Vol. 1 / Number 3 June 1985

The Adam Technical Journal

In this issue:

BI MONTHLY NEWSLETTER

VOL 1 -NO. 3 6/85

CREATED FOR COLECO ADAM OWNERS

A TECHNICAL NEWSLETTER ON HOW TO OUT SMART "SMART BASIC"

SERENDIPITY PRODUCTIONS
P.O. BOX 07592
MILWAUKEE, WI 53207

INTRODUCTION

Well we are back. In the last two issues we covered a great deal of information about the video display processor. We covered reading and writing to the screen and sprites. Well its time to move on. However we'll come back to the VDP time to time to cover special topics such as the 40 column screen.

We will change the pace for a while in this issue and cover the SmartBasic interpreter. I guess some of you may not know what an interpreter is. Well its a way that you can write a program in an interactive way. That is, you can add a line of code and test run immediately without taking any time to change the code into machine language. The way the interpreter works is similar to trying to read a sentence in German and you can only read English. Every time you come to a word you have to look it up in the dictionary and get its translation. This is similar to an interpreter. Even if you are in a loop you have to keep looking up the words. You may ask why doesn't the code get put into machine language on the first pass through the program. This would be called a compiler and the problem is that it takes a great deal of time to change the code. Not only this but usually the code that is generated takes a substantially larger amount of memory.

The interpreter is the ideal way to let the user write large programs and have an interactive and user friendly system. What you sacrifice is speed of execution.

In this issue we will hopfully give an understandable explanation of the interpreter and some very useful routines to help you in your programing.

DISECTING THE SmartBasic INTERPRETER

In this issue we will look at how the commands we type in during program entry are understood and later, acted upon at run time. The first order of business is examining the PRIMARY COMMAND TABLE which will in turn lead us directly to several other tables. The trail ends (for this issue anyway) at the SECONDARY RESERVED WORD TABLE. We will use all of this knowledge to add our new command. When we are done with this, we'll be in perfect position to understand the format of the finished program as it resides in RAM, ready to run. I really think you are going to enjoy this!

SmartBasic PRIMARY COMMAND TABLE

Whenever a new line of a Basic program is being read by the interpreter, whether you are typing it in on the keyboard or a program is executing, SmartBasic is searching for one of 64 primary reserved words. These are words such as GOTO, FOR, NEXT, IF, etc. All of these are located in a table starting at address 272 ($110) along with the necessary information needed by the interpreter to act

upon these words. Let's examine this table in detail since it is close to the very heart of the interpreter (the actual "heart" of the interpreter is the code which uses this table). Once we understand what's in this table, we can find out almost anything else we wish to know!!

The Primary Command table resides from 272-816. The format for each reserved word in the table is as follows:

Byte	Purpose
1	Crunch code token
2-3	Two byte address for parse routine table
4	Length of the reserved word (# of characters)
5-x	Actual reserved word (i.e. GOTO)

The first byte is simply the word's position in the table. The word GOTO is the third entry in the table so the first byte of its entry is a 3. Since this number uniquely identifies the reserved word, it is used in a couple of important ways. First, it is used to represent the reserved word in the actual program in RAM and is called a token. The program doesn't contain the string "GOTO" which takes 4 bytes but instead only contains the value 3 using only 1 byte. For this reason the program in RAM which contains tokens and their associated parameters is called crunch code since it is a condensed numerical code representing English commands which have more meaning to you and me.

The second important function of the crunch code is that it indirectly points to the location of the routine that actually performs the command. There is a table at 6421($1915) containing 64 two-byte addresses. Each entry in the table is the location of a command routine and is arranged in the same order as the primary reserved word table. Therefore, the third two-byte entry in the table is the location of the GOTO routine.

The second and third bytes in the primary command table form the address which is used when accepting a command from the keyboard. When you type the word GOTO the interpreter looks it up in the command table and writes the crunch code found there (byte #1) either into the program in RAM if you are writing a program (i.e. line starts with a number) or to a special temporary buffer for immediate execution at the conclusion of the line. It then goes to the address in bytes 2 and 3 to find out what else has to follow the word GOTO which, in this case, is a line number.

The process of examining a line of keyboard input and acting upon each word is called parsing so we'll call this address the parse address which points into the COMMAND PARSING TABLE. This table resides at 937 through 1055 ($3A9-$41F).

The process of parsing a long program line with multiple statements can be broken up into many small tasks which may or may not be common to other commands. Each task, such as looking for a comma, or a semicolon, or a variable name, is performed by a separate routine. The command parser table lists how many tasks there are and where to find the routine that does them. The GOTO command only requires one task, to accept a line # between 1 and 65535. Its entry in the parser table is a 1 (for one task) followed by the two byte address of the routine which does this. The GOSUB command points to the exact same parser table entry since it requires the same parameter. Commands which require no parameters, such as STOP, END, HOME, CLEAR, etc. have a parser table entry of 0.

The 4th byte in the command table

entry holds the length of the command name (i.e. 4 for GOTO). Starting at the fifth byte is the actual command name.

