, by 2 and return.
This goes on through line #1070 thusly:

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

   Then we go to Line l080. 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 reread,
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 FEEDRACK:
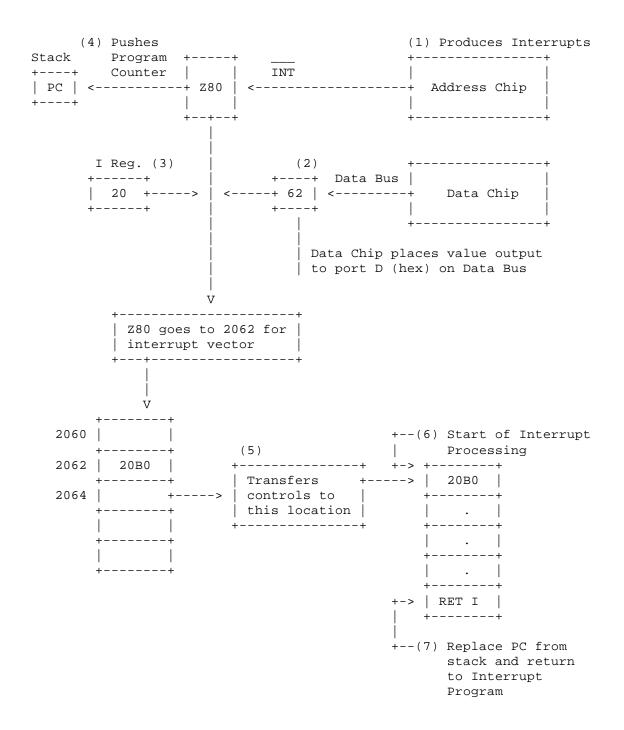---

     This Port works with the IM2 instructions to place on the data bus at each
interrupt, the data that is sent out 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 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 20B0 (hex).

```
                         Diagram of IM 2 INTERRUPT
                         ------------------------


     (4) Pushes                                (1) Produces Interrupts
Stack    Program   +-----+      ___            +----------------+
+----+   Counter   |     |     INT             |                |
| PC | <-----------+ Z80 | <-------------------+   Address Chip  |
+----+             |     |     |               |                |
                   +--+--+                     +----------------+
                      |
                      |
     I Reg. (3)       |          (2)           +----------------+
     +------+         |        +----+ Data Bus |                |
     | 20   +----->   | <-----+ 62 | <---------+   Data Chip     |
     +------+         |        +----+          |                |
                      |          |             +----------------+
                      |          |
                      |          | Data Chip places value output
                      |          | to port D (hex) on Data Bus
                      |
                      V
         +---------------------+
         | Z80 goes to 2062 for |
         | interrupt vector     |
         +---+-----------------+
             |
             |
             V
         +--------+
   2060  |        |                               +--(6) Start of Interrupt
         +--------+          (5)                   |      Processing
   2062  |  20B0  |     +---------------+    +-> +--------+
         +--------+     | Transfers      +----->  | 20B0   |
   2064  |        +---> | controls to    |        +--------+
         +--------+     | this location  |        |   .    |
         |        |     +---------------+         +--------+
         +--------+                               |   .    |
         |        |                               +--------+
         +--------+                               |   .    |
                                                  +--------+
                                             +-> | RET I  |
                                             |    +--------+
                                             |
                                             +--(7) Replace PC from
                                                    stack and return
                                                    to Interrupt
                                                    Program
```

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

     The value output to this part 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 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
'0' 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

     PORT E (Hex) bits Bit   Bit 0 - Light Pen Mode (0 or 1)
       7   6   5   4   3   2   1   0   Bit 1 - Light Pen Interrupt Enable (1 for
     +---+---+---+---+---+---+---+---+   enabled
     | | | | | | | | | Bit 2 - Screen Interrupt Mode (0 or 1)
     +---+---+---+---+---+---+---+---+   Bit 3 - Screen Interrupt (1 for enabled)

You can see from this (refer to INTERRUPT CONTROL BITS chart page 23) that if you want just a Screen Interrupt in Mode 0 (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 Part 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 (INT to Z80) occurs. In our low resolution system only bits 1 - 7 are used with bit 0 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 chips 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:

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

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

   When an On-Board WRITE Routine calls for a MAGIC REGISTER value, this means that it is to modify the data before placing it in memory. This is valid only if the 'write' is from 0 to 16K. What happens is, you write to a location between 0 and 16K. In our low resolution system this only works from 0 to 4K since we only have 4K of memory.

