

\$19.95

**For TRS-80 Models
I, III, and 4**

TRS-80 ROM ROUTINES DOCUMENTED

Written by Jack Decker

- Arithmetic Routines
- String Handling Routines
- Input and Output Routines
- PEEK and POKE Locations
- Model I, III, and 4 ROM Differences

**THE
ALTERNATE
SOURCE**

ERRATA NOTICE - TRS-80 ROM ROUTINES DOCUMENTED - 9/12/84

Please make the following corrections to this edition of TRS-80 ROM ROUTINES DOCUMENTED:

1) Appendix II (starting on page 86) is applicable ONLY to the original versions of the Model 4 ROM. The "new" Model 4 (with green screen and relocated arrow keys) uses a revised ROM with several changes from the original, INCLUDING changes in the area between 0000H and 2FFFH (which was previously unchanged from the Model III ROM).

2) Page 108 - Please make the following correction to line 1030 of the assembly language source code on this page:

Change from:

7FE0 D0 01030 RET NC ;RETURN IF END OF STRING

To:

7FE0 C8 01030 RET Z ;RETURN IF END OF STRING

We apologize for these errors, and regret any inconvenience they may have caused anyone!

TABLE OF CONTENTS

PREFACE by Dave McGlumphy	ii
INTRODUCTION	1
CHAPTER ONE - Input and Output	7
CHAPTER TWO - Arithmetic Routines	21
CHAPTER THREE - Strings and String-Handling Routines	38
CHAPTER FOUR - Miscellaneous ROM Routines	50
CHAPTER FIVE - Reserved RAM Locations	63
APPENDIX I - Model I & III ROM BASIC Compared	78
APPENDIX II - Rom Differences Between the Models III and 4	86
APPENDIX III - Model I ROM Changes	90
APPENDIX IV - TRS-80 vs. PMC-80/Video Genie/System 80/TRZ-80	95
APPENDIX V - Relocatable Program Sample	98
APPENDIX VI - Improved Ampersand Function	104
APPENDIX VII - Line Input Routines for Level II BASIC	109
APPENDIX VIII - BASIC Tokens and Entry Points	119
HEXADECIMAL ADDRESS CROSS-REFERENCE	122
AFTERWORD by Charley Butler	126

There once was a man up north
Who only BASIC spewed forth
'Til he dived in the ROM
and escaped all unharmed
to tell us what he learned perforce.

So it is with Jack Decker.

This book that he has written is the result of days, weeks, and months of continuing study of how the TRS-80 ROM works. It is the summarization of the work you would do if only you could find the time to do it. Maybe you are pretty good at programming in BASIC, but you can't quite figure out how ASSEMBLY is possible. Maybe you are tired of writing the ASSEMBLY routines to print a line on the screen or to get some keyboard input. Perhaps you have not figured out how to do the arithmetic your assembly program needs. You know how to do it in BASIC but do not have the faintest idea where to begin in assembly. If BASIC can do it in the ROM, why not access those same routines from assembly?! Jack shows you how.

This book is more than a disassembly. Jack gives you a good education. He goes beyond one-line comments and gives you the "big picture". He shows you the background and the whys and wherefores of the ROM subroutines. He offers tips and suggestions, and he warns you of the pitfalls that can drive you bonkers when you use a subroutine knowing only enough to be dangerous.

Jack has organized his book into the sections that you will need as you work rather than by ROM address only. It is easier to know how to access the routine you need successfully than it is to try to figure out what all those instructions and comments on a page of disassembly mean. Jack has done the "figuring out". He has saved you a lot of time and trouble.

AND (!), his style is human. It is like having Jack sit in front of your computer with him telling you the neat things he found out.

With that, Arnetta Rose Decker and I give you ...

TRS-80 ROM ROUTINES DOCUMENTED
(or whatever the name of this comic book is gonna be...)

Dave McGlumphy, Chattanooga Microcomputer Club 07/31/83

INTRODUCTION

The use of ROM routines can greatly simplify the task of writing a machine language program. Many programmers like to make use of the various subroutines found in the ROM in order to save memory, and to eliminate the chore of writing a routine that's already available in BASIC. One problem in using ROM routines is that many of them have not been very well documented. This book will attempt to provide more complete information than has been available heretofore on the various routines found in the TRS-80 Model I and III ROMs. The emphasis of this book will be to provide all information necessary for the programmer to actually make use of these ROM routines - information such as how to access the routine, how to pass data between the routine and the calling program, etc. All information given can be assumed to apply to both the Model I and the Model III (and the Model 4, when operated in the "Model III mode"), unless otherwise stated.

- It is my hope that having these ROM routines described in detail will allow the machine-language programmer to make greater use of them, so that he may save memory space and avoid "re-inventing the wheel" at the same time. However, there is another use of these ROM routines that I hope will be made, and that is that by studying them and finding out how they work you will in effect be learning Assembly-Language programming. Microsoft BASIC is generally very well written (with occasional notable exceptions), and it is possible that by looking over these routines you will pick up some good programming techniques without too much effort.

Many of the ROM routines documented in this series have been previously explained by others, in numerous books and magazine articles. Although it is not possible to give credit to each and every person that has made an indirect contribution to these articles, I would like to give special credit to Mr. Lawrence J. Oliveira of Sacramento, California. Some of Mr. Oliveira's heretofore unpublished work has greatly influenced both the style and the content of this book.