Confused? Let's summarize how the interpreter accepts, then executes, a line of BASIC containing the GOTO command. First, you type a line on the keyboard followed by the return key. The Parser examines the line. If the first thing typed was a number it knows you are entering a program so its going to write to RAM program area not to the immediate buffer. Then it tries to match the next word (excluding any spaces) in the typed line against those in the primary command table. It finds a match for GOTO and copies its crunch code from the table into the program being built in RAM. It then uses the parse address to get whatever parameters are needed. The entry in the parse table says there is only one task to complete; find the line # to GOTO and record it in the program. This process is repeated every time you type something on the keyboard.

When the program (or the immediate buffer) is executed, the interpreter picks up the crunch code. It calculates the address of the necessary routine (twice the crunch code + 6421) and jumps to it. The routine is responsible for going back to pick up any needed parameters. Simple, huh?!

SECONDARY RESERVED
WORD TABLE

Between the primary command and parsing tables are some of the secondary reserved words such as the math signs (+,-,<,>,etc) logic symbols (AND, OR, NOT) plus anything normally following FOR, IF or PRINT statements (TO, STEP, THEN, etc). They are secondary because they are used only in the processing of a primary command. Two exceptions are GOSUB and GOTO which can follow a secondary word (such as THEN) as well as being primary commands. Since they are used in support of other commands, the only information needed for their use is a unique crunch code for each. The format of the table entries is very simple; the crunch code (byte #1) followed by the length of the keyword (byte #2), followed by the word itself. The table resides from 818 through 936 ($0332-$03A9).

Table I lists these words along with their token values. Actually, there are several other secondary reserved words which are not in this table. Looking at the way the code was written, one can surmise that the only reason all the secondary words aren't in one table is that the code was written in pieces of increasing complexity and certain capabilities were done very late in the process. Looking at the addresses in table II, we see that all the execute routines are in lower memory between 6000 and 12000. One can imagine developing the interpreter as a series of steps. It is very easy to write a line of tokenized Basic by hand since it is so compact, so it seems natural to start by writing the execute routines to run on simple handwritten lines. Then the parse routines which build the tokenized code were added between 14800 and 16000.

There is a table of 27 secondary words not found in the secondary word table described above. These include all the math functions (SQR, TAN, SIN, LOG, etc), the string functions (MID$, LEN, LEFT$, RIGHT$, etc) plus a variety of others such as PEEK, USR, VPOS, etc. The location of the table is very strange. It floats above LOMEM between the variable name table and the string variable space (described later). Its actual location is stored in the Basic parameter table at 16097-16098 (more about this table later). We'll call this table the MATH WORD TABLE (we love naming tables if you haven't noticed yet). The format is very simple. Byte #1 is the length of each word followed by the word itself. That's all!

Table II lists the contents of the Primary Table in an easy to use format. This table was generated by CMD.LIST (see listing #1) which is

provided so that you can trace through the logic and/or modify it to meet your own needs. Note that there are a few commands in the table which are not listed in the SmartBasic manual such as BREAK, NOBREAK and '&'. BREAK and NOBREAK enable and disable the use of control-C to stop execution of a Basic program.

The '&' command is very interesting. Applesoft Basic (after which SmartBasic is modeled) is famous for its use of the as a 'hook' through which the user can integrate his/her own machine language routines into the Basic language. Although the USR and CALL commands allow you to add your own functions, they provide no mechanism to easily pass parameters to the called routine, especially text parameters. The '&' causes control to be passed via an indirect jump to the address listed in a certain position in the interpreter which the user can set. That is similar to a CALL except for one important feature. After the CALL the interpreter continues to parse the rest of the line but the '&' doesn't. It is stored just like the REM statement (compare the parse address in Table II). This allows parameters to be passed to the user's routine. Huge libraries have been written and published based on the Applesoft '&' vector. We can only assume that SmartBasic left the door open to similar use since the command exists. The execution code, however, appears only to recondition the stack which is a useless operation since stack overflows are detected by SmartBasic and are corrected by reconditioning anyway (without use of the code at 10164). However, since all of Basic is in RAM (Applesoft is in ROM), the '&' command can easily be changed to do what we want whether or not that was the original intent.

Also notice that entry #1 is blank and has the same parse and execute addresses as the LET command. This reflects the fact that "LET a=1" and "a=1" are equivalent statements. The same holds true for PRINT and '?'.

There are several commands which have no parse routines listed in Table II. These require no parameters and can be performed by CALLing the execute address. This makes it easy to perform several functions easily from a machine language routine. More importantly, it will allow us to perform functions after their names have been replaced in the Primary Command table. We will change NOBREAK to a different command later in this issue. But we can still perform the NOBREAK command with a simple CALL to 6351 even though it no longer exists as a reserved word.

