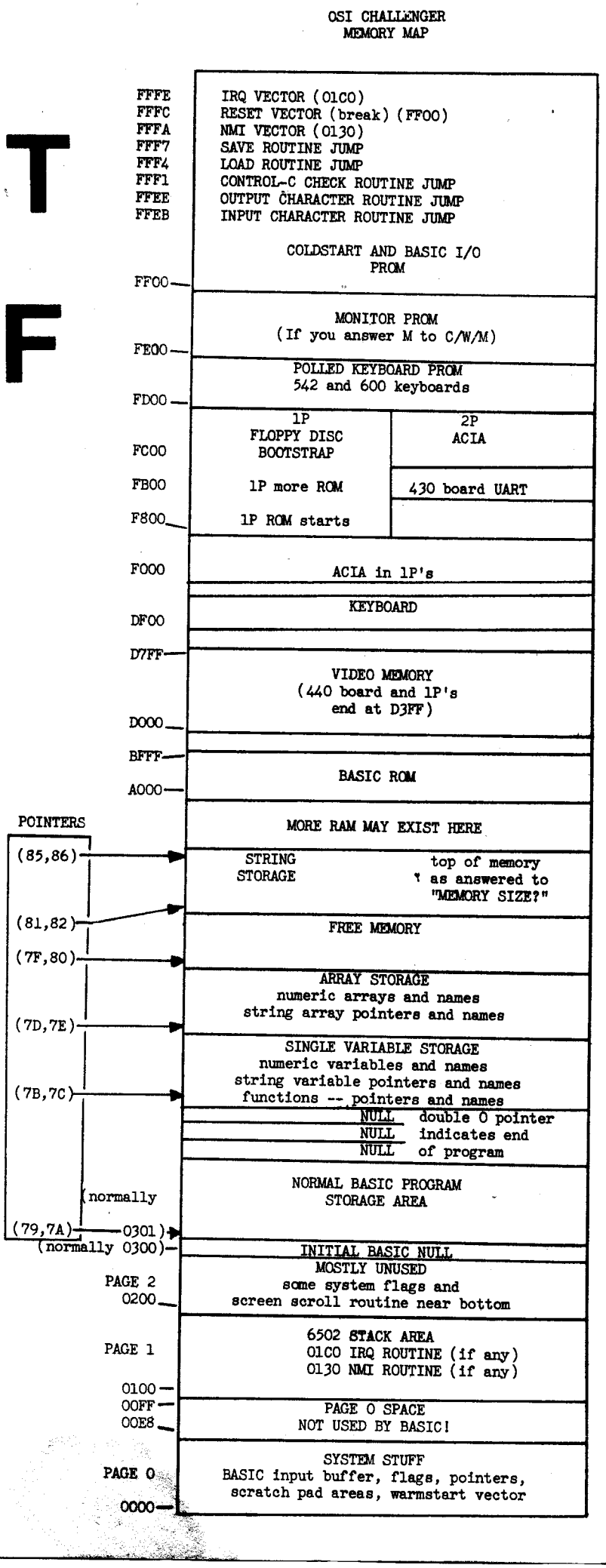
THE FIRST BOOK OF OSI

as told by

Jim Williams
and
George Dorner



THE FIRST BOOK OF OSI

as told by

Jim Williams
George Dorner

AARDVARK TECHNICAL SERVICES
1690 BOLTON
WALLED LAKE, MI 48088

THE FIRST BOOK OF OSI

as told by

Jim Williams
George Dorner

AARDVARK TECHNICAL SERVICES
1690 BOLTON
WALLED LAKE, MI 48088

PREFACE

Jim and George met at their OSI distributor's in the summer of 1978. George, having come to try out a 2P and receiving little attention and less help, cast about the room yearningly and was rescued by Jim, who had driven about an hour to pick up a tape and some documentation for the 2P at the high school where Jim teaches. George, impressed by Jim's finesse at the keyboard, contacted Jim for more info after he acquired his own 2P. Despite the fact that the two lived some fifty miles from each other, they squandered many units of phone calls exchanging ideas, information, and dreams about their 2Ps, thus illustrating how far a computer hobbyist, especially an OSI owner, will go to gather even a crumb of information about his system. The phone bills mounted, since each called the other as soon as some new fact or application was conjectured. When they got together again for one humongeous (sp?) session, much of what is listed here was gleaned or surmised, and Jim later put it all down in notes. He sent a copy to Roger and Jane Olsen of Aardvark Technical Services for their perusal, and he was urged to formalize the notes for the benefit of other OSI users. That is how this document came to be.

Most all the good stuff is due to Jim, George having served chiefly the role of Johnson's Boswell (or was it Boswell's Johnson?).

Some people, other than our wives and the Olsens, should be thanked. One is Jeff Beamsley of TEK-AIDS, INC., who was a lot more helpful than the salesperson mentioned above. Stan Murphy, who contributed the super material on the garbage collection problem, would probably have done the whole thing better. We thank the hardware designers of OSI, but our thanks go to almost no one else in Ohio. Finally, we thank MICROSOFT and Richard W. Weiland, without whom we would have probably been wasting our time on the New York Times crossword.

Jim Williams
George Dorner

March, 1980

TABLE OF CONTENTS

CHAPTERS

APPENDICES

Chapter 0

ASSUMPTIONS AND GROUND RULES

We make many assumptions and probably not enough concessions to our audience. It is assumed that you are pretty familiar with the hexadecimal, binary, and decimal number systems. Almost all our communications are in hex, as God and MOS Technology intended. Conversions to and from decimal are mostly left as an exercise for the reader. ASCII is acknowledged to be the 'lingua franca' between you and the computer when using BASIC, so you should be able to speak it when you look inside BASIC's domain. An ASCII chart is included as a convenience in the appendices. A good working knowledge of BASIC itself, including the quirks and foibles of this particular brand, will be assumed. The keyboard control characters peculiar to Challenger machines, including '?' as 'PRINT', and <shift>s O,P, and N are referred to without explanation.

You should be familiar with the OSI monitor which comes at $FE00-FEFF and with its commands. We highly recommend the OSI Extended Monitor, XMON, or at least some disassembler. Knowledge of the 6502 architecture and at least a nodding acquaintance with its machine language is necessary. (One of us started on the quest of understanding what came in those four ROM chips without any knowledge of the 6502 -- but with a pretty good understanding of its uncle, the 6800.)

We assume that you know that addresses for the 6502 are listed low byte, high byte in memory and machine code listings, but that a hex address may be printed as four hex characters preceded by a '$'. Thus, the address $BD11 is written that way for humans better to understand, but the machine is looking for two bytes, 11 BD, the hex part being understood. Finally, we often use an address as the name of a routine which starts there and sometimes as a label for the location itself, but we assume that the context or the notation 'loc $0206' will help to keep things clear. We sometimes write $79 when we mean $0079, and we often write things like $0079,7A or $79,7A to indicate contiguous locs. We had hoped to be consistent throughout, but readers of authors who have no pride (these authors) can't expect everything. We did try not to confuse the two very different ideas of a location (or an address) and its contents which we tend to designate with parentheses as in the location $000F, but the contents, ($0206), at the location $0206, or the pointer which resides at $7B,7C, which we write as ($7B,7C). You must also know what a pointer is and that $00 may be referred to as a null.

We use a lot of abbreviations -- ML for Machine Language, FPA for Floating Point Accumulator, GC for Garbage Collector, etc.

Finally, we assume that all readers will be charitable enough to forgive typos and outright errors, and, most of all, that they will be smart enough to figure out our very compact and occasionally obtuse style.

HOW TO USE THESE NOTES

We did not sit down a couple of weeks ago and dash off these notes while sipping tea and listening to classical music. Rather, they fell together in fits and spurts as we learned something new or forced ourselves away from our Challenger keyboards to commit some of this stuff to paper when we would rather have stayed at the computer. Consequently, our writing does not have the continuity and plot development of Herman Melville -- it's more like Jack Kerouac or an eighth grader's 'what I did last summer' essay. Moreover, we are both edge-a-katers, and you know how frequently they make something simple hard to understand. You may be able to overcome these disadvantages, actually understand some of what you have in your hands in this document, and put it to good use if you follow these tips:

1. Scan the whole document as soon as you finish reading this list.
2. Study the memory map and memory locations of interest.
3. Have copies of the memory map and BASIC dump at hand while studying the rest of the notes.
4. Be prepared to study the notes slowly, often using your Challenger to verify (we hope) what you read.
5. Take notes so that you know where you've been.
6. GOTO 1., unless interrupted by need for sustenance and other natural causes until you understand it all or are otherwise sated.

Chapter 1

INTRODUCTION TO BASIC IN ROM

THE BASIC INTERPRETER

Not to pull the rug out from under your mental concept of how your computer works, but the BASIC language programs you write are not instructions directly to your computer (the definition of computer enters in now). The only instructions the computer proper ever gets or can understand are binary numbers in its memory. Your BASIC programs never take that form. (But your USR routines necessarily take that form.) The only instructions your computer actually executes are the 8000+ bytes of a wondrous and complex program called the 'BASIC interpreter'. As you enter your BASIC programs at the keyboard, you are simply providing input for this program to interpret. The interpreter stores your BASIC instructions and consults them to decide which of its many tricks to perform -- but the interpreter is the program which is always running and which is in control unless it relinquishes that control to the bare machine through a USR. Your stored BASIC statements are just data which are being operated upon.

At a different level, much closer to where the 6502 lives, it's somewhat like the relationship between the simple BASIC program below and a person who knows little or nothing about the workings of BASIC.

```
10 INPUT "GIVE ME TWO NUMBERS";A,B
20 INPUT "WHAT ARITHMETIC SHALL I DO";A$
30 IF A$="ADD" THEN PRINT A+B
40 IF A$="SUBTRACT" THEN PRINT A-B
50 IF A$="MULTIPLY" THEN PRINT A*B
60 IF A$="DIVIDE" THEN PRINT A/B
70 GOTO 10
```

You might enter this program so you could let a newcomer 'tell the computer what to do'. The newcomer types, say "3,4<cr>" and then 'tells the computer to multiply'. His 'multiply' command doesn't actually multiply, of course. It's just input which your simple BASIC program uses to decide what to do. The program uses the data stored in A$ to decide which of its options to exercise. In exactly the same sense, it is the BASIC interpreter which refers to your stored BASIC statements to decide what to do next. It is residing 'behind' your BASIC program doing the real work and the real communication with the microprocessor.

Some knowledge of how this omniscient (almost) interpreter works will help you understand why some strange things happen to your nice, sensible BASIC programs -- and let you trick the interpreter into doing things it normally wouldn't do (like defeating <ctrl>C, or writing programs that won't LIST, or doing an INPUT which doesn't scroll the screen and mess up your pretty graphics, or avoiding pitfalls like the string array bug and cropped-off long lines during tape SAVEs, or recovering when some turkey hits <break>,C with your almost completed program in memory and you with no copy -- or putting together your own, home-made interpreter which is composed of hunks of code already sitting in your machine, or ...)

BASIC is better when you understand how it works. Also, BASIC is better when you marry it to machine code of your own. These two facts go hand in hand, and in putting together these notes, it is our goal to ease the task of understanding and encourage lots of healthy marriages.

If you're not familiar with the ROM monitor (where you get to with <break>,M), play with it and make it a friend. It is one of the useful features of your Challenger -- letting you directly manipulate memory and write programs in machine language which really do tell the computer what to do. If you're not there yet, you may want to study the 6502 books listed in the Bibliography before you jump in.

STORAGE OF BASIC CODE

A BASIC program is stored, line by line, in a partially digested form normally starting at memory location $0301, which is always preceded by by a null (00) at $0300. This designates the beginning of the program storage. All BASIC keywords, FOR, GOTO, END, =, CHR$, etc., are stored as one-byte 'tokens'. Tokens always have the highest bit on, i.e. they are always greater than $7F. The token for END is $80, that for FOR is $81, the one for LOG is $B5, etc. A complete list appears in Appendix 1. Other parts of BASIC statements, like AA and 123 in LET AA=123, are stored as the ASCII characters you typed in. The line number is stored as a two-byte binary number. (That does not explain why the largest allowed line number is 63999 instead of 65535!) In addition to these data, each stored BASIC line also holds a two byte pointer containing the address of the next stored BASIC line. This lets BASIC search rapidly for a given line number. The format of BASIC statement storage is always like this:

```
   00     pp pp     nn nn     bb bb bb bb bb ....          00

Initial  Pointer   Line #    BASIC Code: tokens & ASCII   Null
Null     to next                                         of this
         line                                            line
```

For example, the following BASIC statement would be stored as shown.

20 LET X=23:PRINT X may be stored as

00 29 03 14 00 87 20 58 AB 32 33 3A 97 20 58 00

beginning at location $0319.

In the example above, the statement was the second line of the BASIC program. The entire program, together with a dump of memory pertaining to the program, is printed in Appendix 3. It should be used for reference here and in the text below. A program which produces a dump like this can be very instructive to write.

The above information alone is enough to get you started on a renumbering program for BASIC. Don't forget the GOTOs, IF-THENs, GOSUBs, etc. Refer to the Bibliography for references to avoid reinventing programs which have already been written if you aren't up to the challenge or if you want to conserve your time for nastier projects.

The phrase "normally starting at memory location $0301" can provide interesting possibilities. BASIC keeps track of what it is doing through the extensive use of "pointers", important addresses which may change during execution, and which thus must be kept in RAM. These and other important data are mostly kept on page zero -- the first 256 bytes of RAM. 'BASIC workspace', the area in memory where your program and variables are stored, begins at whatever address is contained in $0079, 7A (remember: low byte, hi byte). Thus, when the coldstart routine initializes these locations, it puts 01 in $0079 and 03 in $007A. Now, if you change this, either with your trusty ROM monitor or with POKE statements, you can make BASIC store your programs anywhere in RAM you choose. (Well ... almost anywhere.) In fact, you may put one program stored starting at $0301, another at $0901, and another..., all using the same line numbers if you wish!! BASIC will find only one program at a time for RUNning and LISTing -- the one whose beginning is contained in $0079,7A. NOTE: the byte immediately before the first line of each program must be a null or nothing works.

An enlightening (even useful) example of multiple BASIC programs with the same line numbers and living at different places may be found in Appendix 5.

BASIC VARIABLE STORAGE

BASIC also needs space to store variables. These are stored in memory just above the program -- numeric variables, preceded by their names, from the end of the program going up, while string variables are stored from the top of available memory going down, their names being kept in a table together with the addresses of the memory locations where the strings actually live. An understanding of exactly how and where the variables are stored will be interesting, but this is not of great utility to most users. Here goes anyway.

The BASIC workspace will always contain two data areas where name tables are kept. One is for arrays, either string or numeric. The other is for single variables, string or numeric, and functions. Since only seven bits are needed for each character of the variable name, and since two characters are saved as the name, there are two bits available to show what type of variable is stored. If the most significant bit (msb) of the second character is a '1', a string is indicated, while the same bit in the first character indicates a function as in DEF FNAB(X). Both first bits high indicates a string function such as FNAB$, although the system does not support them.

SINGLE VARIABLES

Single variables are stored immediately following the program and starting at the address pointed at by $007B,7C. Thus, the single variables start at (7B,7C). In the example, the variable 'X' is stored beginning at $03AA.

Each single variable is stored in a fixed-length block of six bytes in this area as shown here.

FUNCTION	Nn nn	cc cc		dd dd	
	Function Name (ASCII)	Loc. of 1st char. after = in DEF stmt		Loc. of dummy variable	
NUMERIC VARIABLE	nn nn	ff ff ff ff			
	Variable Name (ASCII)	Floating point value			
STRING VARIABLE	nn Nn	ll	ss ss		00
	Variable Name (ASCII)	Length	Loc. of string		Null

(N indicates hi bit set)

Check this out with the memory dump in Appendix 3, or by looking at actual memory storage in your machine.