I'd like to hear about any useful ROM routines that you have found that do not appear in this book. If you have any questions or comments regarding this series, please feel free to write - perhaps your routines or comments will help shape a future edition of this book. Due to time constraints, I may not be able to send individual replies to all that write, but I definitely will not send a reply if you do not enclose a self-addressed stamped envelope (U.S. or Canadian postage - if you can't provide either because you live in another country, just send the self-addressed envelope). Even then, you may receive only a few notes jotted on the back of your letter. If you need a fair amount of assistance, I suggest that you first ask around at your local TRS-80 user group or computer club - many of these groups have one or two "resident experts" that can answer most of the questions you may have!

Jack Decker
c/o The Alternate Source
704 North Pennsylvania Avenue
Lansing, Michigan 48906

IMPORTANT: All references in this book to the Model III should be assumed to apply equally to the Model 4 when operated in the "Model III mode", unless specifically stated otherwise. There are very few differences between the standard Model III ROM, and the ROM installed in the Model 4. Therefore, the practice in this book will be to assume that anything applicable to the Model III ROM also holds true for the Model 4 ROM, including the places where the text states "Model III only". On the rare occasion where this is not the case, it will be pointed out.

DISK OPERATING SYSTEM CONSIDERATIONS

This book is intended to be applicable to Model I and Model III TRS-80's, whether they be cassette-based or disk-based systems. However, the fact of the matter is that the computer operates differently depending on whether or not a disk system is connected. Before we can effectively use the ROM routines, we need to know what the differences are, and how they might affect our programs.

Most Model I Expansion Interface owners are already familiar with one difference, that being the operation of the computer when the RESET button is depressed. In a system with no Expansion Interface, pressing RESET returns you to the BASIC READY state, and is often useful for escaping from programs that are "locked up". Add the Expansion Interface, however, and you'll find that depressing the RESET button now re-boots the system, starting at DOS READY or MEMORY SIZE?, and usually destroys the resident program. This difference in startup procedures will normally not affect the typical machine language program, but the differences between cassette- and disk-based systems go far beyond that, and to ignore those differences is to invite disaster. A ROM routine that works quite nicely under Level II (non-disk) BASIC can operate differently (and perhaps give us all sorts of grief) when we try to use it on a disk system. Here's why:

Level II BASIC was designed to fit into 12K of ROM (expanded to almost 14K on the Model III). Realizing that some desirable functions and commands would not fit into the 12K, Microsoft placed several Disk BASIC exits in the ROM. These "exits" are jump instructions that jump to an area in reserved RAM. Depending on the particular exit, one of several different actions may be taken.

If a Disk BASIC command or function is entered from the keyboard or used in a program, a jump will be taken to somewhere in the area of memory between 4152H and 41A3H. At that location, three bytes of memory will contain a jump to yet another location - under Level II, all of the jumps are to the L3 ERROR routine at 12DH, while under Disk BASIC the jumps are to the area of Disk BASIC that handles that particular command or function. These exits are commonly referred to as "patch points" or "vectors". The term VECTOR is used to indicate locations in reserved RAM that contain jumps to other locations (or sometimes RET instructions when not in use), and are themselves jumped to from other locations (usually in ROM). The purpose of these vectors is to allow Disk BASIC or other programs to intercept the flow of the BASIC interpreter at certain points. The usual sequence on a

non-disk system is this: BASIC interpreter in ROM takes a jump to a vector in RAM. Vector in RAM contains jump or return back into ROM. Under Disk BASIC, however, the flow goes as follows: BASIC interpreter in ROM takes jump to vector in RAM. Vector in RAM contains jump into Disk BASIC or DOS.

In any event, the vectors in the part of memory between 4152H and 41A3H will usually not trouble us, as they are only used for Disk BASIC commands. In fact, if we have a non-disk system, we can use any of the Disk BASIC command words to access our machine language programs, by replacing the appropriate jump vector with a jump to our program. As an example, suppose we have a machine language program that begins at 7000H. We could access it by typing, for example, the NAME command under Level II BASIC. All we have to do is replace the three bytes starting at 418EH with a JP 7000H instruction, and from then on the use of the NAME command (either from the keyboard or within a BASIC program) will cause a jump to our machine language program.

Another group of DOS exits are associated with the RST instructions. When one of these instructions is executed, a jump is taken from the ROM to a three-byte jump vector that begins somewhere in the area of 4000H through 4012H. Under both Disk BASIC and Level II, RST 8, RST 10H, RST 18H, and RST 20H all jump back into the ROM to provide one byte access to commonly used ROM routines. These RST instructions are used by most of the significant ROM routines, so if these vectors are changed (by the Disk Operating System or another program), calls to ROM routines could have unpredictable results. However, since the DOS does not normally change these vectors, this is generally not as great a problem as it might at first appear.

RST 28H is another potential problem, if any ROM routines are being used that are expected to recognize the BREAK key. Under Level II or Disk BASIC, pressing the BREAK key returns ASCII value of 1, but under normal DOS operation the BREAK key returns an ASCII zero (in other words, it is ignored). Another problem that may crop up in regard to the use of the BREAK key is that if DEBUG is on, pressing BREAK may cause a jump to DEBUG under some Disk Operating Systems. However, since the user of the program would normally not have DEBUG on (unless he were really trying to debug the program!), we can ignore this in most applications. In any event, if either of these BREAK key actions of the DOS turn out to be a problem, they can be canceled by placing a C9H byte (a RET instruction) at memory location 400CH (the location of the vector for RST 28H).

