THE HACKER'S GUIDE TO ADAM™

VOLUME TWO

The Hacker's Guide To Adam, vol. 2

by Ben Hinkle

Published by Peter and Ben Hinkle
117 Northview Rd.
Ithaca, NY 14850

12

Forward

In the year since Ben and I wrote Vol. 1 of the Hacker's Guide (available from us for $12.95, postpaid), we have actually spent less time with our Adam than I anticipated. We have many CP/M programs, but have not used them much, largely because we never have the manuals. We also got a Macintosh (as you can see), which has been a distraction. There is a lot of activity in the Adam community, however, largely with modems and CP/M, and anyone interested should write to the user groups advertized in Family Computing.

Our major project with Adam has been Ben's interpretation of a disassembly printout of SmartBASIC. He took it on out of curiosity last fall, using the information in Vol. 1 and general information from "How to program the Z80" by Rodnay Zaks, "Mapping the Commodore 64" by Seldon Leemon, and "Microsoft BASIC decoded and other mysteries for the TRS-80" by James Lee Farvour. Ben wrote it up, but had to leave for summer school, so Maija and I made the last corrections and printed it out (something that seems to have taken forever). The imperfect copy came from my lack of patience with files that refused to print, and the like. I hope there are not too many errors. If you find any, please let us know.

I think that the Adam is an excellent computer, and a fine introduction to the computer world. This book should be very helpful to anyone who wants to learn SmartBASIC really well, and get beneath the superficial layer of any higher language, into the actual machine. Ben has tried to make vol. 2 more understandable to the novice than vol. 1 was, but you should realize that he is only in the tenth grade, and parts are bound to be hard to follow. If you have questions, please include a SASE in your letter.

Peter Hinkle, July, 1986

At the second printing we thank George Havach and Carl Cummings for pointing out many typos and other errors that have now been corrected.

CONTENTS

1. BASIC Overview	1
2. Zero Page	13
3. Keywords	14
4. Math Routines	19
5. BASIC Commands	29
6. Parser	40
7. Data Table	51
8. Screen Routines	58
9. Tape Routines	64
10. Graphics	73
11. BASIC Changes	77
Appendix 1. Programs	89
Appendix 2. Hello code	90
Appendix 3. Schematics	98
Glossary	104

Chapter 1: BASIC Overview

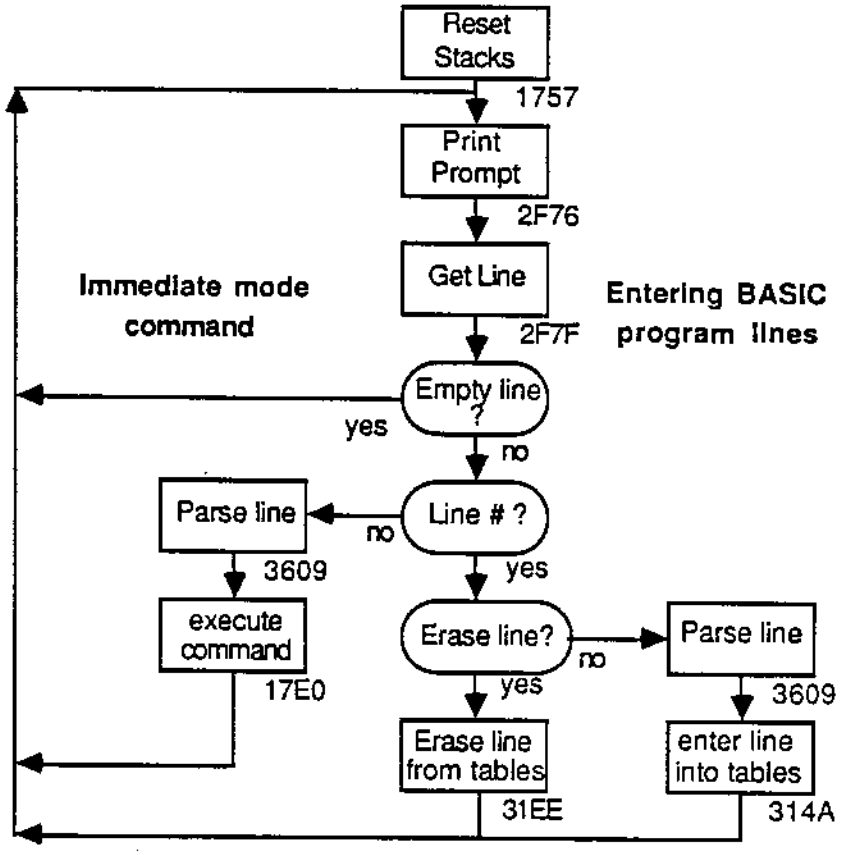
When you load in SmartBASIC by putting the tape in the drive and hitting reset, three things happen: the machine language program that is SmartBASIC is stored in RAM by the boot routine, the Program Counter register of the Z80 microprocessor is set to $100 and the whole thing goes into the labyrinth of subroutines and loops. First it goes to the routine at $4061, which is later replaced by the input buffer. This setup routine writes the interrupt routines to zero page, clears the screen and tape buffers, then looks for and loads the HELLO program from the tape, or, if none exists, prints the message at $043C ("COLECO SmartBASIC V1.0") on the screen. This is what you first see when BASIC is loaded from tape. After the setup routine introduces you to BASIC, it falls into the immediate mode at $3EA3. I call this routine the "Central loop" because it is the heart of BASIC, controlling the flow of execution when you are either typing in a command, waiting as a command is executed, or just watching while the cursor blinks at you. As outlined on the following page, the Central loop reads the keyboard and interprets the line of input.

The Central loop

The Central loop begins by resetting the Z80 Stack Pointer, IX and IY registers to $D380. This clears any GOSUB or FOR...NEXT loops (pointed to by IX and IY), and is used after a program or command has executed, when the stack could have some garbage which is no longer needed. The loop accomplishes this task by calling $1757. After the stacks are reset and the HELLO program or copywrite statement printed, the routine at $2F76 is called to generate the next thing you see on the screen: a return character and a prompt ( ]), which provides you with visual feedback that the loop is working.

The Central loop then calls the subroutine at $2F7F to scan the keyboard for a command or line. This is the routine that is executing when you are in the immediate mode and the cursor is blinking or you are typing in a command. When a program is loaded from tape or disk, the same routine is used but it looks at the tape instead of the keyboard on AdamNet (see vol 1, p. 49). The characters received from the input device are put in a buffer at $3F77 and also printed on the screen. It also acts on any control ASCII you type (e.g. , tab, left arrow, or any other ASCII code in the table at $3051). When a "return" character is received the Z80 returns to the Central loop and parses the line (Parse means interpret. For explanation of jargon please see the glossary). But the loop must first decide if you

actually typed in a line or just pressed the return key without typing anything beforehand. Spaces also don't count as a line. If the line is empty, it prints the prompt again and restarts reading the keyboard. But if the line contains some sort of non-space ASCII, the Central loop calls $3565 to check for the presence of a line number, indicating a program line. This would happen if you were typing in a program, or erasing any unwanted line previously typed in. If a line number is present, it then decides if you want to erase a line (indicated by only typing in the line number). The loop does this by seeing if you typed in anything after the line number. If you didn't, it calls $31EE to erase the line from the tables that store your program as you type or load it in. But if a command follows the line number, it calls $3609 to parse the rest of the line, and then calls $314A to store the line in your growing program tables.



Flowchart of Central Loop

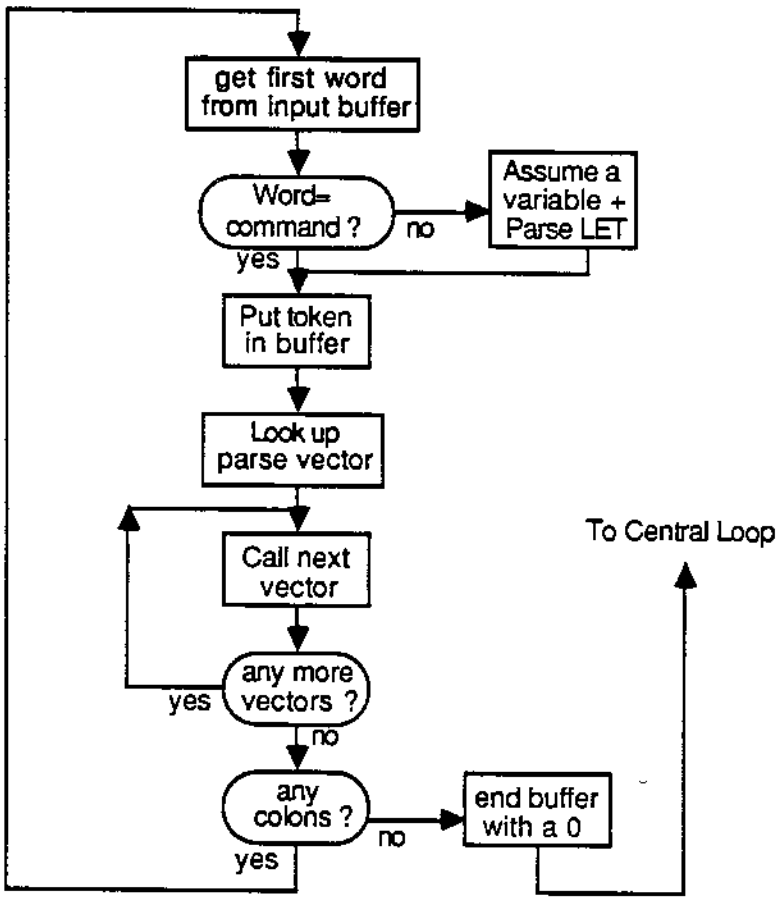
If the Central loop does not find a line number at the beginning of the line, it assumes that you have entered an immediate command (e.g. , SAVE) and it parses the line by calling $3609 and executes the line by calling $17E0, returning to the Central loop when finished.

The Parser

Parsing is the process of replacing the ASCII codes you type in with shorter, more compact tokens in a form called crunch code. It helps speed up execution of the command because less time is spent converting your ASCII line as the command executes. While the speed difference is not noticeable in the immediate mode, it is in programs. This is due to the execution of immediate mode commands directly after parsing, while in programs, the lines are parsed as you type them in. When you RUN the program , the lines are fully parsed and therefore are faster. The routine that parses the input you type in is called "Parse line", or the Parser, and is at $3609. The parser is called by the Central loop, and, after it is done translating the line, returns to the loop, as seen in the flowchart on the following page.

The "Parse line" routine reads the first word in the line and compares it with words in the primary word table at $110. This table has the format of: 1. token, 2. address of vectors in parse vector table lo, hi, 3. number of letters in the command, 4. ASCII of the command. Each command has a different token, which is used during execution. The Parse routine compares the ASCII of the first word with the ASCII of the first command. If they don't match, then it compares it with the second command. This repeats until a match is found or it reaches the end

of the table, when it assumes the first word is a variable and parses it like LET. When a match is found, the routine puts the token of the matched command in a buffer called the Crunch code buffer (e.g. , it would put a '7' in the buffer for a 'PRINT' command because the token for PRINT is 7), and looks at the command's "address of vectors in parse vector table" (2nd and 3rd bytes in primary word table). This address in the parse vector table lists the number of parse vectors for the syntax of that command, followed by the vectors to those Parse routines. An example of this would be "GOTO 30". After finding the GOTO, the Parser only needs to check for a line number, so the entries in the parse vector table that GOTO points to would contain 01 for the number of items that need to be parsed (in this case the line number) followed by the vector to the Parse routine that parses the syntax (in this case the "Parse line number" routine). The reason for doing this is to shorten the amount of space taken up, because many commands share syntax, like



Flowchart of Parse Line

GOSUB 30" and "GOTO 50". Instead of listing the vectors over and over in the primary table, Coleco listed them once, and had the commands point to the required type of syntax.

After the Parser finds the vectors to the command, it calls them in the order in which they are listed. The parse vectors are then responsible for doing most of the parsing. Many times these vectors are looking for commas, equal signs, or words like "at" or "then". These words, and other symbols used only after the commands, are stored in the secondary word table at $0332. Since they are used by Parse routines, they have the format: 1. token, 2. length of word's ASCII, 3. ASCII of word or symbol. As before, when the Parser looked for the command in the primary word table, the parse vectors look for symbols in the secondary word table. But if the

symbol does not match one in the table, an error is printed, and the Z80 returns to the Central loop to get more input from the keyboard. There are some words, like MID, SIN, or POS, called variable commands, that are not in this table. This is probably due to the fact that they require some execution, while words like "at" or commas are only for syntax and do not perform any other function.

As mentioned earlier, the Parser changes the line into a form called crunch code. This code is important if you want to look at execution routines, or if you want to write a new command, because the execution routines use only this form of the line you typed in on the keyboard. A line of crunch code consists of the following: 1. the number of bytes in the crunch code command, 2. the command's crunch code, and 3. a 0 (zero) showing the end of the line. If the line had multiple commands separated by colons, then each command is listed as if it were alone, but with the zero following the last command (e.g. , "PRINT: PRINT: PRINT" would be 1,7,1,7,1,7,0). The crunch code of the command always starts with the token of the command. When you define variables without using LET, the LET token is put in the buffer anyway. What follows the token varies with what you type in, but the symbols in the secondary word table are also listed by their token. For example, "COLOR =..." would start out with: 1. number of bytes in line (depends on what the COLOR equals), 2. $3A (token for COLOR), 3. AA (token for "="), 4. whatever it equals (in crunch code).

Crunch Code Numbers

This brings us to how numbers are stored in crunch code. If the number is a whole number, and is from 0 to 9, then it is the number + $80. Thus 7 becomes $87. If the number is above $A and below $100, then it takes up two bytes: 1. $8A, 2. number in hex. Thus 100 becomes $8A, $64. The $8A and the $80 for the integers are called number types. You can tell what kind of number any number in crunch code is by looking at the number type. Numbers from $100 to $FFFF take three bytes: 1. $8B, 2. number in hex lo, hi. Thus 256 becomes $8B, $00, $01 and 50000 becomes $8B, $50, $C3. Negative numbers that fall into any of the above formats are preceded by $A1 (token for "-"). Any number above $FFFF, in scientific notation, or with a decimal point is stored as a floating point number and has a number type of $92. Thus 1000000 becomes $92, $00, $00, $24, $74, $94.

Crunch Code Variables

Variables are stored in crunch code by assigning a number to any variable you type in. Variables you type first have smaller numbers than later variables, but if you use a variable again later in the program, it still has the same number (e.g. , if the number for variable "x" in line 50 is $20, then if "x" is used again, it still has $20 as its number). Although it would be logical that the first variable you use is assigned the variable number of 1, it isn't, because some ($1B) commands are stored as variables. So the number of the first variable is $01C, and the maximum number is $3FF. The variable is stored in crunch code by adding $8C to the top byte of the variable number (e.g. , $127 becomes $8D, $27). After the number is

stored, it enters any ASCII from the variable name other than the first two characters (e.g. , the "ges" of "pages") in the form: 1. number of characters, 2. ASCII of characters. Thus a variable "point" with a variable number of $01C is stored as: $8C, $1C, $03, $69, $6E, $74. In addition to being stored in crunch code, variables are placed by the Parser in a variable table pointed to by $3EDF. This table is used in execution, and is later explained more fully. String, integer, or dimensioned variables are specified in the variable tables, and not in the crunch code. For example, the crunch code for the line "atom (10, 4) = 3" is $0E, $01, $8C, $1D, $02, $6F, $6D, $B7, $BA, $0A, $B9, $84, $B8, $AA, $83, $00. The interpretation is as follows:

crunch code	meaning
$0E	length of line in bytes
$01	token for LET
$8C, $1D, $02, $6F, $6D	"atom"
$B7	"("
$8A, $0A	"10"
$B9	","
$84	"4"
$B8	")"
$AA	"="
$83	"3"
$00	end of line

Other types of data that need storing are strings and DATA or REM data. Strings are stored in the following form: 1. $91, 2. number of bytes in the string, 3. ASCII of the string. Thus "hello" becomes $91, $05, $68, $65, $6C, $6C, $6F. Things following REM or DATA are similarly stored: 1. $90, 2. number of bytes in DATA, 3. ASCII of DATA. The Parse routine that handles REM and DATA has a bug. For the fix of this DATA Bump Bug, see chapter 11. The following table shows the code that indicates each type of data possible in a line with the length of bytes that follows that type of data.

code ($)	meaning	number of bytes
0- 64	primary word token	1 byte
80-89	number from 0 to 9	1 byte
8A	number from $0A to $FF	2 bytes
8B	number from $100 to $FFFF	3 bytes
8C-8F	variable	at least 3 bytes
90	DATA or REM	at least 2 bytes
91	string	at least 2 bytes
92	floating point number	6 bytes
A0-BD	secondary word token	1 byte

If you are confused about crunch code, there is a program in Appendix 1 that allows you to experiment with it. Even if you are comfortable with crunch code, the program lets you see the crunch code of any command, which helps when you are tracing a command's execution.

Program tables

Once the Parser finishes parsing the line into crunch code, it returns to the Central loop, where it either executes the command, if it is an immediate mode command, or enters it into the program tables. Let us first look at a program line, which is specified by having the first word of ASCII in the line be a number. The Central loop, after calling $3609 to parse the line, calls $314A to enter the crunch code for the line into the program tables, which store your program. The first table, called the crunch code table, is pointed to by $3EE5, and is usually high up in RAM. This is where the crunch code for the line you typed in is stored with all the previous program lines of crunch code. The lines are in descending order, with the first line you typed in being highest in memory. Note that these lines are not in numeric order, but in the order that you typed them in. The crunch code in this table is exactly like the code from the Parser, so it does not include the line number. Line numbers are stored in a table pointed to by $3ED9. The line numbers in this table are in numeric order, and it is usually stored just beneath the crunch code table. The entries have the form: 1. line number lo, hi, 2. address of that line's crunch code in the crunch code table lo, hi. Thus the line numbers are stored in one table, used for optimum speed in finding line numbers for GOTO or GOSUB, and the crunch code for that line is stored in a separate table, as seen in the following diagram.

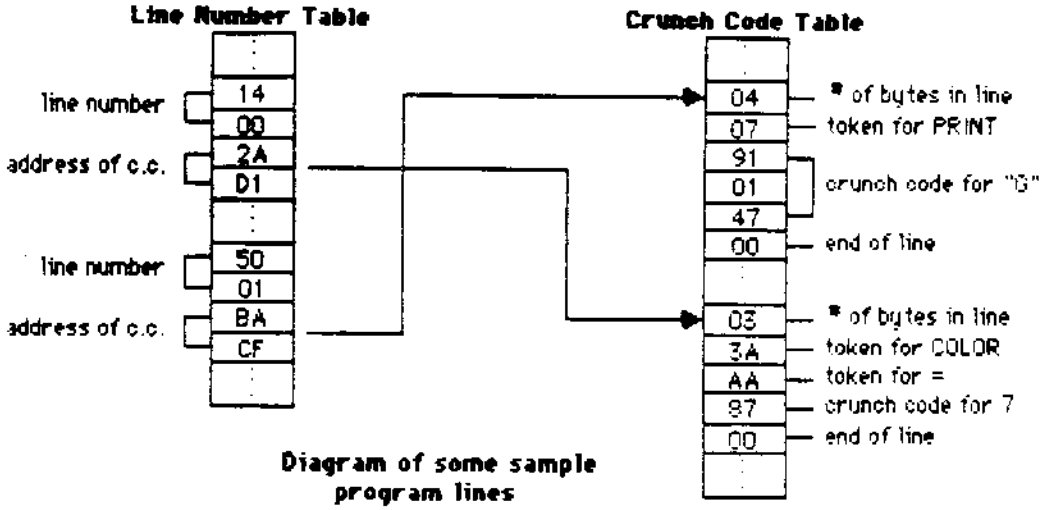
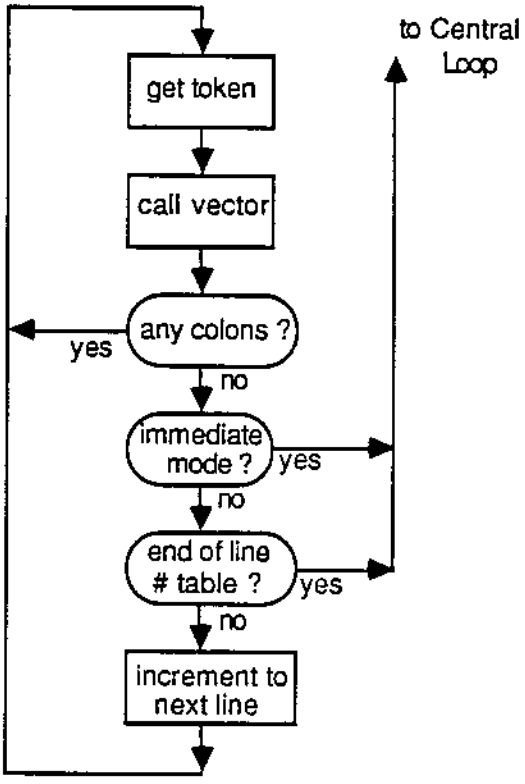


Diagram of some sample program lines

Command Execution

If the line typed in has no line number, then the line requires immediate execution. In these cases, the Central loop calls $17E0, which later falls into $182E, the main execution loop. These routines take the command's token from crunch code put there by the Parser, and pointed to by the DE Z80 register, uses it as an offset to look up the execute routine vector from the table at $1917 and calls it (e.g. , for GOTO, which has a token of 03, the loop would call the third vector in the table). When the command is over, the Z80 returns to the Central loop to read the keyboard for another line of input. If the command is "RUN", the Z80 jumps to a loop at $17E0 which is the "Execution loop". This is called the Program mode because it executes your program. This loop gets tokens from the crunch code table, as pointed to by the line number table, and executes them until an error occurs or it reaches the end of the line number table. When the program ends the Z80 returns to the Central loop to look for keyboard input. The flowchart for the Execution loop is as follows:



Flowchart of Execution loop

While executing a command or program, variables used in the command can change by being assigned a new value or string (e.g. , LET a=6 or INPUT y$). This

is accomplished by changing certain tables: variable table, string space, or the variable value table. The most important of these is the variable table, because it points to the other two. When you create a new variable by typing it in, the Parser makes an entry in this table for some of the variable's essential data. Each entry has the following form: 1. variable type byte, 2. pointer to variable's string or value lo, hi, 3. first two characters of the variable's name (03 if a character isn't present). The type byte has various bits set according to the variable, as follows (a typical numeric variable has a type byte of $01 or $02, strings have $21 or $22):

type byte bit #	meaning
7	variable command (e.g. , MID)
6	variable function (FN)
5	string variable ($)
4	integer variable (%)
3	dimensioned array
2	unused
1	two characters in the name
0	one character in name

The second and third bytes of the entry point to the definition of the variable (string or number). If the variable is a string, then it points to the variable's string in the string space. String space, pointed to by $3EF3 (beginning) and $3EEF (end), has entries in the format: 1. pointer to variable in variable table, 2. number of characters in string, 3. ASCII string. If the variable is a null string, the variable table points to $3F52. If the variable is numeric, then the entry in the variable table points to its value in the variable value table. All the numbers in this table, pointed to by $3EED, are in floating point format. The fourth and fifth bytes of the entry store the variable's ASCII according to bits 0 and 1 of the type byte. Values of integer variables are stored in the variable table as two bytes. Dimensioned arrays are stored in the value table in the format: 1. number of dimensions, 2. depth for each dimension lo, hi, 3. floating point number, pointer to string, or integer number for every entry of the array. This allows for string arrays as well as numeric ones. In defined function variables, the pointer to the string or value points to the function equation in crunch code.

For some strange reason, some commands that can be used in equations are also stored as variables. I call them variable commands, and they take up the first $1B variable entries in the variable table. Instead of the pointer to the string or value, there is a vector to the execution routine of the command. Furthermore, the ASCII characters are replaced by the command's offset into the variable command name table, which stores each command's ASCII characters in the format: 1. number of letters in word, 2. word. This table is pointed to by $3EE1 and $3EE3. The diagram on the following page shows how the variable tables point to each other with some sample variable data.

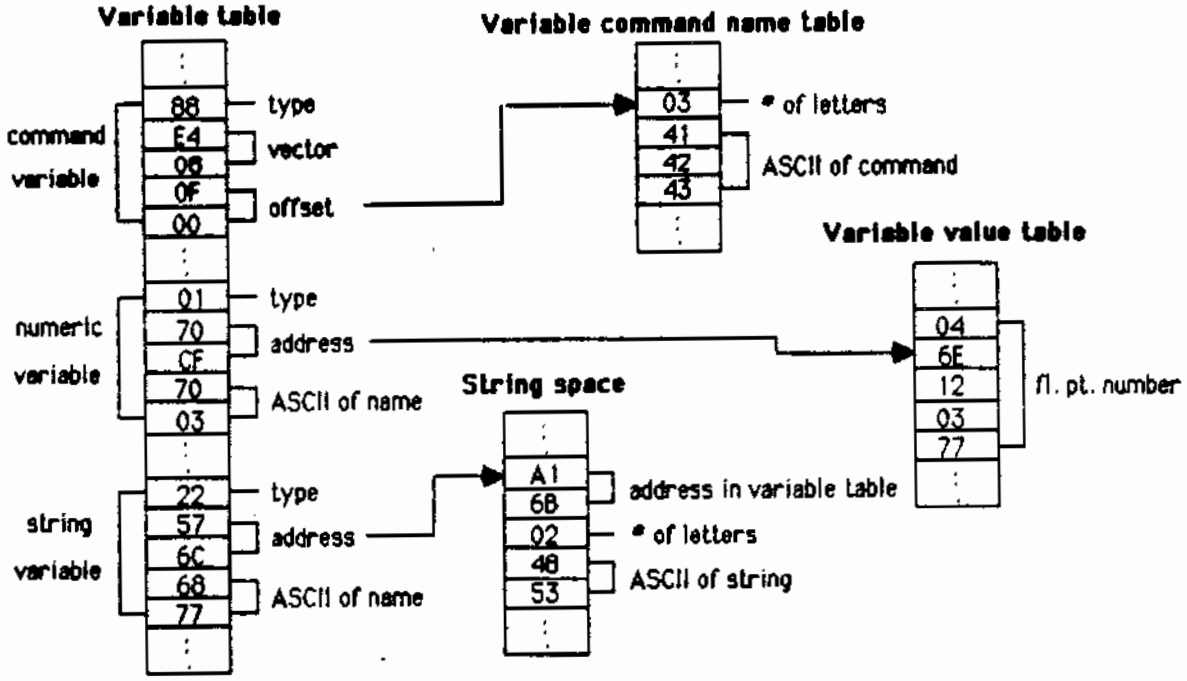


Diagram of some sample variables

The BASIC Stack

The stack in BASIC, when executing a command, has two purposes in addition to being a temporary place to save registers and addresses. It keeps track of nested FOR-NEXT loops and GOSUB commands. When the FOR-NEXT data is pushed to the stack by the FOR routine, IY is pointed to the end of the data, thus indicating the current FOR-NEXT data section as the rest of the stack continues. If IY was already pointing to another section (nested loops), then it is also pushed with the new section, thus preserving it. When the loop is over, it pops off the old IY and continues with that loop. The same theory applies to GOSUB and the IX register. For the exact data pushed, see the FOR or GOSUB commands.

The stack is also used when BASIC computes equations, for priority values (e.g., 2+3*5=17, not 25). Priority is understood by all of us, but how does the stack fit in? Well, BASIC evaluates the equation by comparing the priority of terms as it goes through the line from left to right. Lower priority terms are pushed to the stack to be later popped off when the higher part of the equation is completed. In the example 2+3*5, it first looks at the "2+". Since this is the first term of the equation, nothing is done, yet. It then looks at "3*". Since multiplication has priority over addition, the "2+" is pushed onto the stack and the "3*" now is the current term. When it sees "5 end of equation", it calls the multiplication routine to compute 3*5, pops the "2+" off the stack and adds that to the resulting product, thus getting 17. Variables and commands in equations may seem complex, but they can always be replaced by a single number to let the equation evaluation move on.