To find a single variable, BASIC searches the names, starting at (7B,7C), skipping to the next name six bytes later until a match is found. If a string variable is being sought, the actual string is not here, but is at the address contained in the fourth and fifth bytes. This address will point at the BASIC program code for string variables which are defined in program lines. The search ends if a match is not found before the limit (7D,7E) is reached, and then a new variable is added to the end of the table.

ARRAY VARIABLES

Arrays are stored in blocks of assorted length from (7D,7E) to (7F,80). The blocks consist of two bytes for the name, two bytes for the length of the block, one byte containing the number of subscripts, i.e. 3 for DIM A(2,4,6), then the maximum dimensioned size of each subscript -- two bytes each, followed by four-byte blocks of data for each element as follows:

NUMERIC
ARRAYS

```
 nn  nn    bb  bb    ss    zz  zz    yy  yy   ...   el  el  el  el

Var.       Block    # of   Last      Next to       Array element
Name       Size     SSs    SS        Last SS         (typical)
                                                   ------------
                                                   These four bytes
                                                   repeated for each
                                                   array element.
```

For example, the statement DIM A(2,4,6) would result in the following:

```
41  00  AF  01  03  00  06  00  04  00  02  .... data blocks ....
```

Don't forget that this BASIC uses zeroth elements in arrays and that they are not counted in DIM declarations. Thus, DIM A(3) makes space for four elements, A(0) through A(3).

STRING ARRAYS

```
 nn  Nn    bb  bb    ss    zz  zz    yy  yy    L1  L2  L2  00

Var.       Block    # of   Last      Next to   Elem.  Loc. of
Name       Size     SSs    SS        Last SS   Len.   Elem.
                                                ---------------
                                                These four bytes
                                                repeated for each
                                                string in array.
```

(SS = Subscript)

To find an array element, BASIC starts at (7D,7E) and looks at the name, then skips to the name in the next block, the location of which is determined from the third and fourth bytes, and continues until a match is found. Then it skips four bytes per element until it finds the element it wants. If it's a string, we have the length and location of the string, not the actual string. This table ends at (7F,80).

STRING VARIABLE STORAGE

Strings are actually stored starting at the top of memory as indicated by (85,86). Modifying the contents of these locations -- or having answered a number less than the actual memory size to "MEMORY SIZE?" at coldstart -- will keep the strings from wiping out any other programs or data you may want to tuck safely away in the top of RAM. BASIC uses this space for strings at the top of memory with no regard for saving space or reusing space unless it runs out of room as indicated when (81,82) approaches (7F,80). To keep from creaming the array tables, the first thing it would run into, BASIC employs a 'garbage collection' routine which tries to shuffle the strings around to the top of memory and reclaim space. Unfortunately, there is a bug in the garbage collector that makes it hang up if it has to try to relocate string arrays. This is accompanied by a characteristic blinking of the screen and must usually be aborted by a <break>. Unless you do some fancy string array manipulations in big loops, you probably won't run into trouble, but a patch is included in Chapter 6 in case you need to use string arrays extensively. The FRE(X) routine at $AFAD calls the garbage collector before finding out how much room is left between (81,82) and (7F,80).

NUMERIC VARIABLE REPRESENTATION

The floating point value of a numeric variable is stored in four bytes in normalized binary exponential (scientific) notation:

```
                Sign & implied
  Exponent      most sig. bit
  ^^^^^^^       ^
  10000011      (.)00100000     00000000     00000000
  ^                 ^                             ^
Exponent        Implied binary                  Least sig.
sign            point                           bit
```

This would be read as: .101(base 2)*2^3 = 5(base 10).

The last three bytes contain the number to an accuracy of twenty four bits. The first byte is the power of two -- if you like, the number of places to move the binary point. (The binary point is like the decimal point, except to its right we have the 1/2s column, the 1/4s column, the 1/8s column, etc. It may also be referred to as the radix point.)

The most significant bit of the value (bit 7 of the second byte) is always interpreted as having the value '1'. (If it were '0', we could shift the number to the left -- i.e. shift its binary point to the right -- until it was '1', increasing the exponent by the number of places we moved.) Since this is understood (this is what is meant by the word "normalized"), we can use that actual bit in memory as the sign bit, '1' indicating a negative number. Negative numbers are not represented in two's complement form. The exponent, however, is represented in two's complement form. Here are some examples:

5	10000011	00100000	00000000	00000000
1	10000001	00000000	00000000	00000000
2	10000010	00000000	00000000	00000000
3	10000010	01000000	00000000	00000000
4	10000011	00000000	00000000	00000000
7	10000011	01100000	00000000	00000000
15	10000100	01110000	00000000	00000000
-5	10000011	10100000	00000000	00000000
3/8 = .375	01111111	01000000	00000000	00000000
0	00000000	00000000	00000000	00000000

This representation allows decimal numbers up to 1.70141E38. There is a smallest positive number which may be represented, too. Its discovery is left as an exercise.

If you want to explore this further, the short BASIC program below will read the binary representation of a number from memory. More on number representation may be found in some of the references cited in the Bibliography.

Here is a program to look at the binary representation of a number which is stored in normalized binary exponential form as four bytes in memory. It looks at the second through fifth bytes after (7B, 7C). Deleting line 30 lets you look at the variable name (and the first two bytes of the value).

The program waits for you to input a number, then prints the binary representation of it, then waits for another number.

```
 10 INPUT M
 20 P=PEEK(123)+256*PEEK(124)
 30 P=P+2
 40 FOR J=0 TO 3
 50 N=PEEK(P+J)
 60 GOSUB 200
 70 PRINT "  ";
 80 NEXT
 90 PRINT
100 GOTO 10
200 FOR I=0 TO 7
210 B=N AND 2^(7-I)
220 IF B THEN PRINT "1";:GOTO 240
230 PRINT "0";
240 NEXT
250 RETURN
```

Chapter 2

HINTS AND KINKS

WHAT OSI FORGOT TO TELL YOU ABOUT BASIC

Following are some tips, pretty much in random order, which may be helpful or informative.

-*-

When the system is initiated, answer 'A' to "MEMORY SIZE?". The 'A' is for 'author'.

RICHARD W. WEILAND

-*-

Only the initial quotation mark in a pair is required unless ambiguity would result. For example, PRINT "JIM works fine, but not INPUT "NAME;A$.

-*-

If you want to imbed commas or colons in a line you are typing in response to an INPUT statement, begin the line with a quotation mark. This will also let you enter a line with leading blanks, which are stripped otherwise. The same trick will also let you put commas or leading blanks in DATA statements. The closing quotes are still optional.

Can you come up with a way to INPUT a BASIC line containing a " and a : and a , without rewriting the INPUT routine?

-*-

When using nested loops, a NEXT I,J will substitute nicely for NEXT I:NEXTJ.

-*-

A colon after any response you type to an INPUT statement ends what the INPUT sees, but what you type after the colon will be seen as remarks on the screen. For example, if, in response to INPUT A$, you type JIM:WILLIAMS<cr>, the screen will show JIM:WILLIAMS, but A$ will contain only JIM.

-*-

The colon is also useful in 'formatting' LISTs. Use one as the first character in a line to set it off. Blanks after the colon will be printed in listings.

-*-

Although not documented in the old OSI BASIC manual (we're being charitable), the statement ON X GOSUB mm,nn,pp... works just fine, just the same as ON X GOTO, but calling subroutines.

-*-

CLEAR will reset all the variables to zero.

-*-

Long BASIC lines produce auto carriage return/line feeds when listed. When saving on tape, this causes the last part of the line to be lost. By setting the "TERMINAL WIDTH" to be longer than any BASIC line with a POKE 15,255, the damaging carriage return will be avoided.

-*-

A super-simple decimal to hex converter is as follows: Enter the number you wish to convert to hex. BASIC thinks this is a line number. UP TO 63999

Press <cr>. Then go into the monitor and look at locs $11,12 to find the converted number lo byte, hi byte.

-*-

After warmstart, the stack is usually confused. This causes an OM error with your first immediate mode memory-referencing command after the warmstart. Get in the habit of hitting any character and <cr> immediately after all warmstarts. We like A or P because they are easy to hit rapidly in sequence with a <cr>. See Chapter 7.

-*-

The warmstart vector at $0001,0002 may be set to a return or restart address for convenience while debugging a machine language program. Then you may just hit the 'W' to get where you want to go without entering the monitor after reset.

-*-

To edit a BASIC program without losing the values of variables, find the contents of $7B-82. Then edit the program (shorter only), and replace the pointers with the monitor or POKEs. Do a GOTO to get back into the BASIC program.

-*-

The graphics characters in the error messages are caused by the fact that the hi bit of the second character in the messages is set to allow a routine to tell that it is the last character in the message. Unfortunately, the error message printer doesn't take that into account. Guess we'll have to live with it.

-*-

If you've manufactured tapes directly from a BASIC program, you've probably also seen the message 'OK/SN ERROR/OK' at the end of a tape load. Here's how to avoid that syndrome:

```
POKE 3,96          Turns "OK" off
POKE 3,76          Turns it back on
```

also delete CR on list

-*-

Here's a quick and effective security trick. By making the 'OK' message printer at $03 point to a simulated CONT statement, your BASIC INPUT statements become immune to people hitting <cr> without a response first, and thus having access to the BASIC editor to cream your program (or LIST it and find out how it works.) Since the 'OK' message printer is invoked both by the 'OK' when a null input is given and when BASIC is warmstarted, this trick protects against <ctrl>C and even <break>W! The easiest implementation is to put the code at $02C3 -- where the $03 JMP to $A8C3 can be changed with one POKE.

```
02C3 2065A6 JSR $A665  call the CONT routine
02C6 68      PLA
02C7 68      PLA       fix stack
02C8 4CFCA5 JMP $A5FC  jump back into BASIC exec loop
```

Patch this in with a POKE 5,2 -- or out by putting the $A8 back in. (POKE5,168) Be sure to use prompt messages in your INPUTs; it's disconcerting to try to warmstart and have only a question mark come back! You can, of course, get back to normal by putting the $A8 back in with the monitor.

-*-

The <ctrl>G is implemented as ASCII BEL, even if there's no bell. A funny character is also printed. That's also the character echoed when the input buffer is full.

-*-

Here is a simple 'no LF' input, which is useful at the bottom of a graphics display to avoid a screen roll. In response to an INPUT (asking whether to go on or for more data, say), enter your answer, then type <ctrl>O and <ctrl>M instead of the usual <cr>.

-*-

Here's another 'no LF' input.

BASIC Code	USR code
POKE 100,0	JSR $00C2
X=USR(X)	JMP $A925
POKE 100,128	

-*-

Sometimes when the machine goes astray, as in a bad USR call, a BASIC program is retrievable, even when warmstart won't respond.

Recovery of a BASIC program may be possible if you answer "MEMORY SIZE?" with a number instead of with <cr>. (Once you hit return, BASIC fills memory with test bytes until it doesn't get them back to see how much memory there is. That means your program is completely overwritten and totally irretrievable.) The easiest way to recover is to go into the monitor before you coldstart, find and copy the contents of locations $007B,7C and do the same for $0301,02. Then coldstart, explicitly entering the same memory size you were using before the bomb (i.e. 4096 for a 4K machine, etc.), and after BASIC comes up, go back to the monitor and replace (7B,7C) (the end of program/beginning of BASIC pointer) and replace (301,302) (the pointer from the first BASIC statement to the second), which will have been set to nulls by the coldstart. The rest of the BASIC program should still be there.

If you didn't save (7B,7C), then the contents of $0302 will be 03 always, unless you have hand-manufactured a very unusual BASIC program, and the contents of $0301,02 will always be one higher than the location of the first null byte after $0305.) Now you may warmstart and the program will LIST, but it will bomb itself if you try to RUN it. That is because variables will overwrite the beginning of the program. To fix this, LIST the program, enter the monitor, find the contents of $00AA,AB, add two, and put those values into $007B,007C. Everything should then be back to normal. (Immediately after listing any line, (AA,AB) will be the address of the pointer to the next BASIC statement -- or of the double null immediately before the beginning of variable space.)

-*-

Here's a quick BASIC machine language dump. It makes a monitor format tape for saving machine language very nearly as fast as a machine language program does the same thing.

```
10 SAVE
20 A1=    (fill in start address in decimal)
30 A2=    (fill in end address in decimal)
40 ACIA= 64512      (61440 for 1Ps)
50 ?".HHHH/";        (HHHH is start address in hex)
60 FOR A=A1 TO A2
70 D=PEEK(A)
80 H=INT(D/16)
90 L=D-16*H
100 IF H>9 THEN H=H+7
110 IF L>9 THEN L=L+7
120 ?CHR$(H+48)CHR$(L+48);
130 POKE 14,0
140 WAIT ACIA,2
150 POKE ACIA+1,13
160 NEXT
170 ?".FE00G"
```

-*-

This is a better decimal to hex conversion than that used in the above program.

```
1000 A$="0123456789ABCDEF"
1010 INPUT N
1020 L=N AND 15:H=(N-L)/16
1030 ?MID$(A$,H+1,1)MID$(A$,L+1,1)
1040 GOTO 1010
```

-*-

If you would like to be able to LOAD a BASIC tape and then have it automatically continue and load a machine language tape with the monitor, here is one way. Type the following:

```
SAVE<cr>
LIST(turn recorder on)<cr>
(stop tape when done)
?"POKE251,1:POKE11,67:POKE12,254:X=USR(X)
(restart recorder)<cr>
(stop tape when done)
```

Now put the machine language on the tape using your favorite method.

When you LOAD the tape, it will load the BASIC program, switch to monitor mode (without clearing the screen), and load the last part of the tape.

-*-

Here's a program to prevent LISTing, done by replacing the pointer from line 30 to the next line with a double null. The program is 'found' by replacing the pointer (lines 10,20). The

last line of the program makes it invisible again. The first three lines must be copied exactly as shown, including blanks. Added security may be had by turning off <ctrl>C with a POKE 530,1 after line 30.

```
10 POKE 794,32
20 POKE 795,3
30 REM
      ...   program which
      ...     will not
      ...       list

end with
      POKE 794,0:POKE 795,0
```

-*-

You can defeat <ctrl>C, but <break> still leaves your program vulnerable. Change the warmstart vector at $0001 to that of coldstart or something else to avoid this.

-*-

Here is a fast screen clear in BASIC. It's not as fast as machine language, but much faster than the usual FOR I=1to30:?:NEXT. It uses the screen as string storage space! The address POKEd after the PEEKs is $DFFF.

```
10 A=PEEK(129):B=PEEK(130)
20 POKE 129,255:POKE 130,215
30 A$="              <- 65 blanks ->          "
40 FOR I=1 TO 32:A$=A$+"":NEXT
50 POKE129,A:POKE 130,B
```

-*-

If you wish to look at a program on a tape without writing over a program which is already in memory, the following 'VIEW' program may be useful. It is absolutely relocatable. It reads tapes and writes only on the screen.

```
20 07 BF 20 EE FF D0 F8 F0 F6
```

This won't work on 1Ps, but the following will work on either machine (but this one is not relocatable):

```
1000 A9 80        Set LOAD flag
1002 8D 03 02     Load it
1005 20 EB FF     ACIA input
1008 20 EE FF     Output to screen
100B 4C 00 10     Jump to start
```

-*-

Chapter 3

MEMORY LOCATIONS OF INTEREST

0000	WARMSTART jump-you get here from <break>W-and you may change it to jump wherever you like!
0003	Jump for the 'OK' message printer subroutine. Make location $03 a $60 (RTS) to defeat 'OK'.
0006,7	Address of INVAR (it's AE05).
0008,9	System puts address of OUTVAR here (it's AFC1).
000A-C	USR jump. Put your USR address in $0B,C. It's initialized by the system to point to the FC ERR routine-$AE88.
000D	Number of NULLS to be put in after SAVEd <cr>'s.
000E	Number of chars since last <cr>. Used for auto cr/lf and POS(X).
000F	Number of chars until auto cr/lf. Make it >71 to ensure not losing long lines on tape SAVEs.
0010	Like 000F, but for comma-spaced zones. Equals 14*(INT(line length/14)-1).
0013-5A	Input buffer.
005F	String-variable-being-processed flag. $FF=string.
0061	??
0064	<ctrl>O flag. Hi bit on = suppress PRINTing.
0065	Sometimes contains $68. ??
0079,A	Pointer to beginning of BASIC workspace for RUN,LIST,etc.
007B,C	Pointer to beginning of single variable storage.
007D,E	Pointer to beginning of array storage.
007F,80	Pointer to beginning of FREe memory.
0081,82	Pointer to last used byte of string space. Moves from top of memory down.
0083,4	MEMORY SIZE from coldstart-- used later for string scratch pad.
0085,6	'HIMEM'--top of memory allowed to be used for BASIC.
0087,8	Current line (for BREAK IN LINE XX).
0089,A	Sometimes next line .
008C	CONT flag. If it contains 0, CONT gives error.
008F,90	Pointer to next character after last used DATA. Points to initial null if RESTOREd.
0093,4	Current variable name in ASCII
0095,6	Where AD0B leaves address of variable it found.
0097,8	Address of variable to be assigned by OUTVAR, AFC1.
00A1	General purpose alterable JMP instruction. Target address goes in 00A2,3.
00AA,AB	Scratch pad. Points to pointer of next BASIC line after LIST N, or to middle of 3 final nulls after LIST.
00AC,AF	Primary floating point accumulator. AC is exponent, AD-AF is mantissa.
00AE,F	Where INVAR (that's AE05) leaves its 15-bit signed argument.