There is a third group of DOS exits, and these will give us our biggest headaches. These exits are called by many of the more useful ROM routines, and are used to extend the capabilities of many of the BASIC commands and functions under Disk BASIC. These exits jump to three-byte vectors in the area from 41A6H to 41E2H. Under Level II, each of these vectors contains a RET (C9H) instruction, but under Disk BASIC the vectors are changed to jump to the associated Disk BASIC routines. The problem here is that if Disk BASIC has been loaded, and then a return is made to DOS (or some other machine language program has been activated), the associated Disk BASIC programs may be changed or destroyed. Thus, a jump may be taken by the ROM to an area where

it expects to find a routine supplied by Disk BASIC, but in fact anything could be at that location. If this is a potential problem, the solution is to insert a RET instruction (C9H) at every third byte starting at 41A6H, and ending at 41E2H. The following segment of assembly language code can be used to accomplish this:

```

LD      HL,41A6H
LD      B,15H
LOOP   LD      (HL),0C9H
       INC    HL
       INC    HL
       INC    HL
DJNZ   LOOP

```

The problems mentioned above will occur only if three conditions are met:

- 1) The program in question makes calls to one or more ROM routines that have DOS exits. A stand-alone program that makes no calls to the ROM will not be affected by any of the problems mentioned. Furthermore, any call to a routine that has no DOS exits (within itself, or within any of the subroutines that it may call) is immune to these problems.
- 2) A disk-based system is in use, or a program has been used since power-up that has changed the DOS exits. Under normal circumstances, on a cassette-based system all DOS exits will remain as they were initialized during power up, thus avoiding any of the problems mentioned above.
- 3) Level II or Disk BASIC is not resident. In other words, a machine language subroutine designed to be used with BASIC under the USR function will never be affected by these problems, because BASIC (either Level II or Disk) will be present when the routine is called. Ditto for any utility program designed to work with BASIC (such as a BASIC line renumberer program) - again, we assume that BASIC will be present.

Looking at the matter another way, what sort of program might be expected to run afoul of these problems? One good example might be a computer game program that is written entirely in Assembly language (not part BASIC and part machine language), that makes calls to the ROM for various functions and is expected to run under a disk-based system. What can be done to avoid the problems? There are several possibilities:

- 1) If the program is to be supplied on cassette tape, the user will normally be instructed to load the program using the SYSTEM command. This will normally avoid the problem, since BASIC must be present for the SYSTEM command to be used. However, if the starting address of the program is around 7000H or below (as it would likely be with a game program), it is possible that it might overwrite part of Disk BASIC. The solution here is to instruct the user to load the program only under Level II BASIC (accomplished by powering up the computer with the BREAK key depressed, or by typing BASIC2 from the DOS READY prompt).

2) The above solution stinks as far as most disk users are concerned, since who wants to mess around with cassettes when you have a disk system? Assuming that the user has transferred the program to disk using TAPEDIISK (or a similar utility program), it's possible to load the program without destroying Disk BASIC by using the BASIC CMD"I" command. However, the caution about overwriting Disk BASIC still applies, and in addition the CMD"I" command may work differently, or may not be available at all, under different versions of DOS.

3) The best solution is to include the short segment of Assembly language code shown above to plug the DOS exits with RET instructions. This will solve the problem for most programs. However, there is one other potential conflict of which you should be aware.

If your program accepts input from the keyboard using one of the ROM line input routines (which will accept characters and put them into a buffer until ENTER or BREAK is pressed), you should be aware that BASIC sets up a buffer that can hold up to 255 characters to accept this input. The problem is that the buffer is at different locations under Level II, Model III, or Disk BASIC. The two-byte pointer to the start of the input buffer is located at 40A7H-40A8H. If your program will require the use of this buffer, and you are not sure that the one currently in use will not be overwritten when your program is loaded, you may want to set aside space for your own buffer and then change the pointer to point to the start of the buffer that you have set aside for the purpose. Like the other problems, this one will usually only manifest itself when the program is run on a disk-based system, and BASIC is not present.

In conclusion, I would offer the following piece of advice: If you are writing a program that may fall into the "problem category", write it as if it will be used on a cassette-based system only (save your machine language object code on tape rather than disk, and load it from Level II BASIC only, using the SYSTEM command). When you have it thoroughly debugged, then and only then try to run it under the DOS. If it works, let a few friends try it (it's amazing how your friends can make the computer go bonkers in ways you never thought of). Be sure to go to Disk BASIC, then back to DOS READY and try to reload the program. If it still works each time it's loaded, it will probably run O.K. (but try it under different Disk Operating Systems if you can, just to make sure). If you find you have problems, try disassembling the ROM routines that you're using to find out what DOS exits are being used. Or, just try using the hints given above and see if they take care of the problem. Occasionally you may have to do a little detective work to solve the problem, but in most cases you'll find that the problems are resolved rather easily.

Keep in mind that certain programs (usually commercial programs that you have purchased) have no respect for the use of RAM by the system, and merrily proceed to wipe out all system pointers, etc. If you have run one of these, you may have to reboot the system in order to make your program (or any other) run properly. Your program is probably not at fault if it will always work immediately after a system reboot, unless the simple act of loading Disk BASIC, then

returning to DOS READY and attempting to load the program fails. Of course, you must decide what you consider to be acceptable performance (perhaps you don't mind having to reboot the system before using your program).

CHAPTER ONE - INPUT AND OUTPUT

Before we can make the computer do anything useful, we have to figure out how to get the results of our program into a form that we can make use of. In addition, most programs will at some point require some information that must be input from the "outside world." We may want the results of our program to go to the video display, to a line printer, or to a cassette or disk file. The input for the program might come from the keyboard, or from a cassette or disk file. The process of input and output is usually the most confusing part of assembly-language programming for the beginner, because although he probably has a fairly good idea of how to make his first simple programs do what he wants them to do once the required data has been passed to the program, he has no idea of how to request the data from the keyboard, or how to get the result of the program on the video display in human-readable form. This is where the use of available subroutines in the ROM can turn a complicated task into a few easy instructions.