A NEW COMMAND FOR SMARTBASIC

In the first issue we mentioned that the utilities could easily be appended to an existing program. But if your program already has several data statements, you have to make provisions for jumping over all previous DATA values before calling the setup routine (GOSUB 51000). Otherwise the loop in line 51000 will READ the wrong data for the machine language routine. We didn't highlight this irritating situation in the first issue because we intend to resolve it in this one. We are going to add a new command to SmartBasic which will be very useful to those of you who use DATA statements often in your programs. This little project will accomplish two things. First it adds a useful new feature to SmartBasic and, second, it allows us to use the knowledge gained in the last section.

SmartBasic allows you to store DATA within your program, both numbers and text strings, which can be assigned to a variable using the READ statement. The READ statement simply picks up values in a sequential fashion regardless of where the DATA statements are in your program. This is simple enough if you are using each data value only once and in a known sequence. If the order in which the data will be used can change during program execution, such as when the program

logic branches based upon the user's response to prompts, then the programer must insert dummy READs to skip over the unused values.

Some programs use the same data values repeatedly requiring use of the RESTORE statement which tells the READ to start at the first DATA value again. A large program such as a disassembler or an assembler will typically use 150 or more data values and must jump around these depending on the code being processed. This not only persents a headache for the programer (since he/she must know how many dummy READs to perform to get to a particular set of values) but it also slows down the program while 140 values are read to get at the 141st. Wouldn't it be nice if we could do a RESTORE to a specific line number instead of always going back to the beginning? Of course it would, so we're going to do it!

The program to make the modifications described below is given as Listing #2 at the end of the journal. It should be typed in and SAVEd as HELLO (capital letters are mandatory) on the SmartBasic tape you use to boot the system. The last step in the boot process is to search for this file name. If found, it will run it automatically making it the ideal vehicle for installing patches to SmartBasic. We will be using this again in the future.

We want to make a new SmartBasic command. Since the table where all the statement names are listed is of a fixed size, surrounded on both sides by code, we can't just add to it. So we'll have to cannibalize an existing command that, hopefully, isn't used much. As discussed in the last section, NOBREAK is a great candidate.

The first step is to change the statement name. This is done by poking the ASCII representation of the letters of our new name into memory locations 800-806 which now hold the letters NOBREAK. This is done by lines 20-40 in the HELLO program. We've called our new command RSTOREL (for RESTORE RELATIVE) which has the exact same number of characters as the command we're replacing. If we had entered a program containing a NOBREAK before making this change, we could now list it and find the RSTOREL in its place. The program would operate normally since we have only changed the name not the function of the command.

As we discussed before, the second piece of data for each command is where to find the routine which parses the name into the token stored in RAM. NOBREAK needs no parameters

and its parse address in Table II is the same as all the other parameterless commands. But we need a line number for RSTOREL just like GOSUB and GOTO so line 50 provides a reference to the same parsing routine as they do.

The last change we need to make is the routine used to execute the actual command. As discussed in the previous section, the crunch code is used as an offset into a table of routine addresses. The code is doubled (since every address in the table is two bytes) and added to 6421 ($1915). This final number is the location in memory where we can find the address of the command routine to be executed. Line 55 makes this change for us.

Now, to create a routine to actually perform our relative restore, we must know how the READ and the original RESTORE commands work. There are three pointers which contain all the information needed by the READ command. At 16117 ($3EF5) the current position in the line number table is stored. The current position within that Basic program line is stored in 16119 ($3EF9) and 16121 ($3EFB) holds the number of bytes left in the current line. When a READ is issued, these pointers are used to search for the next data value. If it isn't already at a data value, the code

will march through the program, line by line, until data is found. This can really slow down a large program!

What we really want to do is skip the sequential search and jump right to a particular line. Since we are going to do this using existing SmartBasic code, we will dispense with a detailed explanation. I realize this won't make our more curious readers very happy but we haven't covered nearly enough of the fundamentals of how the interpreter works to do the subject justice. The gist of the code is simply to copy a portion of the GOTO code which searches the line number table to make sure the target line number exists and to get the address in RAM where its actual tokenized code starts. Then we modify the three READ pointers. The next time a READ is issued, it will start looking for data where we told it to, not at the beginning of the program.

We obviously need a place to hide the new code where it will be safe from being overwritten by a large program or by changing LOMEM or HIMEM. There is an area of the interpreter which holds the text for the Coleco banner displayed at boot time. Since the patch is loaded after the logo is displayed, we could use this space. Better yet, there are an additional 33 bytes of text following the banner which contains a copyright statement which is never displayed. This is where line number 120-140 places the GOTO code plus the pointer adjusting code.

WARNING !!

To use the modified Basic, type in Listing #2 and save it as anything but HELLO. The reason for this is that HELLO will be run at bootup even if you made a typo which causes SmartBasic to lock up. Only after you are confident that the patch works as its supposed to should you rename the file to HELLO. Now RUN the file and debug it.

To use the RSTOREL command, just type it in as any other command followed by a legal line number where you want the next READ to start. At run-time it will start searching for a value at the beginning of that line regardless if there is a DATA statement there or not. Entering a non-existant line number will cause the same error as would GOTO or GOSUB (another great reason for using existing code whereever possible, the error handling is done for us). If you attempt to LOAD a program containing a RSTOREL before the patch is applied, the program will load normally without any errors but all the lines referencing the new command will be gone, so be careful.