00B0	Sign bit sometimes.
00BC-D3	Routine to get the next character from wherever $C3,C4 point. Used to work through BASIC lines both from the input buffer and during execution. Leaves the carry flag clear if char is 0-9. Sometimes called CHRGET.
00C2	Entry point to $BC routine to get current char Sometimes called CHRGOT.
00C3,4	Address of current char in $BC routine.
00D1-3	Clobbered by XMON during disassemblies, thus rendering $BC routine useless.
00D4-7	Floating point value of last RND call; will be regurgitated by RND(0).
00E0-5	Apparently unused (by BASIC) Page Zero space.
00E7-FF	Apparently unused (by BASIC) Page Zero space. Note: system can get in mood where it puts stuff in 00F0,F1.
00FB	ROM monitor Load flag (0=keyboard).
00FC	ROM monitor contents of current mem location.
00FD	Printed to screen, then erased by ROM monitor.
00FE,F	ROM monitor address of current mem location.
0100-7	Where B96E leaves ASCII rep of floating point number.
0128	ROM monitor and coldstart prom initialize stack here.
0130-	NMI routine (if you put it in).
01C0-	IRQ routine (if you put it in).
0200	Current screen cursor is D700+(0200); it is initialized to (FFE0).
0201	Temp storage for char to be printed.
0202	Temp storage for CRT driver.
0203	BASIC LOAD flag (hi bit on=input from tape instead of keyboard).
0205	BASIC SAVE flag (0= not SAVE mode).
0206	Time delay for slowing CRT driver.
0207-E	Variable Execution Code block--for system screen scroll; not reuseable by mortals.
020F-11	Apparently not used.
0212	<ctrl>C flag (not 0=ignore <ctrl>C).
0213-16	Polled keyboard temp storage and counter.
0217	Apparently not used.
0218,9	Input routine vector (1P).
021A,B	Output routine vector (1P).
021C,D	<ctrl>C routine vector (1P).
021E,F	LOAD routine vector (1P).
0220,1	SAVE routine vector (1P).
0222-FF	Only used in disc systems.
0300	BASIC initial null (normally).
0301	BASIC code normally starts here with pointer to second line of BASIC code.
0FFF	End of 4K.
1FFF	End of 8K.
A000-37	BASIC initial word jump table; in token order; add 1 to each entry.
A038-65	BASIC non-initial word jumps; actual addresses.
A084-163	BASIC keywords in ASCII; hi bit set as delimiter; in token order.

A164-86	Error messages; hi bit set as delimiter.
BCEE-D05	Code for $BC routine; put in at coldstart.
BE39-4D	'Want SIN-COS-TAN-ATN?' message; a vestige of earlier RAM implementations of BASIC.
BE4E	'Written by' message.
DEXX	Screen size and other latched bit (lo bit=0 for small).
DFXX	Keyboard, both ASCII and polled.
F000,1	ACIA in 1P's.
F700-03	PIA in 2P's (and others) with 500 CPU board.
FB00,1	UART in 430 board.
FC00,1	ACIA in 2P's.
FD00-FF	Polled keyboard prom in 2P's.
FE00-FF	ROM monitor.
FF00-FF	I/O support and RESET (<break>) vector prom in 2P's.
FFE0	Initial cursor position; $64 for 440 video (and 1P's?) and $40 for 540 video (2P's).
FFE1	Default 'TERMINAL WIDTH' (less 1).
FFE2	CRT driver switch: 0 for 440, 1 for 540 video.
FFE3,4	BASIC workspace lower boundary (for 0079,A).
FFE5,6	BASIC workspace upper boundary.
FFE7,8	Variable workspace lower boundary.
FFE9,A	Variable workspace upper boundary.
FFEB	System INPUT routine; returns char in accumulator.
FFEE	System OUTPUT routine; call with char in A.
FFF1	<ctrl>C check routine; exits to BASIC if <ctrl>C detected.
FFF4	LOAD routine (sets flags when BASIC LOAD command is done).
FFF7	SAVE routine (sets flags when BASIC SAVE command is done).
FFFA,B	NonMaskable Interrupt vector (6502-determined).
FFFC,D	Reset (<break>) vector (6502-determined).
FFFE,F	Interrupt ReQuest vector (6502-determined).

See Chapter 7 For Rom Routines

Chapter 4

A WALK WITH BASIC

What we know at this writing about the life and times of OSI ROM BASIC, Version 1.0, rev.3.2 is included here in both text and in the various charts and tables.

A good place to start exploring BASIC is the warmstart entry at $A274. BASIC can also be warmstarted by a jump to loc $0000, where the system puts the instructions $4C/74/A2 after coldstart. At this point, BASIC is looking at the keyboard and waiting for immediate mode commands or BASIC instructions with line numbers to be entered.

Consult the warmstart flowchart, Flowchart A, Appendix 4. BASIC first clears the <ctrl>O flag which is the msb of loc $0064 by an LSR $64. This allows output to occur. The message printer routine is then invoked with the standard convention of pointing A,Y (lo, hi) at the ASCII message whose last character is a null in order to print 'OK cr lf '. The message printer is at $A8C3, the message text starts at $A192, and the null signals $A8C3 to return. A jump to loc $0003 accesses the message printer.

Now, the 'fill the input buffer' routine is called. This routine is located at $A357 and takes input from either keyboard or ACIA, depending on the LOAD flag which is the msb of loc $0203. All I/O is done by vectoring to the FFXX prom. In this case, input is done through a JSR to $FFEB. Characters are received, screened, counted, and stored in the input buffer, locs $0013 through $005A. The screening handles 'backspace', @,<ctrl> O, and <cr>. When it sees a <cr>, it calls $A866 to put a null rather than a <cr> in the buffer, and prints the cr/lf together with extra nulls as defined by the contents of $000D. If needed, the nulls are put in the output stream after cr/lf for a slow device and may be set with a NULL statement or with POKEs to loc $000D. A flowchart is also included for this important routine in Appendix 4.

During the coldstart, a vital routine is copied from $BCEE-BD05 to the locations beginning at $00BC. This code is called very often and puts the next character from the current line into the accumulator. Thus, this routine is referred to as CHRGET. (The current character may also be put in A by calling $00C2, CHRGOT, instead of $00BC.) The routine sets the carry flag for the information of the calling program if the character being passed is numeric. The address of the current character is in locs $00C3,C4 -- the address portion of an LDA instruction. Everybody uses the code at $00BC to find out what's up next, and the stuff at $00C3,C4 is constantly being changed by the programs using $00BC, in addition to being incremented by $00BC each time

it is called.

At this point, the $00BC routine is being used to work through the ASCII in the input buffer as it is being tokenized. $00C3,C4 is set to point at the input buffer. If the first character in the buffer is numeric, the buffer must contain a numbered line of BASIC source code, so we go to $A295 to do the 'tokenize and store in BASIC workspace, updating necessary pointers' job on the input buffer. If the first character in the buffer is not numeric, we must be doing an immediate mode statement, so we call the routine at $A3A6 to tokenize the line in the buffer and put it back into the buffer. Then we jump to $A5F6, the main entry to the 'execute BASIC statements' loop.

When a program is RUN from the beginning, $A5F6 calls the RUN routine at $A477 which does the following:

1) Points $00C3,C4 to the contents of $0079,7A, the beginning of BASIC workspace, usually $0301
2) Resets the string pointer at $0081,82 to the top of memory as recorded in $0085,86
3) Resets the array pointer to the end of the BASIC program which is also the beginning of the single variable work space as kept at $007B,7C. (This pointer is constantly being updated during BASIC editing and program entry.)
4) Resets the 6502 stack pointer to ($01)FC
5) Stores a $00 in locs $008C and $0061 (why??)
6) Stores a $68 in loc $0065 (why??)

and 7) Returns

Then we jump to $A5C2, the top of the 'do the next line of BASIC' loop. Refer to Flowchart A in Appendix 4.

In the main BASIC loop, at $A5C2, we first do a <ctrl>C check and stop, printing 'BREAK IN LINE (contents of $0087,88)' before returning to warmstart if <ctrl>C is detected. If not, we check to see if the next character in whatever line we're working on is a null (indicating the beginning of another BASIC line). If it isn't, it had at least better be a ':' to indicate a multiple statement line, or we go to the syntax error printer, then back to warmstart. If a null was found, the hi byte of the pointer following it will contain a null if we are at the end of the program, so if we find that, we stop. Otherwise, it's on to the next line of BASIC, first storing its line number, then incrementing $00C3,C4 past the pointer and line number. The next sequential instruction in ROM is $A5FC, and we continue executing BASIC statements.

$A5FC is the main entry point to the 'RUN the BASIC program' loop. See its flowchart in Appendix 4.It calls $00BC and checks for a null, exiting to warmstart if it finds one. Otherwise, it calls $A5FF to do the dirty work of executing a BASIC statement before looping back to the top at $A5C2.

$A5FF calls $00BC and checks to see if the first character is greater than $80. If not, it is not a token, so we must be doing a LET statement with an implied LET. In this case, we go

to $A7B9, which calls $AD0B, a very important subroutine that finds the name of the variable to be assigned by the LET, finds its address in variable storage space, puts that address in $0095,96, and also returns with the address in A,Y. $A7B9 stores the variable address in $0097,98 and checks for an '=' (everybody is using $00BC to find the next character), and if it doesn't find one, generates a syntax error. If the '=' is found, the important routine $AAC1, the 'evaluate an expression' routine, is called, which leaves the value from the expression in the floating point accumulator, $00AC,AF. $A7B9 returns by way of a JMP to $B774, the 'store the floating point accumulator into ($0097,98)' routine. ($0097,98) was obtained from $AD0B. Now we return to $A5FC, which loops back to the top at $A5C2. (There will be a short quiz on these addresses at the end of the period.)

If $A5FF finds a token at the beginning of the line, it first verifies that it is an 'initial word' token, (indicated by a value less than $9C), then does an ASL, TAY to multiply the token by two to get an offset for the table of initial words at $A000.

Digression about tokens:

Tokens are functionally divided into 'initial words' like FOR, RUN, or POKE and 'non-initial words' like THEN, =, or SQR. For each initial word, there is a subroutine, the address of which is in a table which begins at $A000. Each address takes two bytes and is stored in order according to its token number. That is, the first address is for token $80, the first token, the next address ($A002,A003) is for token $81, etc. Initial tokens go up through $9B. For non-initial tokens, some (like SQR) are complex enough to require their own subroutines, while others (like =) do not. Tokens $9C through $AC rquire no subroutines; $AD through $C3 do. The first twenty-eight tokens, the initial word ones, take 28*2 bytes in the table, so the non-initial tokens get the addresses starting after the first fifty-six bytes of the table, namely at $A038. Ignoring the hi bit of an initial token and multiplying it by two gives the address in the table of the routine for that token. (If you think that's hard to follow, try to infer it from a disassembled dump of the ROMs!) The words which are tokenized are listed in order beginning at $A084. They have the msb in their last character set, which allows the use of varying lengths of words in that list. It also is the cause of the funny error messages with graphics symbols as the second character.

The other, non-initial, tokens are dealt with within the routines for the initial words. Those that are complex enough to need their own routines are called by the old ASL, TAY trick. The ASL is at $AC27, the TAY is at $AC55. The offset in the Y-register is added to an invented base address of $9FDE to find the routine's address in the jump table. Example for the token AD: $9FDE + 2*($AD with hi bit ignored) =$A038, the address of the pointer to the routine for token $AD. Phew! Unlike the previous case, this jump is not a stack trick, so the addresses

in the jump table are correct as they stand. The $9FDE+Y stuff goes on around $AC56.

End of digression. Where were we?

We had just found a token and gotten its offset. $A5FF now has the address of the subroutine that will do the operation of the BASIC keyword that started the line. It pushes this address onto the stack, calls $00BC for the convenience of the next routine, and an RTS does the actual jump to the needed routine. Again: the address of the routine to do the desired BASIC operation for an initial word is pushed onto the stack - like the return address is for a JSR -- and then an RTS makes the processor jump there. This all happens around $A60D. You will see that $A5FF JMPs to $00BC, while the RTS in that routine is the one which pops the address off the stack and 'returns' there. Since the PC is incremented after popping the return address, the addresses in the jump table for the initial words are all one less than the actual entry addresses.

-*-

Sic transit BASICus

-*-

COLDSTART ET AL

Now that warmstart has been explored, what's left? For starters, there's coldstart. It is entered by a JMP $BD11 from the FFxx startup procedure. There isn't a lot going on now, just lots of initialization of pointers and other data which BASIC has to chew on, so we have not included a flowchart. Nonetheless, there is still some mystery and room for further exploration here.

First of all, the stack pointer is set at $(01)FF and the FF is put at $88. The coldstart address itself is loaded in $01,02 and in $04,05, INVAR and OUTVAR addresses ($AE05 and $AFC1, respectively) are loaded at $06,07 and $08,09 and the JMPs are added just ahead of these four addresses. INVAR and OUTVAR are used with the USR and elsewhere. They are described in some detail in Chapter 7. At this point, either JMP $0000 or JMP $0003 will take us right back to coldstart, but that changes near the end of the coldstart routine. Next, the address $AE88 is loaded into $0B,0C to prepare for a function call (FC) error message in case there is a JMP to ($0B,0C) or to $000A or a USR call before a valid jump address has been entered there by you. The error messages are printed from an indexed table by the routine at $A24E using the contents of X. How these contents are set in two possible ways using the same tricky opcode, 2C, is neat and you should study the code from $AE85 through $AE8C until you understand it. This code is entered at two places, $AE85 or $AE88, depending on the message required. This trick is pulled several times throughout BASIC.

Next, the terminal width is loaded into $0F and $38 is loaded into loc $10 (why?). The variable execution block, CHRGET, which keeps track of characters in the input buffer is loaded into $00BC-00D3 from its permanent location beginning at $BCED. Then nine locations (including the nulls and POS counters at $000D and $000E) are zeroed and a null is pushed onto the stack before we finally get down to something which can be detected by a user sitting at the terminal.