DEVICE CONTROL BLOCKS

One of the first things you will encounter in using the input and output routines are the Device Control Blocks, or DCB's for short. Located in reserved RAM, these 8-byte long blocks contain various pieces of information related to the operation of the keyboard, the video display, and the line printer (and the RS-232-C interface and the I/O Router in the Model III only). It is in these blocks that we find the driver addresses for the devices mentioned, and we can replace these driver addresses with the addresses of substitute driver programs or patches to the ROM drivers if necessary. As an example, the KBFIX program (used in older Model I's to fix the keybounce problem) replaced the driver address in the keyboard DCB with the address of its debounce routine. The debounce routine (which was actually a substitute for some of the code in the ROM keyboard driver) then jumped into the ROM at a different location. Another example of the use of a DCB driver address is the lowercase driver program (used with a hardware lowercase modification) that puts its address into the video DCB, in order to replace a section of "bad" code in the ROM, then jumps into the video driver just past the unwanted ROM code.

The formats of the various DCB's are as follows:

KEYBOARD DEVICE CONTROL BLOCK

4015H Device Type (normally 1 = read only)
 4016H LSB of Driver Address
 4017H MSB of Driver Address
 4018H 0 (in Model III but NOT Model 4, location used by shift-lock routine)
 4019H 0 (in Model III, 0="Upper & Lower Case", else "Caps only")
 401AH 0 (in Model III, time counting byte for blinking cursor)
 401BH "K" (in Model III, status of blinking cursor-on or off)
 401CH "I" (in Model III, Cursor Blink Switch - 0="Blink")
 ("KI"=Keyboard Input - Model I only)

VIDEO DEVICE CONTROL BLOCK

401DH Device Type (normally 7 = read and write)
 401EH LSB of Driver Address
 401FH MSB of Driver Address
 4020H LSB of current Cursor position (3C00H to 3FFFH)
 4021H MSB of current Cursor position
 4022H Character "covered" on video by Cursor character
 4023H "D" (in Model III, ASCII code for Cursor character)
 4024H "O" (Model III: 0=Space Compression, 1="Special" Characters)
 ("DO"=Display Output - Model I only)

LINE PRINTER DEVICE CONTROL BLOCK

4025H Device Type (normally 6 = write only)
 4026H LSB of Driver Address
 4027H MSB of Driver Address
 4028H 67 - Number of lines per page (+1 in Model III only)
 4029H Current Line Number (Model III: Number of lines printed +1)
 402AH 0 (Model III: Number of characters printed on current line)
 402BH "P" (in Model III, max. line length -2, 255="No maximum")
 402CH "R" (in Model 4 only, zero byte)
 ("PR"=Printer - Model I only)

RS-232-C INPUT DEVICE CONTROL BLOCK (Model III only)

41E5H Device Type (normally 1 = read only)
 41E6H LSB of Driver Address
 41E7H MSB of Driver Address
 41E8H Character Received (0 = no character)
 41E9H Bit 1 = Wait/No Wait Flag, Bit 2 = Driver On/Off Flag
 41EBH "F1" key definition in Model 4 only (used by Keyboard Driver)
 41ECH "F2" key definition in Model 4 only (used by Keyboard Driver)

RS-232-C OUTPUT DEVICE CONTROL BLOCK (Model III only)

41EDH Device Type (normally 2 = write only)
41EEH LSB of Driver Address
41EFH MSB of Driver Address
41F0H Character to Transmit
41F1H Bit 1 = Wait/No Wait Flag, Bit 2 = Driver On/Off Flag
41F3H "F3" key definition in Model 4 only (used by Keyboard Driver)
41F4H Keyboard row storage for 3880H¹ in Model 4 only (used by Keyboard Driver)

RS-232-C INITIALIZATION DEVICE CONTROL BLOCK (Model III only)

41F5H Device Type (normally 2 = write only)
41F6H LSB of Driver Address
41F7H MSB of Driver Address
41F8H Baud Rate Codes (Bits 0-3 = Receive, Bits 4-7 = Send)
41F9H RS-232-C Characteristics Switch
41FAH Wait/Don't Wait Switch (0 = Don't Wait)

I/O ROUTER DEVICE CONTROL BLOCK (Model III only, NOT initialized in Model 4)

421DH Device Type (normally 2 = write only)
421EH LSB of Driver Address
421FH MSB of Driver Address
4220H First character of Destination Device Code
4121H Second character of Destination Device Code
4122H First character of Source Device Code
4123H Second character of Source Device Code
4224H Control Key Flag (used by Keyboard Driver - Models III and 4)
(Source & Destination Codes may be any of: KI, DO, RI, RO, PR)

A further clarification of locations 4028H and 4029H (located in the line printer DCB) is in order. 4028H is set to 67 decimal on power-up in both the Model I and the Model III. This number is correct on the Model III (assuming that standard 66-line forms are being used in the printer), but on the Model I it can cause some printers to "creep" up one line per page. If you have this problem, try typing POKE 16424,66 from the keyboard and see if it corrects the problem. The Model III should not have this problem since it counts lines differently. Specifically, on the Model I location 4029H holds the actual number of lines already printed on a page, while on the Model III this location holds the number of lines already printed plus one.

If you are intercepting one of the drivers by storing an address that points to your routine in the DCB, you should be careful to save the address that is already in the DCB. Too many programmers make the mistake of thinking that their routine will be the only one in high memory, so you have a situation where two machine language programs are mutually incompatible. Here's why: Program one loads into high memory and places its address in the DCB. Now program two is loaded in, and without any checking to see what is already in the DCB, it

places ITS entry point address in the DCB. The exit from program two is a jump to the normal ROM driver routine. Because of this, program one is "lost" to the user. If instead, program two (during its initialization segment) gets the address from the DCB and loads that address into the main part of the program (at the program exit instruction), the two programs will (hopefully) be mutually compatible.