By the way, even though we removed the NOBREAK name, we can still use it. Just substitute a CALL 6351 whereever you would place the NOBREAK.

Actually, we haven't added a new command to SmartBasic, we've just traded a seldom used command in on a more useful one. In order to truly add commands we must use one of the positions in the PRIMARY COMMAND TABLE as a gateway to several other commands. This can be done easily using the '&' command described earlier. The '&' code could be modified to jump to our own set of execution routines. Which routine to run would be determined by what followed the '&'. Since the length of a line is limited to 128 bytes during typing, there is room for alot of information. The possibilities seem endless!

BASIC FAST LOADER

The starter Adam system is a very good value in terms of performance per dollar. Part of the reason is the high speed tape drive. It is many times faster than the usual cassette players used on other starter systems but is still reasonably economical. But if you use your system mostly for programing in SmartBasic, much of that speed has been compromised in order to make the most of the SmartWriter software. Most other

microcomputers save a basic program in its tokenized form just the way it sits in RAM, ready to run. Of course doing that precludes the use of SmartWriter to edit and write basic programs so instead, Adam saves the actual text that you see when you LIST the file. The price you pay for that flexibility is loading speed.

When you load a SmartBasic program, the interpreter must convert the text into tokenized code, just as if you were typing it all in again.

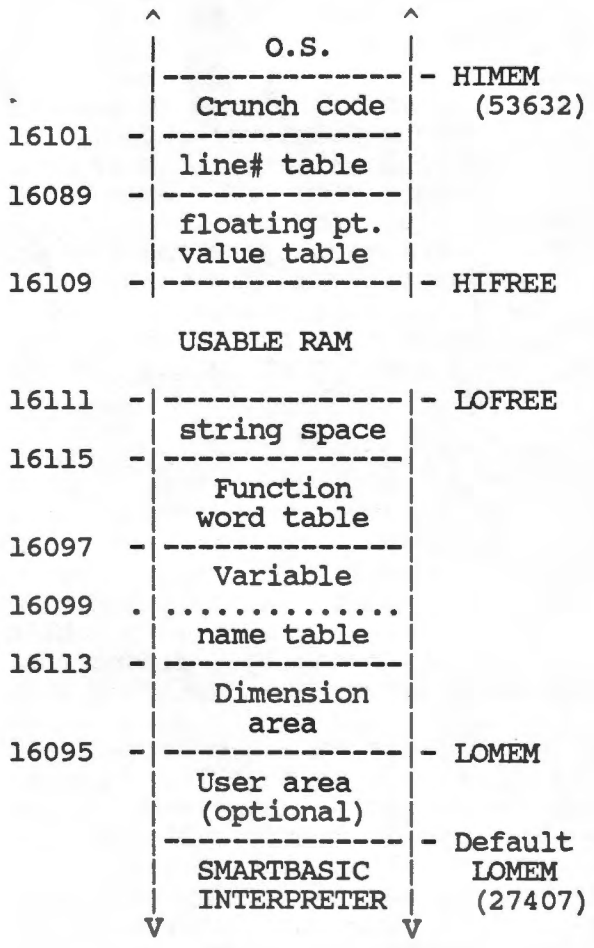
That can take alot of time. When you type in a new line in a long program or edit an existing one, you can actually see the time delay before the cursor comes back. Thats pretty amazing when you consider that the CPU is running at 3 MHz. You have to execute an awful lot of machine code to produce a perceptable delay (about 35,000 instructions for an 1/8 second delay). This is why the tape pauses while reading a long program. It reads a block (1024 characters) then stops the tape while it tokenizes what it read, then starts reading again.

The largest program we have is a 24 block long assembler which takes 266 seconds (4 min 26 sec) to read excluding the time it takes to locate it on the tape. Without having to tokenize anything, the Adam can read 24 blocks in 36 seconds (1.5 sec/block) using BLOAD. Wouldn't it be nice to read a basic program from tape 7 times faster than normal? Ok, so lets do it while we learn more about the way the interpreter works!

First of all, we can't just BSAVE the tokenized program because all the variables are stored separate from the code. Besides, since you can change HIMEM and LOMEM anytime you like, the interpreter must be saving various critical pieces of information somewhere special so it can find the program. We'll call this special place the BASIC PARAMETER TABLE, or parm table for short.

The parm table is in the middle of a rather large data area which serves a variety of purposes but we are interested in one small piece. The 28 bytes from 16089 through 16116 that contain all the information necessary to fully map where a basic program resides.

To understand why we need these 28 bytes, we need to know how a program is set up. Below is a pictorial map of a SmartBasic program. Along the left side are the addresses in the parm table which define the boundaries between the various areas of the map.