First, a <cr/lf> (the BASIC CRT driver lives!), quickly followed by the message 'MEMORY SIZE' and then a call to $A946, the 'print a ?' routine. Then let the user fill the input buffer until he hits a <cr>. Scrutinize a character in the buffer. Is it an 'A'?? (Why 'A'? Try it -- or see the HINTS.) If so, do what that's supposed to do and go back to the top of coldstart for a rerun. Is it '00' (indicating a response of <cr>)? Then test each memory location beginning at $0300 (things would probably be acting very badly, if at all, if memory were not good up to the point where BASIC itself really works) as follows: store the current address, put $92 there, then check to see that it really got there, and if so, shift left (yielding that familiar $24), check to see that it was well-received, and, if so, increment the current address and do it again. If either check fails, a bad memory location or the top of contiguous memory has been found. (They are the same to BASIC.) The current address (the actual top of memory + 1) is then placed in $11,12, in $85,86, where it lives undisturbed until the next coldstart or you alter it, and in $81,82. The value stored is variously referred to HIMEM or MEMTOP on some systems. Its permanent location is $85,86, the other locations simply being set as initial values of pointers which will soon be reset. A decimal response to 'MEMORY SIZE?' is converted to hex by $A77F and placed in the right locations, while any other response results in a syntax error. (Note: there is no check to see that you gave a reasonable decimal answer.)

Well, you know what comes next. It's 'TERMINAL WIDTH''?', followed by another look at the keyboard, a check into the buffer for that null indicating a <cr>, and on to other things, leaving the width where it was set earlier if the null is there. Non-nulls are converted to from decimal to hex if possible and rejected otherwise with a REDO message, until a valid decimal entry (as determined by some mysterious arithmetic around $BDE3) results in a new value for loc $0F.

Coldstart is now on its last legs. The basic BASIC pointer, as copied from the FFXX prom, is placed in that all-important $0079,7A, the critical null is placed where the pointer points, at $0300, and the pointer is incremented and placed in registers A and Y, preparatory to a trip to $A21F to see that memory is big enough for at least a one byte program, and a <cr/lf> is printed. The number of bytes from ($79,7A) to ($85,86) is then computed and printed in decimal by good old $B95E with the hex being sent in A/X (hi/lo), followed by the message 'BYTES FREE'. Note: if you enter a value for MEMORY SIZE which is greater than the

actual amount of memory existing, the number of BYTES FREE will not be adjusted to be correct. For example, if you have an 8K machine and enter '10000' as MEMORY SIZE, you will receive a message of '9231 BYTES FREE' ... a good way to get free write-only memory.

Wrapping it all up, the message printer address, $A8C3, is stored at $0004,5, where it will be used to print all the 'OK's in the next adventures of BASIC beginning at warmstart, $A274, which address is stored in $0001,2 so that the JMP at $BE36 can take us there. Sandwiched in there is a call to the 'NEW' routine which set pointers $0079,7A, $007B,7C, $0081,82, $007D,7E, and $007F,80 and puts $68 at $0065 (wuzzat?). The stack is consulted for the top two bytes, which are put at $01FD,FE, and the stack pointer is aimed at $01FC. After clearing $008C and $0061, coldstart is warm and meets its demise with the JMP to warmstart.

There are other points of interest between $A000 and $BFFF besides warmstart, coldstart, and all the subroutines which compose BASIC statements. The three I/O routines which use the ACIA at $FC0X and which are located between $BF07 and $BF2C have cousins between $BEE4 and $BF06 which are designed to talk to a UART at $FBXX and which are not used in current Challenger machines. A fascinating routine which was mostly listed in the OSI Small Systems Journal of July, 1977 is the CRT Simulator between $B2FD and $BF72. If this routine is entered at $BF2D with an ASCII code in the accumulator, the character will be printed on the screen at $D700 + ($0200) (or $D300 + ($0200) in the case of the 1P -- the screen size is controlled by an address in the prom at $FFE2). The original program used all the memory from $0200 to $020E, but it looks like $0203-205 are not used here. The contents of $0206 control the rate of printing on the screen and could be useful in BASIC games with graphics where acceleration/deceleration effects are desired. The 'scroll the screen' routine, a block of code at $BF83 is relocated into $0207-20E so that it can modify its contents while moving lines up the screen.

We haven't touched any of the math routines here. Deciphering the floating point arithmetic is left to the reader as an exercise. A few of the HINTS in Chapter 2 may also shed insight into parts of BASIC which we have not described in this text.

There are a few stretches of NOPs sprinkled through the code. Perhaps you have listed the BASIC ROMs in ASCII and have wondered what the message "WANT-SIN-COS-TAN-ATN" at $BE39 was. It's a relic, a leftover from the cassette (RAM) version used to allow one to conserve memory at the cost of giving up the trig functions. How many other such relics are buried in there?? Wouldn't it have been nice if they had really optimized the code and given us RENUMBER or DELETE?

-*-

Ad maiorem gloriam Microsofteus.

That's about it, folks. Aside from sundry tables, subroutines, and a little (very little) wasted space, that is BASIC as she is writ for OSI by MICROSOFT. You see, it's really very simple -- nothing that won't yield to a little time (off and on -- mostly off -- for about a year) well spent in cogitation and deduction (we bet Sherlock Holmes and Watson couldn't have done better; hashish and software hacking don't seem to go together), and the satisfaction when one has conquered ....INTERRUPT.... Let's be honest. It was fun, and we think that what we have set down here is more or less correct. But there are still many routines which we haven't touched (from the inside), and there must be a neat idea for an application which you had when reading these notes which deserves to be shared in a magazine article, letter to a column editor, or at least in a note to us. We make this offer:

> If you share some information about the innards of BASIC or about their application in new, unintended ways, we will include the info, with credits, in any subsequent edition of this document.
>
> You may communicate with us by sending a SASE to our publishers:
>
> Aardvark Technical Services
> 1690 Bolton
> Walled Lake, MI 48088
>
> Decisions by the judges as to whether the information is original enough or of sufficiently general interest will be final. Let us hear from you, though.

Then again, maybe we should all just join Disassemblers Anonymous.

Chapter 5

USING THE USER FUNCTION

The USER function, X=USR(Y), lets you pass a value computed in your BASIC program to a machine language program, execute that ML program, pass a value obtained by the ML back to BASIC, and return control to the next statement in the BASIC program. The ML program might drive a printer, do a fast screen clear or reverse line feed, generate a musical note, drive an external device like a PROM programmer, or do anything else the computer is physically capable of. If you have a ML main program, the USER facility is a convenient way to enter it. (Nobody will ever know if you don't return to BASIC!)

Whenever BASIC wants to do a "LET" statement, like "LET X=SQR(Y)" (the LET is optional), it calls some subroutine(s) to evaluate whatever is to the right of the equal sign. To evaluate "LET X=SQR(Y)", it calls a subroutine in ROM. To evaluate "LET X=USR(Y)", it calls a subroutine in RAM located at $000A (10 dec). That subroutine starts with a JMP instruction with the first byte ($4C) at location $0A, and the other 2 bytes -- the address to jump to -- in locations $0B,0C. The system puts the $4C in for you -- and puts the location of the ROM 'Function Call ERROR' subroutine ($AE88) in $0B,0C. In order for control to actually pass to your ML, you must fill in its starting address in the JMP instruction. So if your program starts at $0F00, (near the top of 4K RAM) you must put $00 (0 dec) in location $0B (11 dec) and $0F (15 dec) in location $0C (12 dec). (Remember: lo byte, hi byte.) For a simple example, here's a USER routine that uses ROM routines to let you type a given number of characters on the keyboard, and echoes them -- one character late. Don't just read it, try it!

```
      JSR AE05     get the 16 bit (15+ sign) value passed from BASIC
      LDX AF       to $AE,AF, and put the low part in X-reg as cntr
      LDA 00
      PHA          store a dummy first character
LOOP  JSR FFEB     get a char (no echo) from kbd, using ROM routine
      TAY          save this char in Y register
      PLA          put previous character in accumulator
      JSR FFEE     output previous char to scrn, using ROM routine
      TYA          get current character
      PHA          save it for next time
      DEX          decrement character counter
      BNE LOOP     loop if not down to zero
      PLA          get last saved character
      JSR FFEE     output it to screen, using ROM routine
      RTS          return to BASIC
```

10 POKE 11,0 : POKE 12,15
20 X = USR(X)

THIS WORKS FINE

Here's what's in memory. Use the ROM monitor to put it in. It starts at location $0F00, which contains a $20, with $05 in location $0F01, etc. Don't forget to put the starting address in $0B,0C!

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0F00 20 05 AE A6 AF A9 00 48 20 EB FF A8 68 20 EE FF
0F10 98 48 CA D0 F3 68 20 EE FF 60
```

Now on to bigger and better stuff. When BASIC sees "X=USR(Y)", it:

1) finds where X is stored in memory and stashes the address in $95,96 and $97,98
2) evaluates the expression in the parentheses -- here, simply the value of Y -- and leaves that in the floating point accumulator $AC-AF
3) does a JSR to $0A to do your ML
4) takes whatever value is left in the FPA and assigns it to the variable whose address is in $97,98

If your ML calls INVAR ($AE05), that ROM routine converts the value in the FPA to a 16-bit (15 and sign) straight binary number and leaves it in $AE,AF (hi,lo). If you can handle the floating point value, don't call INVAR -- what you want is already in the FPA.If you want 24 bits' precision instead of the 16 INVAR gives, use the routine in Appendix 6. To pass your own number back to the BASIC program, you may leave your 16 bit value A and Y (hi,lo) and call OUTVAR (AFC1) to convert it to floating point and to put it in the FPA, or you may manufacture a FP number and put it in the FPA yourself. If you change the contents of $97,98 (for example, by calling the routine at $AD0B and doing STA 97;STY 98) whatever value is in the FPA will get assigned to the variable pointed at by the new contents of $97,98 instead of the variable to the left of the equals sign in the BASIC USR statement. The ROM routine at $B774 which does the actual storing is called after your ML program does its RTS. Should you want to use this routine that stores the contents of the FPA in ($97,98), feel free! BASIC leaves $C3,C4 pointing at the first character after the right parenthesis of your USR statement. If you call $C2, you can get that next character that would normally cause a syntax error. (The SN ERROR doesn't happen until after your USR ML has done its RTS.) For example: Point the USR JMP instruction ($0A,0B,0C) to $0F00. Put an RTS ($60) at $0F00 and try this BASIC statement: "X=USR(Y)A". You will get a syntax error, of course. Now change $0F00 to JMP $BC ($4C BC 00). This gets the next character (the "A") and updates $C3,C4 to point past it. The same BASIC statement now gives no error! Make $0F00 read JSR $BC;JMP $BC and you can (must!) put two characters after the right paren to avoid syntax error! Now, since $BC leaves the character it finds right there in the accumulator for you, you may use it -- perhaps to select one of two or more ML programs -- or ??? You can play with this with the following shorty:

```
0F00   20 C2 00   JSR 00C2      get the char
0F03   A8         TAY
0F04   A9 00      LDA 00        set up A,Y to return it
0F06   20 C1 AF   JSR AFC1      put char in FPA to be
                                returned
0F09   4C BC 00   JMP 00BC      point $C3,C4 past char & ret
```

This will return the ASCII value of the 'illegal' character after the right parenthesis to your BASIC variable. Question: Why do you get 165 (dec) instead of 42 (dec) for "X=USR(Y)*"? (Hint: Think about tokens.)

Now for a real toughie. You want a USR function so that X=USR(Y)Z,37*Q,R,S+3 does the following:

1) X gets the value of S+3;
2) Z gets the value of 37*Q;
and 3) R gets the original argument of Y.

The next listing shows a program that does this. Follow through this, and you'll be able to pass just about any arguments you want.

```
1000  A597     LDA $97
1002  85F0     STA $F0      save pointer to variable X
1004  A598     LDA $98
1006  85F1     STA $F1      hi part
1008  2005AE   JSR $AE05    INVAR call to get Y as 16 bits
100B  A5AE     LDA $AE
100D  85F2     STA $F2      save Y lo
100F  A5AF     LDA $AF
1011  85F3     STA $F3      save Y hi
1013  20C200   JSR $00C2    ADOB expects current char in A
1016  200BAD   JSR $ADOB    get ptr to next variable (Z)
1019  8597     STA $97      save pointer to Z lo
101B  8498     STY $98      pointer hi
101D  2001AC   JSR $AC01    check for and pass comma
1020  20C1AA   JSR $AAC1    evaluate next expr (37*Q)
1023  2074B7   JSR $B774    store FPA (37*Q) to ($97,8) (Z)
1026  2001AC   JSR $AC01    get next comma
1029  200BAD   JSR $AD0B    get ptr to next variable (R)
102C  8597     STA $97      save R pointer lo
102E  8498     STY $98      save hi part
1030  A5F2     LDA $F2      get Y lo
1032  A4F3     LDY $F3      get Y hi
1034  20C1AF   JSR $AFC1    16 bits in A,Y to FPA (Y)
1037  2074B7   JSR $B774    store FPA (Y) to ($97,8) (R)
103A  2001AC   JSR $AC01    get next comma
103D  20C1AA   JSR $AAC1    evaluate expr (S+3)
1040  A5F0     LDA $F0
1042  8597     STA $97      restore ptr to variable X
1044  A5F1     LDA $F1      to $97,8 for $B774 which is
1046  8598     STA $98      called after we RTS
1048  60       RTS
```

Have you ever wished you could do a string input and not have to worry about commas, quotation marks, and colons chopping up your input and giving 'EXTRA IGNORED' messages? This routine does it! It is called by A$=USR(X). The string variable contains exactly what you type--even a length zero string if you just hit <cr>! The $B0B4 routine expects the $BC routine to be pointing at the message to be assigned, so we have to save $C3,C4 to be able to continue our BASIC program when we return. $5B and 5C contain the delimiter characters $B0B4 looks for to know when the string is done -- commas and colons and stuff normally -- so we put in a convenient non-alpha character. After letting $B0B4 find the length of the string and set pointers up, etc., we restore $BC's pointer to the BASIC line and go home, letting the normal BASIC operation store the string for us. The two pops before we return bypass a particularly nasty type mismatch check at $AAB0. Happy inputting! (Courtesy of your local USER function.)