Here's an example of checking the driver address and then using it in your routine:

```

00100 START ORG      nnnn
00110          LD       HL,(4016H)    ;Address in keyboard DCB
00120          LD       (USREND+1),HL ;Address of end of routine
00130          LD       HL,USRST     ;Address of start user routine
00140          LD       (4016H),HL   ;User routine address in DCB
.
.
09999 USREND JP      0           ;0 replaced by DCB address

```

Your exact situation may be somewhat different but the idea is the same - when you write a program, try not to "clobber" the address already in the DCB, unless it is unavoidable. It will usually only be unavoidable if you are rewriting a portion of the ROM driver routine itself, then jumping into ROM at other than the normal entry point.

If you are just beginning to program in Assembly Language, don't worry about the above consideration at first. If, however, you are creating a program to sell to other TRS-80 owners, you definitely should keep it in mind.

ROM INPUT AND OUTPUT ROUTINES

The following routines are available for use in the Model I/III ROM. If you are operating under DOS, be sure to read the portion of the introduction to this book entitled "Disk Operating System Considerations" before attempting to use these routines (otherwise you may have unpredictable results).

GENERAL INPUT/OUTPUT ROUTINES

0013H - Inputs a byte from an input device. When calling, DE = starting address of DCB of device. On exit, A = byte received from device, Z set if device ready. Uses AF.

001BH - Outputs a byte to a device. When calling, A = output byte, DE = starting address of DCB of device. On exit, Z set if device ready. Uses AF.

032AH - Output a byte to device determined by byte stored at (409CH) - FFH=Tape, 0=Video, 1=Printer. When calling, A = output byte. Uses AF (may use other registers as well). Warning: This routine CALLs a Disk BASIC link at address 41C1H which may have to be "plugged" with a RETurn (C9H) instruction.

038BH - Reset device type flag at 409CH to zero (output to video display), also outputs a carriage return to the line printer if printer is not at beginning of line (determined by checking the contents of the printer line position flag at 409BH - if flag contains zero, printer is at start of line). Note that if printer line position flag does not contain zero and the printer is not on line, the computer will "hang up" waiting for a "printer ready" signal. See also routine at 2169H (described below).

20FEH - Output a carriage return (0DH) to a device determined by flag stored at (409CH). NOTE: This routine may be CALLED at 20F9H, in which case it will not perform the above action if the video display cursor is already positioned at the beginning of a line, as determined by checking the contents of the cursor position flag at 40A6H (if zero, cursor is at start of line). This routine CALLS the routine at 032AH and also CALLS a Disk BASIC link at 41D0H. See the warning for the routine at 032AH (above).

213FH - TAB function for video or printer (determined by flag at 409CH). On entry: E register contains desired TAB position, HL points to start of message to be displayed (or zero byte if no message). This routine does extensive string processing and may not be the most efficient method of achieving the desired result, particularly if it is desired only to tab over a number of spaces. Also, this routine CALLS several Disk BASIC links which may have to be "plugged".

2169H - Reset device type flag at 409CH to zero (output to video display), also turns off cassette drive if necessary. CALLS Disk BASIC link at 41BEH prior to return. See also routine at 038BH (described above).

2B75H - Output a string to device indicated by device type flag stored at 409CH. String must end with zero byte. On entry, HL registers must point to address of start of string. Calls routine at 032AH (note warning for that routine - see above).

28A7H - Essentially the same effect as the routine at 2B75H (described above), except that text may also end with a quotation mark (22H), creates string vector before output (destroys current contents of ACCUM, sets number type flag at 40AFH to 3 - see chapter two of this book), and also uses BC & DE registers. Depends heavily on BASIC string management routines (use of 2B75H or other routines may be preferable). If string contains a carriage return (0DH) character, a CALL will be made to the Disk BASIC link at 41D0H. Used by BASIC PRINT statement. This routine may also be entered at 28A6H, in which case the HL register pair will be incremented prior to beginning to output string.

NOTE regarding the routines at 032AH, 2B75H and 28A7H: These routines (as well as many others) use a flag byte to determine if the video display is in the 32 or 64 characters per line mode. In the Model I, this flag is located at 403DH and actually contains the CURRENT PORT OFFH OUTPUT BITS, which are organized as follows:

BIT 3 Select video 32 character mode if set
 BIT 2 Turns on cassette tape relay if set
 BITS 1 & 0 Are set for positive and negative audio pulses to the cassette "AUX" plug

In the Model III, this flag is located at 4210H and actually contains the CURRENT PORT 0ECH OUTPUT BITS, which are organized as follows:

BIT 6 Enables fast clock speed if set on Model 4 only
 BIT 5 Disables video wait states if set (not used on Model 4)
 BIT 4 Enables I/O bus if set
 BIT 3 Japanese Kana character set used as "special" characters if set
 BIT 2 Select video 32 character mode if set
 BIT 1 Turns on cassette tape relay if set
 BIT 0 Enables clock display on video if set

The above values take effect when they are OUTput to the proper port (0FFH or 0ECH). However, in all current editions of the Model III ROM, an error exists in that the test for double-width characters at 0348H (used by all three of the above mentioned routines) has not been changed to test the flag at 4210H rather than the flag at 403DH. This can cause serious problems when attempting to use the 32-character mode on the Models III and 4 (even when programming in BASIC). It should also be noted that there is a hardware problem associated with the Model III 32-character mode that sometimes causes characters to be improperly displayed even though they are apparently stored correctly in video memory (this problem does NOT seem to show up when the ROM routines are used, but can appear when the programmer uses a machine language loop to move a text string from somewhere in memory to a video memory location). The solution to this hardware problem is to use an LDIR or LDDR instruction to perform the move, or to insert a NOP instruction after any load (LD) instruction where the destination of the load is video memory (3C00H-3FFFH). Both of the above problems are evident only on the Model III, and then only when using the 32-character video display mode (thanks to Jesse Bob Overholt for providing the solution to the hardware problem mentioned above).