Z80 Registers in BASIC

While executing a command, some registers hold data that is used very often

and would slow down BASIC if it were stored in RAM. These registers are "universal". That is, they are not used casually by one routine to store temporary data. They contain things needed by many routines. If a routine needs the register to hold some temporary data, it pushes the old register to the stack, uses the register for its own purpose, and then pops the original register, thus preserving it for other routines. The table below shows each register with the data it holds. Note that the DE register points to the Input Buffer, and not the crunch code, while BASIC is parsing a line. DE is then set to the crunch code after the line is parsed.

register	function
DE	points to current address in crunch code
DE'	points to the start of the parsed line
HL'	points to the current line number
C'	number of bytes left in crunch code line
B'	status byte in BASIC
IX	points to a Gosub section on the stack
IY	points to a For-Next section on the stack

The status byte (B') is used as a flag register. The following table shows each bit with its function:

B' bit #	function
7	trace (1=on)
6	mode (1=program)
5	For-Next loop started (1=yes)
4	nobreak (1=on)
3	clear variables (1=yes)
2	clear subroutines or loops (1=yes)
1	Onerr executing (1=yes)
0	Onerr (1=on)

Tape commands

You may have noticed that the primary word table does not include any tape commands. Instead, tape commands are stored in their own section at the end of BASIC. I don't know why this is, because it would be easier to include them as primary words. BASIC executes them by doing the following: when you type in the line, it goes to the Parser; this is normal. But when the Parser doesn't find the tape word in the primary word table, assumes it is a variable and doesn't find an equal sign, it calls a routine at $4DAC to see if it is a tape word. If not, it returns to the Parser to print an error and then back to the Central loop. But if the tape command exists, it calls the command's execution routine (found in the vector table at $4F4F). The command is responsible for parsing the rest of the line and executing it. It then returns to the Central loop.

Tape commands from a program are specified by using ctrl-d. When you print

a control-d, the routine at $4C0F sees it, and starts putting anything else printed into the ctrl-d buffer at $4279. When a "return" character is received, it looks up the word as before, and calls it, only this time the command returns to the Print routine instead of the Central loop.

Important tables

Scattered throughout BASIC are tables that would interest the programmer. I have mentioned many of them already, because they are an essential part of BASIC and its functions. They are listed again below to refresh your memory, and provide a lookup chart for the tables that you may use in your programs. The tape and execution error tables store the ASCII of the errors that are printed on the screen when something goes wrong. Parse errors are located in the Parse section, but are not organized into one table. The data table stores many pointers, vectors and buffers, and is fully described in chapter 7.

table	address
Primary word table (commands)	$0110
Tape word table	$4EAA
Variable command table (e.g. , COS)	pointed to by $3EE1
Secondary word table (e.g. , THEN, ;)	$0332
Variable table	pointed to by $3EDF
String space	pointed to by $3EF3
Value table	pointed to by $3EED
Parse vector table	$03AA
Line number table	pointed to by $3ED9
Crunch code table	pointed to by $3EE5
Tape errors	$5E3E
Execution errors	$0480
Data table	$3ED9

The following chapters give a detailed description of the routines in SmartBASIC. I hope the above outline will be sufficient to introduce you to this detail. Questions that arise should be answered by disassembling the routine and following the actual assembly language of the routine, which is how I figured out everything in this book. The operating system routines listed in volume 1 are also important in understanding the disassembled output, since in/out and tape routines use the OS in many cases.

Chapter 2: Zero Page

Zero page is devoted to interrupts. There are three kinds of interrupts for the Z80 microprocessor: a regular interrupt which can be ignored or masked (INT), a nonmaskable interrupt (NMI), and a bus request (BUSRQ). All of these interrupts consist of signals sent from some external device to the appropriate pin of the Z80 CPU. When the Z80 receives a NMI signal, it pushes the program counter to the stack and jumps to $66 in RAM. An INT is more complicated and can either cause the Z80 to put out an in/out request and jump to $38, or perform an indirect jump to an address formed from the I register and a number provided by the external device. The bus request pin causes the Z80 to stop operations and not use the bus until the signal is removed. The pin is used for direct memory access (DMA) such as that by the master 6801 on the Adam.

On the Adam a non-maskable interrupt is sent 60 times per second from the video chip to the Z80. It is serviced by the routine at $66 (102), which switches pointers to two tables in VRAM to create the blinking letters of the FLASH command. This routine is called every 16.7 ms and can cause problems with bankswitching or exact timing requirements. The interrupts from the VDP can be prevented by resetting bit 6 of register 1 of the VDP. The most interesting thing to do with this FLASH routine ($66-$AB) is to change the counter that controls the flash rate. The number at address 159 does this. It is normally 12, and poking it to 3 will cause Flash to definitely get your attention.

Eight memory locations at 0,8,16, etc. are $C9, return from interrupt, which presumably are in case of an INT interrupt, although I am not aware of any. All of zero page except 102-171 is available space for your routines.

Chapter 3: Keywords

A series of tables are stored from $100 to $5CB, including the Primary word table, Secondary word table, Error and Parse tables. The most important of these is the Primary word table at $110, the storage area for the ASCII of the non-tape command keywords (e.g., GOTO, RUN, PRINT). It is used during parsing, when each primary word is compared to the words in the Input buffer. Changing the ASCII of a command in this table allows you to customize commands or keep out nosy people (e.g., changing the ASCII of LIST to HAHA). Printing out this area with Printmem (vol. 1) helps with this task. Its token or crunch code is together with the ASCII. This number is put in the Crunch code buffer by the Parser to save space and speed up execution. The token is used also as an offset into the Command vector table at $1917, which contains the vectors to the execution routine of every command. The Parse vector table at $3AA points to the routines to do this for each command, by having the command in the Primary word table point to the desired group of vectors. The relationship between the Primary word table, the Parse vector table and the Command vector table is shown in the following diagram of a sample command (FOR).

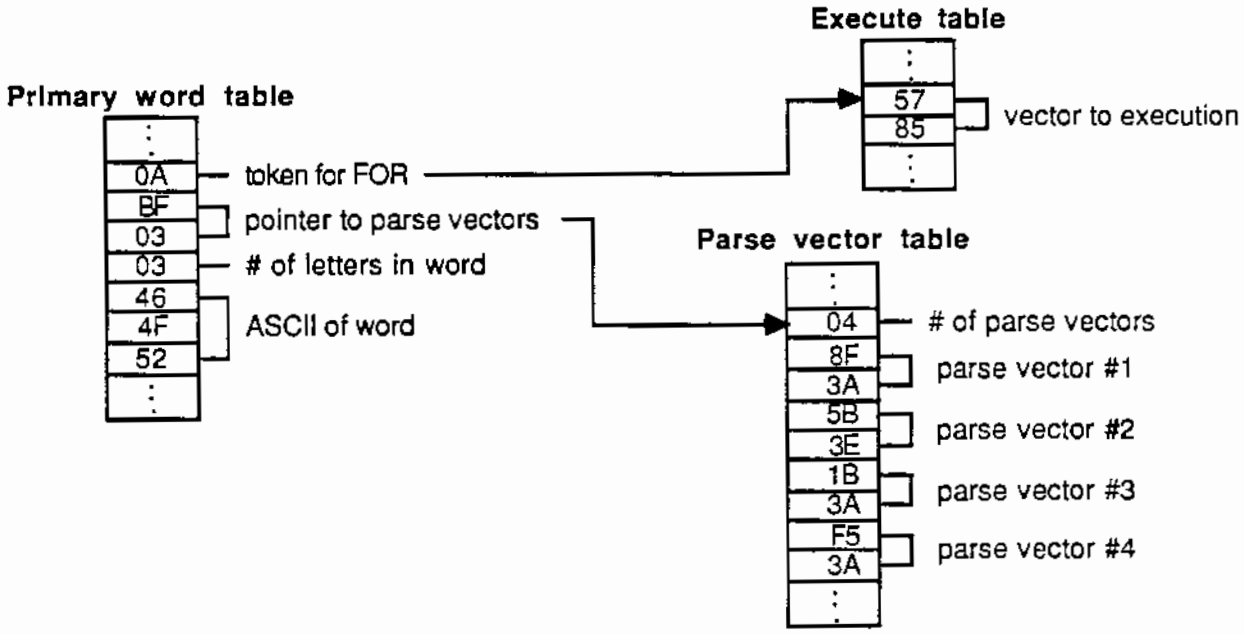


Diagram of a sample primary word

Another table, similar to the Primary word table, is the Secondary word table at $332. It contains the ASCII of most of the symbols that could follow a primary word (e. g., =, THEN, :). They, too, are translated by the Parser into their tokens and put in the Crunch code buffer with the rest of the line.

Ending this section is the ASCII table of execution errors. Changing this table allows your errors to look like anything you want, though, like the Primary word table, the new error should be as long as the one it is replacing. Note that these errors don't include the Parser or Tape errors.

$0100-0102 (256-258) Start Vector
The tape loads JP $3E9D, the cold start address, at $100 but the cold start routine at $4061 changes the address at $101-102 to $3EA3 which is the Central loop.

$0106-010F (262-271) Numbers
The numbers 10000, 1000, 100, 10, 1, are in integer (two byte) format.

$0110-0331 (272-817) Primary word table
This table of BASIC words is in the format: token, pointer to parse vectors, number of letters, word. The token points to the execution address in a table at $1917. The following table gives this information in more convenient form.

Token($)	Word	Execute($)	Execute(dec)	Parse($)
01		1867	6247	3AAC
02	GOSUB	20EB	8427	3D8C
03	GOTO	2096	8342	3D8C
04	INPUT	22FD	8957	3CB7
05	LET	1867	6247	3AAC
06	NEXT	226B	8811	3CCF
07	PRINT	1EAE	7854	3CDC
08	READ	251B	9499	3CD6
09	REM	20E3	11747	3DC9
0A	FOR	216D	8557	3A8F 3E5B 3A1B 3AF5
0B	IF	1E19	7705	3A63 3ABB
0C	DATA	20E3	8419	3DC6
0D	DIM	1B1E	6942	3CD6
0E	ON	20BD	8381	3A1B 3B69
0F	ONERR	1FB2	8114	3E77 3D8C
10	STOP	18EA	6378	
11	RETURN	211D	8477	
12	END	179F	6047	
13	DEF	2034	8244	3B15

14	CLEAR	1FCD	8141		
15	RESUME	2079	8313		
16	NEW	18D4	6356		
17	POP	212D	8493		
18	RUN	180F	6159	3B80	
19	LIST	1CEF	7407	3B8B	
1A	TRACE	18C0	6336		
1B	NOTRACE	18C5	6341		
1C	DEL	1D83	7555	3B8F	
1D	CALL	273A	10042	3A1B	
1E	CONT	18F3	6387		
1F	CLRERR	1FAD	8109		
20	GET	24A2	9378	3C04	
21	POKE	2778	10104	3A1B	3E43
				3A1B	
22	RESTORE	250A	9482		
23	HOME	2B52	11090		
24	DRAW	2C5E	11358	3A1B	3A80
25	XDRAW	2C94	11412	3A1B	3A80
26	FLASH	2B2A	11050		
27	INVERSE	2B2F	11055		
28	NORMAL	2B34	11060		
29	TEXT	2B39	11065		
2A	GR	2B3E	11070		
2B	HGR	2B43	11075		
2C	HGR2	2B48	11080		
2D	HLIN	2BA2	11170	3A1B	3E43
				3A1B	3E69
				3A1B	
2E	VLIN	2BD3	11219	3A1B	3E43
				3A1B	3E69
				3A1B	
2F	HPLOT	2CDF	11487	3AFE	
30	PLOT	2B83	11139	3A1B	3E43
				3A1B	
31	HTAB	2C38	11320	3A1B	
32	VTAB	2C42	11330	3A1B	
33	SHLOAD	2B4D	11085		
34	RECALL	2DF4	11764	3C04	
35	STORE	2DEC	11756	3C04	
36	WAIT	278E	10126	3A1B	3E43
				3A1B	3A79
37	SPEED	2A50	10832	3E36	3A1B
38	ROT	2CC3	11459	3E36	3A1B

39	SCALE	2CD1	11473	3E36 3A1B
3A	COLOR	2B5B	11099	3E36 3A1B
3B	HCOLOR	2B6F	11119	3E36 3A1B
3C	IN	2F34	12084	3E4E 3A1B
3D	PR	2F1A	12058	3E4E 3A1B
3E	HIMEM	2B02	11010	3E27 3A1B
3F	LOMEM	2A76	10870	3E27 3A1B
40	BREAK	18CA	6346	
41	NOBREAK	18CF	6351	
07	?	1EAE	7854	3CDC
42	&	27B4	10164	3DC9

$0332-03A9 (818-937) Secondary word table
These words occur following other words and are always found by a Parse routine. The table is simpler than the previous one, and is arranged: token, number of letters, word.

Token	Word	Token	Word	Token	Word
A0	+	AA	=	B4	TO
A1	-	AB	AND	B5	:
A2	*	AC	OR	B6	#
A3	/	AD	NOT	B7	(
A4	^	AE	GOTO	B8	)
A5	<	AF	GOSUB	B9	,
A6	>	B0	STEP	BA	;
A7	<=	B1	AT	BB	=<
A8	>=	B2	THEN	BC	=>
A9	<>	B3	THEN	BD	><

$03AA-041F (938-1055) Parse vector table
These vectors are listed in the table above, along with the commands that point to them. The format is: number of vectors, vectors. Each vector points to a Parse routine that parses a portion of the command's syntax.

$0420-0478 (1056-1144) Copywrite
This space contains the message printed when you boot BASIC. Since it is not used after the boot, you can use it to store your own data.

$0479-047F (1145-1151)], :, CR
Symbols used in print statements. To change the BASIC prompt from a right

bracket ( ]) to something else, Poke 1145, X , where X is the ASCII of the new symbol.

$0480-05B7 (1152-1463) Error Message Table
Command errors that can occur during RUN or execution are gathered here in the format: number of letters, message. Errors that occur during parsing are scattered throughout the Parse routines, and tape errors are in the tape section.

$05B8-05CB (1464-1483) Offset Table for Error Messages
This list of one byte numbers is used to find messages in the previous table. For example, the third message is $2A into the table.

Chapter 4: Math Routines

The math routines are interesting to disassemble just to see how they work. The information given here should be enough to make the disassembler output interpretable. The calculations are made on numbers in the floating point accumulators (FPA1 and FPA2), which are in floating point format. This format is described in vol. 1, and consists of 4 mantissa bytes and one exponent byte. To experiment with floating point numbers use the program in Appendix 1, which prints out floating point numbers from decimal input (e.g., 5= 00 00 00 20 83). The program enters a number and then PEEKs the variable value table to print out the floating point representation.

Each function (e.g., SIN, LOG, etc.) is calculated as a power series (e.g., a+bx+cx^2+dx^3...). Three general power series routines are at $103C to $1104, which calculate series of odd powers, even powers or all powers, respectively. The number of terms and the constants used in the calculations are specified by the HL register, which points to a table of the number of terms, followed by the constants in floating point format. Such tables are at $11CC to $1257. The constants are not exactly as predicted from the classical infinite series coefficients given in the comments, for reasons which I assume come from the fact that a rather small number of terms are actually used. This is a fairly esoteric subject which I have not found described in engineering or computer books, and may be passed from one generation of programmers to the next by reverse engineering.

$05CD-05DA (1485-1498) Numbers (1-F)
This unused area is filled with the numbers 1 to $0F.

$05DC-05EE (1500-1518) Load HL with number from crunch code.
Calls $1733 to evaluate equation and puts number only (not string) in HL. It prints an error if the number is greater than 255 ($FF) or if it is a string.

$05EF-0610 (1519-1552) Load BC with number from crunch code.
Used by GOTO, etc., to get line numbers from code line. It is like the routine above, only it doesn't call $1733 to get the number. It only accepts numbers in the format $80 to $8B, which is from 0 to $FFFF (see chapter 1).

$0611-0620 (1553-1568) Print FPA1 in decimal.
Calls $0CBC to change FPA1 into an ASCII string, and then calls $2F4E to print it from the buffer at $3F76.

$0621- (1569) Add FPA1 with (HL).
HL must point to a floating point number. The result is in FPA1. It loads FPA2 with HL's number, and falls through to $062F to add them.

$0627- (1575) Subtract FPA2 from FPA1. Result in FPA1.
XORs the top byte of FPA2 (3F2E) with $80 and falls through to the add routine.

$062F-0727 (1583-1831) Add FPA1 and FPA2. Result in FPA1.

$0728-073A (1832-1850) Load **FPA2 to FPA1.** HL lost.

$073B-0752 (1851-1874) Find sign.
Used by multiply and divide routines to prepare the FPAs for calculations. The sign of the result is loaded to 3F17, and top bits of FPAs are set.

$0753-075C (1875-1884) Multiply FPA1*2.
It increments the exponent of FPA1 to multiply it by 2, checking for an overflow error.

$075D- (1885) Multiply (HL)*FPA1
HL must point to a floating point number. The result is in FPA1. Calls $1117 to load the number to FPA2 and falls through to the next routine.

$0760-07E1 (1888-2017) Multiply FPA1*FPA2
A shift and add algorithm is used, but it shifts the running sum right instead of shifting the number left. The result is left in FPA1.

$07E2-08E3 (2018-2275) Divide FPA1 by FPA2
Similar to the multiply routine, only it shifts and subtracts. The result is left in FPA1.

$08E4-08EC (2276-2284) ABS
Resets top bit of FPA1 to 0, making the accumulator positive. HL is lost.

$08ED-0915 (2285-2325) SGN
FPA1 =0 if it (i.e., FPA1) is zero, 1 if it is positive, and -1 if it is negative. A and HL are lost.

$0916-0931 (2326-2353) Toggle FPA1 or FPA2.
FPA2 is toggled if the carry flag is set. Otherwise, FPA1 is toggled. To toggle means to set to zero, if it is not zero, and to set to one, if it is zero.

$0932-0966 (2354-2406) Load FPA1 to HL in integer format.

$0967-09B7 (2407-2487) Load HL to FPA1 or FPA2.
Integer to floating point conversion. FPA2 is used, if the carry flag is set.

$09B8-0A0F (2488-2575) Compare FPA1 with FPA2.
Carry flag is set, if FPA1>FPA2.

$0A10-0B3A (2576-2874) Convert number from ASCII to FPA1.
Number in buffer (DE) is converted from scientific format ASCII (e.g., 3.5E+7) to a floating point number in FPA1. All registers are saved.

$0B3B-0CBB (2875-3259) Table of powers of ten in FP format.
All powers of ten from 1E-38 to 1E+38 are stored in floating point format.

$0CBC-0DFB (3260-3579) Convert FPA1 to decimal ASCII.
The resulting string is at $3F77 and the length of the string is at $3F76.

$0DFC-0E13 (3580-3603) Scale FPA1
If FPA1>10, multiply it by 0.1.

$0E14-0E5D (3604-3677) LOG
Calculates the natural log (ln) of FPA1 and puts the result in FPA1. The routine is based on the equation ln x = 2 [(x-1/x+1) +1/3 (x-1/x+1)^3 + 1/5 (x-1/x+1)^5 ...] when x<1. The number is scaled and the ln calculated by the power series calculator #1 at $103C, using the four constants at $1243.

$0E5E-0E6F (3678-3695) SQR
Calculates the square root of FPA1, with the result in FPA1. Calculated from: SQR (x) = e^1/2 ln x.

$0E70- (3696) Raise to power (^).
Uses the equation: x^y = e^y ln x. x = FPA1 and y = FPA2. The answer is in FPA1.

$0EE8-0F47 (3816-3911) EXP
Exp (x) = e^x. x and answer are in FPA1. Calculated, after scaling, from e^-x= 1 - x + (x^2)/2! - (x^3)/3! . . .

$0F48- (3912) TAN
Angle and answer in FPA1. Calculated from Tan = Sin/Cos. Calls routines below.

$0F6A- (3946) COS
Calculated from Cos x = Sin (x + π/2).

$0F72-103B (3954-4155) SIN
This one does all the work. The equation used is Sin x = x - (x^3)/3! + (x^5)/5! - (x^7)/7! ... Uses power series calculator #2 at $10AE, and the five constants at $1229.

$103C- (4156) Power Series Calculator #1
FPA1 = ((x^2*c0 + c1)*x^2 + c2)*x^2 ... cn)*x. HL is the address of number of terms followed by constants. Input x is in FPA1.

$1054- (4180) ATN
The equation used is ATN(x) = -(x^3)/3 + (x^5)/5 - (x^7)/7 + ...where 0<x<1. Uses power series calculator #1 and six constants at $11CC.

$10AE- (4270) Power Series Calculator #2
FPA1 = ((x^2*c0 + c1)*x^2 + c2)*x^2 + ... cn. HL is the address of number of terms followed by constants. Input x is in FPA1.

$10B6- (4278) Power Series Calculator #3
FPA1 = ((x*c0 + c1)*x + c2)*x + ...cn. HL is the address of number of terms followed by constants. Input x is in FPA1.

$1104-1116 (4356-4374) Load FPA1 to FPA2.
Similar to the routine at $0728, only it moves FPA1 to FPA2.

$1117-111F (4375-4383) Load (HL) to FPA2.
HL must point to a floating point number.

$1120-112B (4384-4395) Load FPA1 with 1.
This and the next four routines destroy DE and HL. It moves the data from $0BF9 to FPA1.

$112C-113A (4396-4410) Push FPA1 to Stack.
On exiting, the stack contains: mantissa byte 1, exponent, mantissa 3, mantissa 2, 0, mantissa 4.

$113B-1149 (4411-4425) Pop FPA1 from Stack.

$114A-1158 (4426-4440) Push FPA2 to Stack.

$1159-1167 (4441-4455) Pop FPA2 from Stack.

$1168-117E (4456-4478) -127 < FPA1 < 127 ?
The Carry flag is reset, if FPA1 is between -127 and 127. Otherwise it is set.

$117F-1191 (4479-4497) Temp ABS
Makes FPA1 positive, if it isn't, and sets the return address to $090E, which sets the sign bit to its original value.

Some Floating Point Constants:

$1192-1196 (4498-4502) 0.693147180 = ln 2

$1197-119B (4503-4507) 1.44269504 = 1/ln 2

$119C-11A0 (4508-4512) 1.57079632 = $\pi/2$

$11A1-11A5 (4513-4517) 0.636619772 = $2/\pi$

$11A6-11AA (4518-4522) 0.785398164 = $\pi/4$

$11AB-11AF (4523-4527) 0.414213562 = SQR (2) - 1

$11B0-11B4 (4528-4532) 2.41421356 = SQR (2) + 1

$11B5-11B9 (4533-4537) -0.5 = - 1/2

$11BA-11BE (4538-4542) -1.41421356 = - SQR (2)

$11BF-11C3 (4543-4547) 0.707106781 = 1/SQR (2)

$11C4-11CB (4548-4555) Four Integer constants.

Groups of power series coefficients.
These coefficients are slightly different from the equations given because of finite series approximations.

$11CC-11EA (4556-4586) ATN coefficients.

-0.060346883	= -1/11 (approximately)
0.105734403	= 1/9
-0.142400777	= -1/7
0.199982167	= 1/5
-0.333333076	= -1/3
0.999999999	= 1

$11EB-1213 (4587-4627) EXP Coefficients.

2.06667101E-5	= 1/8! (approximately)
1.46290047E-4	= 1/7!
1.3386897E-3	= 1/6!
9.61627016E-3	= 1/5!
0.0555044141	= 1/4!
0.240226488	= 1/3!
0.693147181	= 1/2!
1	

$1214-1228 (4628-4648) SIN (routine #1) coefficients.

-4.63313261E-3
0.0796879998
-0.645963956
1.57079632

$1229-1242 (4649-4674) SIN (routine #2) coefficients.

8.95410543E-4
-0.0208554615
0.253668615
-1.23370049
1

$1243-1257 (4675-4695) LOG coefficients.

0.434255942
0.576584541
0.961800759
2.88539007

$1258-12B4 (4696-4788) RND

Puts a random number into FPA1. Seed is at $3F40 (16192), two bytes.

$12B5-12E2 (4789-4834) Push FPA1 to Stack with String Check.

If FPA1 points to a string, indicated by $3F21 = 1, the stack pointer is placed at memory pointed to by $3F22. (In equation evaluation, FPA1 can be used as a string pointer.)

$12E3-1304 (4835-4868) Pop FPA1 from Stack with String Check.

If FPA1 points to a string ($3F21 = 1) , the number $3F22 is loaded to the location pointed to by $3F22.

$1305-1322 (4869-4898) Load FPA1 to FPA2 with String Check.
If FPA1 points to a string ($3F21 = 1), the number $3F2B is loaded to the location pointed to by $3F2B.

$1323-1340 (4899-4928) Load FPA2 to FPA1 with String Check.
If string, "$3F22" is loaded to ($3F22), as above.

Equation Evaluation Routines $1341-1756.
The following section is used when a command wants to compute an equation in crunch code. The main routine is at $1733, which calls all the other routines. Each routine performs a function (e.g., get an integer from 0 to 9. or call the execution routine of a variable command).

$1341- (4929) Get Number from Crunch Code. part 2
A is the number type. This routine calculates the vector of the routine to move the number from crunch code to FPA1 or 2, depending on entry point for part 1 at $16CD or $16E5.

$1363-1375 (4963-4981) Offset Table for number type.
Used by routine above.

$1376 (4982) Get 0-9 integer.
Loads the number from crunch code into either FPA1 or FPA2, depending on whether the Carry bit is set or not.

$1383 (4995) Get $0A-FF integer.
Like the routine above.

$1390 (5008) Get $100-FFFF integer.
Like the routine above.

$139F (5023) Load variable to FPA1.
Calls $199F to see what kind of variable it is, and jumps to the routine according to the table below.

$13A7-13B0 (5031-5040) Table of variable routine vectors.

Variable type	Routine address
FP	$13C3
%	$13D6
$	$13E7
FN	$140F
Command	$140F

$13B1-13B6 (5041-5046) Load variable to FPA2.
Similar to the routine at $139F.

$13B7-13C2 (5047-5058) Table of get routine addresses.
The same as table at $13A7, except FN and Command = $1415.

$13C3-13D5 (5059-5077) Load FP variable from (BC) to (HL).
Moves the variable in floating point format from address pointed to by the BC registers to that pointed to by the HL registers.

$13D6-13E6 (5078-5094) Load Int variable from (BC) to (HL).

$13E7-140E (5095-5134) Load String from (BC) to (HL).

$140F-1414 (5135-5140) Execute Variable Command.
This and the next routine act on commands that are in the string variable table: SPC, TAB,...VPOS. The result is in FPA1.

$1415-142E (5141-5166) Execute Variable Command.
The result is in FPA2.

$142F-14C9 (5167-5321) Variable Command interpreter
Does work for previous two routines. It calls the vector of the variable command found in the variable table and checks for errors.

$14CA-14FD (5322-5373) Move string from crunch code to (FPA1).
Copies a string from current position in crunch code to String space pointed to by FPA1. Entry at $14D0 uses FPA2 instead.

$14FE-151E (5374-5406) Load FP number from (DE) to FPA1.
DE points to the crunch code. Entry point at $1504 for FPA2.

$151F-1528 (5407-5416) Move string from (HL) to (DE).
The first byte is length of string.

$1529-155B (5417-5467) Add (+)
This and the next four routines call the appropriate routine, in this case $062F, POP DE and BC, XOR A, and RET. If the variables are strings, this routine concatenates them.

$155C (5468) Subtract (calls $0627).

$1563 (5475) Multiply (calls $0760).

$156A (5482) Divide (calls $07E2).

$1571 (5489) EXP (calls $0E70).

$1578-1603 (5496-5635) <, >, AND, OR, =, <>, <=, >=.
Executes command on FPA1 and 2, or strings pointed to by them. If true, FPA1 is set to 1. If false, FPA1 is reset to zero, i.e., $3F26 = 0.

$1604-1615 (5636-5653) Compare strings (FPA1) and (FPA2).
Z flag set, if equal.

$1616-1648 (5654-5704) Executes (, -, NOT, for FPA1.

$1649-167F (5705-5759) Executes (, -, NOT, for FPA2.

$1680-16A4 (5760-5796) Check for math symbol in crunch code.
Uses next table to get order of op and address of routine. It points BC to the current op in the table. If the symbol is not a math symbol, BC points to $16CC (end of line).

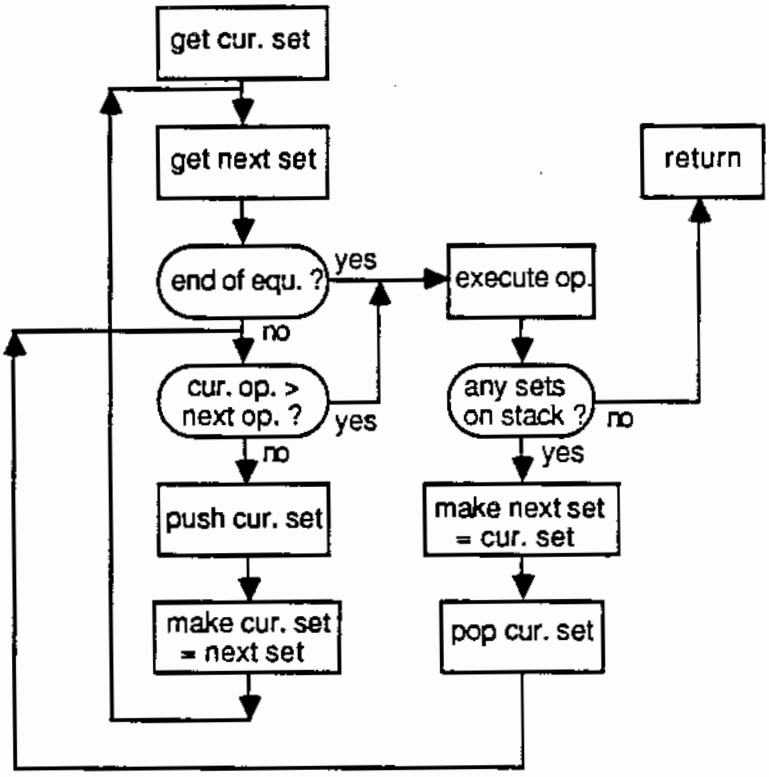
$16A5-16CC (5797-5836) Table of math symbol routines.
Format: order of operation, address low,high.