A multitude of things happen when you enter a line of SmartBasic. First, the keyboard input is parsed into tokens as discussed above. Then the tokenized line is added to

the code being built below HIMEM. The lines are stored in chronological order with the first line on top and the newest line on the bottom, not in order of the line numbers.

Below the crunch code is the line number table which is made up of 4-byte entries; two for the line number and two for the starting address of the crunch code for that line. These entries are in line number order (lowest at bottom, highest on top). This makes it easier to search through the program sequentially, such as when executing a GOTO.

Below this are the floating point variable values (variables not ending with % or $). Only the 5-byte values are stored here. The variable names are stored in another table with pointers to the appropriate locations in this one. As lines are typed in, each area discussed so far grows downward using up RAM as it goes and its all kept straight via the parm table. The value at 16109 marks the end of these tables (or the start of free RAM) so we named that value HIFREE for future reference.

At the other end of memory we have the SmartBasic interpreter. LOMEM defaults to the very top of this code, although you can set it anywhere you like even during program execution. Above this is a 145-byte temporary area with 5-byte entries for managing dimensions, temporary values, etc.

Above this is a two part variable name table. This is followed by the Math Word Table described in the previous sections and, finally, by the string value storage area where the text values for string constants and variables are stored. The highest address in the string value table marks the bottom of the usable free RAM so we'll call it LOFREE.

We will go into more detail on these tables in subsequent issues. Right now we need to understand that as variables are used in either a program or in commands typed on the keyboard, these tables grow upwards in memory. With this arrangement, the program grows both from the top down and from the bottom up. The FRE(0) command simply tells you how much free RAM is left (HIFREE - LOFREE).

By now it must sound like saving a tokenized program is going to be very difficult but its not. We just need a way to save the two halves of the program (top and bottom) plus the critical 28 bytes of the parm table which describe where it all goes. The program CRUNCHER does just this.

CRUNCHER

The CRUNCHER program is given in listing #3. It is made up of 3 pieces essentially. The first 105 bytes does all the important stuff. The very first task is to compress all the variable tables by issuing the CLEAR command. It does this by calling the CLEAR execution routine as listed in Table II. It then obtains the values of HIFREE, LOFREE, and LOMEM.

It moves the block of data between LOMEM and LOFREE up under HIFREE so that the program is now one big piece. Next it copies the Basic parms up immediately under the program. (Actually, it copies 30 bytes instead of just 28 because we like nice round numbers).

Now we could BSAVE this piece to tape but then we would need a second program to uncrunch this file and put the parms back so it would be a runable Basic program. Plus we would have to tell the second program what the starting address was since every program is of different length, and we would have to remember it everytime this file was BLOADed. Certainly not an elegant solution.

We get around this by adding an uncrunching routine (the third part of CRUNCHER) immediately under the parms. Now we're ready to BSAVE the whole thing as a binary file, but to

do this we need to know the starting address for the BSAVE command. Well, register pair DE contains the address just below the start of our file as a consequence of the block load command used to move all this stuff around. We can calculate the file length by subtracting this address from 53632 (HIMEM). All we need now is a way to communicate these values to you so you can use this in the BSAVE command.

Well, lets see. We know a bit about how the tokenized code is put together and we know the address of the PRINT code from Table II. So why not PRINT the entire BSAVE command to the screen so all we have to do is run the cursor over it while supplying the file name and it's done for us!!

Part 2 of cruncher is just a line of crunch code to print the BSAVE command. The last task performed by part 1 of CRUNCHER is to POKE the file length and start address into this tokenized code line and call the PRINT routine.

Let's quickly review the whole process again to make sure we understand it clearly. Only the first 105 bytes of CRUNCER actually run. The next 45 bytes are the tokenized PRINT command which is read by the PRINT code and produces the BSAVE command for you to execute. The last 57 bytes is the UNCRUNCHER routine that is saved with the file for execution at load time.

So we have a Basic program in memory that we want to load fast. Just type "BRUN CRUNCHER". First it issues a CLEAR, then copies all the tables up under HIFREE eliminating the free space in RAM. Then it copies the Basic parms under that, followed by the UNCRUNCHER routine copied under the parms. The last step is to print the BSAVE command on the screen with a default file name and the appropriate length and address. You must now run the cursor over the command line, changing the name as you go and the tokenized program is saved.

To load the program, just BRUN the saved file. It will BLOAD the file and execute the UNCRUNCHER routine. By the way, the BRUN command will always cause an error message to be displayed which insists that you have used an illegal O.S. command. This is another SmartBasic bug. The BRUN command works in spite of the message.

The UNCRUNCHER routine first establishes where in memory it is so it can use absolute addresses to load with. It does this by CALLing a location in high memory which is just a RETURN instruction. A machine language CALL causes the current address to be PUSHed onto the stack so it knows where to return to (which it does immediately in this case). We decrement the stack pointer twice and POP that address into HL. Now we know the address of the first instruction after the CALL so we can calculate any other location in our routine.