FUNCTIONS SET BY MAGIC REGISTER SETS

```
                                    Bit 0 - LSB of shift amount
    7   6   5   4   3   2   1   0    Bit 1 - MSB of shift amount
  +---+---+---+---+---+---+---+---+  Bit 2 - Rotate
  |   |   |   |   |   |   |   |   |  Bit 3 - Expand
  +---+---+---+---+---+---+---+---+  Bit 4 - OR
                                    Bit 5 - XOR
                                    Bit 6 - Flop
```

NOTE: Low resolution DOES NOT allow use of rotate.

     As many as four functions can be done at once. Order of operation is as
follows:

```
    1. Expansion
 ** 2. Rotating and Shifting       ** NOTE: Rotate and Shift, and
    3. Flopping                             OR and XOR, CANNOT be
 ** 4. OR and XOR                           set at the same time.
```


INPUT PORT 8 (HEX) (8 DECIMAL) INTERRUPT 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 0 - 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 0 - 3 are reset to zero.

```
                    INTERCEPT FEED BACK BITS
                    ------------------------


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



                                                                    Bit
                                +-------------- Byte  --------- | ---+
    Relation between byte,, bit and  +  +---+---+---+---+---+---+---+-V-+  |
    Pixel. NOTE: A word is two bytes +- |   |   |   |   |   |   |   |   | -+
                                        +---+---+---+---+---+---+---+---+
                                  Pixel |  #3   |  #2   |  #1   |  #0   |
                                                                \     /
                                                                 Pixel
```

On this page in the original version of the Peek 'n Poke manual there is a Machine Language listing of 'Critter' from page 66 of the October issue of the Cursor newsletter here; it's now in Appendix one. Critter is a program written for the original version of Bally BASIC. It is a small subroutine called from BASIC that moves an object around with no blinking. It uses the same routines that the cartridges use to achieve this effect.

The original listing was typed by hand, so there were numerous errors (the program was also a bit light on comments too). I have retyped the source for Critter so that it can be assembled using the freely distributable assembler called Zmac. I had two objects in mind when I did this: one, the program listing is free of typing errors, and two I added comments so that anyone with the 'Nutting' Manual can follow the code.

ASCII Conversion Chart

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
13	GO	47	/	64	@	81	Q	98	x (Mult)
31	ERASE	48	0	65	A	82	R	99	/ (Divide)
32	SPACE	49	1	66	B	83	S	104	LIST
33	!	50	2	67	C	84	T	105	CLEAR
34	"	51	3	68	D	85	U	106	RUN
35	#	52	4	69	E	86	V	107	NEXT
36	$	53	5	70	F	87	W	108	LINE
37	%	54	6	71	G	88	X	109	IF
38	&	55	7	72	H	89	Y	110	GOTO
39	'	56	8	73	I	90	Z	1ll	GOSUB
40	(	57	9	74	J	91	[	112	RETURN
41	)	58	:	75	K	92	\	113	BOX
42	*	59	;	76	L	93	]	114	FOR
43	+	60	<	77	M	94	Up	115	INPUT
44	,	61	=	78	N	95	Left	116	PRINT
45	–	62	>	79	O	96	Down	117	STEP
46	.	63	?	80	P	97	Right	118	RND
								119	TO