```
1000 2057A3 JSR $A357
1003 A5C3   LDA $C3
1005 A4C4   LDY $C4
1007 8511   STA $11
1009 8412   STY $12
100B A913   LDA $13
100D A000   LDY $00
100F 85C3   STA $C3
1011 84C4   STY $C4
1013 855C   STA $5C
1015 855B   STA $5B
1017 20B4B0 JSR $B0B4
101A A511   LDA $11
101C A412   LDY $12
101E 85C3   STA $C3
1020 84C4   STY $C4
1022 68     PLA
1023 68     PLA
1024 4CC200 JMP $00C2
```

Chapter 6

THE GARBAGE COLLECTION PROBLEM AND A SOLUTION

Stanley P. Murphy

THE PROBLEM

As described above, OSI BASIC in ROM uses a routine referred to as a 'garbage collector' to keep house and clear out 'garbage' in memory which is left over from its ordinary, everyday activities. There is an error in this routine which severely restricts the use of programs with string arrays, particularly if concatenation of strings is extensive.

Strings which are not defined between quotes in a BASIC statement are stored in the 'string space' at the top of memory. For example, if the line '20 A$="A"' appears in a program, then BASIC won't have to store a copy of the string in string space -- it can just set a pointer to the actual location of the string in program storage. However, '30 INPUT B$' will require the value of B$ to be put in string space with a pointer to its location placed in the BASIC variable table.

On concatenation, say '50 C$=C$+A$', C$ is stored in string space. Suppose we start initially with C$="" and A$="A". We then execute C$=C$+A$. C$ will become "A" and will be stored in string space. String space will contain just "A". Suppose we execute C$=C$+A$ again. Now C$ is "AA" and string space contains "AAA". Two of the "A"s are the new C$, and one is just left behind in forming the new C$. If we do it again, C$ is "AAA" and string space consists of "AAAAAA". In this example, three "A"s in string space are for storing the last designation of C$ and the others are previous designations of C$ which are not removed from memory. They are garbage.

Now suppose the following program is run:

```
20 C$="":A$="A"
30 INPUT K
40 FOR I=1 TO K: C$=C$+A$:NEXT
```

C$ is now K bytes long and takes K bytes to store. However, we have used K(K+1)/2 bytes of memory in generating and storing C$. Thus, in the above program, if K=255 (the maximum string length), we need 255 bytes to store the final C$, but we have tried to use 32,640 bytes! Available memory vanishes very quickly if a program contains repetitive operations of this kind.

A memory rearrangement is needed if memory is to be properly utilized. This is accomplished through the Garbage Collection (GC) routine. This routine is called by BASIC when the string

space is full. The GC relocates the valid strings back to the top of memory and defines new pointers in BASIC variable space. Using the above example GC recovers 32,385 bytes of memory for further use.

OSI's GC routine works fine for programs containing numeric variables, string variables, and numeric arrays. The above program can be run on a 4K machine without a problem. However, if the program also contains a string array, then the GC will not work correctly. If, for example, we add the following line to the above program:

```
10 DIM I$(6)
```

and enter K=255, the internal GC will cause the screen to 'pulse' several times at a rate of once every 1.6 seconds, as the GC routine walks through memory. This pulsing is characteristic of GC failure, along with a 'dead' keyboard. Extraneous characters may show up on the screen, and the BASIC program may be altered. The execution time without line 10 is under two seconds, but it exceeds twelve seconds with line 10 added. If the program were entered exactly as written, the pulsing may continue until the <break> key is pressed. Even if the program finally executes, C$ is not placed properly at the top of memory.

A more general program to demonstrate the GC problem is:

```
10 DIM L$(26)
20 INPUT K
30 FOR I=1 TO 26
40 FOR J=1 TO K: L$(I)=L$(I)+CHR$(64+J)
50 NEXT J
60 PRINT L$(I),I:NEXT I
```

Here, a first order array with dimension 26 is established. Each element is formed from the ASCII code starting at A and continuing to Z. At the conclusion of filling each array's Q elements with the alphabet, the element is printed followed by the element number.

For an OSI BASIC-in-ROM machine with 8K of memory, the program will run with L$(Q) DIMensioned for Q<=18. If Q exceeds 18, the GC routine fails. Similarly, if L$ is DIMensioned to allow 62 elements instead of 26, failure occurs for Q>3.

This error is very troublesome for any BASIC program that contains string arrays and does extensive string manipulations (for example, a word processor written in BASIC).

CIRCUMVENTING THE PROBLEM

Ideally, one would like to correct the errors in the ROM program. One could program a 2716 EPROM with the correct code and substitute it, after some wiring changes, for the incorrect ROM. This is neither simple nor inexpensive. There is, fortunately,

a simpler approach which is useful.

The listing below is a BASIC program that, when RUN, places a corrected GC program at the top of memory. It protects the program from being written over by other BASIC programs. It also sets USR function pointers so that the program may be called by X=USR(X). Finally, it displays on the screen two useful pieces of information. It provides data for a POKE statement which may be needed to reset the USR pointers if they are changed by another program. It also provides the location, in decimal, of the GC program called by USR(X).

The steps to use this approach are as follows:

1. Coldstart
2. LOAD the program listed below.
3. RUN the program ONCE. (Each time the program is executed after coldstart, "MEMORY AVAILABLE" is reduced in increments of the GC program length). RUN time is about 15 seconds.
4. Record the POKE and LOCATION data for future use.
5. Type NEW and LOAD the program containing string arrays which you wish to RUN.
6. Insert X=USR(X) in the program after each major concatenation to call the corrected GC. Place the POKE statement before this call if USR(X) is used elsewhere in the program to be RUN.

AN EXAMPLE

Take the general program listed above. Add the following line:

```
60 X=USR(X)
```

This cleans up the garbage left after the completion of each string array element, L$(I). With this addition, an 8K machine will operate with L$(Q) DIMensioned for Q greater than 200. Unfortunately, after 50 or 60 elements are generated, the program slows down noticeably since the GC is moving a large number of strings. This is the penalty paid for being forced to call the GC more frequently than is necessary. It is better to err on the side of conservativism, because the program bombs if the internal GC is triggered.

BASIC Program to Fix GC Problem

```
 10 X=PEEK(133):Y=PEEK(134)
 20 L=256*Y+X:L=L-262
 30 Y=INT(L/256):X=L-256*Y
 40 POKE133,X:POKE134,Y
 50 POKE11,X:POKE12,Y
 60 PRINT"POKE11,";X;"POKE12,";Y
 70 PRINTL;A=45383:B=45644
 80 K=L:FORI=ATOB
 90 IFI<>A+34THEN110
100 M=K+146:GOTO240
110 IFI<>A+59THEN130
120 M=K+140:GOTO240
130 IFI=A+67THENPOKEL,4:GOTO230
140 IFI<>A+84THEN160
150 M=K+209:GOTO240
160 IFI<>A+137THEN180
170 M=K+146:GOTO240
180 IFI=A+216THENPOKEL,2:GOTO230
190 IFI=A+217THENPOKEL,24:GOTO230
200 IFI<>A+261THEN220
210 M=K+4:GOTO240
220 X=PEEK(I):POKEL,X
230 L=L+1:NEXT:PRINT"LOCATION":END
240 Y=INT(M/256):X=M-256*Y
250 POKEL,Y:POKEL-1,X
260 GOTO230
```

A somewhat more efficient technique is to call the GC only when free memory is small enough for there to be some chance of the stock GC being called. By finding how much space is left (not using FRE(X)! That calls the GC!) by looking at the actual pointers, we can save some unneeded GC calls. The following subroutine does just that:

```
200 LO=PEEK(127)+256*PEEK(128)
210 HI=PEEK(129)+256*PEEK(130)
220 IF HI-LO<100 THEN ZZ=USR(8)
230 RETURN
```

The figure of 100 bytes in line 220 is arbitrary; use a number you like there. This subroutine should be called after every major string operation instead of calling the GC directly.

Chapter 7

VERY USEFUL BASIC ROM ROUTINES

These are the routines that are of special interest to people writing applications in machine language, or perhaps trying to use existing routines to do the dirty work in implementing a new language interpreter or the like. Here are text message printer, numeric value printers, system I/O routines, numeric manipulators, and practical and potential hooks into the system. There are doubtless some we have missed or unwisely dismissed, but at least this is a fair start.

0000

This warmstart jump, accessed ~~at any time~~ by <break>W, usually points to BASIC warmstart at $A274. Consider changing it to point at the ROM monitor, or your extended monitor, or anywhere you keep having to go to. The standard warmstart at $A274 does not reset the stack pointer, but if you point this jump at $0000 to code that does, like--LDX #$FE; TXS; JMP $A274--, you can circumvent the annoying OM error which you get on the first immediate mode which makes a memory reference.

0003

This is the 'cr/lf OK cr/lf' message printer jump. You can defeat the 'OK' message by making the JMP instruction in loc $0003 into an RTS (replace the $4C with $60), or you could make it jump elsewhere--perhaps to print a different prompt message--or into an error trap.

00BC (and, of course, 00C2)

This 'CHRGET' routine is very useful in USR statements to change the syntax of the statement. A call to $C2 from your USR routine will get the next character from the USR statement into the accumulator. You may use this character to pass additional information from BASIC, or to select between multiple USR routines; or you may use other ROM routines to get additional values or variables from the BASIC line. Repeated calls to $BC will get the next characters from the line. Carry will be set if the character returned is numeric. This routine would be an essential parser for new interpreters written to run on OSI ROM BASIC machines. If you use it, say in a USR routine in a text processor to do whatever, be sure to save the contents of $C3,C4--the actual pointer-- and restore them before RTSing back to BASIC. The B3F3 routine might be helpful here.

A357

This is the 'fill-the-input-buffer-until-<cr>' routine. It gets input from the current system input device (tape if LOAD mode) through $FFEB, and stores it in the input buffer, starting at $13, using the X register as a pointer through the buffer. It puts a null at the end of the inputted string in place of the <cr>. Normal system conventions, like <shft>O, auto repeat, ignoring control characters, etc apply. All valid characters entered are echoed--including the final <cr>. This last echo is what scrolls the screen on INPUT instructions. (Yes, Virginia, of course the INPUT routine uses $A357!) When $A357 returns with your string in the buffer, it leaves X and Y set with the address of the buffer, all ready to do STX $C3; STY $C4 to point the $BC routine at the buffer to find out what was just entered.

A477

This is an initialization routine used by the BASIC RUN command. Call this routine, then jump to $A5C2, and you'll be RUNning in BASIC--from a machine language start! This has distinct possiblities in LOAD-and-go situations.

A77F

Looks for a decimal number with the $BC routine. Point $C3,C4 at the input buffer (or wherever), call $BC, then call $A77F, and look for the 16 bit binary answer in $11,12. Don't forget to reset $C3,C4 before going back to BASIC.

A8C3

The message printer -- important and easy. Point A,Y (lo,hi) at the ASCII message in memory. Be sure there's a null ending the message.

A925

This is actually an entry into the INPUT routine just after the <ctrl>O flag is cleared. If you first set the <ctrl>O flag by putting $80 in loc $64, and then calling this routine, you will be in an INPUT statement with no echo--including the final <cr> that messes up your nice graphics. If you want BASIC PRINT statements to work again you must clear the flag ($0 to loc $64). Since this routine uses $BC, if you call it from a USR statement, you have to call $C2 first, to get the first character of the 'INPUT' part of the line; your BASIC USR/no-echo-INPUT statement would look like this: X=USR(X)Y in order to INPUT to the variable Y. X here is an unused variable. There seem to be problems using A925 to try to INPUT strings from a USR routine. (The INPUTted strings are stored in non-existent memory.)

AAC1

Here's a goodie. This routine gets a value from the BASIC line (like starting immediately after the ')' of your USR statement), doing all the arithmetic evaluations it finds necessary, and leaves the result in the FPA. This gives you great power to expand the USR statement to get as many values as you like to your ML program. See the USR section for an example of using this powerful routine. The AE05 routine is a natural companion to this. You can use AC01 to find a comma between arguments if you want to get multiple values from the line. AAC1 does no TM error check!

ABF5-AC0C

This series of routines (actually of entry points to one routine) uses the $BC routine to check the line for various delimiters. If you disassemble the ROM here, you will find a classic use of the $2C opcode as a combination NOP and immediate load, depending on where you jump in. ABFB checks for ')'; $ABFE for '('; $AC01 for ','; $AC03 for whatever character you leave in the accumulator when you call it. $ABF5 checks for '(', calls $AAC1 to get a value, then checks for ')'. This would be useful for getting arguments for for functions in some new 'tiny' interpreter.

AD0B

VARPTR is the command in some dialects of BASIC that does what this routine does. A call to this uses the $BC routine to find the variable name that is next in the line, finds the location in memory of that variable (or creates it), and leaves the address of the variable in locs $95,96 and in A,Y. If you store A into $97, and Y into $98, you can call $B774 to store the contents of the FPA into the variable. Of course, you can call OUTVAR, $AFC1, first, to put the 16 bit value in A,Y into the FPA before calling $B774. Who knows what mischief you may come up with, armed with the location of a BASIC variable!

AE05

Often called INVAR, this routine converts the contents of the FPA to straight binary and leaves the 16 bit result in $AE,AF (hi,lo). The argument of your USR call is in the FPA when your ML is executed; that's how you usually get a value from BASIC by calling $AE05.

AFC1

Often called OUTVAR, this routine takes a 16 bit value from A and Y (hi and lo), converts it to floating point, and puts it in the FPA. When you leave your USR routine via an RTS, the machinery that called your USR also calls $B774 to put what you've left in the FPA into the BASIC variable (found by $AD0B) at the left of your USR statement.

B3AE

This is just like $AAC1, except that it gives an FC error if the value is >255 decimal. This is what is used by the POKE instruction to keep you from trying to POKE a too-large number into memory.

B774

Calling this routine stores the contents of the FPA into 4 bytes of memory (presumably variable storage) pointed at by $71,72. $AD0B is useful here.

B95E

Call this routine to print on the screen at the current cursor location the decimal value of the 16 bit number you have left in A,X (hi,lo). Other entry points are

1) $B95A to print current line (from $87,88);
2) $B962 to print the contents of $AD,AE (hi, lo).

B96E

This one creates an ASCII string, starting at $100, to print the value in the FPA, exponential notation (if necessary) and all. It even sets up A,Y to point at $100 so the next instruction can be a call to A8C3 to print the string.

BBC3

This is an entry to the BASIC RND routine. If you call this routine with a 1 in A, it will return with a random number in the FPA. You should get a fair random number if you just use the 8 or 16 bits from $AE,AF.

BF07

Calling this short routine will read a byte from the ACIA (at $FC0X). The read byte, with the highest bit masked off, will be returned in A. This routine is independent of system LOAD and SAVE flags.

BF15

The complement to $BF07, this one outputs a byte from A to the ACIA. (No masking.)

BF2D

With no concern for flags, a call to this puts out a byte from A to the screen at the current cursor location, with auto cr/lf if needed.

FD00

This is the entry point to the polled keyboard routine in 2Ps. The ASCII character typed is returned in A; it won't return until a key is pressed. (Boo!) Stan Murphy has written a new 256 byte version of this routine that makes lower case conveniently usable -- with numbers, cr, etc. working normally in lower case. Both shift keys even do the same thing! This text is being typed with his keyboard routine in PROM.

FE00

This is the entry point to the ROM monitor. It's a good place for your ML programs to go when they're done.

FE43

This one is even documented by OSI! It's a warmstart (no screen clear or stack initialization) to the ROM monitor. If you put other than 0 in $FB and jump here (some POKEs and a USR call, if you like) you will be in ROM monitor Load mode! This is one way to do multi-mode tape loads.

FFEB

This is the address called by the system to input a character from tape or keyboard, as dictated by the LOAD flag. The character is delivered to you in A. Use this routine freely. This and the next 4 routines consist solely of JMP instructions back into ROM. One of the great breakthroughs of the 1P is that these system calling places are indirect JMPs through RAM. ($218-221) The coldstart routine initializes these locations to point into ROM, but you can make them go through your own

routines to screen for special characters (<ctrl>E for your editor, or <ctrl>L (form feed) for instant screen clear, or <ctrl>B or S for big or small letters...my system now does most of those). If you have access to a prom programmer, consider changing those JMPs to go through RAM--at least input and output. Initialization poses some problems, though.

FFEE

System output a character from A to screen and (if SAVE mode) to tape, with all system conventions, nulls, etc. If you vector this through RAM, you can check for ASCII $5F (underscore) and do real backspace, using $200 as cursor location, doing a DEX, and of course decrementing $0E.

FFF1

Call this routine for an easy soft exit from your ML program loops. It's the <ctrl>C check--and it prints the usual 'BREAK' message and goes to warmstart ($A274, not $0000, unfortunately.)

FFF4

This is where BASIC goes when it gets a 'LOAD' command. All it does is set the LOAD flag and clear the SAVE flag. If you vector it through RAM, you could make it do whatever you wanted on a LOAD command! Nobody says it has to have to do with loading anything. It does seem a natural if you have an Exatron Stringy Floppy (TM-Exatron) or something...

FFF7

This is the SAVE routine--another flag-setter. Again, there is no action here--it's a jump back into ROM.

Chapter 8

DATA STATEMENT FILE UTILITIES

DATA STATEMENT GENERATOR

Using knowledge of how BASIC statements are stored, this program moves the variable workspace pointers, gets input from the keyboard, generates DATA statements in memory, and exits (with the operator's help) in such a manner that the DATA statements are then an integral part of the program. They may be listed with the program, or may be listed separately (presumably to tape) to become part of a DATA statement data file.

Line 30 moves the array and bottom of strings pointers to accommodate the new DATA statements; 32 moves the simple variable pointer, thus losing the value of L. Line 40 initializes the program's pointer to where the pointer of the first DATA statement will be stored. The subroutine at 200 POKEs the 2-byte representation of N into P,P+1 -- and updates P for the next call. Line 110 uses 200 to put in the pointer (length of DATA string + 6 bytes overhead: 2 pointer, 2 line , 1 token, 1 null); 120 to put in the line number. Line 125 puts in the token $83 for DATA. The program exits by putting back the high byte of the simple variable workspace pointer. Once that is done, however, BASIC no longer knows where the variables are stored -- so it depends on you to type in the low byte. (It tells you what to type before it loses sight of variable workspace.)

RESTORE TO LINE N

The last part of the listing is a routine to do a BASIC RESTORE function not to the beginning of all DATA statements, but to any desired line number. This makes possible systems of DATA statements which can be semi-randomly accessed: The DATA statements starting with, say, 10000 could be one logical file, those beginning with 11000 another, etc. While DATA statements on tape are a rather awkward file medium, this does open up some possibilities. The RESTORE to line N routine simply searches all the line numbers, beginning with the first at loc 771,2 until it finds the line you want. It fixes up the DATA pointer at $8F,90 to point at the null before that line's pointer. Line 2060 is a simple test READ.