The next step is to copy the parms into the SmartBasic parms table. Now we must copy the tables down to LOMEM but for a large program this may overwrite the UNCRUNCHER routine causing total disaster. The way to avoid this problem, is to relocate the remaining routine to a safe place such as the trusty 56320 (more about why we chose this address at the end of this section). From the new location we can safely move the tables to LOMEM and we're done!

We've glossed over the assembly routine fairly quickly because of its length. If you aren't familiar with Z80 code but want to understand it we suggest buying a book on the subject such as Rodnay Zaks' "How to Program the Z80" (Radio Shack Cat. No. 62-2066). The listing is heavily commented to make it easier to follow.

To enter the program, first POKE a 255 into 16149 and 16150 so that the POKE command will work above HIMEM. The most painless way to enter this rather lengthy program is to type the following immediate command:

```
FOR n=0 TO 206: INPUT da: POKE
56320+n,da: NEXT
```

Now you will be prompted for all 207 values. Just type the decimal values from listing #3 following each with a return. If you get a SmartBasic prompt before you are done or if you're done before it shows up, you know an error was made either by skipping or adding data.

Now use another loop to list the data such as:

```
FOR n=0 TO 206: PRINT PEEK(n);",";
:NEXT
```

Compare the displayed data against the listing. If there are no changes, save it by typing:

```
BSAVE CRUNCHER,a56320,1207.
```

We use the area of memory above 56320 quite a bit. It is a 1K buffer used by the operating system to read from, and write to, a second tape or disk file. The first file opened uses the 1K buffer immediately below 56320. Therefore, as long as we only have one file open at a time this provides a rather large, fairly safe area of memory to work with.

Table I

Secondary Word Table entries in order of occurance in table

Code	Word	Code	Word	Code	Word
160	+	170	=	180	TO
161	-	171	AND	181	:
162	*	172	OR	182	#
163	/	173	NOT	183	(
164	^	174	GOTO	184	)
165	<	175	GOSUB	185	,
166	>	176	STEP	186	;
167	<=	177	AT	167	=<
168	>=	178	THEN	168	=>
169	<>	179	THEN	169	><

Table II

Primary Command Table entries with associated addresses

#	Command	Exec.	Parse Routines			
1		6247	15020			
2	GOSUB	8427	15756			
3	GOTO	8342	15756			
4	INPUT	8957	15543			
5	LET	6247	15020			
6	NEXT	8811	15567			
7	PRINT	7854	15580			
8	READ	9499	15574			
9	REM	8419	15817			
10	FOR	8557	14991	15963	14875	15093
11	IF	7705	14947	15035		
12	DATA	8419	15814			
13	DIM	6942	15574			
14	ON	8381	14875	15209		
15	ONERR	8114	15991	15756		
16	STOP	6378				
17	RETURN	8477				
18	END	6047				
19	DEF	8244	15125			
20	CLEAR	8141				
21	RESUME	8313				
22	NEW	6356				
23	POP	8493				
24	RUN	6159	15232			
25	LIST	7407	15243			
26	TRACE	6336				
27	NOTRACE	6341				
28	DEL	7555	15247			
29	CALL	10042	14875			
30	CONT	6387				
31	CLRERR	8109				
32	GET	9378	15364			
33	POKE	10104	14875	15939	14875	
34	RESTORE	9482				
35	HOME	11090				
36	DRAW	11358	14875	14976		
37	XDRAW	11412	14875	14976		
38	FLASH	11050				
39	INVERSE	11055				
40	NORMAL	11060				
41	TEXT	11065				
42	GR	11070				
43	HGR	11075				
44	HGR2	11080				
45	HLIN	11170 14875	14875	15939	14875	15977
46	VLIN	11219 14875	14875	15939	14875	15977
47	HPLOT	11487	15102			
48	PLOT	11139	14875	15939	14875	
49	HTAB	11320	14875			
50	VTAB	11330	14875			
51	SHLOAD	11085				
52	RECALL	11764	15364			
53	STORE	11756	15364			
54	WAIT	10126	14875	15939	14875	14969
55	SPEED	10832	15926	14875		
56	ROT	11459	15926	14875		
57	SCALE	11473	15926	14875		
58	COLOR	11099	15926	14875		
59	HCOLOR	11119	15926	14875		
60	IN	12084	15950	14875		
61	PR	12058	15950	14875		
62	HIMEM	11010	15911	14875		
63	LOMEM	10870	15911	14875		
64	BREAK	6346				
65	NOBREAK	6351				
7	?	7854	15580			
66	&	10164	15817			