Note: ASCII 94 through 97, the characters are arrows of description

Memory Map

	Decimal
On Board ROM	0 – 8191
Bally Basic ROM	8192 – 12287
Screen Memory Area	16384 – 20479
Bally Basic Graphics/ Program area	16384 – 19983
Bally Basic Scratchpad	20000 – 20463
Tape Input Buffer	20002 – 20049
Variables begin at	20078
Line Input Buffer (104 Characters)	20180 – 20283
Stack Area	20284 – 20462
Text Area	–24576 – –22777
Note Lookup Table	12046

Appendix A - Critter Listing

This is the Assembly listing for Critter. It's called CRITTERP.LST. For Copy
and Paste purposes, Appendix B holds the actual source code.

```
  1:               ; Critter - For the Bally Astrocade
  2:               ; File Name: CRITTERP.ASM
  3:               ; Version 1.1 - February 12, 2002
  4:               ;
  5:               ; This source is as it appears in the PEEK N' POKE
  6:               ; manual.  It has only been changed so that it
  7:               ; assembles under Zmac; the Z80 code it generates is
  8:               ; EXACTLY the same.  Note that Critter does not run
  9:               ; without the first version of Bally BASIC; Critter
 10:               ; does NOT run under Astro BASIC (it needs vector mod-
 11:               ; ifications).
 12:               ;
 13:               ; Compile with:
 14:               ; zmac -i -m -o critterp.bin -x critterp.lst critterp.asm
 15:
 16: 0006              MCALL   EQU   $06
 17: 0008              MRET    EQU   $08
 18: 003E              VECT    EQU   $3E
 19:
 20: 4C80                      ORG   $4C80
 21: 4C80  F3                  DI
 22: 4C81  D9                  EXX
 23: 4C82  3E4C                LD    A,$4C
 24: 4C84  ED47                LD    I,A    ;Load I with page of interrupt vector
 25: 4C86  3EE0                LD    A,$E0
 26: 4C88  D30D                OUT   ($0D),A    ; Load custom chips with line of
 27: 4C8A  D9                  EXX                ; interrupt vector
 28: 4C8B  FB                  EI
 29: 4C8C  C9                  RET
 30:
 31: 4CE0                      ORG   $4CE0
 32: 4CE0  E34C                DW    $4CE3       ; Points to interrupt routine
 33:
 34: 4CE3                      ORG   $4CE3
 35: 4CE3  CDB020              CALL  $20B0       ; Call Bally's interrupt routine
 36: 4CE6  F3                  DI
 37: 4CE7  ED73704C            LD    ($4C70),SP ; Save SP
 38: 4CEB  31704C              LD    SP,$4C70   ; Move SP
 39: 4CEE  F5                  PUSH  AF
 40: 4CEF  C5                  PUSH  BC
 41: 4CF0  D5                  PUSH  DE
 42: 4CF1  E5                  PUSH  HL
 43: 4CF2  DDE5                PUSH  IX
 44: 4CF4  FDE5                PUSH  IY
 45: 4CF6  DB1C                IN    A,($1C)    ; Get KN(1) value
 46: 4CF8  323A4D              LD    ($4D3A),A  ; Place in vector block
 47: 4CFB  FF                  RST   $38        ; On-board call
 48: 4CFC  00                  DB    $00        ;Routine 01 - Start Multiple Calls
```

```
 49: 4CFD  07                    DB    MCALL+1   ; Call V Write Routine
 50: 4CFE  184D                  DW    VWRITE
 51: 4D00  3F                    DB    VECT+1    ; Move vector (see ROM manual)