```
5 REM DATA STATEMENT MAKER
6 REM
10 REM LINE 1000 CONTAINS LENGTH OF DATA BLOCK MAX
20 GOSUB1000
30 D=PEEK(124)+L:POKE126,D:POKE128,D
32 POKE124,D
35 GOSUB1000
40 P=PEEK(123)+256*(PEEK(124)-L)-2
50 INPUT"INITIAL LINE ";LI
60 PRINT:PRINT:PRINT"ENTER '//' TO EXIT
100 PRINTLI;"DATA";:INPUT A$
105 IFA$="//"GOTO180
110 LE=LEN(A$):N=LE+P+6:GOSUB200
120 N=LI:GOSUB200:LI=LI+5
125 POKEP,131:P=P+1
130 FORI=1TOLE:POKEP,ASC(MID$(A$,I,1))
140 P=P+1:NEXT
150 POKEP,0:P=P+1
160 GOTO100
180 N=0:GOSUB200
190 PRINT"TYPE 'POKE123,"PAND255"'
195 POKE124,INT(P/256)
199 END
200 LO=NAND255:HI=(N-LO)/256
210 POKEP,LO:POKEP+1,HI
220 P=P+2:RETURN
1000 L=3:RETURN
1800 REM
1900 REM "RESTORE TO LINE N" ROUTINE
1910 REM
2000 P=769
2010 INPUT"LINE ";N
2015 IFP=0THENPRINT"NOT FOUND":END
2020 LN=PEEK(P+2)+256*PEEK(P+3)
2030 IFLN<>NTHENP=PEEK(P)+256*PEEK(P+1):GOTO2015
2040 P=P-1:POKE143,PAND255
2050 POKE144,INT(P/256)
2060 READA:PRINTA
2999 END
```

APPENDIX 1

BASIC LØØKUP/JUMP TABLES

WØRD LØC	WØRD	TØKEN	JUMP TØ	J TØ LØC
A084	END	80	A639+1	A000
A087	FØR	81	A555+1	A002
A08A	NEXT	82	AA3F+1	A004
A08E	DATA	83	A70B+1	A006
A092	INPUT	84	A922+1	A008
A097	DIM	85	AD00+1	A00A
A09A	READ	86	A94E+1	A00C
A09E	LET	87	A7B8+1	A00E
A0A1	GØTØ	88	A6B5+1	A010
A0A5	RUN	89	A690+1	A012
A0A8	IF	8A	A73B+1	A014
A0AA	RESTØRE	8B	A619+1	A016
A0B1	GØSUB	8C	A69B+1	A018
A0B6	RETURN	8D	A6E5+1	A01A
A0BC	REM	8E	A74E+1	A01C
A0BF	STØP	8F	A637+1	A01E
A0C3	ØN	90	A75E+1	A020
A0C5	NULL	91	A67A+1	A022
A0C9	WAIT	92	B431+1	A024
A0CD	LØAD	93	FFF3+1	A026
A0D1	SAVE	94	FFF6+1	A028
A0D5	DEF	95	AFDD+1	A02A
A0D8	PØKE	96	B428+1	A02C
A0DC	PRINT	97	A82E+1	A02E
A0E1	CØNT	98	A660+1	A030
A0E5	LIST	99	A4B4+1	A032
A0E9	CLEAR	9A	A68B+1	A034
A0EE	NEW	9B	A460+1	A036
A0F1	TAB(	9C		
A0F5	TØ	9D		
A0F7	FN	9E		
A0F9	SPC(	9F		
A0FD	THEN	A0		
A101	NØT	A1		
A104	STEP	A2		
A108	+	A3		
A109	-	A4		
A10A	*	A5		
A10B	/	A6		
A10C	↑	A7		
A10D	AND	A8		
A110	ØR	A9		
A112	>	AA		
A113	=	AB		
A114	<	AC		
A115	SGN	AD	B7D8	A038
A118	INT	AE	B862	A03A
A11B	ABS	AF	B7F5	A03C
A11E	USR	B0	000A	A03E
A121	FRE	B1	AFAD	A040
A124	PØS	B2	AFCE	A042
A127	SQR	B3	BAAC	A044
A12A	RND	B4	BBC0	A046
A12D	LØG	B5	B5BD	A048
A130	EXP	B6	BB1B	A04A
A133	CØS	B7	BBFC	A04C
A135	SIN	B8	BC03	A04E
A139	TAN	B9	BC4C	A050
A13C	ATN	BA	BC99	A052
A13F	PEEK	BB	B41E	A054
A143	LEN	BC	B38C	A056
A146	STR$	BD	B03C	A058
A14A	VAL	BE	B3BD	A05A
A14D	ASC	BF	B39B	A05C
A150	CHR$	C0	B2FC	A05E
A154	LEFT$	C1	B310	A060
A159	RIGHT$	C2	B33C	A062
A15F	MID$	C3	B347	A064

TABLE AS BELOW EXAMPLES

EN— 45 4E C4
FOR 46 4F D2
NEXT 4E 45 58 D4

```
10 PRINT"BASIC LØØKUP/JUMP TABLES"
15 PRINT:PRINT
20 PRINT      "WØRD                JUMP     J TØ"
30 PRINT      "LØC   WØRD  TØKEN   TØ       LØC"
33 PRINT
35 AA=40960
37 T=128
40 FØR A=41092 TØ 41200
50 D=A:GØSUB 1000
60 PRINTH$"  ";
70 GØSUB2000
80 PRINTW$;
85 ND=2:D=T:GØSUB 1005
86 T=T+1
87 PRINTTAB(14);H$;
90 GØSUB3000
110 PRINTTAB(18);
120 PRINTH$;
125 PRINT"+1";
140 PRINTTAB(26);
150 D=AA
160 GØSUB1000
170 PRINTH$
180 AA=AA+2
190 NEXT A
195 PRINT
200 FØR A=41201 TØ 41236
210 D=A:GØSUB1000
220 PRINTH$;"  ";
230 GØSUB 2000
235 PRINTW$;
240 ND=2:D=T:GØSUB1005
242 T=T+1
245 PRINTTAB(14);H$
250 NEXT A
260 PRINT
270 FØR A=41237 TØ 41315
280 D=A:GØSUB1000
290 PRINTH$;"  ";
300 GØSUB2000
305 PRINTW$;
310 ND=2:D=T:GØSUB1005
315 PRINTTAB(14);H$;
317 T=T+1
320 GØSUB3000
330 PRINTTAB(18);
340 PRINTH$;
350 PRINTTAB(26);
360 D=AA
370 GØSUB1000
380 PRINTH$
390 AA=AA+2
400 NEXT A
999 END
1000 ND=4
1005 H$=""
1010 FØR I= ND-1 TØ 0 STEP -1
1020 H=INT(D/16↑I)
1030 D=D-H*16↑I
1040 IF H>9THENH=H+7
1050 H$=H$+CHR$(48+H)
1060 NEXT
1070 RETURN
2000 W$=""
2010 W=PEEK(A)
2020 W$=W$+CHR$(W)
2030 IF W<127 THEN A=A+1:GØTØ 2010
2040 RETURN
3000 D=PEEK(AA)
3010 D=D+256*PEEK(AA+1)
3020 GØSUB1000
3030 RETURN
```

Appendix 2

BASIC ROM ROUTINES AND ENTRY POINTS

These notes do not claim to be complete or even completely error-free. They are the result of our disassemblies and input from other sources. They constitute a fairly useful reference to have at hand while looking through other, as yet unexplored parts of ROM BASIC.

0000	Warmstart jump
0003	Through-RAM jump for system 'OK' message printer
00A1	General purpose alterable JMP instruction
00BC	CHRGET - get next char from line of BASIC or whatever
00C2	CHRGOT - get previous char
A1A1	Look back through stack ??
A212	Check for OM and stack overflow
A24C	'OM' error routine
A24E	Error printer; caller sets X-reg to error code
A357	Input to buffer 'til <cr>; put null at end of buffer
A386	Input from FFEB
A399	Toggle <ctrl>O flag
A432	Find BASIC line whose is in $11,12; put addr of ptr of that line in $AA,AB
A477	Point $C3,C4 at $301; reset str and array ptrs; reset stack to $1FC;put 0301 in $8F,90;0 in $8C; 0 in $61; $68 in $65 (??)
A491	Clear stack; 0 in $8C and $61
A5C2	Top of main BASIC execution loop
A5FC	Entry to BASIC execution loop
A5FF	Do a line of BASIC
A629	JMP $FFF1 for <ctrl>C check
A636	<ctrl>C entry point
A77F	Get dec from buffer using BC; put value in $11,12
A866	Put null at end of buffer;cr/lf; nulls from $0D
A86C	Output cr/lf w/ nulls from $0D
A8C3	Message printer; A,Y (lo,hi) point to message, which ends with a null
A8E0	Output one blank
A8E3	Output '?'
A8E5	Output char in A; update $0E and do cr/lf if necessary

A925 BASIC INPUT routine less
clear <ctrl>O flag function
A946 Output '? '; jump to A357
AAC1 Get 16-bit value from BASIC line using BC routine;
a subsequent call to AE05 will put
the value in $AE,AF; no TM check
AAAD Like AAC1, but does TM error check
ABA0 Put 0 in $5F; call $BC; goto B887 if
numeric character ???
ABD8 16 bit complement using AE05/AFC1 ?
ABF5 Checks for '(', calls AAC1, checks for ')'
ABFB SN error if next char not ')'
ABFE SN error if next char not '('
AC01 SN error if next char not ';'
AC03 SN error if next char not what's in Acc
AC0C SN error printer
AD0B Get variable name from BASIC line using
$BC; put addr of var in $95,6; also
in Acc and Y register
AD53 Expects variable name in $93,94; finds
addr of variable and puts it in
$95,96 and Acc and Y; puts 0 in $61
AE05 INVAR; converts FP value in FPA to 15-bit
signed binary and puts it in $AE,AF
AE85 BS error
AE88 FC error
AFC1 OUTVAR; puts 0 in $5F; converts 15-bit
signed binary in Acc,Y (hi,lo) to
floating point and leaves it in FPA
B0AE Does some housekeeping for message printer
B3AE Get 8 bit value from BASIC line using $BC;
put it in $AE,AF ?
B3F3 Restore saved contents of $BC routine pointer:
put ($BA,BB) into $C3,C4
B4D0 Normalize FP value ??
B774 Store FP value in FPA into 4 bytes
of variable storage starting at ($71,2)
B887 Check for arith operators; long!
B95A Prints current line as contained
in $87,88
B95E Prints 16-bit value in Acc,X (hi,lo)
on screen at current cursor location
(in decimal)
B962 Prints contents of $AD,AE (hi,lo)
on screen in decimal
BD11 Coldstart entry point
BEE4 Input char from UART (for 1883
chip at FB0X,like 430 board)
BEF3 Output char in Acc to UART
BEFE Initialize UART at FB0X;
8 bits, 2 stop, no parity
BF07 Input char to Acc from ACIA;
(for 6850 chip at FC0X, like
C II 4P's)
BF15 Output char in Acc to ACIA

BF22	Initialize ACIA; 8 bits, 2 stop, x16 clock
BF2D	CRT driver: output car in Acc
FD00	Get char from polled keyboard
FE00	ROM monitor coldstart entry
FE43	ROM monitor warmstart entry
FF00	<break> routine entry
FFEB	System input char routine (to Acc)
FFEE	System output char (from Acc)
FFF1	<ctrl>C check routine
FFF4	System LOAD routine
FFF7	System SAVE routine

Appendix 3

BASIC DEMO PROGRAM AND DUMPS

Here is a simple sample BASIC program, along with dumps of memory showing exactly how the program and all variables were stored, along with page zero with its flags and pointers. Follow through this, and there should be little question where and how the interpreter finds and keeps track of things.

```
10 REM BASIC STORAGE DEMO
20 LET X=23:PRINT X
30 INPUT YESSIRREE:PRINTYE
40 LET A$="ABC":PRINTA$
50 INPUT B$:PRINTB$
60 DIM A(3,2)
70 FOR I=1TO3:FOR J=1TO2
80 A(I,J)=(J<I):PRINTA(I,J);
90 NEXT J,I
100 END
```

Remembering that the BASIC storage format is:

[pointer lo,hi] [line lo,hi] [pgm text ASCII & tokens] [null]

and armed with your list of tokens, you should be able to make it to the double null pointer at the end of the program at $03A8,9. Single variables go through $03CD, and the array goes through $0406.

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0300  (00) [1A] 03 0A 00 [8E] 20 42 41 53 49 43 20 53 54 4F
0310  52 41 47 45 20 44 45 4D 4F (00) 29 03 14 00 [87] 20
0320  58 [AB] 32 33 3A [97] 20 58 (00) 3D 03 1E 00 [84] 20 59
0330  45 53 53 49 52 52 45 45 3A [97] 59 45 (00) 50 03 28
0340  00 [87] 20 41 24 [AB] 22 41 42 43 22 3A [97] 41 24 (00)
0350  5D 03 32 00 [84] 20 42 24 3A [97] 42 24 (00) 6A 03 3C
0360  00 [85] 20 41 28 33 2C 32 29 (00) 7E 03 46 00 [81] 20
0370  49 [AB] 31 [9D] 33 3A [81] 20 4A [AB] 31 [9D] 32 (00) 98 03
0380  50 00 41 28 49 2C 4A 29 [AB] 28 4A AC 49 29 3A [97]
0390  41 28 49 2C 4A 29 3B (00) A2 03 5A 00 [82] 20 4A 2C
03A0  49 (00) A8 03 64 00 [80] (00) 00 00 [58 00] 85 38 00 00
```

○ — NULL
—— POINTER
~~~ LINE #
□ TOKEN

SIMPLE VARIABLES FROM $03AA-$03CD
~~~

```
03B0  59 45 8E 40 E4 00 41 80 03 47 03 00 42 80 07 F9
03C0  1F 00 49 00 83 00 00 00 4A 00 82 40 00 00 41 00
03D0  39 00 02 00 03 00 04/00 00 00 00/00 00 00 00/00
03E0  00 00 00/00 00 00 00/00 00 00 00/00 00 00 00/81
03F0  80 00 00/81 80 00 00/00 00 00 00/00 00 00 00/00
0400  00 00 00/81 80 00 00 24 24 24 24 24 24 24 24 24

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1FF0  24 24 24 24 24 24 24 24 24 42 59 45 2D 42 59 45

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  4C 74 A2 4C C3 A8 05 AE C1 AF 4C 88 AE 04 00 48
0010  38 FF FF 94 3A 99 00 3A 00 49 53 54 00 00 29 3A
0020  97 41 28 49 2C 4A 29 3B 00 3B 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040  FB FF FF FF FF FB FF FF FF FF FF FB F7 FF FB FF
0050  FF FF FF FF FF FF FF FF FF FF FF 22 22 44 00 FF
0060  01 00 00 04 00 68 65 00 06 92 A1 FF FF FF FF FF
0070  FF 92 A1 47 B9 FF 04 00 FF 01 03 AA 03 CE 03 07
0080  04 F9 1F 00 20 00 20 64 FF 64 00 A7 03 00 00 00
0090  03 1A 00 49 00 12 03 04 03 FF 00 00 BC 00 68 00
00A0  03 4C 02 00 07 04 F9 1E FD 00 A8 03 06 92 68 00
00B0  20 00 00 80 00 00 40 00 92 A1 98 A1 E6 C3 D0 02
00C0  E6 C4 AD 16 00 C9 3A B0 0A C9 20 F0 EF 38 E9 30
00D0  38 E9 D0 60 80 4F C7 52 FF FF 80 00 DC 00 20 01
00E0  FF FF FF FF FF FF FF 20 FF FF FF FF FF FF 40 D7
00F0  40 D7 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0100  20 31 30 30 00 30 30 30 FF FF FF FF FF FF FF FF
0110  FF FF 7F FF FF FF 07 BA E0 06 00 22 0D BA E0 18
```

03B0 59 45 = YE; 8E 40 E4 00 41 80 = A$; 03 47 03 00 42 80 = B$; 07 F9

03C0 1F 00 49 00 = I; 83 00 00 00 4A 00 = J; 82 40 00 00 41 00 = A

ARRAY SPACE FROM $03CE TO $0406

03D0 39 00 = LEN; 02 = #SSS; 00 03 00 04 / 00 00 00 00 / 00 00 00 00 / 00

03E0 00 00 00 / 00 00 00 00 / 00 00 00 00 / 00 00 00 00 / 81

03F0 80 00 00 / 81 80 00 00 / 00 00 00 00 / 00 00 00 00 / 00

0400 00 00 00 / 81 80 00 00 24 24 24 24 24 24 24 24 24

VALUE OF -1. SEE CHAPTER 1.