KEYBOARD ROUTINES

NOTE: All of the following keyboard routines (except for the Model III fast BREAK check at 028DH) use the ROM keyboard driver routine, which executes a RST 28H instruction prior to returning to the calling routine if the BREAK key was depressed. A JP 400CH instruction is found at 0028H, and under a non-disk system 400CH contains a RET instruction (unless modified by a user program). However, under a disk-based system RST 28H is also used for DOS overlay requests, and the vector at 400CH contains a jump into the Disk Operating System. More information on the RST 28H routine can be found in chapter four of this book.

002BH - Loads DE with address of keyboard DCB and scans keyboard. On exit, if no key pressed the A register will contain a zero byte, else the character input from the keyboard will be returned in A. Character is not echoed to video. Uses AF,DE (to save DE use routine at 0358H).

0040H - Line input routine. Jumps to 05D9H (see description of that routine below).

0049H - Scan keyboard until key pressed (continuous calls to 002BH). Returns with character in A. Uses AF,DE (to save DE use routine at 0384H).

028DH - Model III only. Very fast check for BREAK key only. On exit, BREAK was pressed if Z flag is reset (NZ status). Uses AF. For Model I users that may wish to duplicate this action, the ROM code used is as follows:

```
LD      A,(3840H)
AND    04H
RET
```

0358H - Calls Disk BASIC link at 41C4H, then saves DE and calls 002BH (Warning: Disk BASIC link may need to be "plugged", or you may bypass this link entirely by entering the routine at 035BH).

0361H - Input a line from the keyboard into the BASIC input buffer (starting address of buffer is found at locations 40A7H-40A8H). Maximum allowable input length is 240 characters. On exit, returns starting address of buffer minus one (location prior to first character) in HL. End of input is marked by zero byte in buffer. C flag is set if input is terminated by BREAK key. Uses AF,DE,HL. CALLs Disk BASIC link at 41AFH which may have to be "plugged". See also routine at 0040H (above).

0384H - Scan keyboard until key pressed (continuous calls to 0358H - see above warning for this routine). Returns with character in A. Uses AF (see also routine at 0049H).

03E3H - Model I ROM Keyboard driver address (as found in keyboard DCB).

05D9H - Line input routine. This routine displays each character as it is entered from the keyboard, and takes action on the control keys.

On entry:

HL points to start of buffer that will hold input.

B = maximum number of bytes to input.

Input is terminated by ENTER or BREAK (see note below).

On return:

HL points to start of text.

B = actual input length (less terminator).

C = original contents of B register.

A = terminating character (BREAK or ENTER).

C flag set if input terminated by BREAK.

When this routine is used under a Disk Operating System and Disk BASIC is not initialized, the BREAK key is ignored and will not act as an input terminator for this routine (a zero byte will be returned when the BREAK key is pressed). When used with Level II or Disk BASIC, pressing the BREAK key returns a 01H character which is then acted

upon as shown above, except that some Disk Operating Systems allow the user to set a flag that specifies that the BREAK key is to be ignored at all times (even when Disk BASIC is in use). Anytime that the Disk Operating System has disabled the BREAK key, this routine will ignore it as well. Also, note that in some cases (such as when using TRSDOS with DEBUG on), pressing the BREAK key may cause an immediate entry into DEBUG.

1BB3H - Displays prompt character (question mark followed by space) and then jumps to routine at 0361H (see above).

1D9BH - ROM BASIC check for BREAK or SHIFT-@ characters. Calls routine at 0358H (note warning for this routine - see above), decrements the A register, and takes appropriate action if either key is pressed (suspends program execution on SHIFT-@; or sets certain BASIC pointers, displays BREAK message, and goes to BASIC READY state on BREAK) - otherwise returns with keyboard character minus one in A.

21C9H - ROM BASIC "INPUT" routine. Print prompt string (if any) and get user input to variable. On entry: HL points to quotation mark (start of prompt string) or first character of variable name (if no prompt string) immediately following "INPUT" command. On exit: HL points to zero byte if input was valid, else prints "? REDO" (if certain flags are set properly) and requests more input. Syntax must be legal for BASIC "INPUT" statement, for example:

"Prompt String"; VARIABLE NAME <zero byte>
or simply: VARIABLE NAME <zero byte>

Multiple variables may be INPUT with one statement:

VARIABLE NAME, VARIABLE NAME, ... VARIABLE NAME <zero byte>

A colon may be used in place of the zero byte. See chapter two of this book for more information on the portion of this routine that begins at 21E3H, including information on which flags must be set to assure that the "?REDO" message is printed in the event of an input error, and how to determine if an error has occurred. NOTE: The routine that displays the prompt string (if any) may be CALLED separately at 21CDH. This routine will return immediately if the HL register pair does not point to a quotation mark (22H) on entry. Otherwise, the prompt string must be terminated with a quotation mark followed by a semicolon (3BH), or else a syntax error will result.

3024H - Model III ROM Keyboard driver address (as found in keyboard DCB).

VIDEO ROUTINES

0033H - Loads DE with address of video DCB, and displays character stored in A at current cursor position (cursor position stored at 4020H-4021H). This routine also performs control functions. Uses AF,DE (to save DE use routine at 33AH).