 52: 4D01  384D                  DW    VBLOCK    ; Vector Block Address
 53: 4D03  204D                  DW    LIMITS    ; Limit Table
 54: 4D05  07                    DB    MCALL+1   ; Call V Write Routine
 55: 4D06  184D                  DW    VWRITE
 56: 4D08  02                    DB    $02       ; Routine 02 - End Multiple Calls
 57: 4D09  FDE1                  POP   IY
 58: 4D0B  DDE1                  POP   IX
 59: 4D0D  E1                    POP   HL
 60: 4D0E  D1                    POP   DE
 61: 4D0F  C1                    POP   BC
 62: 4D10  F1                    POP   AF
 63: 4D11  ED7B704C              LD    SP,($4C70) ; Return SP
 64: 4D15  FB                    EI
 65: 4D16  C9                    RET
 66:
 67: 4D18                        ORG   $4D18
 68: 4D18  1F        VWRITE:     DB    $1F       ; Routine 30 - VWRITR
 69: 4D19  384D                  DW    $4D38     ; Vector
 70: 4D1B  244D                  DW    $4D24     ; Pattern
 71: 4D1D  08                    DB    MRET
 72:
 73: 4D20                        ORG   $4D20
 74:              ; Limit Table
 75:              ; Note: This part of the source doesn't match the ML
 76:              ;       listing.  I had to fudge it to make it match;
 77:              ;       because of this, I have typed, but commented
 78:              ;       out, the four lines of 'weird' ML listing.
 79:              ;       Notice that the assembly hex codes are reverse
 80:              ;       ordered; this is how the manual lists the lines.
 81:              ; 00 98    DW    152   ; X boundaries
 82:              ; 00 40    DW    64    ; Y boundaries
 83:              ; 00 00    DW    0     ; (0,0) Position
 84:              ; 02 08    DW    520   ; 2 byte, 8 line pattern size
 85:
 86: 4D20  0098       LIMITS: DB    00D,152D      ; X boundaries
 87: 4D22  0040               DB    00D,64D       ; Y boundaries
 88: 4D24  0000               DB    00D,00D       ; (0,0) Position
 89: 4D26  0208               DB    $02,$08       ; 2 byte, 8 line pattern size
 90:
 91:              ; Critter pattern
 92: 4D28  0AA0               DW    $A00A
 93: 4D2A  2288               DW    $8822
 94: 4D2C  AAAA               DW    $AAAA
 95: 4D2E  2AA8               DW    $A82A
 96: 4D30  0820               DW    $2008
 97: 4D32  2008               DW    $0820
 98: 4D34  0820               DW    $2008
 99: 4D36  0000               DW    $0000
100:
```

```
101:                    ; VECTOR BLOCK (See 'Nutting' ROM Manual, page 39-41)
102: 4D38  20           VBLOCK: DB      $20      ; Magic Register value
103: 4D39  80                   DB      $80      ; Vector Status
104: 4D3A  00                   DB      $00      ; Time Base - Holds KN(1) value
105: 4D3B  0500                 DW      $0005    ; Delta X
106: 4D3D  0000                 DW      $0000    ; X Position
107: 4D3F  03                   DB      $03      ; X Checks Mask -Bounce off walls
108: 4D40  0500                 DW      $0005    ; Delta Y
109: 4D42  0000                 DW      $0000    ; Y Position
110: 4D44  03                   DB      $03      ; Y Checks Mask -Bounce off walls
**** work\critterp.asm ****
```