Symbol	order	address
^	0	$1571
*	4	$1563
/	4	$156A
+	5	$1529
-	5	$155C
<	6	$1597
>	6	$15A5
=<	6	$15B5
=>	6	$15C5
<>	6	$15D3
=	6	$15F1
AND	7	$1578
OR	8	$1589
end of equation	FF	

$16CD-16E4 (5837-5861) Load FPA1 from crunch code, part 1.
Finds number type and calls routine at $1341.

$16E5-16FC (5862-5884) Load FPA2 from crunch code, part 1.
Finds number type from crunch code and calls routine at $1341.

$16FD-1732 (5885-5938) Equation Evaluation
During RUN or an immediate command's execution. Coming in FPA1 = first number and BC points to math operation. This routine gets the next number in FPA2 and the next operation (op), which together are called a set. If the second op has a higher priority, it pushes the first set to the stack, moves FPA2 to 1, puts second op in BC, and calls itself. This continues until the end of the equation. In this way the math operations are done in order of decreasing priority. The following flowchart shows how equations are evaluated by BASIC during execution. See chapter 1 for more details.



Flowchart of Equation Evaluation

$1733-1756 (5939-5974) Get Equation from crunch code.
Gets one number and operation from crunch code and calls $16FD.

Chapter 5: BASIC Commands

This section of BASIC is where most of the action occurs. It can be considered the "brain" of BASIC, because it does important tasks like setting variables to a value or string, keeping IF... THEN statements straight, interpreting variable commands or gathering information from crunch code for graphics commands. It is needed after the Parser has translated the input and the line requires execution. Each command has a routine that does only what that command is supposed to do. These routines are vectored through a table at $1917, which stores them in the order of each command's token. Changing the vector of a command allows you to change what the command does, and is therefore helpful in adding new commands.

By using the token of a command as an offset into the Command vector table at $1917, the Execution loop at $182E looks up the address for the next immediate or program command in the crunch code, and calls it to execute the command. This loop repeats endlessly if BASIC is in the program mode (RUN), reading and executing tokens from the program's Crunch code table. If BASIC is in the immediate mode, then the loop only looks up and executes the one command in the Crunch code buffer. If this command is RUN, then it enters the program mode, and starts executing the program in the Crunch code table.

After the Execution loop calls the command's routine, the command is responsible for getting needed data from the crunch code (e.g., the 10 in GOTO 10). Other routines like "Get one number" ($05DC) are called by the command to do this. It also needs to update the registers that point to or contain data needed by other routines. They must contain the same information exiting as when they entered. Pushing and then popping them off the stack help in keeping them intact. These registers are: DE, pointing to the current address in crunch code; HL', pointing to the current address in the Line number table; C', the number of bytes left in the line of crunch code; B', the status byte of BASIC; IX and IY, the pointers to the GOSUB and FOR... NEXT data on the stack. Remembering to keep these registers intact and up to date during the command's execution is important, because the Execution loop and other commands need them.

$1757-1769 (5975-5993) Stack setup
IX, IY, and SP = $D380, the top of the stack which extends down to $D1FF. BC and the top to the stack are loaded with $1EDC (Print error). There is only one stack, but the index registers of the Z80 are used to keep track of locations on the stack where return information is kept for Gosub (IX) and For-Next (IY).

$176A-177F (5994-6015) Find first line number address.
Called at the beginning of RUN. It points DE to crunch code line. HL' points to the second line number. C' is the length of the crunch code line.

$1780-179E (6016-6046) Find next line number address.
In: HL' is the address of the current line number. On exit HL' points to the next line number, and DE points to the next crunch code line of length C'.

$179F-17B4 (6047-6068) END
Saves pointers for possible CONT, (DE at $3EFA and HL' at $3EFC). It then returns to the Central loop at $3EA6.

$17B5-17DF (6069-6111) TRACE routine.
It prints "#", line number, space if bit 7 of B' is set. Returns to the Central loop if BASIC is in the immediate mode. Otherwise it gets the address of the next line number.

$17E0-180C (6112-6156) Execute command
Checks mode, does restore, jumps to Execute loop ($182E).

$180F-182D (6159-6189) RUN
Clears stack, gets first line number or immediate number (RUN 30) address, and falls through to next routine.

$182E-1866 (6190-6246) Execute loop
Loops endlessly until control-**C** or S is pushed or program ends. Loads (DE) to A, gets command address from table at $1917 and calls it. Upon return it checks for Trace or Break, and loops again. The Trace routine called at $17B5 also checks for the immediate mode, and jumps to the Central loop, thus exiting the Execution loop.

$1867-18BF (6247-6335) LET
This routine is called even if LET is not written (e.g., a=7, or LET a=7). It checks variable type, calls get-equation ($1733), sets variable to what follows equal sign, and checks for errors.

$18C0-18C4 (6336-6340) TRACE
Sets bit 7 of B' (to one).

$18C5-18C9 (6341-6345) NOTRACE
Resets bit 7 of B' (to zero).

$18CA-18CE (6346-6350) BREAK
Resets bit 4 of B' (to zero).

$18CF-18D3 (6351-6355) NOBREAK
Sets bit 4 of B' (to one).

$18D4-18E9 (6356-6377) NEW
Resets stack, clears variables, pointers, jumps to the Central loop at $3EA3.

$18EA-18F2 (6378-6386) STOP
Prints "BREAK IN (line number)" and jumps to Central loop at $3EA3.

$18F3-1916 (6387-6422) CONT
Loads DE' and DE with (3EFA), HL' with (3FFC), C' with (DE), and jumps to Execute loop at $182E. If the Temp pointers (3EFA) and (3FFC) are zero, "Can't Continue" is printed and it jumps to the Central loop at $3EA3.

$1917-199A (6423-6554) Command vector table
Two byte vectors of each command are stored here in order of token values. These addresses are listed in the command list in chapter 3. Each vector points to the execution routine of the command. Changing commands and vectors lets you change the function of a command because you can create a new command, even though you lose an old one.

$199B-1B1D (6555-6941) Get variable type
The variable is in crunch code at (DE). A is loaded with the type number and BC points to the number in the Variable table. The Z80 then jumps to an address in a table which follows the call statement that called this routine.

Variable	Type #	JP address
FP	$00	1st
%	$10	2nd
$	$20	3rd
math command	$80	5th
Def FN	$C0	4th
Dim array	$08	

The variable type number is the number above plus 1 or 2 depending on whether the name has one (1)or more letters (2).

$1B1E-1C04 (6942-7172) DIM
Sets up definition of array in the Variable value table. For example, DIM A(12,44,7)

has three dimensions and the first dimension has twelve elements. The format of the definition is as follows: number of dimensions, number of elements in dimension 1 (two bytes, max $7FFF), number of elements in dimension 2, etc., followed by the actual numbers in the array. To begin these are zeros for numbers and $3F52 for strings, which is the address of a null string. A sample array is shown in the following diagram.

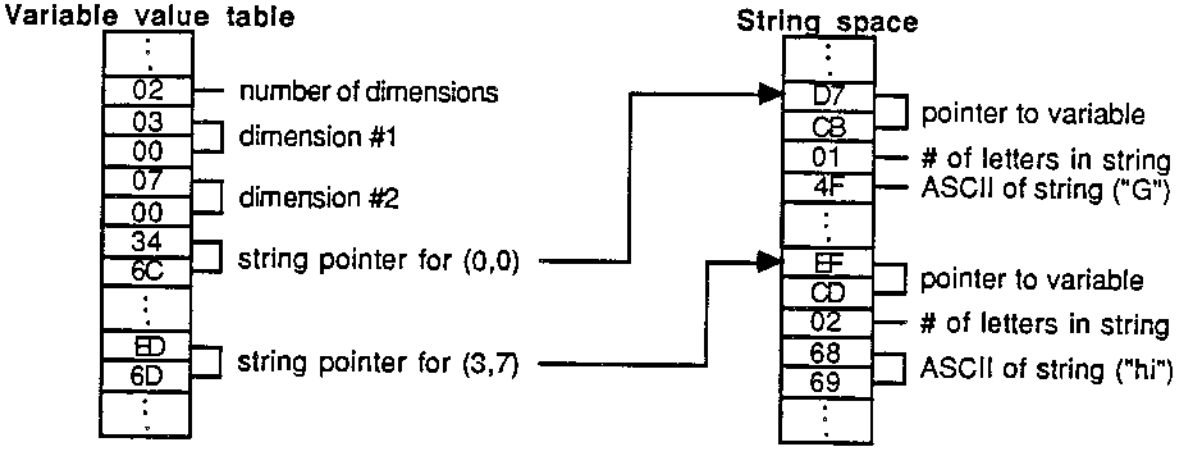


Diagram of a sample string array

$1B9A-1BB8 (7066-7096) Multiply HL x DE
Part of DIM. The result is in HL. Carry flag is set on overflow.

$1C05-1C5F (7173-7263) Check DATA length
Error if number of commas is greater than 256. On return C = number of commas +1.

$1C60-1C82 (7264-7298) Make string definition.
In A = string length, HL = address of variable name. Out (3EEF) = end of String space and HL = start of String space.

$1C83-1C8E (7299-7310) Check stack
If stack goes below $D1FF "stack overflow" is printed.

$1C8F-1CBC (7311-7356) Check String space
If table is too long (HL + ($3EEF) >$FFFF), it calls FRE ($27E1) to remove strings that are not pointed to by the Variable table (garbage collection). If the table is still too long, "Out of memory" is printed.

$1CBD-1CDA (7357-7386) Print program.
Used by LIST. Starts at current line number and prints to last line.

$1CDB-1D82 (7387-7554) LIST
Checks for line number, "-", or ",". Actual print routine is at $3493. Finds words from tokens in crunch code.

$1D83-1E18 (7555-7704) DEL
Checks text following DEL and calls $31EB to delete a line. If there is more than one line, it calls $31F2 . It jumps to Central loop ($3EA3) when done.

$1E19-1E3B (7705-7739) IF
Calls $1733 to evaluate the condition following the 'IF'. If it is true (A≠0), it returns if "THEN" is found to continue the Execute loop. If the THEN is not there, GOTO is assumed, and the routine jumps to $2096. If the condition is false (A=0), it calls $1780 to drop down to the next line number.

$1E3C-1EDB (7740-7899) PRINT
Calls $1733 to get the numbers or strings for printing. It loops until the line ends, checking for "," or ";".

$1EDC-1FAC (7900-8109) Print command errors
This is jumped to by any command when an error is detected. It prints the following: (return) "?" (error) "Error "(and "In line#" if in program mode). If bit 0 of B' is set (onerr), then it "GOTO"s to the line number at $3EFE. The following addresses print the corresponding errors:

address	ASCII string of error
$1EEB	Illegal Mode
$1EEE	Divide By Zero
$1EF1	Overflow
$1EF4	Redimensioned Array
$1EF7	Out Of Memory
$1EFA	Out Of Data
$1EFD	Formula Too Complex
$1F00	Illegal Quantity
$1F03	Type Mismatch
$1F06	Incorrect Function Usage
$1F09	String Too Long
$1F0C	Syntax
$1F0E	error code in A

$1FAD-1FB1 (8110-8113) CLRERR
Clears an onerr command by resetting bit 0 of B' to 0.

$1FB2-1FCC (8114-8140) ONERR
Sets bit 0 of B' and puts the line number to goto at $3EFE.

$1FCD-2033 (8141-8243) CLEAR
Resets pointers so that all variables are set to 0 or null strings.

$2034-2078 (8244-8312) DEF
Sets the variable to a function variable, and points it to the function in crunch code.

$2079-2095 (8313-8341) RESUME
Checks for mode or syntax, and restores old pointers ($3EFC)=HL' and ($3EFA)=DE to continue execution.

$2096-20BC (8342-8380) GOTO
Gets a line number from crunch code and calls $30F0 to make sure it exists. Line pointers are then set to that line number. GOTO is very useful in the immediate mode, because, unlike RUN, it does not reset variables. For possible changes to GOTO, see chapt. 11.

$20BD-20E2 (8381-8418) ON
If the offset number in the crunch code is 0, the line is skipped over. Otherwise DE is set to the correct line number entry and continues at GOTO or GOSUB.

$20E3-20EA (8419-8426) REM or DATA
DE is incremented so that it points to the next line in the crunch code.

$20EB-211C (8427-8476) GOSUB
Checks stack and saves the current position in the program by pushing IY, IX, HL', DE, and $2122 onto the stack. IX is adjusted so that it points to the current GOSUB entry on the stack. It then enters the program mode and continues execution at the given line number. The $2122 entry is for the RETURN routine.

$211D-212C (8477-8492) RETURN
Pops the old pointers saved by GOSUB off the stack in the order: DE, HL', IX, and IY. It uses the $2122 entry to allow the machine language return command to continue execution for the BASIC RETURN command. Changing the $2122 to another address allows the RETURN routine to be vectored to the routine you want. Execution continues at the new line number saved on the stack.

$212D-2143 (8493-8515) POP
Pops off the GOSUB pointers pointed to by IX. This is like RETURN, but DE is not changed, so execution continues with the next command after POP.

$2144-216C (8516-8556) ON GOSUB
This is a continuation of ON that executes a GOSUB instead of a GOTO.

$216D-2230 (8557-8752) FOR
Gets the necessary data from the crunch code and pushes it to the stack in the following order: IX, IY, address of variable, final loop number (in floating point), STEP number in floating point (default is 1), HL', DE, $2231. IY is then updated to point to the new FOR-NEXT entry on the stack. The entry $2231 is for the NEXT routine. Changing this address allows the NEXT routine to be vectored to another routine.

$2231-22FC (8753-8956) NEXT
Actual entry point is at $226B, but it starts at $2231. Updates the data on the stack pushed by FOR. If the loop is over, the data is popped off, and IY is set to the next FOR-NEXT loop on the stack. Leaving the variable off (e.g., NEXT instead of NEXT x) increases the speed of the loop.

$22FD-24A1 (8957-9377) INPUT
INPUT prints out any message or question, and then scans the keyboard until the return key is pressed. Multiple variables of string or numeric contents can be defined by using commas between them. "?Extra Ignored" or "?Reenter" is printed in case of errors.

$24A2-2509 (9378-9481) GET
Calls Input at $2F69 to get one character from the keyboard or other device. This character is then assigned to the desired string or numeric variable.

$250A-251A (9482-9498) RESTORE
Resets all the DATA pointers ($3EF7-9) to 0.

$251B-2702 (9499-9986) READ
Uses the DATA pointers at $3EF7-9 to get the numeric or string variable from crunch code. If more than one variable is present, it loops until all of them are read.

$2703-2739 (9987-10041) Get memory address
Calls $1733 to get an integer from crunch code. It is then placed in HL.

$273A-2758 (10042-10072) CALL
Calls $2703 to get the memory address from crunch code, checks stacks, saves

DE, DE', BC', HL', IY, and IX on stack, and then calls the address, popping the registers when done.

$2759-276A (10073-10090) USR
Similar to CALL, only it calls the routine at $3F02 instead of the address obtained from crunch code.

$276B-2777 (10091-10103) PEEK
Checks for numeric data type in FPA1, and then loads the contents of that address into FPA1.

$2778-278D (10104-10125) POKE
Gets an address from crunch code, checks to see if it is over the limit pointed to by $3F15, and loads it with the next number in crunch code if it is low enough. To poke anywhere in memory, simply POKE 16149 and 16150 with 255.

$278E-27B3 (10126-10163) WAIT
Loops endlessly until value1 AND (value2 XOR data from port) =0. Port number, value1 and value2 are found in the crunch code.

$27B4-27CF (10164-10191) &
Like USR, except it calls the routine at $3F04 instead of $3F02.

$27D0-2844 (10192-10308) FRE
Erases all strings that are not being used by a variable. It does this by stepping through String space, checking each string for its variable, and moving it to the new string space inside the old one if it is being used. Exits with the amount of free RAM (end of String space to start of numeric value table) in FPA1.

$2845-286E (10309-10350) VAL
Checks for correct variable type, moves the string to $3F77 for processing by $0A10, which gets the numeric value of the string.

$286F-2882 (10351-10370) ASC
Checks for string variable type, finds the desired string , and moves the ASCII value of it into FPA1.

$2883-28AA (10371-10410) CHR
Checks for numeric variable type, and creates a new string with a length of 1. FPA1 is then moved into the string.

$28AB-28D5 (10411-10453) STR
Checks for numeric variable type, and creates a new string with decimal equivalent of FPA1 as its content.

28D6-28DF (10454-10463) LEN
Checks for variable type, moves the length of the string (third byte in the definition) pointed to by FPA1 to FPA1.

28E0-290B (10464-10507) Check string length
Used by LEFT, RIGHT, and MID to compare the number following the command with the length of the wanted string. Carry set if number is larger.

$290C-2920 (10508-10528) LEFT
Checks for errors, loads C with the number from crunch code (right end), and loads A with 0 (left end). It then jumps to $2978 to cut up the string.

$2921-2938 (10529-10552) RIGHT
Checks for errors, loads C with the length of string -1 (right end), and A with the number from crunch code (left end). $2978 is jumped to for processing the string.

$2939-2977 (10553-10615) MID
Checks for errors, and sets up A as the first number (left end), and C as the first number + the second number-2 (right end). It falls through to $2978 to make the new string.

$2978-29AF (10616-10671) Cut string
Creates a new string with A being the left boundary, C being the right boundary, and its contents being the wanted portion of the old string.

$29B0-2A3D (10672-10813) INT
Cuts the decimal remainder off of FPA1, and leaves the result in FPA1 and A.

$2A3E-2A4F (10814-10831) ERRNUM
Uses the error number at $3F00 as the offset for the table at $05B7, and places the number found in $05B7 to FPA1.

$2A50-2A5B (10832-10843) SPEED
Gets the number from crunch code, and places it at $3F01.

$2A5C-2A68 (10844-10856) POS
Calls $6641 to get the horizontal position of the cursor, and puts it in FPA1.

$2A69-2A75 (10857-10869) VPOS
Calls $6648 to get the vertical position of the cursor, and puts it in FPA1.

$2A76-2B01 (10870-11009) LOMEM
Checks to see if the address obtained from crunch code is too big or less than

$6B0F. Moves the Variable table and Variable command name table to new location, but not the String space. So you should set new Lomems before defining strings to be sure the Lomem area is not already being used by strings.

$2B02-2B29 (11010-11049) HIMEM
Gets the address from crunch code, calls Clear ($1FD0), checks for errors, and puts the address at $3EED.

$2B2A-2B5A (11050-11098) Screen commands
Saves DE on the stack and then calls the actual screen command. SHLOAD, which means shape-load and was used by Apple to save shape tables on tape, is only a return. The following commands are rerouted by this routine:

old address	command	new address
$2B2A	FLASH	$6633
$2B2F	INVERSE	$661D
$2B34	NORMAL	$6627
$2B39	TEXT	$4815
$2B3E	GR	$483C
$2B43	HGR	$638C
$2B48	HGR2	$631A
$2B4D	SHLOAD	$18C9
$2B52	HOME	$4B68

$2B5B-2B6E (11099-11118) COLOR
Puts a number from crunch code into C and calls $492F to place the color value at $4188.

$2B6F-2B82 (11119-11138) HCOLOR
Like COLOR, except it calls $4928 to put the color at $4189.

$2B83-2BA1 (11139-11169) PLOT
Gets the x and y from crunch code, places them in C and B, respectively, and calls $4A9E to plot the point.

$2BA2-2BD2 (11170-11218) HLIN
Sets up C as the y, B as x1, E as x2, and calls $4975 to plot the horizontal line.

$2BD3-2C03 (11219-11267) VLIN
Sets up E as the x, B as the y1, C as the y2, and calls $49FC to plot the vertical line.

$2C04-2C37 (11268-11319) SCRN
Gets the x from FPA1, and y from crunch code, and puts them in C and B. It then calls $4AFB to get the color of the block and puts it in FPA1.

$2C38-2C41 (11320-11329) HTAB
Loads C with the number from crunch code, and calls $664F to move cursor.

$2C42-2C56 (11330-11350) VTAB
Calls $666B with the crunch code number in C to move the cursor down.

$2C57-2C8C (11351-11404) DRAW
Loads E with the shape number, B with the y, C with the x, and calls $67DC to draw the shape.

$2C8D-2CC2 (11405-11458) XDRAW
Like DRAW, but it calls $6904 instead of $67DC.

$2CC3-2CD0 (11459-11472) ROT
Loads C with a number from crunch code, and calls $66E8 to rotate the shape.

$2CD1-2CDE (11473-11486) SCALE
Sets up C with the number and calls $66DD to perform the scaling.

$2CDF-2D62 (11487-11618) HPLOT
Calls $6401 if a point is wanted, $6456 for a line, and $64C5 for a continuation of a line (e.g., HPLOT TO x,y).

$2D63-2D82 (11619-11649) PDL
Loads C with FPA1, and calls $6918 to scan the paddles. Exits with result in FPA1.

$2D83-2DFD (11650-11773) STORE, RECALL
The STORE entry point is at $2DEC, while RECALL is at $2DF4. This rather long and complex routine, which stops after the setup, appears to be an initial attempt to implement the similar Apple II commands. They were used by Apple for cassette storage, and are essentially archaic.

Chapter 6: Parser

The Parser is the portion of BASIC that translates your typed in line into a shorter and more efficient form, called crunch code. Along the way it checks the syntax or format of the line, making sure it's legible. The Parser doesn't execute the command, it only makes it more readable for the execution routines in Chapter 5. Of course, each command has a different syntax, so many different routines are needed to parse the line. The Parse routines are listed in a table at $3AA. Though different, some commands share similar syntax (e.g., PLOT x,y and HPLOT x,y). In order to save space, there are routines that parse a variable, an equation, or some other common syntactic structure. The parsed lines are placed in the Crunch code buffer at $4077, and in the Crunch code table, if there is a program line number present. See chapter 1 for more detail on the Parser. The diagram below of a sample line ("PRINT x") shows the line in the Input buffer and in the Crunch code buffer.

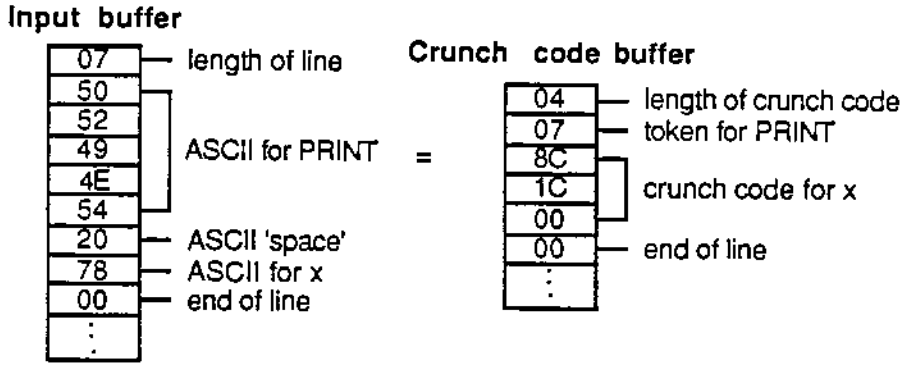


Diagram of a sample line

While creating your own commands, you usually can use old Parse routines, thus saving space and your time. But there could be moments when the format you want can't be done with the current Parse routines. In these situations, you must remember to keep the following registers intact as you write your own Parse routine: DE, pointing to the current address in the input line; B', the status byte; IX and IY, pointing to the stack. Keeping them intact means that when the routine is exited, they must contain the same information they had when the routine started. To do this, you can either not use the register in the routine, or you can push the register to the stack, use it, and then pop off the old contents. If you write your own Parse routine, you must create a Parse vector entry to point to the Parse routine.

The entry and the routine can be placed anywhere in RAM, but the primary command must point to the vector entry and the vector must point to the Parse routine.

$2E00-2E09 (11774-11785) Read buffer
Reads the next non-control ASCII byte from a buffer pointed to by DE into A. The Zero flag is set if the byte is zero, indicating the end of the buffer.

$2E0A-2E0E (11786-11790) Set word scan
Sets B up with the length of the table pointed to by HL, and C with the length of the table at DE. It then falls through to the next routine to scan the tables.

$2E0F-2E3D (11791-11837) Word scan
Compares the two tables pointed to by HL and DE with each other, with the length of comparison in B, or C, whichever is shorter. If the alternate Carry flag is not set upon entering, then the routine also checks for upper case ASCII. The Zero flag is set if the tables are equivalent.

$2E3E-2E4A (11838-11850) Save registers and Set word scan

$2E4B-2E57 (11851-11863) Save registers and Word scan

$2E58-2E6B (11864-11883) Letter check
If the ASCII byte pointed to by DE is from $41 to $5B, or $61 to $7B, then the Carry flag is set.

$2E6C-2E73 (11884-11891) Number check
If the ASCII byte pointed to by DE is from $30 to $3A, then the Carry flag is set.

$2E74-2E8F (11892-11919) Reset program pointers
Sets $3ED9, the pointer to the first line number address, to $D180. Sets $3EDB, the number of line numbers, and $3EDD, the length of the line number table, to 0. It also sets the random number seed ($3F3E) to $FB40 and $D291.

$2E90-2ED9 (11920-11993) Print parse errors
It first calls $4DAC to see if the error occured because of a tape command. If it did not, then a "^" is printed followed by the string placed after the call-routine. "Expected" is printed if $2E91 was called; nothing, if $2E90 was called. It then returns to the Central loop at $3EA3.

$2EDA-2EE9 (11994-12009) Print character with PR
Calls the current PR routine pointed to by $3F49 ($2EEA for pr#1, and $2F0B for pr#0). A contains the ASCII character to be printed.

$2EEA-2F0A (12010-12042) Print to printer
Prints the character in A on the printer by calling $FC66, and falls through to also print it on the screen.

$2F0B-2F19 (12043-12059) Print to screen
Pauses to execute the SPEED counter at $3F01, then jumps to $4C0F to print the character on the screen.

$2F1A-2F33 (12060-12083) PR
Uses the next number in crunch code as an offset into the PR table at $3F55. The new address found in the PR table is moved to $3F49 to vector the current PR routine.

$2F34-2F4D (12084-12109) IN
Like PR, except that the IN table is at $3F65, and the new address is vectored through $3F43.

$2F4E-2F5F (12110-12127) Print table
Prints the table pointed to by HL via PR routine. The table's first byte is its length, followed by the rest of the ASCII table.

$2F60-2F68 (12128-12136) Print a return

$2F69-2F75 (12137-12149) Input using IN
Calls the routine vectored through $3F45 to get input into A.

$2F76-2F7E (12150-12158) Print prompt
Prints the contents of $0479.

$2F7F-3050 (12159-12368) Input line
Reads input device by calling $2F69, and places the characters received into the Input buffer at $3F75 with $3F75 being the maximum length of the buffer, $3F76 being the length of the buffer, and $3F77 being the start of the characters. Checks for control characters and acts accordingly if one is encountered. It prints the characters on screen only, and loads DE with $3F76 when the line is over, indicated by the ASCII return character or the overflowing of the buffer limit.

$3051-3062 (12369-12386) Control character table
Contains the ASCII codes for control characters like return, arrow keys, ctrl-N, etc. The above routine checks input characters with these for action.

$3063-3083 (12387-12419) Control address table
This table is similar to the one above in that it is used when control codes are found

in input. Its format is in the same order, though reversed from the previous one, where the address of the first code used to be last (ie. ctrl-2 is the last entry in the above table; return is the first):

address	control ASCII code
$2FA4	crtl-2
$2FF9	ctrl-arrow
$2FFB	ctrl-L,home,down or up arrow
$3044	ctrl-X
$2FF5	ctrl-O
$2FF1	ctrl-N
$3007	left arrow, backspace
$3016	right arrow
$3024	ctrl-I, tab
$2FBD	return

$3084-3092 (12420-12434) Vectored screen print
Calls the routine vectored through $3F4B to print only on the screen.