0150H - Routine used by BASIC POINT, SET, and RESET functions. To use in an assembly-language program, follow this procedure: PUSH the RETurn address on stack. Load HL with address of a right parenthesis character (29H). Load A register with 0 for POINT, 80H for SET, or 1 for RESET, then PUSH AF. Load A register with X-coordinate value in range 0 to 7FH, then PUSH AF. Load A register with Y-coordinate value in range 0 to 2FH. Finally, jump to 150H (don't CALL it). If POINT function is selected, on return from routine arithmetic accumulator (see chapter two of this book) will contain zero if POINT(X,Y) is off, -1 if on (integer precision).

01C9H - Clears screen and homes cursor (BASIC CLS command).

021BH - Model III only. Video line display routine. Displays the text starting at the address in HL. String may contain control characters. Text must be terminated by carriage return character (0DH, which will be printed), or end of text marker (ASCII ETX=03H, which will NOT be printed). On return HL points to the first character following the terminator character. Uses AF,DE,HL.

0298H - Model III only. Turn on the clock display (SET bit 0 of memory location 4210H). Uses AF.

02A1H - Model III only. Turn off the clock display (RESet bit 0 of memory location 4210H). Uses AF.

033AH - Saves contents of DE register and calls routine at 0033H (see above), also updates cursor line position indicator at 40A6H (used by TAB function). Uses AF.

0348H - Get current cursor position on line in A register. On return from this routine the A register will contain a value in the range 0 through 63 if the video display is in the 64 characters per line mode, or 0 through 31 if the 32 characters per line mode is selected. See note under routine at 28A7H (under GENERAL INPUT/OUTPUT ROUTINES) for information on Model III ROM error in this routine.

0458H - Model I ROM Video driver address (as found in video DCB).

0473H - Model III ROM Video driver address (as found in video DCB).

20F9H - Outputs a carriage return to video if cursor is not already at the beginning of a line. See routine at 20FEH (under GENERAL INPUT/OUTPUT ROUTINES) for more information.

LINE PRINTER ROUTINES

003BH - Loads DE with address of printer DCB and checks memory location 37E8H to determine printer status. If the A register contains zero upon entry, the routine exits with the Z flag indicating printer status, otherwise the routine waits until the printer is ready and then outputs the contents of the A register to the printer. On the Model III only, the BREAK key may be used to escape from this routine if the printer is not ready. On exit, Z flag is set if printer was ready.

01D9H - Model III only. Output all 1024 characters on video display to printer (graphics characters are changed to periods). Uses AF,DE,HL.

0214H - Model III only. Output a carriage return (0DH) to printer. A register contains zero (Z flag set) on return. Uses AF,DE.

0394H - Output a carriage return to printer. Loads A register with 0DH prior to calling routine at 039CH (see description below). A register contains 0DH on return. Uses AF.

039CH - Output character in the A register to the printer. Also maintains the line width counter at 409BH, which is incremented for each character printed, except that it is reset to zero if character is line feed (0AH), form feed (0CH), or carriage return (0DH). Line feed (0AH) characters are changed to carriage return (0DH) characters prior to calling the printer output routine. CALLs routine at 003BH (see above for description of that routine).

03C2H - Model III ROM Printer driver address (as found in printer DCB).

0440H - Model III only. Wait for "printer ready" status or BREAK key depressed. To use, PUSH address where program should jump if BREAK is depressed onto stack, then CALL 0440H. Returns to calling routine if printer is ready. If BREAK is pressed, POPs the return address off of the stack and then executes a RET instruction. Uses AF.

044BH - Model III only. Test printer status. Same as Model I routine at 05D1H (see description of that routine below).

058DH - Model I ROM Printer driver address (as found in printer DCB).

05D1H - Model I only. Test printer status. On return, Z flag set if printer is ready to accept output. This routine can be used to check the printer status prior to attempting output to the printer, thus avoiding the possibility of "hanging up" the computer if the printer is not ready. This routine can be used in programs designed to run on either Model (I or III) - to do this, first LD A,(37E8H) and then CALL 05D4H. Uses AF.

CASSETTE INPUT/OUTPUT ROUTINES

01D9H - Model I only. Make a cassette pulse. Uses AF,B,HL.

01F8H - Turn off cassette drive. Uses AF.

01FEH - Model I only. Define cassette drive. On entry, if HL points to a 23H byte ("#"), then a cassette drive number (-1 or -2) followed by a comma is expected in the bytes immediately following the one pointed to by HL, and the cassette drive so specified is selected. Otherwise drive one is used by default. The code at 0212H (see description below) is then executed. Uses AF, and if HL points to a 23H byte also uses BC,DE,HL.

0212H - Model I only (also may be used with Model III). This routine will select the proper cassette drive according to the value stored in the A register upon entry (0 = first cassette, 1 = second cassette), and will turn on the drive motor. NOTE: This routine is not present in the Model III. However, in order to accommodate existing Model I software, location 0212H in the Model III contains an XOR A instruction followed by a RET instruction. Uses AF.

0215H - Model I only. Turn on cassette drive. Same as routine at 0212H (see description above) except does not select cassette drive (previously selected drive is used). Uses AF.

021EH - Model I only. Reset cassette input port OFFH. Uses AF,HL.

022CH - Model I only (also may be used with Model III). Blinks right asterisk during tape load operations. NOTE: On Model III this location has jump to 0212H (zeroes A register and returns). Uses AF.

0235H - Read one byte from the tape into the A register. Uses AF.

0241H - Model I only. Read one bit and shift into A register. Uses AF,HL.