Statistics:

```
    6       symbols
  110       bytes
```

Symbol Table:

```
limits          4d20      vblock          4d38
mcall       =   6         vect        =   3e
mret        =   8         vwrite          4d18
```

Appendix B - Bally BASIC version of Critter

This is the original "Critter" program (and article) as printed in the October
1980 issue of the Cursor newsletter (Vol.2, pg. 66-67). This article (which is
NOT included in the original PEEK n' POKE manual) is included here because this
is the exact program as in Appendix A, but with a BASIC loader to get the
program into memory. After the program is run, the Astrocade holds EXACTLY the
same data in the EXACT same memory locations as the assembled Z80 code in
Appendix A.

Critter
By Brett Bilbrey

This program will place a Space Invader type "CRITTER" on the screen that will
bounce from top to bottom and side to side without disturbing anything that is
already on-screen. This "CRITTER" will run independent of anything else you
wish to do. If you press "HALT," he won't! His speed is controlled by Hand
Control Knob #l.

After you have "RUN" this program, do NOT scroll to the bottom line! Use
"CY=40" to keep any text away from the area in the bottom of the screen that is
"twinkling" (also, do not use "CLEAR").

Once the BASIC program has been "RUN," it can be erased and replaced with
whatever you want. Use ":RETURN" to stop the routine, and "CALL 19584" to start
it up again. One problem is that when BASIC tries to print on top of the
"CRITTER," some small screen glitches appear. You can create an invisible
screen by altering the value of Port 15 (&(15) Interrupt Line Port). It is set
to 99 which is the minimum size of invisible screen. The Interrupt Line Port
determines the number of lines scanned before the next interrupt (for a complete
explanation of all the interrupt ports etc., refer to CURSOR "PEEK n' POKE"
manual. To give you an example of a use for this type of routine, input the
following line after you have "RUN" the program: key-in ":RETURN" and hit "GO"
before you key in the line:

     1 CALL 19584;&(15)=99;INPUT A;:RETURN ;STOP

As you know, when the computer hits an "INPUT" command, it will just sit there,
waiting for you to give it a value, it will not allow anything else to happen
until you key in a value. With this one line program, it will start the
"CRITTER" bouncing around the screen as soon as it hits the input line and will
stop the "CRITTER" as soon as you input a value.

"So what," you ask? Well, instead of having a "CRITTER," we could have a clock
decrementing from one minute. If you don't get your answer into the computer
before the clock hits zero, you lose your turn and control switches to the next
player. This would provide for truly sophisticated software. So, don't lose
heart, we are on the opening stages of an exciting software era.

Brett notes that he received a great deal of help from Tom Wood, Dave Ibach, and
John Perkins, without whose help he doubts he could have written this program.

"Critter"

```
10 CLEAR ;&(15)=99
20 A=19584;B=A;C=640
30 D=-9741;GOSUB C
40 D=19518;GOSUB C
50 D=18413;GOSUB C
60 D=-8130;GOSUB C
70 D=3539;GOSUB C
80 D=-1063;GOSUB C
90 D=201;GOSUB C
100 A=19680
110 D=19683;GOSUB C
120 A=19683
130 D=-20275;GOSUB C
140 D=-3296;GOSUB C
150 D=29677;GOSUB C
160 D=19568;GOSUB C
170 D=28721;GOSUB C
180 D=-2740;GOSUB C
190 D=-10811;GOSUB C
200 D=-8731;GOSUB C
210 D=-539;GOSUB C
220 D=-9243;GOSUB C
230 D=12828;GOSUB C
240 D=19770;GOSUB C
250 D=255;GOSUB C
260 D=6151;GOSUB C
270 D=16205;GOSUB C
280 D=19768;GOSUB C
290 D=19744;GOSUB C
300 D=6151;GOSUB C
310 D=589;GOSUB C
320 D=-7683;GOSUB C
330 D=-7715;GOSUB C
340 D=-11807;GOSUB C
350 D=-3647;GOSUB C
360 D=31725;GOSUB C
370 D=19568;GOSUB C
380 D=-13829;GOSUB C
390 A=19736
400 D=14367;GOSUB C
410 D=9293;GOSUB C
420 D=2125;GOSUB C
430 D=19744
440 A=-26624;GOSUB C
450 D=16384;GOSUB C
460 D=0;GOSUB C
470 D=2050;GOSUB C
480 D=-24566;GOSUB C
490 D=-30685;GOSUB C
500 D=-21846;GOSUB C
510 D=-22486;GOSUB C
```

```
520 D=8200;GOSUB C
530 D=2080;GOSUB C
540 D=8200;GOSUB C
550 D=0;GOSUB C
560 D=-32735;GOSUB C
570 D=1280;GOSUB C
580 D=0;GOSUB C
590 D=768;GOSUB C
600 D=5;GOSUB C
610 D=0;GOSUB C
620 D=3;GOSUB C
630 CALL B;STOP
640 %(A)=D;A=A+2;RETURN
```