$3093-30A2 (12435-12450) Print return on screen
Prints the table at $047E, a return, through the above routine. Calling $3098 prints any table with the length being the first byte of the table.

$30A3-30D3 (12451-12499) Check number size
Converts a number in ASCII form pointed to by DE into an integer in HL. "Number Too Big" is printed if the number >$FFFF.

$30D4-30EF (12500-12527) Get length of line
Loads $3F4E with the length of the crunch code line pointed to by DE.

$30F0-3149 (12528-12617) Look for line number
Scans the line number table for the line number stored at $3F4F. Because of the method used for scanning, it is faster to place a wanted line number to GOTO either in the middle or above, or the very last line of the program. This practice only slightly increases speed, but is useful when dealing with long programs or frequent loops.

$314A-31EA (12618-12778) Insert line number into table
Erases any line number if the line already exists. It also moves the tables down, and enters the line number data into the Line number table, and crunch code line into the Crunch code table.

$31EB-329E (12779-12958) Delete line number
HL=the line number to delete. It moves the Line number table and Crunch code

table so that the specified line number and crunch code is erased.

$329F-32ED (12959-13037) Print (HL) with PR
Call $32A7 to print DE as ASCII. Converts the number pointed to by HL to its ASCII equivalent, and prints it. This routine, and the following routines, all print their data through $2F4E and the PR routine.

$32EE-3312 (13038-13074) Print primary word
Prints the primary words corresponding to the token in crunch code pointed to by DE.

$3313-336B (13075-13163) Print number
Takes the crunch code format of a number, turns it into its ASCII form, and prints it.

$336C-33CF (13164-13263) Print variable name
Gets the variable number from crunch code, looks it up in the Variable table, and prints out the name (2 letters), with any more letters in crunch code, along with the variable type.

$33D0-33D8 (13264-13272) Print "FN"

$33D9-33E4 (13273-13284) Print data
Prints the data from crunch code indicated by $90, which is in the format of length of data, followed by the data itself.

$33E5-33FF (13285-13311) Print string
This routine is similar to the above routine, except that the string is placed in quotes and is indicated by a $91.

$3400-3430 (13312-13360) Print secondary word
Looks up the ASCII for the symbol from crunch code in the Secondary word table, and prints it.

$3431-3442 (13361-13378) Check type of secondary word
In: B=code of symbol. The Carry flag is set if the symbol is a word (e.g., AND, STEP, etc.).

$3443-3464 (13379-13412) Find primary word
Call $3443 for primary words, or $344D for secondary words. Scans the table until the crunch code in A is matched with one in the table. HL is then pointed to the word following the code.

$3465-347B (13413-13435) Print command
Prints a line of crunch code pointed to by DE, with the beginning of the crunch code

being the length of the line, and the end, a colon.

$347C-3492 (13436-13458) Print line
Prints a command by calling the above routine, and then continues printing commands to the end of the line.

$3493-34A4 (13459-13476) Print line number and line
In: HL points to the line number in the line number table. It calls $329F to print the line number, and then $347C to print the crunch code.

$34A5-34C8 (13477-13512) Print floating point number
Moves the floating point number pointed to by DE (in crunch code) to FPA1, and calls $0611 to print it in decimal.

$34C9-3517 (13513-13591) Move first string
If the first string in the String table is not being used, then it is erased, otherwise it is moved to the end of the table, erasing the original.

$3518-355A (13592-13658) Make first string
Makes room in front of the string table for a string of length A.

$355B-35C5 (13659-13765) Check type of character
Sets C, B or the Carry flag depending on whether (DE) is a letter, number, :, ?, or equality symbol. For example, if (DE) is a letter, then C=the length of the word, B=0, and the Carry flag is set.

$35C6-3608 (13766-13832) Compare word to tables
This routine scans the primary word tables to see if the word you typed in exists. Call $35C6 for the Primary word table, $35CE for the Secondary word table, or $3602 for both. The word is pointed to by DE, and the length is in C. If a match was found, then the Carry flag is set, the crunch code is placed in A, and HL is pointed to the word in the table.

$3609-3679 (13833-13945) Parse line
Resets the Crunch code buffer pointer ($3EE7) to $4077, and calls $367A to parse the command. If a colon is present, it continues parsing, ending when the line is over, and setting DE to the start of the crunch code.

$367A-3690 (13946-13968) Parse command and finish buffer
Calls $36A8 to parse the command, places the length of the Crunch code buffer at the start of it, and a zero at the end.

$3691-36A7 (13969-13991) Check for end of line
Looks at the Input buffer to see if any colons exist after the command. If so, the

Carry flag is set.

$36A8-3701 (13992-14081) Parse command
This routine is called to parse the line recieved from an input device, and to put the crunch code in a buffer pointed to by $3EE7, called the Crunch code buffer. It calls $35C6 to see if the first word is a primary word. If it is, it puts the crunch code of the word in the buffer and calls the keyword's parse routines, according to the table at $03AA. If no primary word was found, it assumes that the word is a variable, and is parsed by "LET".

$3702-3764 (14082-14180) Look for variable
In: A=variable type to look for, DE points to the name in the Input buffer. The routine searches the Variable table until a match is found, then it sets the Carry flag, and puts the variable's number in HL. The Carry flag is reset if no match was found.

$3765-3850 (14181-14416) Make variable
Moves the Variable tables to allow for the new variable. It creates an entry in the Variable table of type A and length C.

$3851-3892 (14417-14482) Hold new variables
This routine temporarily holds new variables in a string until the line is finished parsing. The string is pointed to by $3EE9, and contains the type of variable, its length, the position in Input buffer, and the pointer to its crunch code position. A maximum of $29 variables can be held.

$3893-38C1 (14483-14529) Make new variables
This routine enters any variables stored in the string pointed to by $3EE9 into the Variable tables by calling $3765 for each individual variable.

$38C2-38D4 (14530-14548) Fill Crunch code buffer
Loads A to the Crunch code buffer pointed to by $3EE7. The buffer is then incremented to the next byte, and is checked for being too long (>$FF)

$38D5-38DA (14549-14554) Add C to DE

$38DB-390C (14555-14604) Check for symbol
Looks for the wanted symbol in the Input buffer. If it is found, the crunch code is placed in the Crunch code buffer, and the Carry flag is set. Call the following entry points for the desired symbols:

entry point	symbol parsed
$38DB	,
$38DF	;
$38E3	+
$38E7	-
$38EB	NOT
$38EF	(
$38F3	)
$38F5	crunch code in A

$390D-3923 (14605-14627) Check for NOT, +, -
Looks at buffer for one of the above, and puts it in the Crunch code buffer.

$3924-393A (14628-14650) Check for + or -
Like the above routine, except it does not look for NOT.

$393B-3952 (14651-14674) Check and Parse data
Checks for any signs by calling $390D, and then calls the next routine to parse the data.

$3952-397B (14675-14715) Parse data
Parses any type of number or string. If B=1, then it jumps to $3D25. If B=0, then it jumps to $3C23. If a quote is found, then it jumps to $3DD2. If a "(" is found, then it parses the next type of data, as long as it ends with a ")".

$397C-3995 (14716-14741) Parse secondary words
This routine scans the Secondary word table to see if the next byte in the Input buffer is a secondary word. If it is, the token is put in the Crunch code buffer, and the Carry flag is set.

$3996-39A2 (14742-14754) Table of math priorities
The priorities of the secondary words from + to OR are stored here.

$39A3-39B8 (14755-14776) Print "Illegal Equation"
Prints the message, and returns to the Central loop.

$39B9-3A16 (14777-14870) Equation evaluation in parsing
Steps through the equation, and puts the crunch code into the buffer, following the equation as it does so.

$3A17-3A78 (14871-14968) Parse equation
Call $3A17 for numeric equations without error messages, $3A1B for errors. Call $3A32 for string equations without errors, and $3A36 to check for errors. The

Crunch code buffer is filled with the parsed equation.

$3A79-3A7F (14969-14975) Parse WAIT

$3A80-3A8E (14976-14990) Parse DRAW
Checks for "AT x,y".

$3A8F-3AAB (14991-15019) Parse FOR
Checks for a numeric variable and equation.

$3AAC-3ABA (15020-15034) Parse LET
Checks for a varaible, "=", and an equation.

$3ABB-3AF4 (15035-15092) Parse IF
Checks for "GOTO", "THEN", and commands following them.

$3AF5-3AFD (15093-15101) Parse FOR
Checks for "STEP".

$3AFE-3B14 (15102-15124) Parse HPLOT
Checks for "TO x,y".

$3B15-3B53 (15125-15187) Parse DEF
Looks for "FN", and continues at $3A8F to parse equation.

$3B54-3B7F (15188-15231) Parse ON
Checks for "GOTO" or "GOSUB", followed by a series of line numbers and commas.

$3B80-3B8A (15232-15242) Parse RUN
Looks for a word or line number following the RUN.

$3B8B-3BCA (15243-15306) Parse LIST, DEL
Checks for a line number, "," or "-" followed by another line number.

$3BCB-3BEA (15307-15338) Parse variable type
A is loaded with the variable type: $20 for strings, $10 for integers, 0 for floating point. If a "(" is found, then 8 is added to the type (e.g., an integer array has a variable type of $18).

$3BEB-3C9B (15339-15515) Parse variable
If the variable name is a command, then an error is given. $3BCB is called to get the variable type. If the variable is new, it is placed in the temporary variable string for later entry.

$3C9C-3CB6 (15516-15542) Parse dimensioned variables
Checks for a "(" and commas.

$3CB7-3CDB (15543-15579) Parse INPUT
Sees if a line number is present and then falls into parsing NEXT, REM and DIM.

$3CDC-3D12 (15580-15634) Parse PRINT

$3D13-3D8B (15635-15755) Parse number
Puts the number into one of the numeric formats for the crunch code.

$3D8C-3DC5 (15756-15813) Parse line number
Used by GOTO or GOSUB to format a line number.

$3DC6-3E1D (15814-15901) Parse DATA, REM, or quotes
REM and DATA are parsed as $90 type , and quotes are $91. This is where the Data-Bump-Bug originates. When you run the cursor over the DATA line, or one is LOADed in, this routine adds an extra space at the beginning of the data. To fix this, simply add the following line in your HELLO program: 10 POKE 15830,8: POKE 15831, 55: POKE 15832, 19: POKE 15824, 216.

$3E1E-3E26 (15902-15910) Parse =
Prints "Illegal Command" if an error occurs.

$3E27-3E35 (15911-15925) Parse :

$3E36-3E42 (15926-15938) Parse =
Prints " '=' expected" if an error occurs.

$3E43-3E4D (15939-15949) Parse ,

$3E4E-3E5A (15950-15962) Parse #

$3E5B-3E68 (15963-15976) Parse TO

$3E69-3E76 (15977-15990) Parse AT

$3E77-3E86 (15991-16006) Parse GOTO

$3E87-3E9C (16007-16028) Print errors
Prints "Line Number" or ":"+" Expected".

$3E9D-3EA2 (16029-16034) Boot routine
This is the routine jumped to when BASIC is first loaded from a tape or disk. It calls

a routine at $4061, which is later written over as an Input buffer as you use BASIC. This routine sets up pointers and looks for a HELLO program. It then falls through to the next routine, the Central loop.

$3EA3-3ED8 (16035-16088) Central loop
This is the immediate mode loop, which is the "heart" of BASIC. Some routines jump to $3EA3, which resets the stack, while other routines jump to $3EA6, which keeps the stack intact. The Central loop prints a return, a prompt, and calls the routine to read the keyboard. If the typed in line has a line number, then it parses the line, and enters it into the Crunch code table. Otherwise, the line is assumed to be an immediate command, and, after parsing, the command is called without moving the crunch code from the Crunch code buffer into the Crunch code table. For a complete description of the Central loop, see chapter 1.

Chapter 7: Data table

A mass of pointers, vectors and other varying data is stored here. Though it contains many types of data, it can generally be broken down into smaller sections as follows: program pointers ($3ED9), math and FPA data ($3F17), input and output vectors ($3F43), Input and Crunch code buffers ($3F75), graphics data ($417B), tape or file data ($4194), screen data ($4239), and the control-d pointers ($4276). This organization helps in both understanding the table and being able to find the pointers you want in it.

The majority of the space is taken up by two byte pointers and vectors. These are likely to be the most interesting or useful to you, the programmer. For instance, by POKing to them you can change the size of the screen or create your own output routines. By PEEKing them you can look for a variable or know the current color or speed. If you use your own pointers (e.g., in a new command), you can use spaces like the one at $3F09, though they must not be permanent, because other routines also temporarily use these areas. This is useful when you are pressed for space, but when free RAM is no problem, it is safer and easier to keep them elsewhere.

$3ED9 (16089) Pointer to start of Line number table

$3EDB (16091) Number of line numbers

$3EDD (16093) Length of Line number table

$3EDF (16095) Pointer to start of Variable table (LOMEM)

$3EE1 (16097) Pointer to end of Variable table

$3EE3 (16099) Pointer to end of Variable command name table

$3EE5 (16101) Pointer to start of Crunch code table

$3EE7 (16103) Pointer to end of Crunch code buffer

$3EE9 (16105) Pointer to the string of new variables
When the Parser parses a line, it puts all the variables that haven't already been used in a previous line into string pointed to by this location. After it parses the line, it goes back and enters each variable into the Variable table.

$3EEB (16107) Number of variables

$3EED (16109) Pointer to start of Variable value table
This pointer points to the beginning of the table that stores the values of all the numeric variables. It also is a temporary place for other values that are used during execution, so sometimes it is simply called the Value table.

$3EEF (16111) Pointer to end of String space

$3EF1 (16113) Temporary pointer to end of String space

$3EF3 (16115) Pointer to start of String space

$3EF5 (16117) Pointer to current DATA line number

$3EF7 (16119) Pointer to current DATA crunch code

$3EF9 (16121) Number of remaining bytes in DATA crunch code

$3EFA (16122) Storage of DE for CONT

$3EFC (16124) Storage of HL' for CONT

$3EFE (16126) Line number for ONERR

$3F00 (16128) Command error number
This is the offset that is used to print errors. It does not include parse or tape errors.

$3F01 (16129) Current SPEED

$3F02 (16130) Vector to USR routine

$3F04 (16132) Vector to & (ampersand) routine

$3F06 (16134) ASCII code for break (ctrl-c)

$3F07 (16135) ASCII code for pause (ctrl-s)

$3F08 (16136) Indicator of pause

$3F09 (16137) Temporary storage area

$3F14 (16148) ASCII code for indenting line numbers
This is used by LIST to indent the line number of the program. The default code is a space (32).

$3F15 (16149) Pointer to POKE limit

$3F17 (16151) Sign for the result of operations

$3F18 (16152) Temporary FPA data and pointers

$3F1E (16158) FPA1 data used in division

$3F21 (16161) FPA1 status byte
If the byte is 0, then the FPA1 is a floating point number, otherwise, it means the FPA1 is pointing to a string.

$3F22 (16162) FPA1 mantissa and exponent

$3F27 (16167) FPA2 data used in divsion

$3F2A (16170) FPA2 status byte
0= floating point number, ≠0 means FPA2 points to a string.

$3F2B (16171) FPA2 mantissa and exponent

$3F30 (16176) Maximum width of printer line

$3F31 (16177) Position of head on printer

$3F32 (16178) Temporary FPA for Sin, Cos, etc

$3F37 (16183) Temporary FPA for calculations

$3F3E (16190) Random seed number

$3F42 (16194) Sign of floating point numbers
This is like the pointer at $3F17, but this temporary storage area is more generic than the other one.

$3F43 (16195) IN vector used by READ
This pointer stores the old IN vector while READ is using the tape through this vector. When READ is done, then it changes the IN vector back to its original value.

$3F45 (16197) Vector to recieve data from device (IN)

$3F47 (16199) Storage of PR vector for writing to tape
This is used by the LOAD and WRITE command to "remember" the old vector while the command executes. It is similar in function to the pointer at $3F43.

$3F49 (16201) Vector to transmit data to device (PR)

$3F4B (16203) Vector to printing on screen
This vector points to the routine that will print the ASCII code in A to the screen only, without printing it on the printer, etc.

$3F4D (16205) Length of Crunch code buffer

$3F4F (16207) Line number to GOTO, GOSUB, etc.
Stores the last line number that was jumped to, because it is used by GOTO and GOSUB to store the line number while it checks the Line number table.

$3F51 (16209) Temporary ASCII code for line indenting

$3F52 (16210) Null string
This is pointed to by any variable that does not have a string assigned to it yet.

$3F55 (16213) PR vector table
The 8 addresses for each PR routine are stored in increasing order here (e.g., PR#0 is the first vector, PR#1 is the second vector and the others are the same as PR#0).

$3F65 (16229) IN vector table
Like the PR vector table, only the 8 addresses vector the IN routines.

$3F75 (16245) Maximum length of Input buffer ($80)

$3F76 (16246) Length of Input buffer

$3F77 (16247) Input buffer
The Input buffer is where the Central loop places the line typed in on the keyboard. All characters are in ASCII form, with the end indicated by a 0.

$4076 (16502) Length of Crunch code buffer

$4077 (16503) Crunch code buffer
This is where the parser places the crunch coded line of input. If the line is meant for a program, then this buffer is copied into the Crunch code table. It ends with a 0.

$417B (16763) Coordinates of last plotted hi-res point

$417D (16765) Current SCALE

$417E (16766) Pointer to shape table

$4180 (16768) Used for DRAWing and ROTating

$4188 (16776) Current COLOR

$4189 (16777) Current HCOLOR

$418A (16778) PDL buffer
Contains the following data from the last PDL command: joystick, right button, left button, keypad, spinner. The data of the second paddle follows that of the first.

$4194 (16788) Binary file header data
Consists of the following data needed for the beginning of binary files: 1,0,2, followed by the address of the binary file in RAM.

$4197 (16791) Address of file in RAM

$4199 (16793) Length of file

$419D (16796) Temporary name of file in first file buffer

$41A9 (16809) Temporary name of file in second file buffer

$41B5 (16821) Device number for drive

$41B6 (16822) Temporary storage for files
Used by the tape routines to hold file numbers and data temporarily.

$41BD (16829) Vector to NO/MON I

$41BF (16831) Vector to NO/MON C

$41C1 (16833) Vector to NO/MON L

$41C3 (16835) Vector to NO/MON O

$41C5 (16837) Header for first file buffer
This is in the format: mode (A), file number (B), FCB address, length, address of name, 0,0.

$41CF (16847) Header for second file buffer
Same format as the above buffer.

$41D9 (16857) Name and length of first file buffer

$41E7 (16871) Name and length of second file buffer

$41F5 (16685) Complete file entry in directory

$4210 (16912) Temporary name holder
Used by CATALOG and APPEND to hold the ASCII names of files.

$4237 (16951) Temporary storage by APPEND

$4239 (16953) ASCII code of cursor

$423A (16954) ASCII code of blank character (space)

$423B (16955) ASCII code of current character

$423C (16956) Left margin for screen

$423D (16957) Right margin for screen

$423E (16958) Top margin for screen

$423F (16959) Bottom margin for screen

$4240 (16960) Buffer for screen routines

$4260 (16992) Unused RAM

$4261 (16993) Number of lines on screen (y) for HOME

$4262 (16994) Number of columns on screen (x) for HOME

$4263 (16995) Starting column number for HOME

$4264 (16996) Starting line number for HOME

$4265 (16997) Address in VRAM of Name table

$4267 (16999) Address in VRAM of Pattern table

$4269 (17001) Current line (y) position of cursor

$426A (17002) Current column (x) position of cursor

$426B (17003) Current input byte

This is the last ASCII byte read from the keyboard or tape.

$426C (17004) ASCII base

$426D (17005) Blinking cursor indicator
0 indicates the cursor is blinking, ≠0 means the cursor does not blink.

$426E (17006) ASCII base for cursor

$426F (17007) Current Name table
$0F means the first Name table in VRAM is being used, $FF indicates the second Name table is in use.

$4270 (17008) Current screen or graphics mode
This location holds the current screen mode. 0=TEXT, 1=GR, 2=HGR, 3=HGR2.

$4271 (17009) Print character indicator
$FF means the characters are printed on screen, 0= they are not.

$4272 (17010) Flash character indicator
$FF means some characters are flashed, while 0 means they are not being flashed.

$4273 (17011) Frequency of flashing

$4274 (17012) VRAM address of Name table for flashing

$4276 (17014) Ctrl-d indicator
0= no ctrl-d was pressed. 4= ctrl-d was pressed or printed.

$4277 (17015) Temporary storage of output

$4278 (17016) Length of Ctrl-d buffer

$4279 (17017) Ctrl-d buffer
The print routine places all characters to be printed into this buffer if it encounters a ctrl-d. It ends when a "return" (13) ASCII is printed.

$4290 (17040) Pointer to Ctrl-d buffer
This pointer points to the current position in the Ctrl-d buffer.

$4292 (17042) Temporary pointer to file names

$4294 (17044) Pointer to default file name
Points to the strings "$$$$1" or "$$$$2".

Chapter 8: Screen routines

The screen routines consist of the lo-res GR routines, the TEXT routine, and the routines that print characters on the video display. They do not handle printing to the printer or other device other than VRAM. The most important routines in this chapter are the ones at $4296, because it sets up the TEXT mode and is fun to change, $4352, because it calls all the other routines to print the character in A on the screen, $437B, because it loops until you press a key, flashing the cursor while it scans the keyboard, and $48B1, because it sets up the needed table in VRAM for GR. Other routines, like the Scroll screen routine at $46C0, are also interesting because they let you do things not easily attainable in a BASIC program.

While BASIC does not implement it, the Video Display Processor in Adam is capable of 40 columns of text. The reason why Coleco did not use it for the TEXT mode is unknown to me, because they wanted to keep compatability with Apple, which has 40 columns. But in case you wish to have 40 columns, a program in Chapter 11 lets you do this, even though it can't change the number of columns in GR or HGR.

$4296-4349 (17046-17225) Set TEXT
Called by TEXT to set up the video registers and VRAM with graphics mode 1. To change the color of the TEXT screen, POKE 17115 with the color of the foreground (pixel set) being in the top nibble, and the background (pixel off) color in the bottom nibble. For changing this routine to 40 columns, see chapter 11.

$434A-4351 (17226-17233) Pattern of a character
The pattern of character number $1F is replaced with the pattern stored here by the Set TEXT routine.

$4352-437A (17234-17274) Print character
This routine prints the character whose ASCII is in A. Control codes are printed if needed, along with scrolling and updating the cursor's position.

$437B-43B5 (17275-17333) Read keyboard
Reads the keyboard until a key is pressed, flashing the cursor when necessary. The rate of the cursor's flashing is stored at $438A and $438B. The default rate is $400.

$43B6-43EC (17334-17388) Init screen
In: B=number of columns, C=number of lines, D=top column, E=top line, HL=address of Name table in VRAM, A=address of Pattern table lo, A'=address of

Pattern table hi. Moves the data in the registers to page $42 to set up the screen for printing.

$43ED-43F3 (17389-17395) Reset ($4271) to 0

$43F4-4407 (17396-17415) Print cursor
Prints the $7F or $FF character depending on the cursor's position.

$4408-4427 (17416-17447) Print with control characters
This routine checks for control characters before printing the ASCII character in A. If a contol code is in A, then the routine to print the code is called depending upon the table at $4791. It falls through to the next routine if the character in A is not a control code.

$4428-44CA (17448-17610) Print without control characters
Prints the charcter in A on screen without checking for control codes.

$44CB-4639 (17611-17977) Control printing routines
The routines to print control code are gathered here according to the table at $4791.

$463A-464E (17978-17998) Clear buffer
Loads the $20 byte buffer at $4240 with the clear character ($423A).

$464F-467E (17999-18046) Clear screen
In: H=starting column, L=starting line, B=number of lines to clear. Loads both Name tables in VRAM with the clear character.

$467F-46A4 (18047-18084) Clear rest of line
Clears the remainder of the line, and writes it to VRAM.

$46A5-46BF (18085-18111) Read rest of line
Moves ASCII from VRAM to $4240 until the end of the line is reached.

$46C0-4714 (18112-18196) Scroll screen
Moves all lines up one position, filling the last line with the clear ASCII.

$4715-4745 (18197-18245) Update cursor
Flashes the cursor by erasing the cursor character in the first Name table.

$4746-4759 (18246-18265) Read character from screen
The character at x,y ($4269) is read into $423B.

$475A-476F (18266-18287) Calculate Name table position
In: H=column position (x), L=y. Calculates the address in VRAM of an x,y location for reading or writing by loading DE with y*32+x+base address of Name table in VRAM.

$4770-477F (18288-18303) Calculate pattern position
In: A=pattern number. Sets DE up like above except it points to the position in the Pattern table in VRAM, and BC=8.

$4780-4790 (18304-18320) Table of control ASCII
This table contains the ASCII codes of all the characters that require special printing routines. They are in the reverse order of the next table.

$4791-47B2 (18321-18354) Table of control addresses
The vectors of all of the following control characters are stored here:

address of routine	ASCII character
$47CB	ctrl-p
$457A	delete
$4542	insert
$44AD	ctrl-i, tab
$450A	right arrow
$4526	left arrow, ctrl-h, backspace
$44D9	ctrl-d, down arrow
$44FA	up arrow
$45F1	ctrl-/
$45CD	ctrl-x
$4619	ctrl-g
$45C4	home
$45B2	ctrl-l
$44CB	ctrl-m, return

$47B3-47CA (18355-18378) Calculate relative position
Loads DE with the cursor's distance from the edges of the window.

$47CB-4814 (18379-18452) Print control-p
Prints the rest of the line on the printer and on the screen.

$4815-483B (18453-18491) TEXT
Checks to see if the cursor ($4239) and the clear ($423A) characters are ASCII. It then calls $4296 to set up VRAM and set the mode pointer ($4270) to 0.

$483C-4883 (18492-18563) GR
Sets the mode byte to 1, and calls $48B1 to set up the VRAM tables.

$4884-489B (18564-18587) Lo-res block
This table of character patterns creates the 6x4 lo-res blocks for VRAM.

$489C-48B0 (18588-18608) GR video addresses
Contains the following data for the GR VRAM tables and registers (reg. 0=02, reg. 7=01):

$1F80	Sprite attribute table
$3800	Sprite pattern table
$1800	Name table
$2000	Pattern table
$0000	Color table

$48B1-4927 (18609-18727) Set GR
Called by GR to move the lo-res blocks into VRAM and set up the other tables and registers according to the above tables.

$4928-492E (18728-18734) Put HCOLOR
In: C=Coleco color. Calls $4936 to translate Coleco color into TI color, and puts it in $4189.

$492F-4935 (18735-18741) Put COLOR
Same as above, except the TI color is put in $4188.

$4936-4942 (18742-18754) Get color
Call $4936 for HCOLOR, $493B for COLOR. Translates the Coleco color to TI form, and puts the color in A.

$4943-494C (18755-18764) Translate color
In: A=TI color. Translates the TI color into Coleco color, putting it in A.

$494D-495C (18765-18780) HCOLOR table
TI color numbers in order of COLECO hi-res color scheme.

$495D-496C (18781-18796) Color table
TI color numbers in order of COLOR numbers.

$496D-49F3 (18797-18931) Plot HLIN
The actual entry point is at $497A. It sets the color bytes in VRAM to the current COLOR, looping until the end of the horizontal line is reached.

$49F4-4A95 (18932-19093) Plot VLIN
The entry point is at $49FC. It plots the vertical line by setting the color bytes of the blocks to the current color.

$4A96-4AF2 (19094-19186) PLOT point
$4A9E is the entry point. Sets the color bytes in VRAM to the current COLOR to plot the point.

$4AF3-4B3F (19187-19263) Do SCRN
The entry point is at $4AFB. Loads A with the color of the lo-res block pointed to by B (x), and C (y).

$4B40-4B56 (19264-19286) Read foreground color
In: DE=address in VRAM to be read. It loads A with the top nibble (foreground) of the byte pointed to by DE.

$4B57-4B67 (19287-19303) Read background color
Similar to the above routine, only it loads A with the bottom nibble.

$4B68-4B72 (19304-19314) Home screen
Checks to make sure the mode is not HGR2, and prints the ASCII 0C (home).

$4B73-4B85 (19315-19333) Load video registers with address
Loads the table pointed to by HL into the desired video registers, looping until the table is over. The table is in the format: register number, address byte lo, address byte high. It ends with an $FF as the register. The diagram on the following page shows a table of some sample data for this and the next routine.

$4BB6-4B93 (19334-19347) Load video registers
Same as the above routine, only that the table is in the format of register number and then the desired contents of that register, instead of an address.