← THAT FAMILIAR $24 FREE MEMORY

1FF0 24 24 24 24 24 24 24 24 24 42 59 45 2D 42 59 45

← FREE MEM. B Y E — B Y E

NOBODY HAS DONE ANY USR'S SINCE LAST COLDSTART

MY SYSTEM NEEDS EXTRA NULLS FOR TAPE. I TYPED "NULL 4" BEFORE SAVING THE SAMPLE PROGRAM

0000 4C 74 A2 4C C3 A8 05 AE C1 AF 4C 88 AE 04 00 48

LAST CHAR TYPED WAS <CR>; CHAR CNTR RESET

0010 38 FF FF 94 3A 99 00 3A 00 49 53 54 00 00 29 3A

0020 97 41 28 49 2C 4A 29 3B 00 3B 00 00 00 00 00 00

0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0040 FB FF FF FF FF FB FF FF FF FF FF FB F7 FF FB FF

0050 FF FF FF FF FF FF FF FF FF FF FF 22 22 44 00 FF

INPUT BUFFER WITH BITS OF HISTORY. NOTE TOKENIZED "SAVE: LIST" ENDING AT $16; REMAINS OF WORD "LIST" ENDING AT $1B; ASCII REMAINS OF LINE 80 ENDING AT $27.

0060 01 00 00 04 00 68 65 00 06 92 A1 FF FF FF FF FF

0070 FF 92 A1 47 B9 FF 04 00 FF 01 03 AA 03 CE 03 07

SINGLE VARS; ARRAYS; BEGINNING OF FREE MEMORY

0080 04 F9 1F 00 20 00 20 64 FF 64 00 A7 03 00 00 00

8K MACHINE; BASIC PGM; LAST USED STRING SPACE

0090 03 1A 00 49 00 12 03 04 03 FF 00 00 BC 00 68 00

00A0 03 4C 02 00 07 04 F9 1E FD 00 A8 03 06 92 68 00

00B0 20 00 00 80 00 00 40 00 92 A1 98 A1 E6 C3 D0 02

GOOD OL' CHRGET - STILL POINTING AT THE INPUT BUFFER FROM AN IMM. MODE COMMAND

00C0 E6 C4 AD 16 00 C9 3A B0 0A C9 20 F0 EF 38 E9 30

00D0 38 E9 D0 60 80 4F C7 52 FF FF 80 00 DC 00 20 01

00E0 FF FF FF FF FF FF FF 20 FF FF FF FF FF FF 40 D7

00F0 40 D7 FF FF FF FF FF FF FF FF FF FF FF FF FF FF

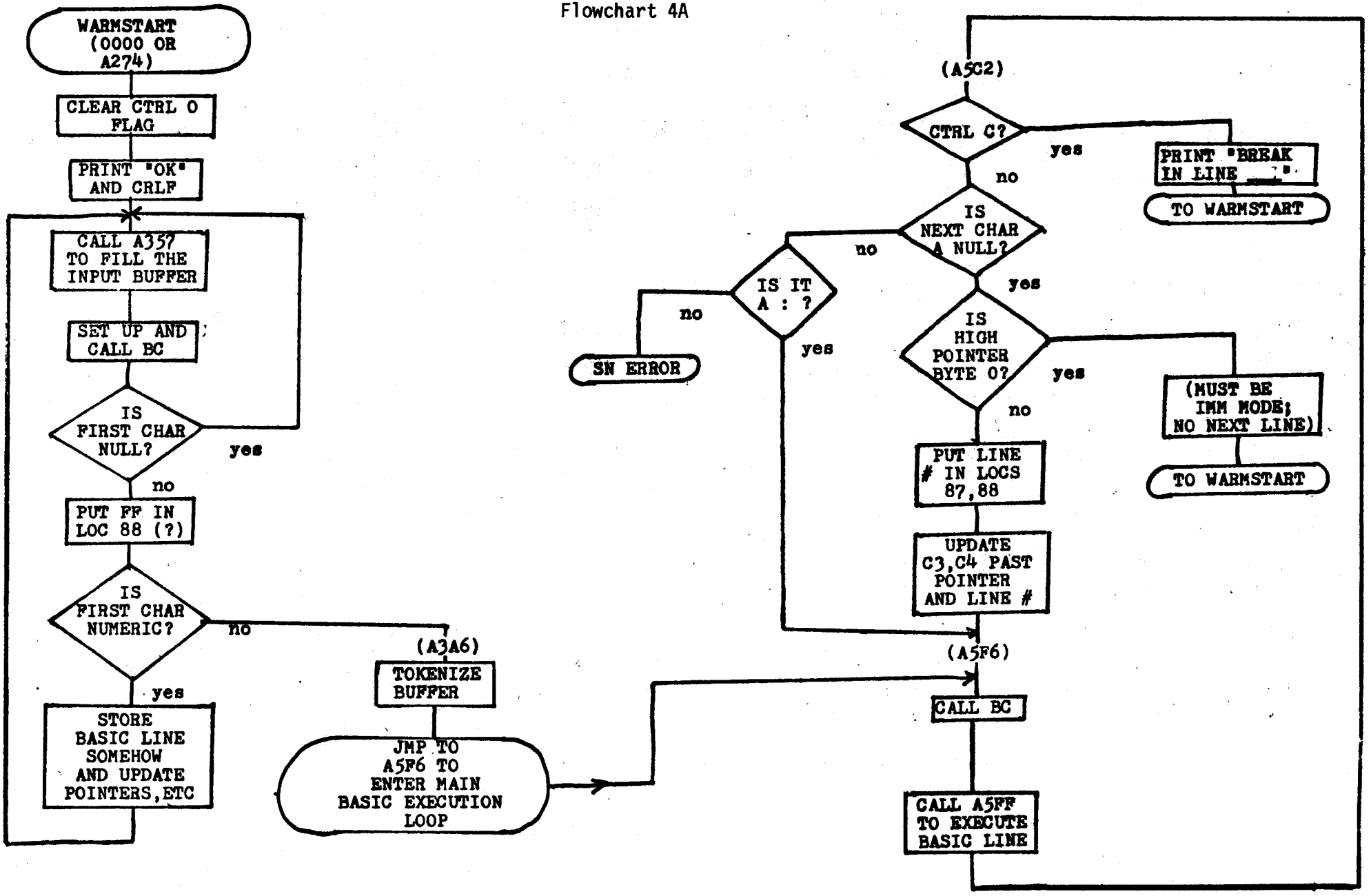
0100 20 31 30 30 00 30 30 30 FF FF FF FF FF FF FF FF

THE LAST NUMBER IT PRINTED WAS (LINE#)100

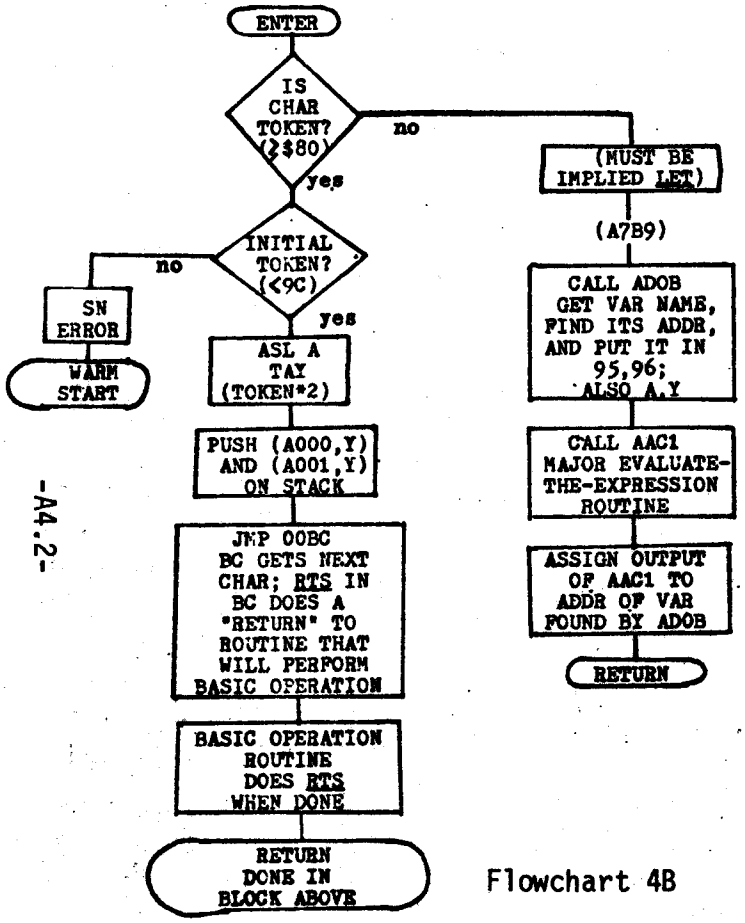
0110 FF FF 7F FF FF FF 07 BA E0 06 00 22 0D BA E0 18

WARMSTART AND MAIN BASIC EXECUTION LOOP

Flowchart 4A

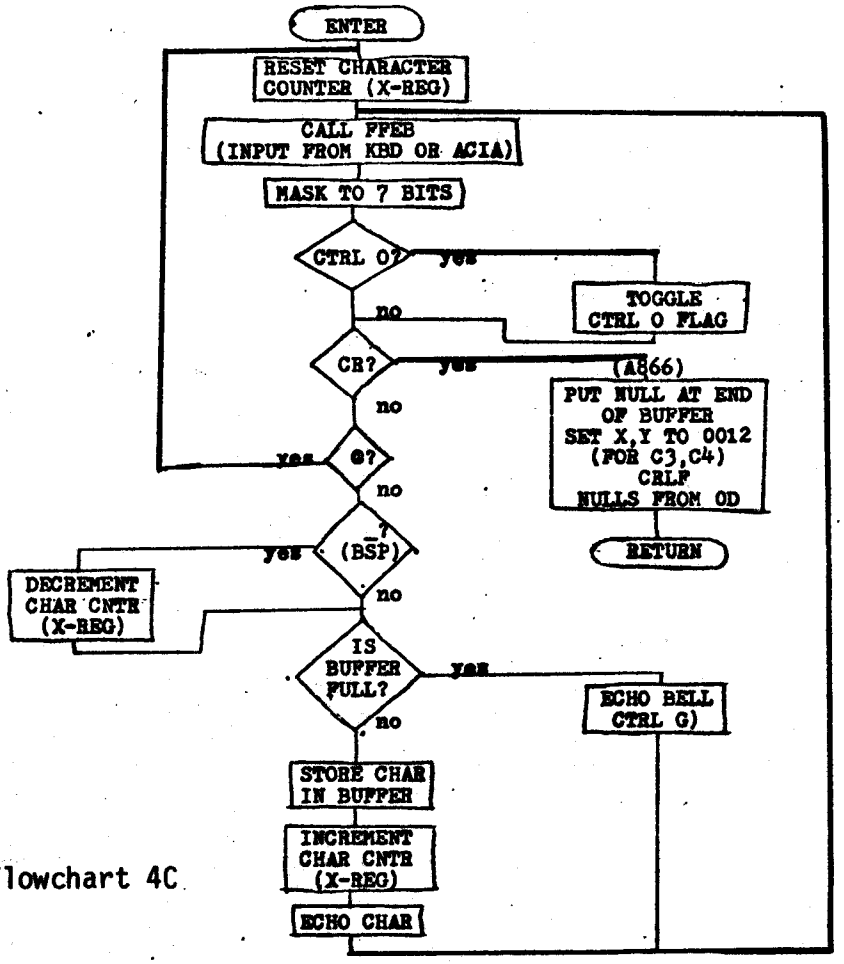


A5FF "EXECUTE THIS LINE OF BASIC" ROUTINE



Flowchart 4B

A357 "FILL THE BUFFER" ROUTINE



Flowchart 4C

Appendix 5

MULTIPLE BASIC PROGRAMS

The following is a sequence of instructions that will put three BASIC programs in different places in memory at the same time, along with a menu program that lets you select which of the programs you want to run.

```
10 INPUT"PROGRAM  (1/2/3)";N
20 POKE 122,3+2*N
30 RUN

POKE122,5:POKE 768+512,0:NEW

10 FOR I=1 TO 20
20 ?I;I↑2
30 NEXT
40 ?"TYPE 'RUN 100' TO CONTINUE
50 END
100 POKE 122,3:RUN

POKE 122,7:POKE 768+1024,0:NEW

10 ?"NOW IS THE TIME FOR ALL GOOD
20 ?"MEN TO COME TO THE AID OF
30 ?"THEIR COUNTRY."
40 ?:?:?
50 ?"TYPE 'RUN 100' TO CONTINUE
60 END
100 POKE 122,3:RUN

POKE 122,9:POKE 768+512+1024,0:NEW

10 FOR I=1 TO 30:?:NEXT
20 FOR I=1 TO 200
30 X=60*RND(8):Y=25*RND(8)
40 POKE 53248+INT(X)+64*INT(Y),42
50 NEXT
60 ?"TYPE 'RUN 100' TO CONTINUE"
70 END
100 POKE 122,3:RUN
RUN 100
```

The immediate mode POKEs reset the BASIC workspace pointer and put in initial nulls; the NEWs set the other pointers. Try it!

Appendix 6

23-BIT INVAR ROUTINE

This routine does about the same thing as INVAR ($AE05), but to 23 bits' precision instead of 16. The philosophy is to shift the FP number in the FPA by just enough to right justify it. The idea would work for 24 bits, but the loop decrementing X doesn't work for zero times through. Nothing tricky -- but it's useful when 16 bits isn't enough. (Or when you're too lazy to mess around in BASIC to make a number >2↑15 by making a negative number.)

```
1000 A918    LDA $18
1002 38      SEC
1003 E5AC    SBC $AC
1005 297F    AND $7F
1007 AA      TAX
1008 A980    LDA $80
100A 05AD    ORA $AD
100C 85AD    STA $AD
100E 46AD    LSR $AD
1010 66AE    ROR $AE
1012 66AF    ROR $AF
1014 CA      DEX
1015 D0F7    BNE $100E
1017 60      RTS
```

Appendix 7

OTHER 'SOFT BASICS

The folks at Microsoft are no dummies. Not only did they write the first widely used BASIC for small computers, they did it in a way which enabled them to deliver essentially the same interpreter to a number of micro manufacturers very quickly -- thus, giving them more return on their initial investment and establishing their version as the industry standard.