Listing #1

```
  5 PR #1
 10 REM
 11 REM   LIST.CMDS  reads the Basic command table at 256 and
 12 REM   calculates the address of the command handler.  The
 13 REM   output lists the command number (ie crunch code), the
 14 REM   command itself, the address of the execute code, and
 15 REM   the various parsing routine addresses used at entry time
 16 REM
 17 REM   ****  Serendipity Productions 10/27/84  ******
 18 REM
 40 ad = 272
 60 PRINT "   #   Command     Exec.     Parse Routines"
 70 PRINT "  --   --------    -----    ----------------------"
 80 PRINT
100 cm$ = ""
102 ix = 6421+2*(PEEK(ad))
104 rt = PEEK(ix)+256*PEEK(ix+1)
106 nu = PEEK(ad)
108 po = PEEK(ad+1)+256*PEEK(ad+2): ad = ad+3
109 pc = PEEK(po)
110 cc = PEEK(ad)
115 IF cc = 0 THEN  GOTO 145
120 FOR i = 1 TO cc
130 ad = ad+1
140 cm$ = cm$+CHR$(PEEK(ad)): NEXT
145 rt$ = STR$(rt)
150 PRINT "   "; nu; TAB(8); cm$; TAB(20); rt; SPC(6-LEN(rt$));
160 IF pc = 0 THEN  GOTO 210
170 po = po+1
180 FOR i = 0 TO pc-1
185 ad$ = STR$(PEEK(po+(2*i))+256*PEEK(po+(2*i)+1))
190 PRINT SPC(9-(LEN(ad$))); ad$;
200 NEXT
210 PRINT
220 ad = ad+1
230 IF nu = 55 THEN  FOR n = 1 TO 8: PRINT: NEXT: GOTO 60
240 IF ad < 800 GOTO 100
250 PR #0: END
```

]

Listing #2

```
  5 REM       RESTOR-REL changes the NOBREAK command into a
  6 REM       relative data RESTORE command which allows
  7 REM       a RESTORE to a line number instead of to the
  8 REM       first DATA statement of the program
 10 str$ = "RSTOREL"
 15 DATA      34,245,62,62,0,50,249,62,201
 20 FOR n = 1 TO 7
 30 POKE 799+n, ASC(MID$(str$, n, 1))
 40 NEXT
 49 REM       Patch parse vector
 50 POKE 797, 173: POKE 798, 3
 54 REM       Patch execute vector
 55 POKE 6551, 89: POKE 6552, 4
110 REM       copy GOTO execute code
120 FOR n = 0 TO 17: POKE 1113+n, PEEK(8342+n): NEXT
130 REM       Add the RSTOREL code
140 FOR n = 0 TO 8: READ m: POKE 1131+n, m: NEXT
150 PRINT: PRINT "  RSTOREL modification is done"
160 NEW
```