$4B94-4BAF (19348-19375) Calculate GR offsets
In: B=x, C=y. Loads E with the offset from the right side (6x+8), and D with the offset from the top of the screen (y/2). A=the type of block.

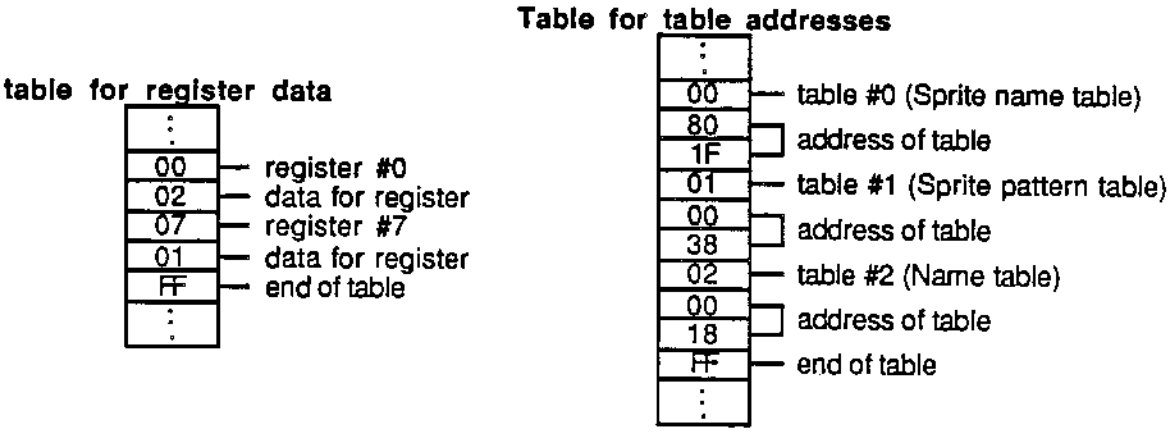


Diagram of some sample data for the register tables

$4BB0-4BDE (19376-19422) Plot top block
Sets the top nibble of the color byte pointed to by DE in VRAM to the current color, thus plotting the top GR block.

$4BDF-4C0E (19423-19470) Plot bottom block
Like the above routine, only it sets the bottom nibble to the current color.

Chapter 9: Tape routines

The tape routines and commands provide you with a way of storing your programs on a tape, disk, or other device. Changing the device is possible by poking the device number to $41B5 (e.g., "POKE 16821, 4" makes the first disk drive the device). As explained in chapter 1, the commands can normally be accessed in either the immediate mode or the program by using ctrl-d. After a tape command is typed in the immediate mode, the routine at $4DAC is called. That routine sorts out which command it is and what to call in order to execute it. It compares the command to the words in the ASCII table at $4EAA. If no match is found, it returns to print an error. But if a match is found, then the corresponding vector in a table at $4F4F is called. This vector routine, which is in a jump table at $4E03, checks for "Illegal Form Of OS Command" errors before and after the command's execution. It also calls the actual execution routine of the command. Since a tape command's syntax is usually short and rather simple, the execution routine itself can get parameters from the input buffer, and so no parse routine is needed.

The path a tape command follows from a program is very similar to the immediate mode tape word. The differences are in the compare routine (at $4C0F), the vector table ($4F73), and the jump table (at $4CED, checking for "Syntax" errors).

The process of adding a new tape command is similar to adding a normal command, only no parse routine is needed, and you must change the execution address of the old routine in two tables ($4CED and $4E03) instead of only one ($1917). Besides adding new commands, you can also change the ASCII of old commands (e.g., "LOAD" can be "DAOL") or errors (e.g., "I/O Error" can be "Bad Tape!").

$4C0F-4CEC (19471-19692) Print with tape check
In: A=ASCII of character to print. If A=04 (ctrl-d), then all the ASCII that it gets after it is put into the Ctrl-d buffer at $4279, until a return ASCII is given. It then looks up the first word in the Ctrl-d buffer in a table at $4EAA and that vector is called. If A≠04, it prints the character on the screen, checking the keyboard for pause or break.

$4CED-4DAB (19693-19883) Ctrl-d tape routines
The tape commands which can be used in programs with the ctrl-d are gathered here to check for errors before and after the command's execution. This group of

routines, along with the immediate group of routines at $4E03 are outlined in the following table, listing the command's entry point and actual routine address:

name	ctrl-d	imm.	routine
CATALOG	$4CED	$4E03	$5298
DELETE	$4CF5	$4E0F	$4F2A
RENAME	$4D00	$4E21	$4FF5
LOCK	$4D07	$4E28	$50A1
UNLOCK	$4D0E	$4E2F	$50A0
BSAVE	$4D15	$4E36	$5171
BLOAD	$4D1C	$4E3D	$5201
BRUN	$4D23	$4E44	$5294
CLOSE	$4D2A	$4E4B	$6024
MON	$4D31	$4E52	$5A07
NOMON	$4D38	$4E59	$5A02
LOAD	$4D3F	$4E60	$5DA8
SAVE	$4D46	$4E67	$5D05
OPEN	$4D4D	-	$5FB1
APPEND	$4D54	-	$53E5
WRITE	$4D5B	-	$57B7
READ	$4D62	-	$5621
POSITION	$4D69	-	$54D3
PR	$4D70	-	$6166
IN	$4D77	-	$616F
FP	$4D7F	$4E6E	$4FC3
INT	$4D88	$4E77	$4FC0
INIT	$4D91	$4E80	$62B3
RUN	$4D9A	$4E89	$5DCC
RECOVER	$4DA3	$4E92	$5034

$4DAC-4E02 (19884-19970) Immediate mode tape checker
This routine is called when a normal match for an immediate command is not found, or the command is not in variable form. It looks in the table at $4E9F for the command, ignoring commands only used in programs (OPEN, APPEND, etc.), and calls the vector of the routine.

$4E03-4E9A (19971-20123) Immediate tape routines
This group of command routines are together for error checking. It is like the ctrl-d routines ($4CED) in that they both call the same routine for the command, differing in their error printing or checking. For this reason, the entry points are listed in the above table with the ctrl-d entry points.

$4E9B-4EA9 (20124-20137) First letters of commands
The first letters (ASCII) of all the tape routines are stored here.

$4EAA-4F4E (20138-20302) Table of tape command ASCII

The ASCII names of all the tape commands are stored here in the format: number of letters in command, ASCII of command, offset into vector table. It starts with OPEN to IN, and then CATALOG to RUN, thus setting apart the immediate mode commands from the ctrl-d commands.

$4F4F-4F72 (20303-20338) Vectors of immediate commands
This table consists of the vectors of the immediate commands starting from CATALOG to RECOVER.

$4F73-4FA4 (20339-20388) Vectors of ctrl-d commands
This table is similar to the above one, except it starts at OPEN and goes to RECOVER, and they vector the ctrl-d commands.

$4FA5-4FBF (20389-20415) Tape error ASCII
The ASCII of "Illegal Form Of OS Command", preceded by its length, is stored here.

$4FC0-4FC9 (20416-20425) FP or INT
Call $4FC0 for FP, and $4FC3 for INT. This routine replaces the prompt with a '>' for INT, and a ']' for FP. These commands were included to provide compatability to Apple's Integer or Floating Point BASICs.

$4FCA-4FF4 (20426-20468) DELETE
Calls $FCE1 to delete the file, which can be either an 'A' or 'H' file, on any drive and with the name pointed to by DE.

$4FF5-5033 (20469-20531) RENAME
Calls $FCDE to rename the file on any drive with the names pointed to by DE.

$5034-509F (20532-20639) RECOVER
Makes an 'a' or 'h' file into a 'A' or 'H' file. This routine has a bug: it does not recover binary files. To change this, POKE 20619,72.

$50A0-50F9 (20640-20729) LOCK or UNLOCK
Call $50A0 for UNLOCK, $50A1 for LOCK. Sets the write protect bit of the file's attribute byte on or off, and sets the permanent bit off.

$50FA-5170 (20720-20848) Get address or length
Call $50FA for length, $50FD for address. Checks the buffer pointed to by DE for a comma, an 'L' or 'A', and a decimal or hexidecimal number. $4197 is loaded with the address, and $4199 with the length.

$5171-5200 (20849-20992) BSAVE
Creates a file consisting of the following: length of header high (1), length of header lo (0), type of file (2), address of file in RAM lo, high, binary data from the given address with a given length.

$5201-5293 (20993-21139) BLOAD
Opens and loads in the binary file from any drive into the wanted address, or to the original address of the file.

$5294-5297 (21140-21143) BRUN
Calls $5201 to load the binary file, and then it jumps to the beginning of the file, so no return is expected from the file.

$5298-5352 (21144-21330) CATALOG
Reads the first block on the tape or disk to get the directory. It calls $5353 to print "Volume:", and the name of the tape. It then prints the files in order if they are not deleted (bit 2 set), or it is not a system file (bit 3 set). If bit 0 of the attribute byte is set, then the directory is over, and the number of free blocks on the tape is printed.

$5353-5367 (21331-21351) Print tape or disk name
Prints the string at $53C4 ("Volume:"), and the name of the tape or disk, which is stored $41F5.

$5368-5397 (21352-21399) Print file data
Prints the following from the directory entry at $41F5: a space, the lock status, the file type, length of file, a space, and the file name.

$5398-53A4 (21400-21412) Print lock status
Prints an asterisk (*) if bit 7 of the attribute byte ($4201) is set.

$53A5-53C3 (21413-21443) Read file name
Moves the name pointed to by HL to the buffer at $4210. It ends when an 03 character is reached, or the name is $C characters long.

$53C4-53E4 (21444-21476) Words for directory
$53C4=the ASCII for "DIRECTORY". $53CE="Volume:". $53D7="Blocks free".

$53E5-54D2 (21477-21714) APPEND
Opens a file, whose name is pointed to by DE, and skips to the end of it for further writing. It then sets up the Print screen vector to Write to tape, so that any characters printed will be set to the file.

$54D3-554E (21715-21838) POSITION
Opens a file and skips to the record number pointed to by DE. Records are separated by a return chatacter.

$554F-5552 (21839-21842) Write to tape
This routine is called when a character is meant to be printed on screen, but it is written to the tape or disk for a file. It jumps to NO/MON O.

$5553-5556 (21843-21846) Read from tape
Similar to the above routine, only it jumps to NO/MON I to read the tape.

$5557-555F (21847-21855) MON I
Reads a character from the tape into A, and prints it on the screen.

$5560-55DC (21856-21980) NOMON or MON O
Entry point for NOMON is at $5572, MON is at $556D. It writes the character in A to the tape, and updates the length of the file. If MON O was called, the character is also printed on the screen.

$55DD-5620 (21981-22048) NOMON I
Reads a character from the tape into A and updates the file pointers.

$5621-57B6 (22049-22454) READ
Checks the buffer pointed to by DE for a file name, and optional record number or length of records. It then skips to the desired record in the file, and changes the input vector to Read from tape ($5553) to get a character from tape instead of the keyboard.

$57B7-598A (22455-22922) WRITE
Like the above routine, except it changes the Print vector to Write to tape ($554F) to fill a file instead of only being printed on the screen.

$598B-5A01 (22923-23041) Check for record number or length
Looks for an upper or lower case 'B' or 'R' at the buffer pointed to by DE, and, if either is found, loads HL with the 'B' number, and DE with 'R'.

$5A02-5ABE (23042-23230) NOMON or MON
Entry point for MON is at $5A07, NOMON is at $5A02. It looks for any of the following letters, and, depending on the command, makes the following changes:

letter	vector	MON	NOMON
C	$41BD	$4BF8	$5AA9 (a return)
I	$41BF	$5557	$55DD
O	$41C1	$556D	$5572
L	$41C3	$4352	$5AA9

$5ABF-5AD0 (23231-23248) Legal file ASCII
A list of acceptable ASCII for file names is stored here.

$5AD1-5ADE (23249-23262) Default file names
The ASCII for "$$$$1" is stored at $5AD1, and "$$$$2" is at $5AD8.

$5ADF-5AEE (23263-23278) Drive to device table

The ASCII characters for S, V, and D are stored at $5ADF, $5AE2 stores all the combinations of these three characters, and the following table at $5AE9 matches a drive number to its device number:

drive	device	drive	device
1	$08	4	$19
2	$18	5	$04
3	$09	6	$05

$5AEF-5B07 (23279-23303) Check ASCII of file name
Sets the Carry flag if the character pointed to by DE is a letter, number, or one of the legal ASCII characters stored at $5ABF.

$5B08-5B1B (23304-23323) Get second file name
Looks for a comma, and falls through to the next routine, putting the name at $41A9.

$5B1C-5B43 (23324-23363) Get first file name
Calls $5AEF to see if the ASCII pointed to by DE is legal, and if it is, then the name is moved to $419D, adding an 'A' and 03 on the end of it.

$5B44-5BB4 (23364-23476) Get drive number
Looks at the buffer pointed to by DE to check for an S, V, or D. If one is found, the number after it is placed in the current drive pointer ($41B5). The Carry flag is also set if this occurs.

$5BB5-5BC0 (23477-23488) Skip over file name
This routine skips over the name of the file pointed to by HL. The ending of the name is shown by an 03. HL is then pointed to the end of the name.

$5BC1-5BCF (23489-23503) Change file type
The entry point for the buffer at $41A9 is at $5BC1, and call $5BC6 for the buffer at $419D. This routine skips over the name in the wanted buffer, and changes the file type to the ASCII code in A.

$5BD0-5BE0 (23504-23520) Look for default name
If either default names ("$$$$1" or "$$$$2") are found in the FCB, then the Zero flag is reset, otherwise the "No Buffer Available" error is printed.

$5BE1-5C52 (23521-23634) Skip over header on tape
If the file in the buffer at $419D is an 'A' file, then it returns, because normal programs don't have a header. If the file is an 'H' file, then it moves the tape over the header, and returns.

$5C53-5CD8 (23635-23768) Update file backups
In: A=ASCII of new file type. It checks the directory for any old 'A' or 'H' files,

renames them to 'a' or 'h' files, deleting any old 'a' or 'h' files, and saves the new 'A' or 'H' file.

$5CD9-5CF3 (23769-23795) Read one byte from tape
Reads one byte from the device pointed to by $41B6 into A, setting Zero flag if the file being read is over.

$5CF4-5D04 (23796-23812) Write one byte
Sends the byte in A to the device pointed to by $41B7.

$5D05-5D7E (23813-23934) SAVE
Gets the name of the file from the buffer pointed to by DE. It then calls $6277 to see how long the file will be, and Makes the file. $5F23 is called to write the program to the tape, after which it updates any backups.

$5D7F-5DA7 (23935-23975) Input routine for LOAD
This routine is called by 'Input line' to read a byte from the tape instead of the keyboard. It calls $5CD9 to read one byte into A. If the file is over, it restores the original pointers and closes the file.

$5DA8-5DF5 (23976-24053) LOAD or RUN
LOAD is at $5DA8, and RUN is at $5DCC. It opens the file whose name is pointed to by DE, and redirects the vectors for 'Input line', etc. so that they read the tape instead of the keyboard. To load a new program without erasing the old one, simply POKE 24010,163: POKE 24011,62. See chapter 11 for more information on this change.

$5DF6-5E22 (24054-24098) Input routine for RUN
Similar to the Input routine for LOAD at $5D7F, except it jumps to RUN when the file is over.

$5E23-5E3D (24099-24125) Write program to tape
Used by SAVE to write a program to the tape. It calls $3493 to print the program, only the SAVE routine changes the vector so the file goes to tape.

$5E3E-5EE8 (24126-24296) File error table
The strings printed when an error occurs are stored here in the format: length of string, and the string. The following table lists the error numbers, the address to print the error, and the errors.

Error #	Address	Error
$01	$5F0A	Range Error
$02	$5F0D	Write Protected
$03	$5F10	End of Data
$04		-
$05	$5F13	File Not Found
$06		I/O Error
$07	$5F19	No More Room
$08	$5F1C	File Locked
$09	$5F1F	Syntax Error
$0A	$5F22	No Buffers Available
$0B	$5F25	File Type Mismatch
$0C		-
$0D		-
$0E	$5F07	Control Buffer Overflow

$5EE9-5F62 (24297-24418) Print file errors
The entry points for certain errors can be seen in the above table. The routine jumps to the Central loop when it is done printing the error.

$5F63-5FB0 (24419-24496) Close files
This routine is called after an error has occured. It restores any read or write pointers, and closes any file buffers in RAM and on tape.

$5FB1-6023 (24497-24611) OPEN
This routine gets a name and drive from the buffer at DE, and checks to see if the file already exists. It creates a new one if it doesn't, and sets up the buffers at $41D9 or $41E7 with the file's name.

$6023-6165 (24612-24933) CLOSE

$6166-616E (24934-24942) PR
This routine is the same as the other PR command at $2F1A.

$616F-6177 (24943-24951) IN
This routine is also like the IN routine at $2F41.

$6178-6193 (24952-24979) Read DE for a number 0-7
Looks at the buffer pointed to by DE to see if the ASCII is a number from 0 to 7. HL is loaded with the number if it is.

$6194-61FE (24980-25086) Set up File data buffer
In: HL=pointer to file name, B=mode, A=file number. It finds an empty buffer at

$41C5 or $41CF, and moves the following data into it: mode, file number, FCB address, length, length of name.

$61FF-621B (25087-25115) Close File data buffer
Looks for the data buffer containing the file whose number is in B. If it is found, then the first byte is set to zero, thus closing it.

$621C-623E (25116-25150) Buffer check
If the name pointed to by HL is not in one of the buffers at $41D9 or $41E7, then the Carry flag is reset, otherwise, it sets the Carry flag.

$623F-625D (25151-25181) Find File data buffer
Looks for the buffer containing the file whose number is in A. If it is found, then A is loaded with the file's mode byte.

$625E-626A (25182-25194) Fill end of File data buffer
It finds the buffer for the file number in A, and loads the last two positions with C and B.

$626B-6276 (25195-25206) Get length of name
Searches the buffer pointed to by HL for an 03, and loads C with its length.

$6277-6293 (25207-25235) Get length of program
Used by SAVE to see how long a program is. It acts like it will print the program, but diverts the print routine to $6294 to increment a counter each time a byte is supposed to be printed. $4197 is loaded with the length.

$6294-62A8 (25236-25256) Increment block counter
This routine increments the 4 byte counter at $4197 to see the length of the program.

$62A9-62B2 (25257-25266) ASCII for BASIC file name
Contains the ASCII characters for the BASIC program's directory entry on the tape.
$62B3-6309 (25267-25353) INIT
Remakes the directory of the tape in the drive. It makes the first directory entry the tape's name, and the second is the 'BOOT' program. It does not change a tape with the BASIC program on it.

$630A-6311 (25354-25361) Set date
Calls $FCD8 to set the date to 13/10/57.This is presumably the birth date of an author of SmartBASIC.

$6312-6319 (25362-25369) Init's data
$6315 contains the ASCII characters for "BOOT", and $6312 is the Boot routine (JP $FCE7).

Chapter 10: Graphics

The section of Basic from $631A to $6B0A handles the hi-res graphics and game paddles. The routines in it are called by the routines in the command section (chapter 5). The 'Hplot' routine in chapter 5 loads the Z80 registers with data from crunch code and calls the 'Hplot' routine in this chapter to plot the point or line. It plots a point by changing bits in a special group of memory called VRAM (Video RAM), in which each bit is a pixel on the screen. Since there is 16K of VRAM and only about 6K of pixels, there is 10K left over for the storage of color and sprites.

The TI video chip that handles VRAM organizes each section of data into tables pointed to by registers in the video chip. These registers are not like the ones in the Z80, because the video registers can only be written to. Register 0 and 1 hold information concerning the mode the chip is in. Graphics mode 1 is the TEXT mode, and mode 2 is the GR, HGR, or HGR2 mode. Register 2 points to the Name table. Each byte in the Name table corresponds to a region on the screen, and the number in the table specifies the pattern to be displayed there. Register 3 points to the Color table, which holds the foreground and background colors for each group of 8 pixels in the pattern table. Register 4 points to the Pattern table. The Pattern table stores pixels in blocks of 8 (one byte) that form a pattern displayed on the screen according to the Name table. Register 5 and 6 point to the Sprite attribute table, which holds the color and position of each sprite, and the Sprite generator table, which is like the Pattern table in that it stores the shape of each sprite. For more information, see vol. 1 or chapter 11 for sprites. In mode 1, the Name table is 768 bytes long. Each entry in the table points to a group of 8 bytes in the Pattern table. Since this allows the ASCII code of a character to point directly to the entire character pattern, it is used by the TEXT mode. In mode 2, the name and pattern tables are three times as long as in mode 1. This lets each entry in the name table point to a separate pattern of 8 bytes, unlike in mode 1, where patterns have to be reused. Each byte in the Pattern table also has its own background and foreground colors. This mode is used in GR by loading the Pattern table with repeating groups of 6 pixels set, 6 off, etc. It has the entries in the Name table point to separate patterns, and only changes the Color table to plot the point. A similar setup is used in HGR2, only the patterns and colors are changed to plot a point. HGR fixes up the ends of the tables from HGR2 to create four lines of text.

$631A-6343 (25370-25411) HGR2
Sets the mode to HGR2 by loading $4270 with 03. It then calls $6359 to set up the tables in VRAM.

$6344-6358 (25412-25432) VRAM addresses for HGR
The table at $6344 lists the address for each kind of VRAM table, as seen below. The table at $6354 lists immediate data for some video registers.

address	table
$1F80	Sprite attribute table
$3800	Sprite generator table
$1800	Name table
$2000	Pattern table
$0000	Color table

$6359-638B (25433-25483) Set HGR2
Sets up the video registers according to the above tables, loads the color table with black foreground and background colors, and erases the pattern table. $66A0 is finally called to fill the name table.

$638C-6400 (25484-25600) HGR
It sets the mode to HGR by loading $4270 with 02. It then calls $6359 to set up the tables in VRAM, but when it is returned to, it modifies the name table and the pattern table to allow four lines of text at the bottom.

$6401-6455 (25601-25685) HPLOT x,y
This routine calculates the address in VRAM of the point in B (x) and C (y), moves the wanted pattern and color bytes to the tables at $661B and $6613, and calls $6543 to plot the point. It then moves the data back into VRAM by calling $65EF.

$6456-6542 (25686-25922) HPLOT x,y to x1,y1
Entry point for HPLOT TO x,y is at $64C5. This routine takes the two endpoints of a line (BC to DE) and plots the points in between them. It reads the pattern and color bytes from VRAM, plots the line, and writes the data back into VRAM as the line is plotted.

$6543-65D7 (25923-26071) Plot a point
Checks to make sure the point is on the screen, and the pattern and color bytes are in the buffers. It then updates the last point plotted byte ($417B). The HCOLOR byte at $4189 contains both the color and the indicator to either plot the point (bit 7=0), or to erase it (bit 7=1). If it is to be plotted, then the color byte is changed to the color, and the pattern bytes are changed even if the point will be erased. Note that it does not write the data back into VRAM.

$65D8-65EE (26072-26094) Calculate offset for patterns
This routine calculates the pattern number of the point, whose x is in B, and y is in C, in the pattern table.

$65EF-660A (26095-26122) Write pattern and color
Writes the 8 byte pattern buffer at $660B to the VRAM address pointed to by BC. It also writes the color buffer ($6613) to the color table.

$660B-6612 (26123-26130) Pattern buffer
An 8 byte buffer for storing the pattern of a position in VRAM is located here.

$6613-661C (26131-26140) Color buffer
This buffer is like the one above, only it stores a pattern's color bytes. A temporary pointer used in plotting for VRAM is stored at $661B.

$661D-6626 (26141-26150) INVERSE
Sets $426C to $80, and $426E to 00.

$6627-6632 (26151-26162) NORMAL
Sets $426C and $426E to 0 if it is in TEXT mode.

$6633-6640 (26163-26176) FLASH
Sets $426E to $80, and $426C to 0 if it is in TEXT mode.

$6641-6647 (26177-26183) Do POS
Calls $47B8 to get the cursor's horizontal position into A.

$6648-664E (26184-26190) Do VPOS
Calls $47B8 to get the vertical position into A.

$664F-666A (26191-26218) Do HTAB
Resets the cursor's horizontal position to the number in C, rounded to the nearest third position. It wraps around if needed.

$666B-669F (26219-26271) Do VTAB
Sets the cursor's vertical position to the number in C, checking to make sure it is on the screen.

$66A0-66B8 (26272-26296) Fill name table
Fills the name table in VRAM with 0 to $FF, repeating 3 times, so that each pattern is pointed to by only one name table position.

$66BC-66CC (26297-26316) Do XDRAW with no x or y
Sets the top bit of $4189 (hcolor) to one, draws the shape by calling $66CD, and then return $4189 to its original value.

$66CD-66DC (26317-26332) Do DRAW with no x or y
Gets the last point plotted ($417B), and draws the shape there.

$66DD-66E7 (26333-26343) Do SCALE
Puts the new scale number in C to $417D.

$66E8-67CD (26344-26573) Do ROT
Rotates the shapes according to the number in C. It does this by using some data to switch the shape's pattern.

$67CE-67DB (26574-26587) Default shape table
The default shape table is stored here. It contains the shape used for demonstrations in the BASIC manual.

$67DC-6903 (26588-26883) Do DRAW
This routine looks up the shape whose number is in E. It then plots the shape by tracing its steps, using the data at $4180 to help the routine decipher it.

$6904-6917 (26884-26903) Do XDRAW
Like the XDRAW routine at $66B9, only it draws the shape with $67DC.

$6918-6B0E (26904-27406) Do PDL
Calls $FD3E to read the game paddles, then depending upon what is wanted (joystick, button, etc.), A is loaded with its status. It uses the buffer at $418A to store the data from all the paddle options.

Chapter 11: BASIC Changes

This chapter summarizes minor fixes for SmartBASIC and gives examples of how to add your own commands for sprites and sound. SmartBASIC is one of the best BASICs around. But it has bugs like adding spaces to REM and DATA lines, or not Recovering 'h' files. In order to fix these bugs, you can either change BASIC on the tape or disk, or you can Poke changes in after BASIC is booted. Using a HELLO program makes the changes easy and almost invisible to the user. If you wish to make the changes permanent, run the following program with your BASIC tape or disk in the drive. The program asks for the drive number, the address you want to change, and the contents you want the address changed to.

```
 2 REM  ---BASIC-ed.(changes the BASIC tape)---
 3 LOMEM :40000: INPUT "Drive (8=tape, 4=disk)?"; dd
 4 PRINT "Inset BASIC tape or disk into drive"
 5 DATA  62,8,1,0,0,17,0,0,33,48,117,205,243,252,201
 7 FOR x = 0 TO 14: READ d: POKE 29000+x, d: NEXT: POKE 29001, dd
10 PRINT: INPUT "Address to change?"; ad
20 INPUT " new contents for address?"; n
30 ad = ad-256: ap = INT(ad/1024): ab = ad-ap*1024+30000
40 POKE 29006, ap+2: CALL 29000
50 POKE ab, n: POKE 29012, 246: CALL 29000
60 RESTORE: GOTO 5
```

BASIC from disk

If you have a disk, you can copy your SmartBASIC from tape to disk with the Backup program in vol. 1 or any other program that lets you copy BASIC. However, BASIC still looks for a HELLO program on tape. The program above can be used to change the device that BASIC looks to for a HELLO program. This lets you put both BASIC and the HELLO program on disk. The address of the device used to look for HELLO is 16641. By using BASIC-ed., you can change it from an 8 to a 4 and whenever you boot BASIC, it will look to the disk for the HELLO program.

Recovering binary files

The tape "Recover" command has a bug that won't let it recover 'h' files. The Recover routine at $5034 checks to see if the file is an 'a' file. If it is, then the Recover routine changes it to an 'A' file. But if it is an 'h' file, the routine changes it to 'h' instead of 'H'. This is a simple fix, because all we have to do is change the second 'h' to 'H'. The location of the 'h' is 20619, and the ASCII for 'H' is 72, so Poking 20619,72 allows Recovery of binary files.