It seems to us that BASIC was written in a higher level language (was it PL/M?), subsequently modified per specifications from manufacturers or distributors, and then assembled into the appropriate machine code version. That is the reason that the guts of any Microsoft BASIC and its use of memory are so similar from machine to machine. As we noted elsewhere, this also resulted in some unused code here and there. About the time we were writing about the unused "WANT-SIN-COS-TAN-ATN" mentioned in Chapter 4, we were also reading up on the KIM cassette version which allows you to scrap the trig functions in favor of more available RAM. When we got our hands on a SYM version, we found that it too gave this option -- even though this is a ROM version. Some other goodies (e.g. direct use of hex from BASIC) and completely different I/O routines replace the trig functions which must be loaded from tape or disk.

With some care it is possible to RUN a simple BASIC program from one machine on one made by another manufacturer, but that is too much trouble to be really useful. For ML dabblers who read the hobby magazines, intimate knowledge of the Microsoft BASIC bloodline can be more useful -- sometimes for the good program which may be implemented on your Challenger with the change of a few POKE or PEEK addresses and occasionally for obtaining more information about how your BASIC works by reading what someone else has discovered about their AIM or APPLE. Indeed, what is now Chapter 8 began as an attempt to illustrate this point by modifying Virginia Brady's APPLE program in MICRO 19. It turned out that it was better to rewrite the whole program, but that still makes the point -- we (and now you) have a good program because of our knowledge of the inner workings of BASIC.

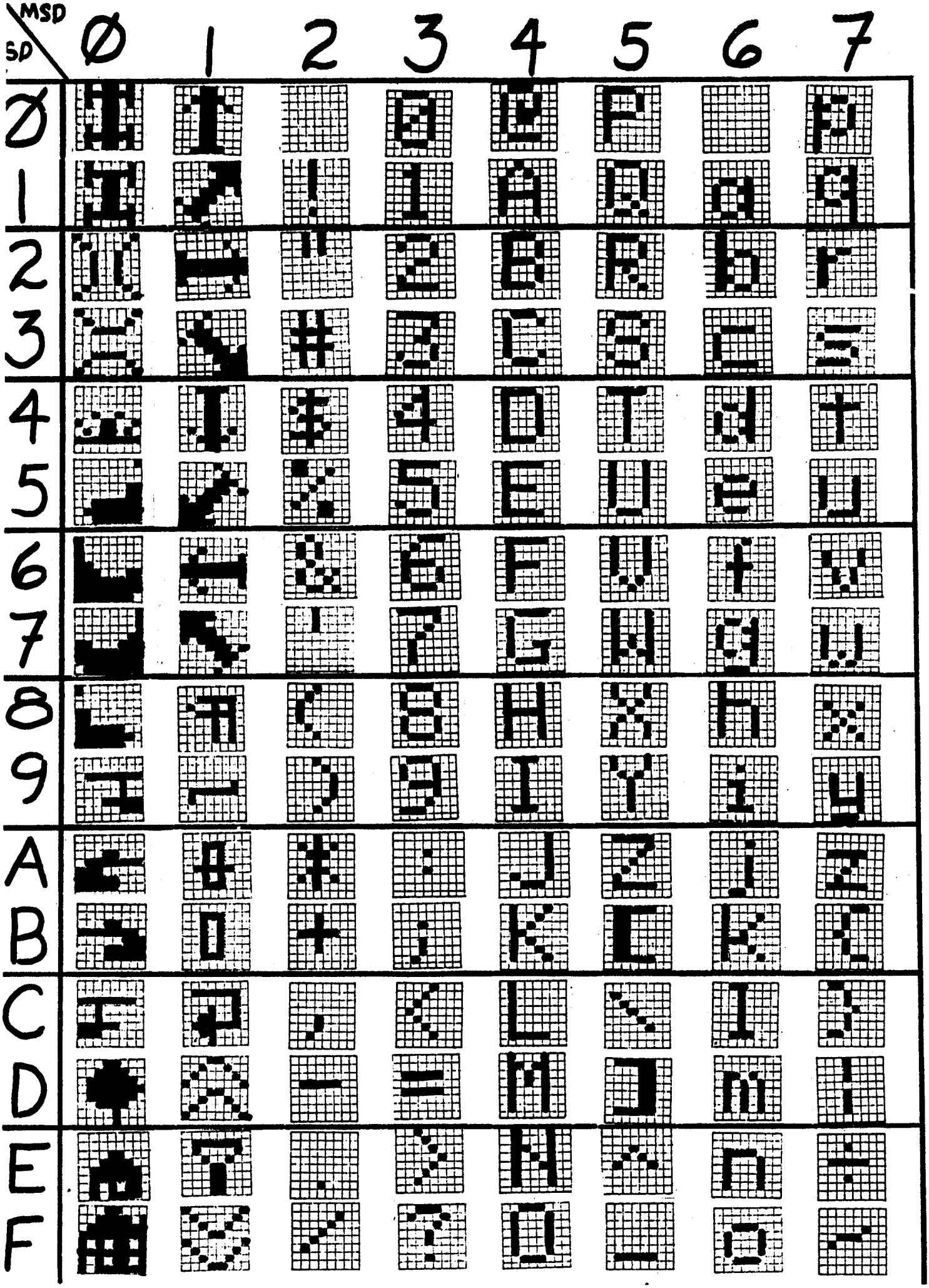
The cross-reference on the next page will be of help if you want to dabble (we almost said plagiarize) with programs from other machines. We have included only 6502 information, but even scrutiny of the TRS-80 can be enlightening. We just read a review of the TRS-80 DISASSEMBLED HANDBOOK by R.M. Richardson which makes this point.

BRIEF MICROSOFT 6502 CROSS-REFERENCE

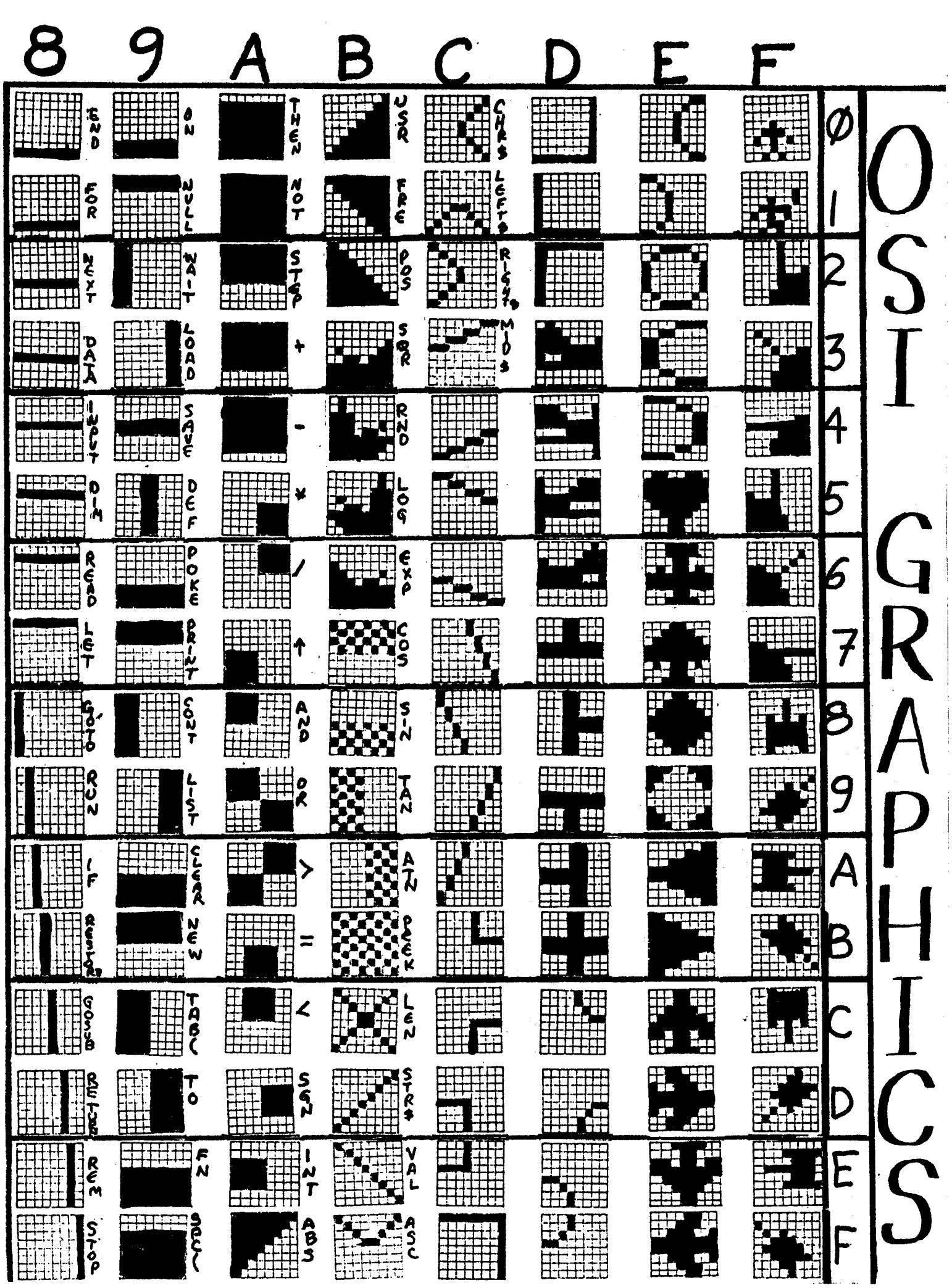
	OSI	KIM	SYM	AIM	APPLE	OLD PET	NEW PET
Input buffer	13-5A	1B-62	1E-65	16-5D	200-FF	0A-59	20-50
CHRGET	BC	C0	CC	BF	B1	C2	70
POINTERS TO:							
Workspace	79,A	78,9	7B,C	73,4	67,8	7A,B	28,29
Variables	7B,C	7A,B	7D,E	75,6	69,A	7C,D	2A,B
Arrays	7D,E	7C,D	7F,80	77,8	6B,C	7E,F	2C,D
FREe memory	7F,80	7E,F	81,2	79,7A	6D,E	80,1	2E,F
Strings	81,2	80,1	83,4	7B,7C	6F,70	82,3	30,1
Top of memory	83,4	84,5	87,8	7F,80	73,4	84,5	84,5

More detailed lists of memory locations for the machines listed may be found in old issues of the magazines MICRO, 6502 USER NOTES, and COMPUTE. Some explicit references are found in the Bibliography.

LSD \ MSD	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX CTRL-C	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	END	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	÷
F	SI CTRL-O	US	/	?	O	_	o	~


MSD
LSD
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

OSI GRAPHICS



BIBLIOGRAPHY

6502

MCS6500 MICROCOMPUTER FAMILY PROGRAMMING MANUAL, MOS Technology, Norristown, PA, 1976

Rodnay Zaks, PROGRAMMING THE 6502, SYBEX, 1979

Lance A. Leventhal, 6502 ASSEMBLY LANGUAGE PROGRAMMING, OSBORNE/McGraw-Hill, 1979

NUMBER SYSTEMS

Donald Knuth, THE ART OF COMPUTER PROGRAMMING. VOLUME III; SORTING AND SEARCHING, Addison-Wesley, 1978

OSI

E.D. Morris, "TOKENS", MICRO, August 1979, p. 20

S.R. Murphy, "Some Useful Memory Locations for OSI BASIC in ROM", MICRO, November 1979, pp.18:9-18:10

Alvin L. Hooper, Letter in response to Jim Butterfield's "Inside PET BASIC" article (listed below), MICRO, July 1979, pp. 14:15-14:16

OTHER BASICS

Virginia Brady, "Data Statement Generator", MICRO, December 1979, pp19:5-19:7

Jim Butterfield, "BASIC MEMORY MAP (Page 0)", COMPUTE, January/February, 1980, Issue 2, p. 41

Jim Butterfield, "Inside PET BASIC", MICRO, December 1978-January 1979, pp.8:39-8:41

Gary A. Creighton, "A Partial List of PET Scratch Pad Memory", MICRO, August-September 1978, Back Cover

William F. Leubbert, "What's Where in the APPLE", MICRO, August 1979, pp. 15:29-15:36

Gregory Yob, "Personal Electronic Transactions", CREATIVE COMPUTING, September 1979 5, # 9, pp. 178-182 and October 1979, 5, #10, pp.180-183

INDEX

```
10 REM BASIC STORAGE DEMO
20 LET X=23:PRINT X
30 INPUT YESSIRREE:PRINTYE
40 LET A$="ABC":PRINTA$
50 INPUT B$:PRINTB$
60 DIM A(3,2)
70 FOR I=1TO3:FOR J=1TO2
80 A(I,J)=(J<I):PRINTA(I,J);
90 NEXT J,I
100 END
```

[pointer] [line] [pgm; ASCII & tokens] (null)

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0300  00 1A 03 0A 00 8E 20 42 41 53 49 43 20 53 54 4F
0310  52 41 47 45 20 44 45 4D 4F 00 29 03 14 00 87 20
0320  58 AB 32 33 3A 97 20 58 00 3D 03 1E 00 84 20 59
0330  45 53 53 49 52 52 45 45 3A 97 59 45 00 50 03 28
0340  00 87 20 41 24 AB 22 41 42 43 22 3A 97 41 24 00
0350  5D 03 32 00 84 20 42 24 3A 97 42 24 00 6A 03 3C
0360  00 85 20 41 28 33 2C 32 29 00 7E 03 46 00 81 20
0370  49 AB 31 9D 33 3A 81 20 4A AB 31 9D 32 00 98 03
0380  50 00 41 28 49 2C 4A 29 AB 28 4A AC 49 29 3A 97
0390  41 28 49 2C 4A 29 3B 00 A2 03 5A 00 82 20 4A 2C
03A0  49 00 A8 03 64 00 80 00 00 00 58 00 85 38 00 00
03B0  59 45 8E 40 E4 00 41 80 03 47 03 00 42 80 07 F9
03C0  1F 00 49 00 83 00 00 00 4A 00 82 40 00 00 41 00
03D0  39 00 02 00 03 00 04 00 00 00 00 00 00 00 00 00
03E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 81
03F0  80 00 00 81 80 00 00 00 00 00 00 00 00 00 00 00
0400  00 00 00 81 80 00 00 24 24 24 24 24 24 24 24 24
```

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1FF0  24 24 24 24 24 24 24 24 24 42 59 45 2D 42 59 45
                                  B  Y  E  -  B  Y  E
```

WARMS · "OK" · INVAR · OUTVAR · USR · NULLS · TERM WIDTH

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  4C 74 A2 4C C3 A8 05 AE C1 AF 4C 88 AE 04 00 48
0010  38 FF FF 94 3A 99 00 3A 00 49 53 54 00 00 29 3A
0020  97 41 28 49 2C 4A 29 3B 00 3B 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040  FB FF FF FF FF FB FF FF FF FF FF FB F7 FF FB FF
0050  FF FF FF FF FF FF FF FF FF FF FF 22 22 44 00 FF
0060  01 00 00 04 00 68 65 00 06 92 A1 FF FF FF FF FF
0070  FF 92 A1 47 B9 FF 04 00 FF 01 03 AA 03 CE 03 07
0080  04 F9 1F 00 20 00 20 64 FF 64 00 A7 03 00 00 00
0090  03 1A 00 49 00 12 03 04 03 FF 00 00 BC 00 68 00
00A0  03 4C 02 00 07 04 F9 1E FD 00 A8 03 06 92 68 00
00B0  20 00 00 80 00 00 40 00 92 A1 98 A1 E6 C3 D0 02
00C0  E6 C4 AD 16 00 C9 3A B0 0A C9 20 F0 EF 38 E9 30
00D0  38 E9 D0 60 80 4F C7 52 FF FF 80 00 DC 00 20 01
00E0  FF FF FF FF FF FF FF 20 FF FF FF FF FF FF 40 D7
00F0  40 D7 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0100  20 31 30 30 00 30 30 30 FF FF FF FF FF FF FF FF
0110  FF FF 7F FF FF FF 07 BA E0 06 00 22 0D BA E0 18
```

BUFFER · POINTERS · CHRGET