Listing #3

```
*
* Program:  CRUNCHER formats the SmartBasic tokenized code into
*           a binary file which can then be BSAVEd.
*
* Written           SERENDIPITY PRODUCTIONS
*      by:              P.O. Box 07592
*                    Milwaukee, WI  53207
*
*         Decimal
*Addr.     Value              Op-code          Comments
*======  ==========        ==============  ========================
*
 56320: 205 205 31          CALL $1FD3      ;Call SmartBasic CLEAR
 56323: 197                 PUSH BC
 56324: 229                 PUSH DE
 56325: 213                 PUSH HL
 56326: 245                 PUSH AF
 56327: 237  91 223  62     LD   DE,($3EDF) ;Load LOMEM from parm tbl.
 56331: 237 107 239  62     LD   HL,($3EEF) ;Load LOFREE  "   "    "
 56335: 229                 PUSH HL         ;  and save it on stack
 56336: 167                 AND  A          ;Clear the carry flag
 56337: 237  82             SBC  HL,DE      ;Calc. # bytes in tables
 56339: 229                 PUSH HL         ;Transfer it to BC via the
 56340: 193                 POP  BC         ;  stack and add one to it
 56341:   3                 INC  BC         ;  This is the byte count
 56342: 237  91 237  62     LD   DE,($3EED) ;Load HIFREE, subtract one
 56346:  27                 DEC  DE         ;   to get destination adr
 56347: 225                 POP  HL         ;POP the source addr. and
 56348: 237 184             LDDR            ;  copy the tables up
 56350:   1  30   0         LD   BC,$001E   ;Load constant (30 bytes)
 56353:  33 246  62         LD   HL,$3EF6   ;Load source and copy the
 56356: 237 184             LDDR            ;  30 critical bytes up
 56358:   1  55   0         LD   BC,$0037   ;Byte count for uncruncher
 56361:  33 206 220         LD   HL,$DDCE   ;Source addr of code and
 56364: 237 184             LDDR            ;  copy it under parms
 56366:  33 128 209         LD   HL,$D180   ;Load default LOMEM
 56369: 167                 AND  A          ;  prepare the carry and
 56370: 237  82             SBC  HL,DE      ;  calc total file length
 56372:  34 149 220         LD   ($DC95),HL ;Put it into PRINT code
 56375:  19                 INC  DE
```

```
56376: 203 122          BIT  7,D          ;If its <32768 then jump
56378:  40  17          JR   Z,$DC4D      ;  to 56397 (decimal)
56380: 122              LD   A,D          ;Otherwise,
56381: 230 127          AND  $7F          ;  subtract 32767 ...
56383:  87              LD   D,A          ;
56384:  62 255          LD   A,$FF        ;  and make the first
56386:  50 136 220      LD   ($DC88),A    ;  constant = 32767
56389:  62 127          LD   A,$7F        ;  We'll explain why next
56391:  50 137 220      LD   ($DC89),A    ;  month
56394:  19              INC  DE           ;Adjust the count
56395:  24   8          JR   $DC55        ;Jump to PRINT rtn @ 56405
56397:  62   0          LD   A,$00        ;Make the first constant
56399:  50 136 220      LD   ($DC88),A    ;  zero since we don't
56402:  50 137 220      LD   ($DC89),A    ;  need it (ie < 32768)
56405:   0              NOP
56406: 237  83 140 220  LD   ($DC8C),DE   ;Store start addr of file
56410: 217              EXX               ;  in PRINT code
56411:  14  38          LD   C,$26        ;Put PRINT line length in
56413: 217              EXX               ;  C' for PRINT command
56414:  17 112 220      LD   DE,$DC70     ;Load addr of PRINT cmd.
56417: 205 174  30      CALL $1EAE        ;Call SmartBasic PRINT rtn
56420: 241              POP  AF           ;Restore registers
56421: 209              POP  DE           ;
56422: 225              POP  HL           ;   and ...
56423: 193              POP  BC           ;
56424: 201              RET               ;     We're done!!!
56425:   0              NOP
56426:   0              NOP               ;Room for expansion
56427:   0              NOP
56428:   0              NOP
56429:   0              NOP
56430:   0              NOP

;
;The following data represents the tokenized version of the line...
;
;        PRINT "]BSAVE PROGRAM.MA,a";32767+0;",l";0   -Run under whole LINE
;
;  where the zeroes are replaced with the appropriate constants.

                                                    39
56431:  39   7 145  19  93  98 115  97 118 101  32  80  82  79  71
56446:  82  65  77  46  77  65  44  97 186 139   0   0 160 139   0
56461:   0 186 145   2  44 108 186 139   0   0   0

;Now the routine to UNCRUNCH the file into a runnable form.  This
;  routine is saved with the condensed program.

56472: 197              PUSH BC           ;Save the registers
56473: 229              PUSH HL
56474: 213              PUSH DE
56475: 205 133 251      CALL $FE85        ;CALL a RET instruction
56478:  59              DEC  SP           ;  and get the address
56479:  59              DEC  SP           ;  where we are now in
56480: 225              POP  HL           ;  memory
56481:  17  18   0      LD   DE,$0012     ;Calc. the source address
56484:  25              ADD  HL,DE        ;  for the copy instr.
56485:  17   0 220      LD   DE,$DC00     ;Load it as the dest.
56488:   1  31   0      LD   BC,$001F     ;Load the routine length
56491: 237 176          LDIR              ;Copy it to safe area
56493: 195   0 220      JP   $DC00        ;  and execute it there
56496:   1  30   0      LD   BC,$001E     ;Load byte count
56499:  17 217  62      LD   DE,$3ED9     ;  destination for the
56502: 237 176          LDIR              ;  parms and copy them
56504: 229              PUSH HL           ;Save the ending address
56505: 237  91 223  62  LD   DE,($3EDF)   ;Load LOMEM value
56509: 237 107 239  62  LD   HL,($3EEF)   ;  and LOFREE
56513: 167              AND  A            ;Calc. the # of bytes in
56514: 237  82          SBC  HL,DE        ;  Basic tables
56516: 229              PUSH HL           ;Transfer it to BC as
56517: 193              POP  BC           ; byte count in LDIR
56518:   3              INC  BC           ;Adjust it
56519:   3              INC  BC
56520: 225              POP  HL           ;Restore the source addr
56521: 237 176          LDIR              ;  and copy tables
56523: 209              POP  DE           ;Replace the registers
56524: 225              POP  HL           ;  and ...
56525: 193              POP  BC           ;       TAH-DAH!!! ...
56526: 201              RET               ;We're done!!!
```

The ADAM TECHNICAL JOURNAL is published bi-monthly by Serendipity Productions. Subscription rates are $18.00 per year in the U.S. and Canada, $23.00 per year in any other country, payable by check or money order only. Single issues are available for $5.00. All inquiries and payments should be made to Serendipity Productions, P.O. Box 07592, Milwaukee, WI 53207.

Preview of the next issue

- We'll examine the tokenized Smartbasic program in even greater detail
- New utility will print a Basic program in three formats... hexidecimal, tokens and normal list form
- We'll explore the format of the tape directory
- New utility will sort the tape directory and recover deleted files
- All concepts are demonstrated in an example program

Errors and Corrections

An error was made in the printing of the utilities in the first issue which we would like to correct. We did not realize that the printer we used made it impossible to distinguish between a one and a lower case L. Therefore, to make listing #1, page 10 correct, make certain that all references to the variable (ul) contain ones not lower case L in lines 54010, 54030, 54040 and 54070. To prevent future occurances of this problem we will use a different printer where necessary.

SERENDIPITY PRODUCTIONS
P.O. BOX 07592
MILWAUKEE, WI 53207