DATA bump bug

The routine at $3DC6, which parses DATA and REM statements, has a bug. It adds a space at the start of your data when you type the line in, run your cursor over it, or load it in from tape or disk. If you are making many updates to a REM or DATA line, the spaces can pile up, and may push your data off the end. This destructive bug can be fixed by including the following POKEs in your HELLO program:

POKE 15830,8: POKE 15831,55: POKE 15832,19: POKE 15824,216

Interesting Pokes

The Data table of Chapter 7 stores many important pointers and other structures. Some of the things stored there are not very interesting, and should not be changed. Others, like many screen pointers, provide results not possible with normal commands. You can change the color of the screen, size of the screen, or the ASCII codes of many characters displayed on it. A few interesting pointers are listed below with their address and function.

address	function
16953	ASCII code of the cursor (0-255; default=95)
16954	ASCII code of a blank character (default=32)
16956	left margin of the screen (0-31; default=1)
16957	right margin (default=31)
16958	top margin (0-24; default=0 in TEXT, 20 in GR and HGR)
16959	bottom margin (default=23)
16993	# of line to HOME (default=24 in TEXT, 4 in GR and HGR)
16994	# of columns to HOME (default=30)
16995	top margin for HOME (default=0)
16996	left margin for HOME (default=1)
17001	y position of the cursor (same as VPOS)
17002	x position of the cursor (same as POS)
159	rate of FLASH (default=12)
17115	color of text and background installed by TEXT
18711	color of text and background in GR
25568	color of text and background in HGR
16134	ASCII code for Break (default=ctrl-c)
16135	ASCII code for Pause (default=ctrl-s)
16149	Poke limit (2 bytes in lo, hi format)
16148	ASCII code for indenting lines in LIST (default=32)
16763	coordinates of the last hi-res point plotted (2 bytes)

File names

On the Apple II file names can have spaces. For some unknown reason, Coleco does not allow file names to have spaces, though file names can have other non-letter or number ASCII. A list of these ASCII codes can be found at $5ABF. If we replace one of these codes with the code for a space, then file names can include spaces in them. Poking 23240,32 replaces the ASCII character @ ($40) with a space.

CHAINing programs

The SmartBASIC that came with our first Adam had the command CHAIN, even though it did not perform its function. It is used on the Apple II to load in one program over another without erasing the first. This is helpful when you are using libraries, because you can load in the subroutines that you need for a specific program. Of course, each subroutine must have unique line numbers, because the programs are loaded in just as if you were typing them, so a line that has the same line number as an existing line replaces the old line. The newer SmartBASICs don't have the CHAIN command, but the following Pokes change the LOAD command so that it is like CHAIN:

```
POKE 24010, 163: POKE 24011,62
```

In order to remove the changes and have the old LOAD command back, POKE 24010,212 and POKE 24011, 24.

Line numbers

Did you ever wish you could 'GOTO x*10'? It could replace lengthy ON... GOTO lines with a simple (or complex) equation. The following Pokes let you do this to both GOTO and GOSUB:

```
10 DATA 0,0,0,205,3,39,68,77
20 FOR x=0 TO 7: READ d: POKE 8342+x, d: POKE 8437+x, d: NEXT
30 POKE 15756, 195: POKE 15757, 27: POKE 15758, 58
```

40 Columns

The following program changes the TEXT mode so that it uses 40 columns of text instead of 31. It does not work in GR, HGR or HGR2, because it uses the text mode on the Video Display Processor. It makes all the needed changes, including changing the offset calculation routine, the TEXT SETUP routine, and the 40 byte screen buffer, which is relocated to 28094 below LOMEM. The Poke for changing the color of the letters and the screen is still at 17115. Sprites cannot be displayed in this mode because of the VDP's restrictions.

```
 30 LOMEM :28400
 98 REM ----40 COLUMNS----
 99 REM ---Poke in TEXT changes---
110 DATA 1,240,7,205,32,253,24,14,41,197,229,41
115 DATA 41,193,58,112,66,183,32,1,9,193,201
120 FOR x = 0 TO 7: READ d: POKE 17114+x, d: NEXT
125 POKE 17177, 192: POKE 17166, 192: POKE 17988, 40
130 POKE 17215, 240: POKE 17199, 39
135 REM ---Change offset routine---
140 FOR x = 0 TO 14: READ d: POKE 16976+x, d: NEXT
155 REM ---Change these addresses---
160 DATA 17985,18036,18098,18162,18174,18188,18401,18410,18430
170 FOR x = 1 TO 9: READ y: POKE y, 190: POKE y+1, 109: NEXT
180 POKE 18272, 205: POKE 18273, 80: POKE 18274, 66
185 TEXT
```

Macros

It is possible to have a string of ASCII printed when you hit a certain key (e.g., hitting the 'Store/Get' key prints "LOAD "). The following program lets you have 30 macros (keys that store a string in them), which are stored in buffers at 28203 and 28234. The program predefines some macros for your use, but the real fun is in making your own macros. To do this, all you have to do is add or change the Data statements from line 910 and up. The data is in the format: ASCII code of the key, string to to be printed. If the string has a '&' in it, then the next two bytes store an ASCII code in hex (e.g., "CATALOG&0D" would print "CATALOG" and a "return" i.e., CHR$(13)). The predefined keys and their strings are:

key	string
clear	NEW
delete	DELETE space
print	PRINT space
insert	&0E
store/get	LOAD space
shift+store/get	SAVE space
I	CATALOG return
II	RUN return
III	LIST return
IV	TEXT return
V	PR#1 return
VI	PR#0 return

```
  30 LOMEM :28400
 700 REM ----MACRO----
 710 REM ---Poke in 'Input line' changes---
 720 DATA E5,C5,D5,21,3B,6E,ED,5B,3B,6E,7B,B2,20,1A,CD
 725 DATA 69,2F,E5,21,5A,6E,01,1E,00,ED,B9,E1
 730 DATA 20,18,11,5B,6E,13,1A,B7,20,F9,0D,20,F8,13,1A
 740 DATA B7,73,23,72,20,05,36,00,2B,36,00,01,C1,E1,C9
 750 FOR x = 0 TO 56: READ d$: GOSUB 800: POKE x+27407, d: NEXT
 753 POKE 12197, 15: POKE 12198, 107
 755 REM ---Poke in macros---
 760 POKE 28221, 0: POKE 28252, 0: mk = 28222: md = 28253
 770 READ a: IF a = 0 THEN 850
 780 POKE mk, a: mk = mk+1: READ a$: FOR x = 1 TO LEN(a$): d = ASC(MID$(a$, x,
1))
 785 IF d = 38 THEN  d$ = MID$(a$, x+1, 2): x = x+3: GOSUB 800
 790 POKE md, d: md = md+1: NEXT: POKE md, 0: md = md+1: GOTO 770
 799 REM ---Change hex to dec---
 800 d = 0
 805 FOR i = 2 TO 1 STEP -1
 810 j = ASC(MID$(d$, 3-i, 1))
 820 IF j > 64 THEN  j = j-55
 830 IF j > 47 THEN  j = j-48
 840 d = d+j*i^4: NEXT i: RETURN
 850 END
 900 REM ---Preset macros---
 910 DATA 150,NEW,151,DELETE
 915 DATA 149,PRINT ,148,&0E
 920 DATA 147,LOAD ,155,SAVE
 925 DATA 129,CATALOG&0D,130,RUN&0D,131,LIST&0D
 930 DATA 132,TEXT&0D,133,PR#1&0D,134,PR#0&0D
 998 DATA 0
 999 END
```

Sound

Volume 1 described the sound chip and gave examples of how to drive it. The best way to make sounds, however, is to have a command in BASIC. The following program creates such a command, called "Sound". It starts by Poking the execution routine into RAM. After this, line 570 Pokes the new parse address into the Break command, the command that Sound will replace. This is the part of the Primary word table's entry that points to the command's parse vectors, which is what line 573 Pokes in. The parse vectors are in the format: # of vectors, vectors. Thus the sound command's parse vector entry is 01, 119, 59. Lines 575 and 576 change the Break entry's ASCII in the Primary word table to 'SOUND'. Line 580 changes the execution vector to point to the new Sound command. Lines 590 to 607 SETUP the various tables for the Sound command. Locations 1 to 7 in page 0 point to the current note being played for each voice. Locations 9 to 11 store the note's number (0 to 9). Locations 17 to 22 point to the latest note entered, and locations 25 to 27 store that note's number for each voice. Locations 27974 to 28093 is a 120 byte buffer that stores the notes for each voice in the format: length, frequency lo, frequency hi, volume. Lines 610 to 690 poke in the interrupt routine that looks at each voice's note after each interrupt and counts the length of notes. This routine is jumped to when the VDP creates an interrupt. It sends out notes to the sound chip and updates the sound notes and pointers. It seems that the interrupt routine has a bug of some sort; when the screen is being used for a long time without typing TEXT(or other mode commands that reset VRAM), it randomly inserts either text or cursors into VRAM that are displayed on the screen. Don't be alarmed if suddenly an "h" doesn't look like an "h" anymore, it's only on the screen and not in your program. If someone finds the problem and sees how to fix it, please contact me. I think it has to do with the timing of the VDP or the length of the routine.

The syntax for the command is:

SOUND [voice 1-3], [length 0-255], [pitch 0-1023], [volume 0-15]

For the length of notes, the higher numbers specify longer lengths. For pitch and volume,
however, the higher the number, the lower or softer the note is. Following the program that installs the sound command is a program that demonstrates the command.

```
 30 LOMEM :28400
 90 POKE 15756, 195: POKE 15757, 27: POKE 15758, 58
499 REM  ----SOUND----
500 REM  ---Clear 0 page---
510 POKE 102, 237: POKE 103, 69
515 REM  ---Poke in Sound routine---
520 DATA  205,220,5,125,245,135,198,15,111,229
525 DATA 126,35,102,111,229,217,13,217,19,205,220,5
530 DATA  125,193,2,3,217,13,217,19,197,205,3
535 DATA 39,193,125,230,15,2,3,41,41,41,41
540 DATA 124,230,63,2,3,217,13,217,19,197,205
545 DATA 220,5,193,125,2,3,225,241,229,198,24,111,38,0,52,126
550 DATA  254,10,32,8,54,0,33,216,255,9,229,193,225,113,35,112,201
560 FOR x = 0 TO 87: READ d: POKE x+27755, d: NEXT
565 REM  ---Change tables for Sound command---
570 POKE 788, 230: POKE 789, 109: POKE 28134, 1: POKE 28135, 119: POKE 28136,
59
575 DATA  83,79,85,78,68
576 FOR x = 0 TO 4: READ d: POKE 791+x, d: NEXT
580 POKE 6549, 107: POKE 6550, 108
585 REM  ---Setup note tables---
590 DATA   70,109,110,109,150,109
600 FOR x = 1 TO 6: READ d: POKE x, d: POKE x+16, d: NEXT
605 FOR x = 27974 TO 28093: POKE x, 0: NEXT
607 FOR x = 9 TO 12: POKE x, 0: POKE x+16, 0: NEXT
609 REM  ---Poke in 0 page routine---
610 DATA  213,229,197,245,6,3,14,0,97,104,41,43,229
615 DATA 126,35,102,111,175,182,209,40,94,213,35,203,126
620 DATA  43,32,36,229,197,5,120,135,203,121,40,1,60
625 DATA 6,4,203,39,16,252,35,182,246,128,211,250
630 DATA  203,121,193,32,8,35,126,211,250,203,249,24,222,225,53,35,203,254,35,
35,35,209,32,41,213
635 DATA  197,5,120,135,60,6,4,203,39,16,252,246,143,211,250,193,235
640 DATA  120,198,8,111,38,0,52,126,254,10,32,7,54,0,33,216,255,25
650 DATA 235,225,115,35,114,16,144,205,35,253,241,193,225,209,237,69
660 FOR x = 0 TO 126: READ d: POKE x+27847, d: NEXT
665 REM  ---Set up 0 page---
670 DATA  195,199,108
680 FOR x = 1 TO 3: READ d: POKE 101+x, d: NEXT
690 CALL 64803: REM  ---restart interrupts---
```

```
 5 REM random demo program for SOUND
 7 REM  run HELLO first
10 v = INT(RND(1)*3)+1
20 d = INT(RND(1)*200)+1
30 f = INT(RND(1)*1000)+50
40 vo = INT(RND(1)*15)+1
50 SOUND v, d, f, vo
60 GOTO 10
```

Sprites

Coleco did not include a sprite command in SmartBASIC because the Apple II did not have sprites, even though the VDP chip has hardware capable of 32 sprites. In order to use the 32 sprites, as described in vol. 1, you had to use complex Pokes and machine language routines. The following program lets you easily create and draw up to 31 sprites in BASIC.

A sprite is a group of 64 pixels arranged in an 8x8 pattern, with the pixels stored as bits in 8 bytes (or 256 pixels in a 16x16 pattern, because16x16 sprites have 4 groups of 8x8 patterns). See vol. 1 for more information on sprites. Each sprite can be displayed independently of any other in any mode except for the 40 column TEXT mode.

In order to use sprites in your own programs, I have created four new commands: SETUP, DEFINE, SPDRAW, BUMP. Lines 10 to 20 replace the VPOS variable command with the "BUMP" ASCII and vector. Lines 210 to 250 change the Primary word table and Command vector table to replace STORE, RECALL and SHLOAD with SETUP, DEFINE and SPDRAW.

Lines 255 to 290 Poke in the 'SETUP' command. The SETUP command is needed to switch from little to big sprites, or vice versa. Its effects are seen immediately upon any sprite that is being dispayed when you enter the SETUP command. It has the following syntax:

SETUP [magnification], [size]

[magnification]=0 for normal sized sprites, and =1 for sprites that are twice as big (each pixel is expanded to 4 pixels). [size]=0 for 8x8 sprites, and =1 for 16x16 sprites.

Lines 305 to 340 Poke in the 'DEFINE' command. DEFINE loads the sprite's pattern into VRAM so that they can be drawn. It has no visual output by itself. DEFINE's syntax is:

DEFINE [sprite #1-31], [byte 1], [byte 2]...

[sprite #] is the sprite that will be defined.[byte 1]... is the data for that sprite. You should have 8 bytes for 8x8 sprites, and 32 bytes for 16x16 sprites.

Lines 355 to 400 Poke in the 'SPDRAW' command. It performs the same function as the shape table's DRAW command (drawing shapes or patterns on the screen). Sprites are drawn in the current HCOLOR, and are erased from their previous position if you redraw them at a different location. Sprites can be drawn in any screen mode except for the 40 column TEXT mode. SPDRAW has the syntax:

SPDRAW [sprite #1-31] AT [x coordinate], [y coordinate]

[sprite #] is the sprite to be drawn.[x coordinate] and [y coordinate] specify the location at which the sprite will be drawn. They are like the x and y coordinates of any shape table drawn with DRAW.

Lines 410 to 440 Poke in the 'BUMP' routine. The BUMP command can only be used in equations, (e.g., x= BUMP(10)*50). It replaces the VPOS command in the variable command tables. BUMP returns the number of the lowest number sprite that is overlapping with the sprite in parenthesis (e.g., if sprite #5 is at 100,100, sprite #10 is at 97,100, and sprite #27 is at 100,99, then BUMP(10)=5 and BUMP(27)=5. If there has not been a collision BUMP (x)=0. Unfortunately, it cannot check for collisions with patterns made with HPLOT, PLOT, DRAW, etc. Bump's syntax is:

BUMP ([sprite #1-31])

The following program installs the sprite commands, followed by a demonstration of the sprite commands.

```
  5 REM ---Change tables for 'bump'---
 10 POKE 27548, 38: POKE 27549, 108: DATA 66,85,77,80
 15 i = PEEK(16098)*256+PEEK(16097)+121
 20 FOR x = 0 TO 3: READ d: POKE i+x, d: NEXT
 30 LOMEM :28400
 90 POKE 15756, 195: POKE 15757, 27: POKE 15758, 58
199 REM ----SPRITE----
200 REM ---Change tables for sprite commands---
210 DATA 0,4,6,83,80,68,82,65,87,52,230,109,6,68,69,70,73,78,69
220 DATA 53,249,3,5,83,69,84,85,80
230 FOR x = 677 TO 704: READ d: POKE x, d: NEXT: REM Primary word  table
235 POKE 28134, 1: POKE 28135, 119: POKE 28136, 59
240 DATA 72,107,167,107,226,107
243 FOR x = 6523 TO 6528: READ d: POKE x, d: NEXT: REM Command vector table
245 REM ---sprite setup---
250 DATA 205,220,5,125,254,2,210,0,31,14,224
255 DATA 177,79,217,13,217,19,197,205,220,5,193
257 POKE 25413, 0: POKE 18589, 0: POKE 17104, 0
260 DATA 125,254,2,210,0,31,135,177,50,176
265 DATA 254,79,6,1,213,205,32,253,33,0,31,62,0
270 DATA 205,41,253,33,0,56,62,1,205,41,253
275 DATA 175,33,0,31,17,1,0,205,38,253,209,201
280 FOR x = 0 TO 67: READ d: POKE x+27618, d: NEXT
305 REM ---sprite define---
310 DATA 205,220,5,125,183,202,0,31,254,32,210,0,31,229,33,176,254
320 DATA 203,78,225,40,2,41,41,41,41,41,1,255
325 DATA 55,9,229,217,121,13,217,183,40,18,19
330 DATA 205,220,5,125,225,35,229,213,17,1,0,205,38,253,209,24,231,225,201
340 FOR x = 0 TO 58: READ d: POKE x+27559, d: NEXT
355 REM ---sprite draw---
360 DATA 205,220,5,125,183,40,2,254,32,210,0,31,229,217,13,217,19,205,220,5
370 DATA 34,178,254,217,13,217,19,205,220,5
375 DATA 125,33,177,254,119,43,203,78,225,229
380 DATA 125,40,2,135,135,33,179,254,119,58
385 DATA 137,65,35,119,225,229,41,41,1,0,31,9,213,235
390 DATA 33,177,254,1,4,0,205,26,253,209,225,213,0,175,103,1
395 DATA 233,109,41,9,17,178,254,26,119,35,27,26,119,209,201
400 FOR x = 0 TO 94: READ d: POKE x+27464, d: NEXT
405 REM ---Sprite bump---
410 DATA 194,3,31,205,50,9,218,0,31,125,254,32,210,0,31,41,1,233,109,9
415 DATA 126,230,248,87,54,255,35,126,230,248,95
420 DATA 54,255,229,96,105,6,32,126,35,230,248
425 DATA 186,32,6,126,230,248,187,40,5,35,16,240
430 DATA 6,32,62,32,144,225,115,43,114,38,0,111,195,103,9
440 FOR x = 0 TO 68: READ d: POKE 27686+x, d: NEXT
```

```
  5 REM demo of sprites
  7 REM run HELLO first
 10 HGR: SETUP 0, 0: DIM x(31), y(31)
 20 FOR x = 1 TO 10
 30 DEFINE x, 28, 28, 8, 29, 42, 20, 98, 4
 40 NEXT
 45 t = t+.2
 50 FOR x = 1 TO 10
 60 HCOLOR  = x
 70 SPDRAW x AT x(x), y(x)
 80 x(x) = INT(60*SIN(x/5+t))+70
 90 y(x) = INT(60*COS(x/5+t))+70
100 NEXT x: GOTO 45
```

In vol. 1, there is a program that lets you edit sprites. With a few modifications, this same program can be used to edit sprites for the new commands. The following program is similar to the older one, but it prints the sprite's definition for DEFINE when you are done.

```
]
  2 REM     sprite-editor by B. Hinkle
  3 DIM i(33)
 10 PRINT: PRINT: PRINT: PRINT "Would you like to have an:": PRINT: co = 1
 12 PRINT "  1. 8x8 sprite ": PRINT "  2. 16x16 sprite ": PRINT: INPUT "(1,2)?
"; s
 20 IF s < 1 OR s > 2 THEN  TEXT: GOTO 10
 30 rb = s*8+11: bb = s*8+1
 50 GR: COLOR  = 10: x = 11: y = 1
 60 VLIN 0, bb AT 10: VLIN 0, bb AT rb: HLIN 10, rb AT 0: HLIN 10, rb AT bb
 65 REM     print commands on screen
 70 PRINT "    arrow keys to move cursor"
 80 PRINT "'a'-plot", "'d'-erase"
 90 PRINT "'return' when done with sprite"
 95 PRINT "sprite #"; d;
 99 REM     main loop
100 COLOR  = 6: PLOT x, y
110 GET a$: p = ASC(a$)
120 IF e = 1 THEN  COLOR  = 8: PLOT x, y: GOTO 140
130 COLOR  = 0: PLOT x, y
135 REM     check for special commands
140 IF p = 97 THEN  COLOR  = 8: PLOT x, y
150 IF p = 100 THEN  COLOR  = 0: PLOT x, y: e = 0
155 IF p = 13 THEN 200
157 REM     check for arrow keys
160 IF p = 163 AND x-1 > 10 THEN  x = x-1: e = 0
165 IF p = 161 AND x+1 < rb THEN  x = x+1: e = 0
167 IF p = 160 AND y-1 > 0 THEN  y = y-1: e = 0
170 IF p = 162 AND y+1 < bb THEN  y = y+1: e = 0
180 IF SCRN(x, y) = 8 THEN  e = 1
190 GOTO 100: REM     go back to main loop
199 REM     print sprite's data
200 IF s = 2 THEN 280
205 REM     8*8 sprite figuring
210 aa = 8: ab = 1: ac = 18: ad = 11: GOSUB 230
220 GOTO 500
229 REM   compute an 8*8 block
230 FOR y = ab TO aa: i = 0
240 FOR x = ac TO ad STEP -1
250 IF SCRN(x, y) = 8 THEN  i = i+2^(ac-x)
260 NEXT x: i(co) = i: co = co+1: NEXT y
270 RETURN
```

Continued next Page

```
  279 REM       16*16 sprite figuring
  280 aa = 8: ab = 1: ac = 18: ad = 11: GOSUB 230
  290 aa = 16: ab = 9: ac = 18: ad = 11: GOSUB 230
  300 aa = 8: ab = 1: ac = 26: ad = 19: GOSUB 230
  310 aa = 16: ab = 9: ac = 26: ad = 19: GOSUB 230
  499 REM     save sprites on tape
  500 TEXT: PRINT: PRINT: FOR x = 1 TO s^2*8-1: PRINT i(x); ","; : NEXT: PRINT i
(x)
  501 PRINT "would you like a hard copy?(y/n)"
  502 INPUT a$: IF a$ = "y" THEN  GOSUB 600
  505 PRINT: PRINT: INPUT "Would you like to plot another sprite  (y/n)?"; a$
  510 IF a$ < > "y" AND a$ < > "n" THEN 500
  520 IF a$ = "n" THEN  PRINT "End of program": END
  530 GOTO 10
  600 PR #1
  610 FOR x = 1 TO s^2*8-1
  620 PRINT i(x); ","; : NEXT: PRINT i(x)
  630 PR #0: RETURN
```

In order for you to have all of the features listed above at once, they have been grouped together into a HELLO program (except for the 'BASIC from disk' and CHAIN fixes) which you can type in and save on your BASIC tape or disk. Any HELLO program you have been using can be RENAMEd to be BELLO, and the HELLO program below will load it from tape and execute it as if it were a HELLO program. If you don't have a BELLO program on the BASIC tape or disk, then the HELLO program enters the immediate mode like it does when no HELLO program exists. All the fixes are stored under a LOMEM of 28400, so you will have to modify programs that use any RAM lower than this, or else the program will conflict with the new commands and fixes. The LOMEM is divided into the following sections:

address	contents
27407	macro routine
27464	SPDRAW routine
27559	DEFINE routine
27618	SETUP routine
27686	BUMP routine
27755	SOUND routine
27847	interrupt routine for sound
27974	note table for 3 voices
28094	40 column screen buffer
28134	parse vector for DEFINE and SOUND
28137	sprite coordinate buffer for BUMP
28201	pointer to current macro being printed
28203	table of macro keys
28233	table of macro definitions

In order for you to fully understand the commands and help you make your own commands, the assembly code for the routines (locations 27407 to 27973) have been included in appendix 2.

```
]
    3 REM ----HELLO program to install BASIC changes (save on BASIC tape)-----
    5 REM ---Change tables for 'bump'---
   10 POKE 27548, 38: POKE 27549, 108: DATA 66,85,77,80
   15 i = PEEK(16098)*256+PEEK(16097)+121
   20 FOR x = 0 TO 3: READ d: POKE i+x, d: NEXT
   30 LOMEM :28400
   35 FOR x = 27407 TO 28399: POKE x, 0: NEXT
   40 POKE 20619, 72: REM Recover fix
   50 POKE 15830, 8: POKE 15831, 55: POKE 15832, 19: POKE 15824, 216: REM Data-B
ump-Bug
   60 POKE 23240, 32: REM spaces in file names
   70 DATA 0,0,0,205,3,39,68,77
   80 FOR x = 0 TO 7: READ d: POKE 8342+x, d: NEXT: REM line number fix
   90 POKE 15756, 195: POKE 15757, 27: POKE 15758, 58
   97 REM
   98 REM ----40 COLUMNS----
   99 REM ---Poke in TEXT changes---
  110 DATA 1,240,7,205,32,253,24,14,41,197,229,41
  115 DATA 41,193,58,112,66,193,32,1,9,193,201
  120 FOR x = 0 TO 7: READ d: POKE 17114+x, d: NEXT
  125 POKE 17177, 192: POKE 17166, 192: POKE 17988, 40
  130 POKE 17215, 240: POKE 17199, 39
  135 REM ---Change offset routine---
  140 FOR x = 0 TO 14: READ d: POKE 16976+x, d: NEXT
  155 REM ---Change these addresses---
  160 DATA 17985,18036,18098,18162,18174,18188,18401,18410,18430
  170 FOR x = 1 TO 9: READ y: POKE y, 190: POKE y+1, 109: NEXT
  180 POKE 18272, 205: POKE 18273, 80: POKE 18274, 66
  185 TEXT: TEXT: PRINT "Just a moment"
  198 REM
  199 REM ----SPRITE----
  200 REM ---Change tables for sprite commands---
  210 DATA 0,4,6,83,80,68,82,65,87,52,230,109,6,68,69,70,73,78,69
  220 DATA 53,249,3,5,83,69,84,85,80
  230 FOR x = 677 TO 704: READ d: POKE x, d: NEXT: REM Primary word  table
  235 POKE 28134, 1: POKE 28135, 119: POKE 28136, 59
  240 DATA 72,107,167,107,226,107
  243 FOR x = 6523 TO 6528: READ d: POKE x, d: NEXT: REM Command vector table
  245 REM ---sprite setup---
  250 DATA 205,220,5,125,254,2,210,0,31,14,224
  255 DATA 177,79,217,13,217,19,197,205,220,5,193
  257 POKE 25413, 0: POKE 18589, 0: POKE 17104, 0
  260 DATA 125,254,2,210,0,31,135,177,50,176
  265 DATA 254,79,6,1,213,205,32,253,33,0,31,62,0
  270 DATA 205,41,253,33,0,56,62,1,205,41,253
  275 DATA 175,33,0,31,17,1,0,205,38,253,209,201
  280 FOR x = 0 TO 67: READ d: POKE x+27618, d: NEXT
  305 REM ---sprite define---
  310 DATA 205,220,5,125,183,202,0,31,254,32
  315 DATA 210,0,31,229,33,176,254
  320 DATA 203,78,225,40,2,41,41,41,41,1,255
  325 DATA 55,9,229,217,121,13,217,183,40,18,19
  330 DATA 205,220,5,125,225,35,229,213,17,1
  335 DATA 0,205,38,253,209,24,231,225,201
  340 FOR x = 0 TO 58: READ d: POKE x+27559, d: NEXT
  355 REM ---sprite draw---
  360 DATA 205,220,5,125,183,40,2,254,32,210
  365 DATA 0,31,229,217,13,217,19,205,220,5
  370 DATA 34,178,254,217,13,217,19,205,220,5
  375 DATA 125,33,177,254,119,43,203,78,225,229
  380 DATA 125,40,2,135,135,33,179,254,119,58
  385 DATA 137,65,35,119,225,229,41,41,1,0,31,9,213,235
  390 DATA 33,177,254,1,4,0,205,26,253,209,225,213,0,175,103,1
  395 DATA 233,109,41,9,17,178,254,26,119,35,27,26,119,209,201
  400 FOR x = 0 TO 94: READ d: POKE x+27464, d: NEXT
  405 REM ---Sprite bump---
  410 DATA 194,3,31,205,50,9,218,0,31,125,254,32,210,0,31,41,1,233,109,9
  415 DATA 126,230,248,87,54,255,35,126,230,248,95
  420 DATA 54,255,229,96,105,6,32,126,35,230,248
  425 DATA 186,32,6,126,230,248,187,40,5,35,16,240
  430 DATA 6,32,62,32,144,225,115,43,114,38,0,111,195,103,9
  440 FOR x = 0 TO 68: READ d: POKE 27686+x, d: NEXT
  498 REM
```

```
499 REM  ----SOUND----
500 REM  ---Clear 0 page---
510 POKE 102, 237: POKE 103, 69
515 REM  ---Poke in Sound routine---
520 DATA  205,220,5,125,245,135,198,15,111,229
525 DATA 126,35,102,111,229,217,13,217,19,205,220,5
530 DATA  125,193,2,3,217,13,217,19,197,205,3
535 DATA 39,193,125,230,15,2,3,41,41,41,41
540 DATA 124,230,63,2,3,217,13,217,19,197,205
545 DATA 220,5,193,125,2,3,225,241,229,198,24,111,38,0,52,126
550 DATA  254,10,32,8,54,0,33,216,255,9,229,193,225,113,35,112,201
560 FOR x = 0 TO 87: READ d: POKE x+27755, d: NEXT
565 REM  ---Change tables for Sound command---
570 POKE 788, 230: POKE 789, 109: POKE 28134, 1
573 POKE 28135, 119: POKE 28136, 59
575 DATA  83,79,85,78,68
576 FOR x = 0 TO 4: READ d: POKE 791+x, d: NEXT
580 POKE 6549, 107: POKE 6550, 108
585 REM  ---Setup note tables---
590 DATA   70,109,110,109,150,109
600 FOR x = 1 TO 6: READ d: POKE x, d: POKE x+16, d: NEXT
605 FOR x = 27974 TO 28093: POKE x, 0: NEXT
607 FOR x = 9 TO 12: POKE x, 0: POKE x+16, 0: NEXT
609 REM  ---Poke in 0 page routine---
610 DATA  213,229,197,245,6,3,14,0,97,104,41,43,229
615 DATA 126,35,102,111,175,182,209,40,94,213,35,203,126
620 DATA  43,32,36,229,197,5,120,135,203,121,40,1,60
625 DATA 6,4,203,39,16,252,35,182,246,128,211,250
630 DATA  203,121,193,32,8,35,126,211,250,203
631 DATA 249,24,222,225,53,35,203,254,35,35,35,209,32,41,213
635 DATA  197,5,120,135,60,6,4,203,39,16,252,246,143,211,250,193,235
640 DATA  120,198,8,111,38,0,52,126,254,10,32,7,54,0,33,216,255,25
650 DATA 235,225,115,35,114,16,144,205,35,253,241,193,225,209,237,69
660 FOR x = 0 TO 126: READ d: POKE x+27847, d: NEXT
665 REM  ---Set up 0 page---
670 DATA  195,199,108
680 FOR x = 1 TO 3: READ d: POKE 101+x, d: NEXT
690 CALL 64803: REM  ---restart interrupts---
693 REM poke in routine for loading the BELLO program
695 DATA 205,87,23,33,163,62,229,195,250,64
697 FOR x = 0 TO 9: READ d: POKE 64+x, d: NEXT
699 REM
700 REM ----MACRO----
710 REM ---Poke in 'Input line' changes---
720 DATA E5,C5,D5,21,3B,6E,ED,5B,3B,6E,7B,B2,20,1A,CD
725 DATA 69,2F,E5,21,5A,6E,01,1E,00,ED,B9,E1
730 DATA 20,18,11,5B,6E,13,1A,B7,20,FB,0D,20,F8,13,1A
740 DATA B7,73,23,72,20,05,36,00,2B,36,00,D1,C1,E1,C9
750 FOR x = 0 TO 56: READ d$: GOSUB 800: POKE x+27407, d: NEXT
753 POKE 12197, 15: POKE 12198, 107
755 REM ---Poke in macros---
760 POKE 28221, 0: POKE 28252, 0: mk = 28222: md = 28253
770 READ a: IF a = 0 THEN 850
780 POKE mk, a: mk = mk+1: READ a$
781 FOR x = 1 TO LEN(a$): d = ASC(MID$(a$, x, 1))
785 IF d = 38 THEN  d$ = MID$(a$, x+1, 2): x = x+3: GOSUB 800
790 POKE md, d: md = md+1: NEXT: POKE md, 0: md = md+1: GOTO 770
799 REM ---Change hex to dec---
800 d = 0
805 FOR i = 2 TO 1 STEP -1
810 j = ASC(MID$(d$, 3-i, 1))
820 IF j > 64 THEN  j = j-55
830 IF j > 47 THEN  j = j-48
840 d = d+j*i^4: NEXT i: RETURN
850 TEXT: POKE 16681, 66: CALL 64: END
900 REM ---Preset macros---
910 DATA 150,NEW,151,DELETE
915 DATA 149,PRINT ,148,&0E
920 DATA 147,LOAD ,155,SAVE
925 DATA 129,CATALOG&0D,130,RUN&0D,131,LIST&0D
930 DATA 132,TEXT&0D,133,PR#1&0D,134,PR#0&0D
998 DATA 0
999 END

]
```