Appendix C - Critter Source

This is the Assembly source for Critter (it's called CRITTERP.ASM). It is
included for Copy and Paste purposes, therefore there are no page numbers on
these pages. Copy the file, make changes (if you want) and then assemble it
using Zmac. As noted in the source code, Critter is not executable without
BASIC, so don't expect it to run as a cartridge.


```
; Critter - For the Bally Astrocade
; File Name: CRITTERP.ASM
; Version 1.1 - February 12, 2002
;
; This source is as it appears in the PEEK N' POKE
; manual.  It has only been changed so that it
; assembles under Zmac; the Z80 code it generates is
; EXACTLY the same.  Note that Critter does not run
; without the first version of Bally BASIC; Critter
; does NOT run under Astro BASIC (it needs vector mod-
; ifications).
;
; Compile with:
; zmac -i -m -o critterp.bin -x critterp.lst critterp.asm

MCALL   EQU   $06
MRET    EQU   $08
VECT    EQU   $3E

        ORG  $4C80
        DI
        EXX
        LD   A,$4C
        LD   I,A   ;Load I with page of interrupt vector
        LD   A,$E0
        OUT  ($0D),A    ; Load custom chips with line of
        EXX             ; interrupt vector
        EI
        RET

        ORG  $4CE0
        DW   $4CE3      ; Points to interrupt routine

        ORG  $4CE3
        CALL $20B0      ; Call Bally's interrupt routine
        DI
        LD   ($4C70),SP ; Save SP
        LD   SP,$4C70   ; Move SP
        PUSH AF
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
        PUSH IY
```

```
        IN    A,($1C)     ; Get KN(1) value
        LD    ($4D3A),A   ; Place in vector block
        RST   $38         ; On-board call
        DB    $00       ;Routine 01 - Start Multiple Calls
        DB    MCALL+1     ; Call V Write Routine
        DW    VWRITE
        DB    VECT+1      ; Move vector (see ROM manual)
        DW    VBLOCK      ; Vector Block Address
        DW    LIMITS      ; Limit Table
        DB    MCALL+1     ; Call V Write Routine
        DW    VWRITE
        DB    $02         ; Routine 02 - End Multiple Calls
        POP   IY
        POP   IX
        POP   HL
        POP   DE
        POP   BC
        POP   AF
        LD    SP,($4C70) ; Return SP
        EI
        RET

        ORG   $4D18
VWRITE: DB    $1F         ; Routine 30 - VWRITR
        DW    $4D38       ; Vector
        DW    $4D24       ; Pattern
        DB    MRET

        ORG   $4D20
; Limit Table
; Note: This part of the source doesn't match the ML
;       listing.  I had to fudge it to make it match;
;       because of this, I have typed, but commented
;       out, the four lines of 'weird' ML listing.
;       Notice that the assembly hex codes are reverse
;       ordered; this is how the manual lists the lines.
; 00 98       DW    152    ; X boundaries
; 00 40       DW    64     ; Y boundaries
; 00 00       DW    0      ; (0,0) Position
; 02 08       DW    520    ; 2 byte, 8 line pattern size

LIMITS: DB    00D,152D      ; X boundaries
        DB    00D,64D       ; Y boundaries
        DB    00D,00D       ; (0,0) Position
        DB    $02,$08       ; 2 byte, 8 line pattern size

; Critter pattern
        DW    $A00A
        DW    $8822
        DW    $AAAA
        DW    $A82A
        DW    $2008
        DW    $0820
        DW    $2008
        DW    $0000
```

```
; VECTOR BLOCK (See 'Nutting' ROM Manual, page 39-41)
VBLOCK: DB      $20      ; Magic Register value
        DB      $80      ; Vector Status
        DB      $00      ; Time Base - Holds KN(1) value
        DW      $0005    ; Delta X
        DW      $0000    ; X Position
        DB      $03      ; X Checks Mask -Bounce off walls
        DW      $0005    ; Delta Y
        DW      $0000    ; Y Position
        DB      $03      ; Y Checks Mask -Bounce off walls
```