Appendix 1: Programs

The following programs were mentioned in earlier chapters (BASIC Overview and Math Chapters). They have been reprinted here for your convenience. The Crunch code viewer program lets you examine the crunch code of any line you can type in. Simply enter the line as line 1000, and RUN the program. Line 1000 will never be executed, so you don't have to worry about its affects on the rest of the program. The program looks in the Line number table for line 1000. When it finds it, the program finds the corresponding crunch code and prints it out. If you want to print the crunch code on the printer for a hard copy, include the following line:

2 PR#1

The program following the Crunch code viewer prints the floating point representation of a number you provide. The number can be positive, negative, whole, or with a decimal point. If you don't want it to print the floating point number on the printer, erase the PR#1 and PR#0 commands in line 40 and 90. The program finds the floating point number by assigning the number you type in to a variable (w). BASIC then converts it to floating point, and stores it in the Variable tables. The program looks in the Variable value table, and prints out the first value in the table, because "w" was the first numeric variable assigned a value. That way you can see the floating point format for any number.

```
  3 REM   ---crunch code viewer for line 1000---
  5 LIST 1000: x$ = "0123456789ABCDEF"
 10 p = PEEK(16090)*256+PEEK(16089)
 20 IF PEEK(p) = 232 AND PEEK(p+1) = 3 THEN 40
 30 p = p+4: GOTO 20
 40 y = PEEK(p+3)*256+PEEK(p+2)
 50 x = PEEK(y): GOSUB 100: PRINT d$; ","; : IF x = 0 THEN  END
 55 FOR i = 1 TO PEEK(y): x = PEEK(i+y)
 60 GOSUB 100: PRINT d$; ","; : NEXT i: PRINT "00": END
100 d$ = MID$(x$, INT(x/16)+1, 1)+MID$(x$, x-INT(x/16)*16+1, 1): RETURN
998 END
999 REM  line 1000 will not be executed, only printed
1000 REM replace this line with the one you want to see
```

```
 10 REM   Prints floating point representation in hex
 20 h$ = "0123456789ABCDEF"
 30 INPUT "Enter number in decimal"; w
 40 PR #1: PRINT w; "   ";
 50 FOR x = 0 TO 4: a = PEEK(53340+x)
 60 b = a/16: c = INT(b): GOSUB 100
 70 c = a-c*16: GOSUB 100
 80 PRINT " "; : NEXT
 90 PRINT: PR #0: RUN
100 PRINT MID$(h$, c+1, 1); : RETURN
```

Appendix 2: HELLO code

In chapter 11, there is a HELLO program that installs 40 columns, macros, sound and sprites to BASIC. The assembly language needed to create these changes is printed below. They serve as examples of how to write new commands. Note that the sound and sprite commands often decrement C' and increment DE, because DE points to the crunch code for the line and C' hold the length of the line. Registers C' and DE are often Pushed to the stack when the routine needs an extra register to do something, and then Poped off when it is done using them for its own purpose. Other registers, like HL' and DE', also need to be Pushed and Poped when you use them for someting other than their original purpose. The page 0 routine Pushes every register at the start and then Pops them all off at the end, because the routine is called during an interrupt, which can occur at any time in the middle of any routine, so the interrupt routine has to Push and Pop every register that it uses. The BUMP routine is interesting because it replaces a variable command. By looking at it, you can get a feel for this type of command. They often call similar subroutines to either change FPA1 into the HL register, or vice versa. The number within the parenthesis of any variable command (numeric) is stored in FPA1 when it calls the command. This way the command doesn't have to bother getting the number. For the BUMP routine, and other variable command routines, register BC is the only register that needs to be Pushed and Poped. The DE register is already Pushed by the routine that calls the variable command. The macro routine is an example of diverting an already existing routine so that it can perform some other function as well as its original one. The HELLO program replaces three byte in the original routine so that it calls the new routine. The new routine performs the action of these three bytes, and then does what it wants, in this case it checks the keyboard for the macro keys and prints any necessary macros. When the routine returns to the original one, the original continues on its way, usually without noticing the change. However you use these printouts, whether to learn from them or to change them, I'm sure they will be appreciated.

Macro routine, called by "input line" at $2F7F.

```
27407 6B0F  E5          PUSH  HL
27408 6B10  C5          PUSH  BC                    save registers
27409 6B11  D5          PUSH  DE
27410 6B12  213B6E      LD    HL,nn     6E3B
27413 6B15  ED5B3B6E    LD    DE,(nn)   6E3B
27417 6B19  7B          LD    A,E                   is macro being printed?
27418 6B1A  B2          OR    D                     yes. get next ASCII
27419 6B1B  201A        JR    NZ,e  6B37            no. get input from keyboard
27421 6B1D  CD692F      CALL  nn    2F69
27424 6B20  E5          PUSH  HL
27425 6B21  215A6E      LD    HL,nn     6E5A
27428 6B24  011E00      LD    BC,nn     001E
27431 6B27  EDB9        CPDR                        compare input with macro table
27433 6B29  E1          POP   HL                      at $6E5A
27434 6B2A  2018        JR    NZ,e  6B44            if match is found look up string
27436 6B2C  115B6E      LD    DE,nn     6E5B
27439 6B2F  13          INC   DE
27440 6B30  1A          LD    A,(DE)                get macro ASCII to print
27441 6B31  B7          OR    A
27442 6B32  20FB        JR    NZ,e  6B2F
27444 6B34  0D          DEC   C
27445 6B35  20F8        JR    NZ,e  6B2F
27447 6B37  13          INC   DE
27448 6B38  1A          LD    A,(DE)                increment pointers
27449 6B39  B7          OR    A
27450 6B3A  73          LD    (HL),E                if word over load pointer with 0
27451 6B3B  23          INC   HL
27452 6B3C  72          LD    (HL),D
27453 6B3D  2005        JR    NZ,e  6B44
27455 6B3F  3600        LD    (HL),n
27457 6B41  2B          DEC   HL
27458 6B42  3600        LD    (HL),n
27460 6B44  D1          POP   DE                    pop registers and return
27461 6B45  C1          POP   BC
27462 6B46  E1          POP   HL
27463 6B47  C9          RET
```

SPDRAW routine.

```
27464 6B48  CDDC05      CALL  nn    05DC            get sprite #
27467 6B4B  7D          LD    A,L                   print error if <0 or >31
27468 6B4C  B7          OR    A
27469 6B4D  2802        JR    Z,e   6B51
27471 6B4F  FE20        CP    n
27473 6B51  D2001F      JP    NC,nn     1F00
27476 6B54  E5          PUSH  HL
27477 6B55  D9          EXX
27478 6B56  0D          DEC   C                     save it on stack
27479 6B57  D9          EXX                         get x-coord #
27480 6B58  13          INC   DE
27481 6B59  CDDC05      CALL  nn    05DC
27484 6B5C  22B2FE      LD    (nn),HL   FEB2        save it at $FEB2
27487 6B5F  D9          EXX                         get y-coord #
27488 6B60  0D          DEC   C
27489 6B61  D9          EXX
27490 6B62  13          INC   DE
27491 6B63  CDDC05      CALL  nn    05DC
27494 6B66  7D          LD    A,L
27495 6B67  21B1FE      LD    HL,nn     FEB1
27498 6B6A  77          LD    (HL),A                save it at $FEB1
```

SPA/24W, cont.

```
27499 6B6B  2B        DEC   HL
27500 6B6C  CB4E      BIT   1,(HL)
27502 6B6E  E1        POP   HL
27503 6B6F  E5        PUSH  HL
27504 6B70  7D        LD    A,L
27505 6B71  2802      JR    Z,e  6B75          multiply sprite # by 4
27507 6B73  87        ADD   A,A                if in 16X16 sprite mode
27508 6B74  87        ADD   A,A
27509 6B75  21B3FE    LD    HL,nn    FEB3
27512 6B78  77        LD    (HL),A             save # at $FEB3
27513 6B79  3A8941    LD    A,(nn)    4189
27516 6B7C  23        INC   HL
27517 6B7D  77        LD    (HL),A             save color at $FEB4
27518 6B7E  E1        POP   HL
27519 6B7F  E5        PUSH  HL
27520 6B80  29        ADD   HL,HL
27521 6B81  29        ADD   HL,HL
27522 6B82  01001F    LD    BC,nn    1F00
27525 6B85  09        ADD   HL,BC
27526 6B86  D5        PUSH  DE
27527 6B87  EB        EX    DE,HL
27528 6B88  21B1FE    LD    HL,nn    FEB1       load 4 bytes from $FEB1
27531 6B8B  010400    LD    BC,nn    0004       to VRAM in the sprite name taole
27534 6B8E  CD1AFD    CALL  nn    FD1A
27537 6B91  D1        POP   DE
27538 6B92  E1        POP   HL
27539 6B93  D5        PUSH  DE
27540 6B94  00        NOP
27541 6B95  AF        XOR   A
27542 6B96  67        LD    H,A
27543 6B97  01E96D    LD    BC,nn    6DE9
27546 6B9A  29        ADD   HL,HL
27547 6B9B  09        ADD   HL,BC
27548 6B9C  11B2FE    LD    DE,nn    FEB2
27551 6B9F  1A        LD    A,(DE)             move x and y from $FEB1
27552 6BA0  77        LD    (HL),A             to sprite entry in table
27553 6BA1  23        INC   HL                 at $6DE9 (used for BUMP)
27554 6BA2  1B        DEC   DE
27555 6BA3  1A        LD    A,(DE)
27556 6BA4  77        LD    (HL),A
27557 6BA5  D1        POP   DE                 return
27558 6BA6  C9        RET
```

DEFINE routine

```
27559 6BA7  CDDCD5    CALL  nn    05DC       get sprite #
27562 6BAA  7D        LD    A,L
27563 6BAB  B7        OR    A
27564 6BAC  CA001F    JP    Z,nn    1F00
27567 6BAF  FE20      CP    n                  error if <0 of >31
27569 6BB1  D2001F    JP    NC,nn    1F00
27572 6BB4  E5        PUSH  HL
27573 6BB5  21B0FE    LD    HL,nn    FEB0
27576 6BB8  CB4E      BIT   1,(HL)
27578 6BBA  E1        POP   HL
27579 6BBB  2802      JR    Z,e  6BBF
27581 6BBD  29        ADD   HL,HL
27582 6BBE  29        ADD   HL,HL              mult # by 64 if
27583 6BBF  29        ADD   HL,HL              in 16X16 mode
27584 6BC0  29        ADD   HL,HL
27585 6BC1  29        ADD   HL,HL
27586 6BC2  01FF37    LD    BC,nn    37FF
27589 6BC5  09        ADD   HL,BC              add # to base of
27590 6BC6  E5        PUSH  HL                 sprite pattern table
27591 6BC7  D9        EXX                      ($3800)
27592 6BC8  79        LD    A,C
27593 6BC9  0D        DEC   C
27594 6BCA  D9        EXX
```

DEFINE, cont

```
27595 6BCB  B7        OR    A                 is crunch code over?
27596 6BCC  2812      JR    Z,e  6BE0         yes. then return
27598 6BCE  13        INC   DE                no. get # (data for pattern)
27599 6BCF  CDDC05    CALL  nn    05DC
27602 6BD2  7D        LD    A,L
27603 6BD3  E1        POP   HL
27604 6BD4  23        INC   HL
27605 6BD5  E5        PUSH  HL
27606 6BD6  D5        PUSH  DE
27607 6BD7  110100    LD    DE,nn   0001
27610 6BDA  CD26FD    CALL  nn    FD26        send it to VRAM
27613 6BDD  D1        POP   DE
27614 6BDE  18E7      JR    e  6BC7
27616 6BE0  E1        POP   HL                loop to get next #
27617 6BE1  C9        RET                     return
```

SETUP routine

```
27618 6BE2  CDDC05    CALL  nn    05DC        get magnification #
27621 6BE5  7D        LD    A,L               error if >1
27622 6BE6  FE02      CP    n
27624 6BE8  D2001F    JP    NC,nn   1F00
27627 6BEB  0EE0      LD    C,n
27629 6BED  B1        OR    C
27630 6BEE  4F        LD    C,A
27631 6BEF  D9        EXX
27632 6BF0  0D        DEC   C
27633 6BF1  D9        EXX
27634 6BF2  13        INC   DE
27635 6BF3  C5        PUSH  BC
27636 6BF4  CDDC05    CALL  nn    05DC        get size #
27639 6BF7  C1        POP   BC
27640 6BF8  7D        LD    A,L
27641 6BF9  FE02      CP    n
27643 6BFB  D2001F    JP    NC,nn   1F00        error if >1
27646 6BFE  87        ADD   A,A
27647 6BFF  B1        OR    C                 combine mag and size
27648 6C00  32B0FE    LD    (nn),A  FEB0      to put in register 1 of VDP
27651 6C03  4F        LD    C,A               save # at $FEB0
27652 6C04  0601      LD    B,n               (for draw and define)
27654 6C06  D5        PUSH  DE
27655 6C07  CD20FD    CALL  nn    FD20        send # to reg 1
27658 6C0A  21001F    LD    HL,nn   1F00
27661 6C0D  3E00      LD    A,n
27663 6C0F  CD29FD    CALL  nn    FD29        clear sprite name table
27666 6C12  210038    LD    HL,nn   3800
27669 6C15  3E01      LD    A,n
27671 6C17  CD29FD    CALL  nn    FD29
27674 6C1A  AF        XOR   A                 set registers to point to
27675 6C1B  21001F    LD    HL,nn   1F00      $3800 for sprite pattern table
27678 6C1E  110100    LD    DE,nn   0001
27681 6C21  CD26FD    CALL  nn    FD26
27684 6C24  D1        POP   DE
27685 6C25  C9        RET                     return
```

```
                                                BUMP routine
27686 6C26  C2031F    JP     NZ,nn    1F03     error if string
27689 6C29  CD3209    CALL   nn       0932     convert FPA1 to HL
27692 6C2C  DA001F    JP     C,nn     1F00
27695 6C2F  7D        LD     A,L
27696 6C30  FE20      CP     n                 error if HL too big
27698 6C32  D2001F    JP     NC,nn    1F00
27701 6C35  29        ADD    HL,HL
27702 6C36  01E960    LD     BC,nn    60E9
27705 6C39  09        ADD    HL,BC             use HL as offset into $6DE9
27706 6C3A  7E        LD     A,(HL)
27707 6C3B  E6F8      AND    n
27709 6C3D  57        LD     D,A
27710 6C3E  36FF      LD     (HL),n            load D and E with
27712 6C40  23        INC    HL                x and y of sprite
27713 6C41  7E        LD     A,(HL)
27714 6C42  E6F8      AND    n
27716 6C44  5F        LD     E,A
27717 6C45  36FF      LD     (HL),n
27719 6C47  E5        PUSH   HL
27720 6C48  60        LD     H,B
      6C49  69        LD     L,C               search table for
      6C4A  0620      LD     B,n               coordinates of other sprites
27724 6C4C  7E        LD     A,(HL)
27725 6C4D  23        INC    HL
27726 6C4E  E6F8      AND    n
27728 6C50  BA        CP     D
27729 6C51  2006      JR     NZ,e  6C59
27731 6C53  7E        LD     A,(HL)
27732 6C54  E6F8      AND    n
27734 6C56  BB        CP     E
27735 6C57  2805      JR     Z,e   6C5E
27737 6C59  23        INC    HL
27738 6C5A  10F0      DJNZ   e  6C4C           if match is not found
27740 6C5C  0620      LD     B,n               then loop again
27742 6C5E  3E20      LD     A,n
27744 6C60  90        SUB    B
27745 6C61  E1        POP    HL
27746 6C62  73        LD     (HL),E
27747 6C63  2B        DEC    HL
27748 6C64  72        LD     (HL),D            get # of the sprite that
27749 6C65  2600      LD     H,n               bumps and put it in FPA1
27751 6C67  6F        LD     L,A
27752 6C68  C36709    JP     nn     0967

                                                SOUND routine

27755 6C6B  CDDC05    CALL   nn     05DC       get voice #
27758 6C6E  7D        LD     A,L
27759 6C6F  F5        PUSH   AF                push it
27760 6C70  87        ADD    A,A               mult by 2 and add 15
27761 6C71  C60F      ADD    A,n               to point to next note
27763 6C73  6F        LD     L,A               push pointer
27764 6C74  E5        PUSH   HL
27765 6C75  7E        LD     A,(HL)
27766 6C76  23        INC    HL
27767 6C77  66        LD     H,(HL)
27768 6C78  6F        LD     L,A
27769 6C79  E5        PUSH   HL
27770 6C7A  D9        EXX
27771 6C7B  0D        DEC    C
27772 6C7C  D9        EXX
27773 6C7D  13        INC    DE
27774 6C7E  CDDC05    CALL   nn     05DC       get length #
27777 6C81  7D        LD     A,L
27778 6C82  C1        POP    BC
27779 6C83  02        LD     (BC),A
27780 6C84  03        INC    BC                save it in note table
```

Sound Cont.

```
27781 6C85  D9        EXX
27782 6C86  0D        DEC   C
27783 6C87  D9        EXX
27784 6C88  13        INC   DE
27785 6C89  C5        PUSH  BC
27786 6C8A  CD0327    CALL  nn    2703      get pitch #
27789 6C8D  C1        POP   BC
27790 6C8E  7D        LD    A,L
27791 6C8F  E60F      AND   n
27793 6C91  02        LD    (BC),A          save bottom nibble
27794 6C92  03        INC   BC              in note table
27795 6C93  29        ADD   HL,HL
27796 6C94  29        ADD   HL,HL           mult pitch by 16
27797 6C95  29        ADD   HL,HL
27798 6C96  29        ADD   HL,HL
27799 6C97  7C        LD    A,H
27800 6C98  E63F      AND   n
27802 6C9A  02        LD    (BC),A
27803 6C9B  03        INC   BC              save top part in note table
27804 6C9C  D9        EXX
27805 6C9D  0D        DEC   C
27806 6C9E  D9        EXX
27807 6C9F  13        INC   DE
27808 6CA0  C5        PUSH  BC
27809 6CA1  CD0C05    CALL  nn    05DC      get volume #
27812 6CA4  C1        POP   BC
27813 6CA5  7D        LD    A,L
27814 6CA6  02        LD    (BC),A
27815 6CA7  03        INC   BC              save in note table
27816 6CA8  E1        POP   HL
27817 6CA9  F1        POP   AF
27818 6CAA  E5        PUSH  HL
27819 6CAB  C618      ADD   A,n
27821 6CAD  6F        LD    L,A
27822 6CAE  2600      LD    H,n
27824 6CB0  34        INC   (HL)            update pointer (# of notes)
27825 6CB1  7E        LD    A,(HL)
27826 6CB2  FE0A      CP    n
27828 6CB4  2008      JR    NZ,e  6CBE      loop the buffer if more
27830 6CB6  3600      LD    (HL),n          than 10 notes
27832 6CB8  21D8FF    LD    HL,nn   FFD8
27835 6CBB  09        ADD   HL,BC
27836 6CBC  E5        PUSH  HL
27837 6CBD  C1        POP   BC
27838 6CBE  E1        POP   HL              save pointer to note table
27839 6CBF  71        LD    (HL),C
27840 6CC0  23        INC   HL
27841 6CC1  70        LD    (HL),B
27842 6CC2  C9        RET                   return
27843 6CC3  00        NOP
```

sound interupt routine (jumped to from zero page)

```
27844 6CC4  00        NOP
27845 6CC5  00        NOP
27846 6CC6  00        NOP
27847 6CC7  D5        PUSH  DE              save registers
27848 6CC8  E5        PUSH  HL
27849 6CC9  C5        PUSH  BC
27850 6CCA  F5        PUSH  AF
27851 6CCB  0603      LD    B,n             setup 380 registers
27853 6CCD  0E00      LD    C,n
27855 6CCF  61        LD    H,C
27856 6CD0  68        LD    L,B
27857 6CD1  29        ADD   HL,HL
27858 6CD2  2B        DEC   HL
27859 6CD3  E5        PUSH  HL              get pointer to current note
27860 6CD4  7E        LD    A,(HL)
27861 6CD5  23        INC   HL
27862 6CD6  66        LD    H,(HL)
```

```
27863 6CD7  6F        LD    L,A
27864 6CD8  AF        XOR   A
27865 6CD9  B6        OR    (HL)
27866 6CDA  D1        POP   DE
27867 6CDB  285E      JR    Z,e  6D3B
27869 6CDD  D5        PUSH  DE
27870 6CDE  23        INC   HL
27871 6CDF  CB7E      BIT   7,(HL)
27873 6CE1  2B        DEC   HL
27874 6CE2  2024      JR    NZ,e  6D08
27876 6CE4  E5        PUSH  HL
27877 6CE5  C5        PUSH  BC
27878 6CE6  05        DEC   B
27879 6CE7  78        LD    A,B
27880 6CE8  87        ADD   A,A
27881 6CE9  CB79      BIT   7,C
27883 6CEB  2801      JR    Z,e  6CEE
27885 6CED  3C        INC   A
27886 6CEE  0604      LD    B,n
27888 6CF0  CB27      SLA   A
27890 6CF2  10FC      DJNZ  e  6CF0
27892 6CF4  23        INC   HL
27893 6CF5  B6        OR    (HL)
27894 6CF6  F680      OR    n
27896 6CF8  D3FA      OUT   (n),A
27898 6CFA  CB79      BIT   7,C
27900 6CFC  C1        POP   BC
27901 6CFD  2008      JR    NZ,e  6D07
27903 6CFF  23        INC   HL
27904 6D00  7E        LD    A,(HL)
27905 6D01  D3FA      OUT   (n),A
27907 6D03  CBF9      SET   7,C
27909 6D05  18DE      JR    e  6CE5
27911 6D07  E1        POP   HL
27912 6D08  35        DEC   (HL)
27913 6D09  23        INC   HL
27914 6D0A  CBFE      SET   7,(HL)
27916 6D0C  23        INC   HL
27917 6D0D  23        INC   HL
27918 6D0E  23        INC   HL
27919 6D0F  D1        POP   DE
27920 6D10  2029      JR    NZ,e  6D3B
27922 6D12  D5        PUSH  DE
27923 6D13  C5        PUSH  BC
27924 6D14  05        DEC   B
27925 6D15  78        LD    A,B
27926 6D16  87        ADD   A,A
27927 6D17  3C        INC   A
27928 6D18  0604      LD    B,n
27930 6D1A  CB27      SLA   A
27932 6D1C  10FC      DJNZ  e  6D1A
27934 6D1E  F68F      OR    n
27936 6D20  D3FA      OUT   (n),A
27938 6D22  C1        POP   BC
27939 6D23  EB        EX    DE,HL
27940 6D24  78        LD    A,B
27941 6D25  C608      ADD   A,n
27943 6D27  6F        LD    L,A
27944 6D28  2600      LD    H,n
27946 6D2A  34        INC   (HL)
27947 6D2B  7E        LD    A,(HL)
27948 6D2C  FE0A      CP    n
27950 6D2E  2007      JR    NZ,e  6D37
27952 6D30  3600      LD    (HL),n
27954 6D32  21D8FF    LD    HL,nn    FFD8
27957 6D35  19        ADD   HL,DE
27958 6D36  EB        EX    DE,HL
27959 6D37  E1        POP   HL
27960 6D38  73        LD    (HL),E
27961 6D39  23        INC   HL
27962 6D3A  72        LD    (HL),D
27963 6D3B  1090      DJNZ  e  6CCD
27965 6D3D  CD23FD    CALL  nn    FD23
27968 6D40  F1        POP   AF
27969 6D41  C1        POP   BC
27970 6D42  E1        POP   HL
27971 6D43  D1        POP   DE
27972 6D44  ED45      RETN
```

Sound interrupt cont.

is a note being played?

no. then loop for next voice

has the note been sent to VDP?

no. then get data from note table
and send it out

decrement length counter

set flag that note has been sent

is note over (counter =0)?
no. then loop for next note

send out vol. of $F (off)
(its not obvious, but that is what it
does. see volume 1 about sound)

update # of notes

if > 10 loop note table pointer
to start of table

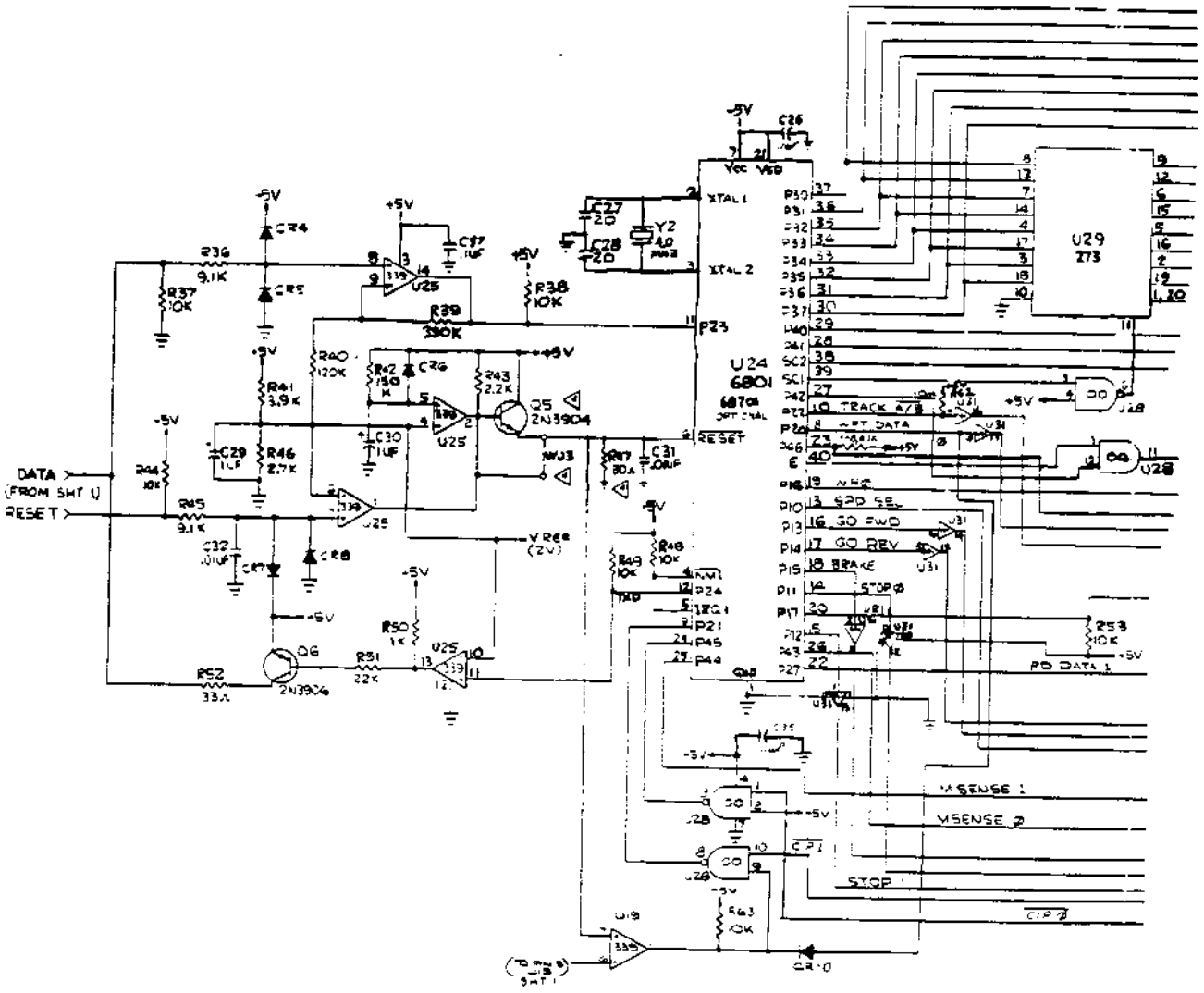
save pointers

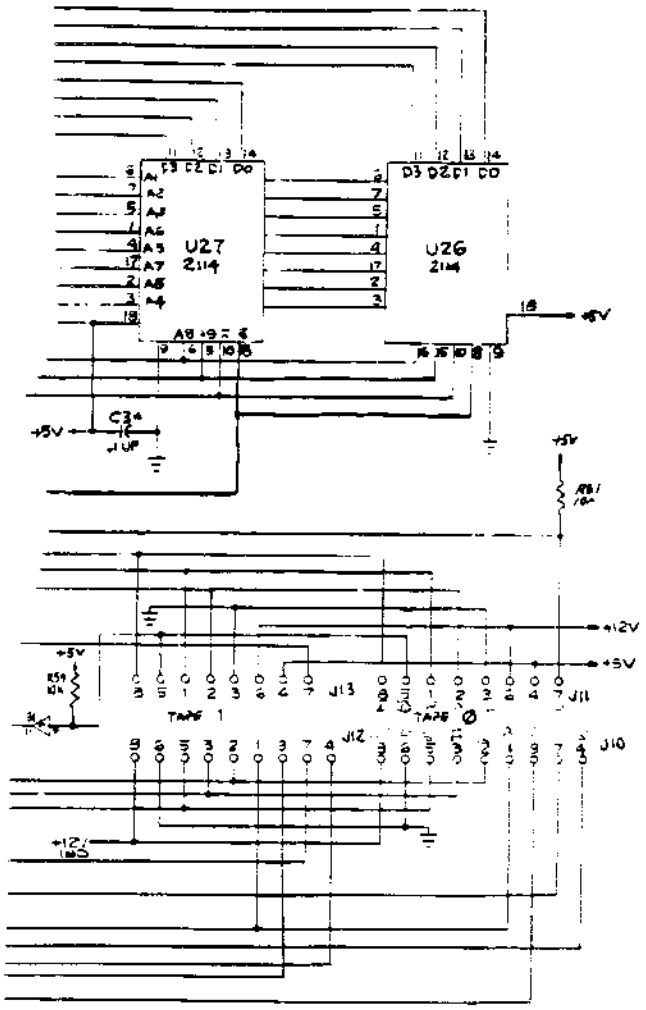
loop for next voice

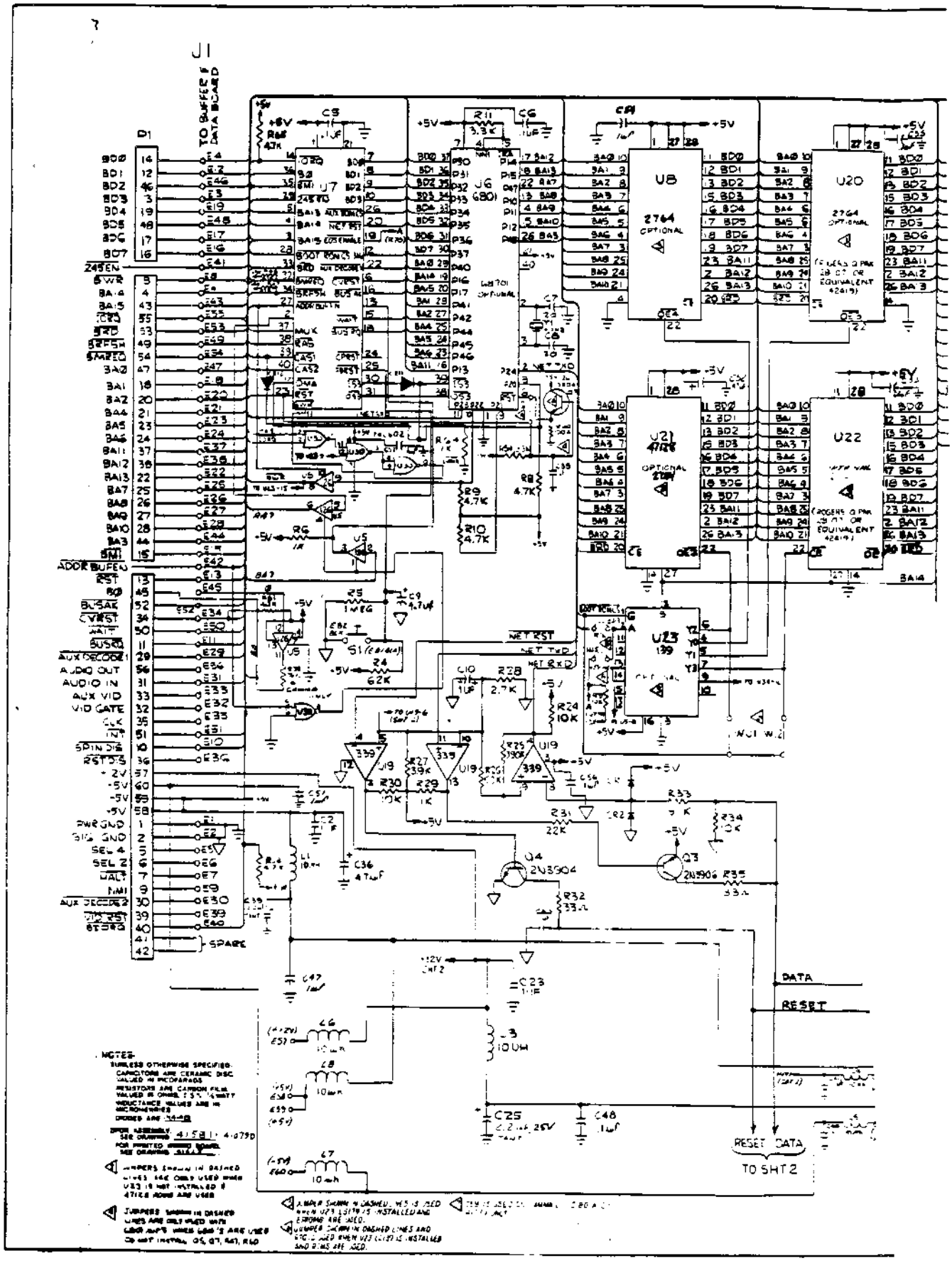
pop registers when all voices
are checked, and return

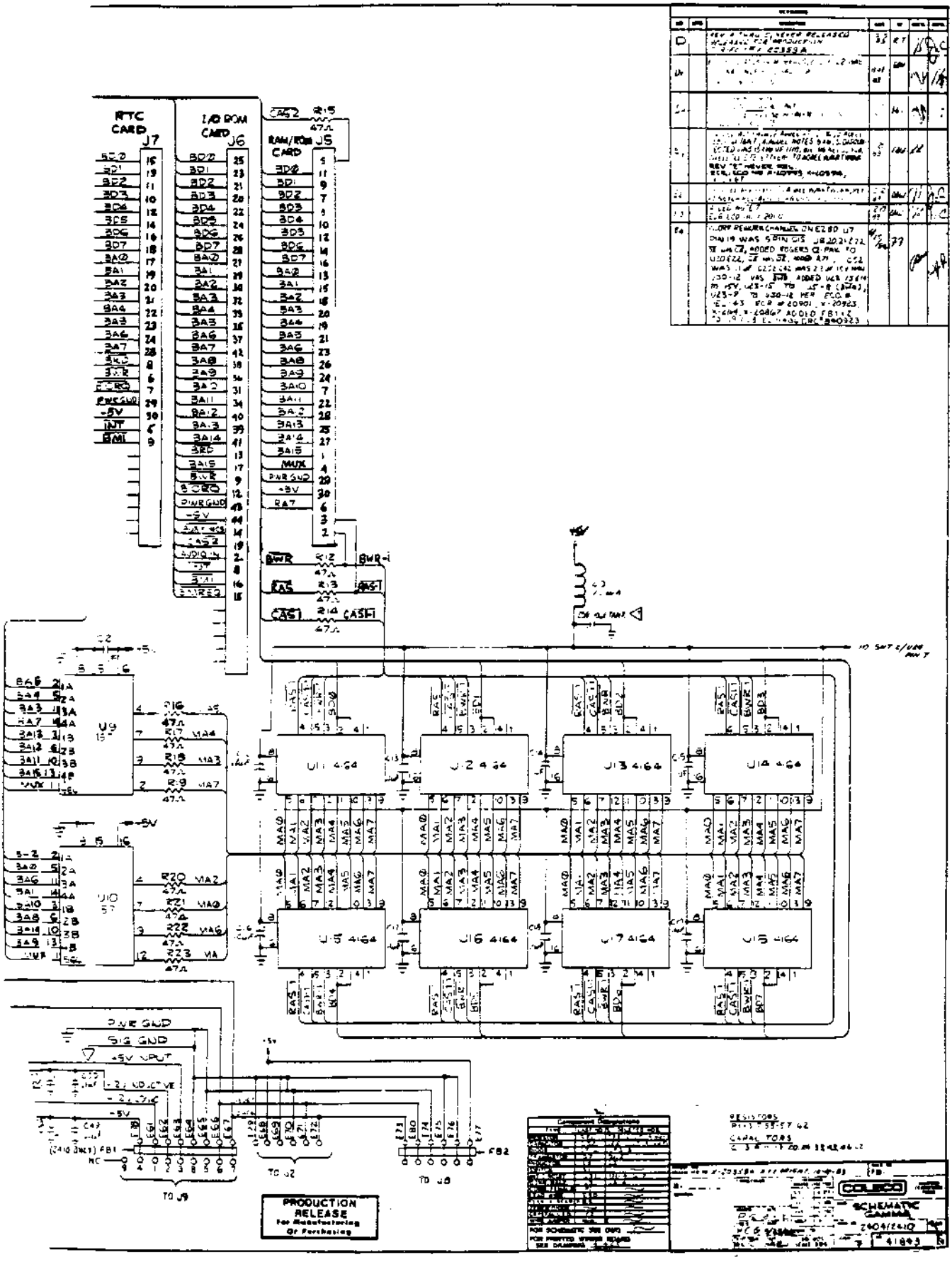
Appendix 3: Schematics

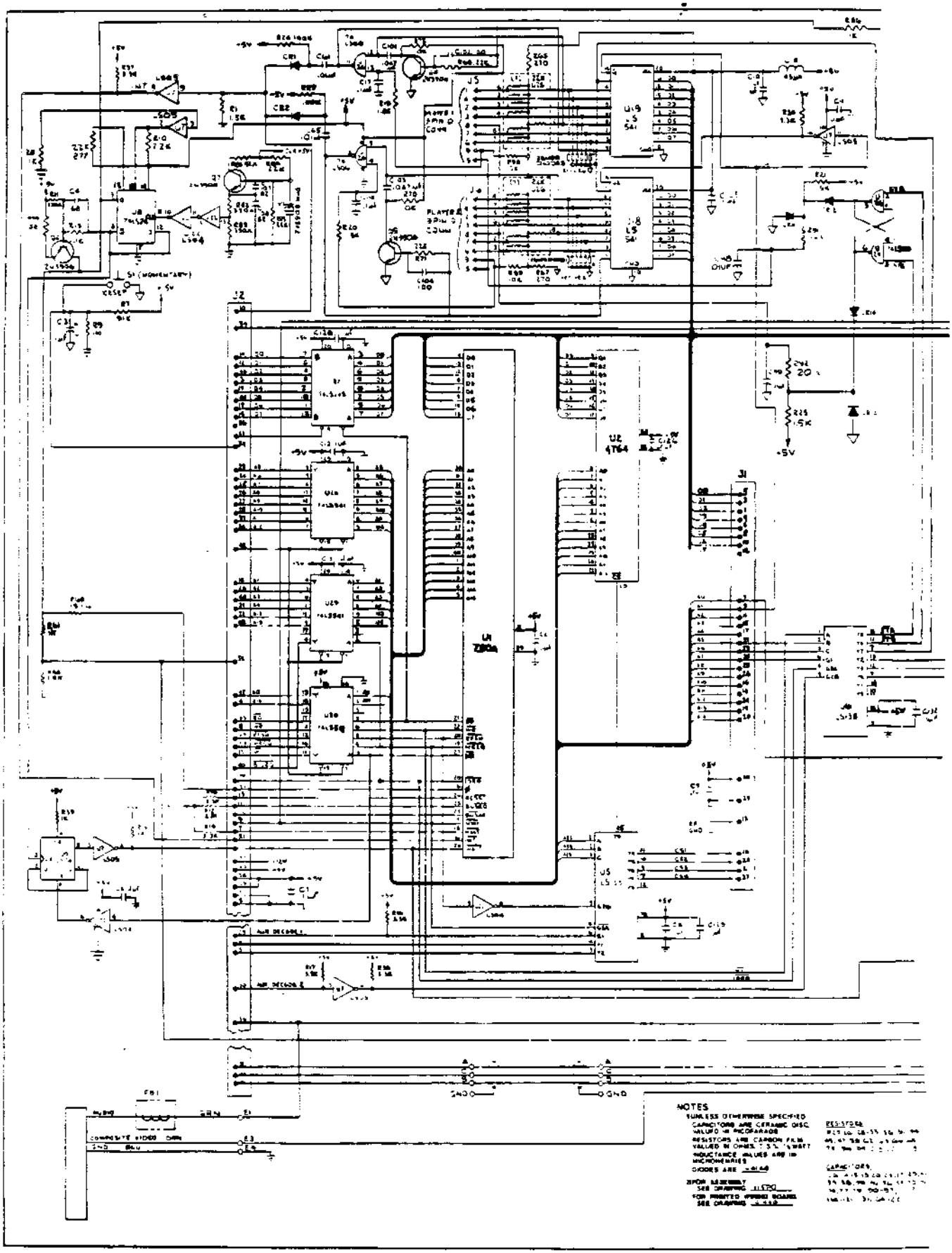
We received the following pages of scematics after we completed the first volume. Due to the many questions we were asked concerning them and the possibilities they have, we have reprinted them for any of you who are interested. But for those who don't know a chip from a DIP switch, this appendix can safely be ignored, because it is not crucial to understanding BASIC or the Adam. I also think that it is very unlikely that you will be able to get your Adam fixed when it goes, and suggest getting spare tape drives, power supply and keyboard for about $10 (and that's not hex) each, as listed by several surplus dealers.

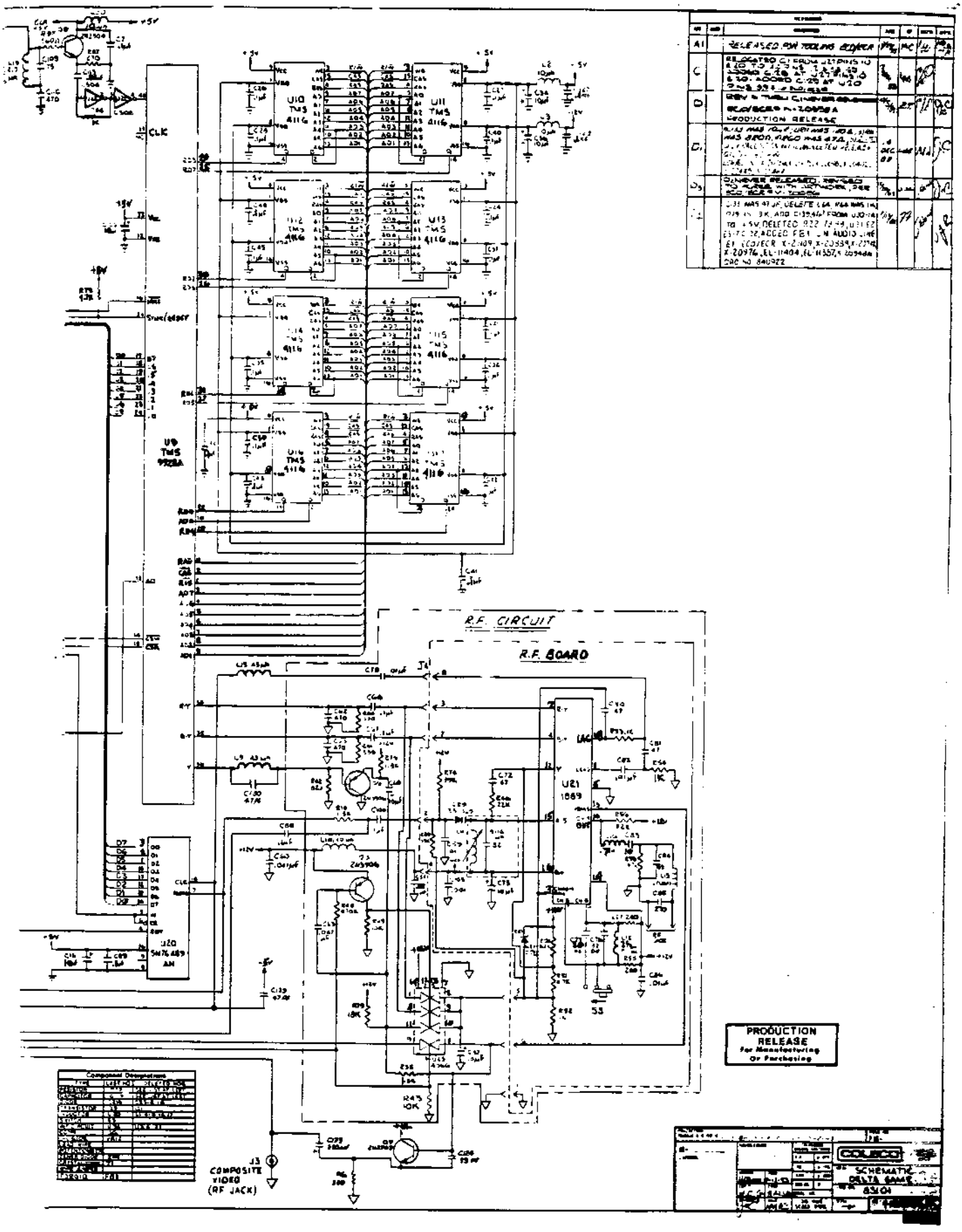



U27
2114
U26
2114
D3 D2 D1 D0
+5V
+12V
TAPE 1
TAPE 0
J13
J12
J11
J10


J1
TO BUFFER & DATA BOARD
P1
U8
2764
U20
2764
U21
4732
U22
U23
139
U7
U6
6801
U9
339
U19
339
Q4
2N3904
Q3
2N3906
DATA
RESET
RESET DATA
TO SHT 2
SPARE


RTC CARD J7
I/O ROM CARD J6
RAM/ROM CARD J5
U1 4164
U2 4164
U3 4164
U4 4164
U5 4164
U6 4164
U7 4164
U8 4164
U9
U10
PRODUCTION RELEASE
For Manufacturing Or Purchasing
SCHEMATIC CAMMAS




R.F. CIRCUIT
R.F. BOARD
PRODUCTION RELEASE
For Manufacturing Or Purchasing
J3 COMPOSITE VIDEO (RF JACK)
SCHEMATIC DELTA GAME 8304

Glossary

ASCII	American Standard Code for Information Interchange. It is a standard set of letters, characters or symbols that assigns each character a unique number represented by 7 bits (the top bit is 0). See the SmartBASIC manual.
bit	One binary digit. It can either be on (1) or off (0), depending upon its voltage. There are 8 bits in a byte, and 4 bits in a nibble.
block	A group of 1024 (1K) bytes. It is the basic unit used on the tape or disk.
boot	To load a program into RAM. The Operating System loads the first block of the tape or disk (the Boot), and executes it, loading whatever it is told to.
buffer	An area in RAM that is used by some routines to temporarily store data that will change. It is similar to tables, but tables usually don't change as much as buffers do.
byte	A group of 8 bits that represent a number from 0 to 255. A byte is often shown in its hexadecimal form.
Central loop	The routine in BASIC that oversees the input, translation and execution of commands.
Color table	The area in VRam that stores the color of each of the patterns in the Pattern table. (see vol. 1)

command	An order given to BASIC to perform a function. A command can be a part of a line. It has a Primary word followed by any crunch code needed to execute that function.
Command vector table	This table stores the vectors for the commands listed in the Primary word table. The token for each command is used to look up the vector for that command in this table.
CPU	Central processing unit, the microprocessor or Z80.
crunch code	A group of tokens or codes that represent the string you typed in. While a token is a single byte, crunch code can refer to many bytes.
Crunch code buffer	A buffer that stores the crunch code for the last line you typed in. If a line number exists for the line, then this buffer is copied to the Crunch code table.
Crunch code table	A table in RAM that stores the parsed line you typed in a form called crunch code. Lines in this table are part of a program, with each line number entry in the Line number table pointing to that line's crunch code in this table.
device	A piece of hardware that accepts commands from AdamNet to either write to or read data from some sort of storage system. The keyboard, tape and disk drives are examples of devices.
Execution loop	The routine that loops endlessly until the end of the Crunch code buffer or table, or the execution of the END or STOP command. It gets a token from crunch code and calls the vector in the Command vector table to execute the command.

floating point	A form of representing numbers by having a 4 byte mantissa and an exponent. It is often used for large numbers or numbers with a decimal point.
Floating Point Accumulator (FPA)	An area in RAM that is used to store a floating point number. It is like the Z80's Accumulator, because it is used for many calculations. On the Adam, the FPA can either hold a floating point number, or it can point to a string.
hexadecimal ($)	Notation that uses base 16 to represent a byte with the normal decimal digits and the letters A, B, C, D, E and F (A=10, and C=12, etc). Binary numbers can easily be shown in hex, because they take two hex digits.
keyword	A keyword is a primary, secondary or tape word. They are represented by tokens when a line is parsed into crunch code.
line	A line can refer to the screen or to everything typed before a return. It is usually the latter.
Line number table	An area in RAM that stores your program's line numbers in ascending order. The entries in this table also point to the line's crunch code in the Crunch code table.
macro	A string that is printed when you hit a key as if you had typed it in. Up to 30 macros can be stored by the HELLO program in chapter 11.
Name table	A table in VRAM that stores the offsets for the Pattern table. It reflects the current patterns displayed on the screen. In the TEXT mode, the name table stores ASCII codes, in HGR or GR, it is filled with 0 to FF repeating.

nibble	A group of either the upper four bits , or lower four bits of a byte.
Operating System (OS)	A group of subroutines that can be used in order to perform tedious or routine functions. Adam's Operating System is called EOS (E for Elementary or extended). It also uses OS-7 (which EOS is built on) for cartridges.
page	A group of 256 ($FF) bytes.
parse	"To resolve into elements". The process of replacing the ASCII codes of a line with tokens and crunch code.
Pattern table	A table in VRam that stores the patterns of the characters to be displayed on the screen in groups of 8 bytes.
pointer	Two bytes in RAM that point to a different memory location that stores some sort of data. A pointer can contain data in place of the other memory location. (Although it is then not a pointer).
primary word	A string of ASCII that can be found in the Primary word table. Primary words are the first words in a line or command. Typical primary words are: GOTO, TEXT, and IF
Primary word table	The table that stores the ASCII strings of all the primary words. The command's token and pointer to parse vector is stored with the string.

RAM	Random access memory, or, more accurately, read write memory (but RWM does not sound as good).
ROM	Read only memory.
secondary word	A string of ASCII that can be found in the Secondary word table. Secondary words usually are used after Primary words, and help in the syntax of the command. Typical secondary word are THEN, = and *.
Secondary word table	This table stores the ASCII strings of all the secondary words. Along with the string is the word's token, which is used to identify the word in crunch code.
shape table	A set of directions that BASIC follows in order to draw a shape in the hi-res screen. An unlimited number of shapes can be stored in a shape table.
sprite	A pattern defined by 8 or 32 bytes that can be moved on the screen by defining x and y coordinates.
Sprite name table	A table in VRam that has 32 entries. Each entry points to a pattern for that sprite, and the entry stores the color and coordinates for that sprite.
Sprite pattern table	A table in VRam that stores the pattern of each sprite. Each pattern takes up either 8 bytes (8x8 sprites) or 32 bytes (16x16 sprites). The bits in these bytes represent the pixels on the screen that are on or off.

stack	Reserved area of memory where the CPU stores two byte addresses or registers by Pushing to it, or Popping data off it. Data is stored in the first-in-last-out method in the downward growing table.
string	A group of ASCII characters that is normally preceded by the length of the string.
String space	The area in RAM where strings are stored either for a string variable or for temporary use.
table	An area in memory that groups similar data into one place. Though some tables may change, most remain the same and store the same data needed continually by a routine or routines.
Tape vector table	A table in RAM that stores the vectors of the tape commands in both the program mode and the immediate mode. Vectors are stored in the order used by the Tape word table.
tape word	A string of ASCII that can be found in the Tape word table. Typical tape words include SAVE, LOAD, and CATALOG.
Tape word table	This area in RAM stores the ASCII strings for each tape command. BASIC compares the command you type in with the strings in this table. It also holds the offset for the command's entry in the Tape vector table.
token	A number used to represent a primary, secondary or tape word. It helps speed up execution and saves memory by reducing the amount of space taken up to store the parsed line.

variable command	A type of command that is stored as a variable. They are used in equations and require parameters in parentheses. Typical variable commands include COS, LEFT and VPOS.
Variable table	The table in memory that stores all the variables that are used in a program. Each variable points to its definition, which is either a string or a number. Strings are stored in String space, and numbers are stored in the Variable value table.
Variable value table	A table in RAM that stores the current values for all the numeric variables in floating point format.
vector	A pointer in RAM that points to a routine that will be executed. A pointer only points to data, while a vector points to a routine.
VRAM	16K of RAM used by the Video Display Processor to store the tables for the screen and sprites. It is accessed by using I/O space on the Z80.