0264H - Write the byte stored in the A register to tape.

0284H - Turn on cassette motor, write leader and sync byte. In Model I, makes CALL to 01FEH (see description above) to select the proper cassette drive prior to executing routine at 0287H, while in the Model III this location contains a JP 0287H instruction (only one cassette drive is used with the Model III). Uses AF (except in Model I may use additional registers as noted for the routine at 01FEH).

0287H - Write leader and sync byte (A5H). Also turns on cassette motor on Model III only. Uses AF.

0293H - Define drive (on Model I only), turn on cassette motor, searches for leader and sync byte on cassette, then puts two asterisks in upper right corner of video. In Model I, makes CALL to 01FEH (see description above) to select the proper cassette drive prior to executing routine at 0296H, while in the Model III this location contains a JP 0243H instruction (0296H contains a JR 0243H instruction on the Model III). Uses AF (except in Model I may use additional registers as noted for the routine at 01FEH).

0296H - Turns on cassette motor (on Model III only), then searches for leader and sync byte. When found, puts two asterisks in upper right corner of video. Uses AF.

0314H - Reads two bytes (LSB/MSB) and transfers to HL registers (used when reading SYSTEM format tapes). Uses AF,HL.

2BF5H - BASIC CSAVE routine. Saves a BASIC program to tape. On entry, the HL register pair must point to the start of a valid filename sequence (a quotation mark followed by a single character filename, which in turn may be optionally followed by a second

quotation mark. The entire filename sequence must be terminated by a zero byte or colon).

2C1FH - BASIC CLOAD routine. Loads a BASIC program from tape. On entry, the HL register pair must point to a valid argument for the CLOAD command, or to a zero byte or colon terminator. A valid argument could consist of a filename sequence (as explained for above CSAVE routine), a question mark (to verify the program on tape against the program in memory), etc. Arguments must be properly terminated with a colon or zero byte. NOTE: This routine does NOT return to the calling program, but instead exits to the BASIC command level ("READY").

3000H - Model III only. Writes 500 baud leader and sync byte. Also places address of slow write routine into cassette write vector at 420CH.

3003H - Model III only. Writes 1500 baud leader and sync byte. Also places address of fast write routine into cassette write vector at 420CH.

3006H - Model III only. Searches for 500 baud leader and sync byte. Also places address of slow read routine into cassette read vector at 420EH.

3009H - Model III only. Searches for 1500 baud leader and sync byte. Also places address of fast read routine into cassette read vector at 420EH.

300CH - Model III only. Turn off cassette (routine at 01F8H jumps here).

300FH - Model III only. Turn on cassette (also blanks video locations normally occupied by asterisks while loading tape).

3042H - Model III only. Prompts the user to set the cassette baud rate (displays "Cass?"). If user enters "L" the 500 baud rate is selected. If user enters "H" or "ENTER" the 1500 baud rate is selected. Cassette baud rate switch at 4211H is set to zero for 500 baud rate, 0DH or 48H ("ENTER" or "H") if 1500 baud selected.

NOTE REGARDING MODEL III CASSETTE READ AND RS-232-C I/O ROUTINES: The Model III allows the user to press the BREAK key to abort a cassette load or RS-232-C input or output, however pressing BREAK in this manner will immediately return the user to the BASIC "READY" prompt. The BREAK key vector used for this purpose is located at 4203H, and normally contains a JP 022EH. 022EH contains an EI (Enable Interrupts) instruction followed by a jump to 1A19H ("READY"). If it is not desirable for the user to be able to return to BASIC, it is suggested that the vector at 4203H be changed to provide the desired result. Note that the location of the stack pointer may be unpredictable, therefore best results may be obtained by plugging the vector with a RET instruction to return to the calling routine.

RS-232-C ROUTINES (MODEL III ONLY)

0050H - Model III only. Receive one character from the RS-232-C interface. If RS-232-C Wait is enabled, this routine waits for a character to be received, or until the BREAK key is pressed. If Wait is not enabled, it returns whether or not a character is received. Character received (if any, or zero if no character) is stored at 41E8H and also in the A register. Uses AF,DE.

0055H - Model III only. Transmit one character to the RS-232-C interface. If RS-232-C Wait is enabled, this routine waits until the character is transmitted, or until the BREAK key is pressed. If Wait is not enabled, it returns whether or not the character was transmitted. On entry, if a non-zero byte is stored at 41F0H, that is the character that will be output to the RS-232-C, otherwise whatever character was stored in the A register on entry will be used. The buffer at 41F0H is cleared after the character is sent. Uses AF,DE.

005AH - Model III only. Initialize the RS-232-C interface. On entry, memory location 41F8H is expected to contain the send and receive baud rate codes (send in the most significant four bits, receive in the least significant four bits), 41FAH contains the Wait/Don't Wait switch (0 = "Don't Wait", any other value indicates "Wait"), and location 41F9H contains the Characteristics switches as follows:

Bits	Meaning
7	Parity (1=Even, 0=Odd)
6,5	Word Length (00=5, 01=6, 10=7, 11=8 bits)
4	Stop Bits (0=One bit, 1=Two bits)
3	Parity On/Off (0=Parity, 1=No Parity)
2	Transmit On/Off (0=Disable, 1=Enable)
1	Data Terminal Ready (0=No, 1=Yes)
0	Request To Send (0=No, 1=Yes)

This routine uses the AF,DE registers.

301BH - Model III RS-232-C Initialization driver address (as found in RS-232-C Initialization DCB).

301EH - Model III RS-232-C Input driver address (as found in RS-232-C Input DCB).

3021H - Model III RS-232-C Output driver address (as found in RS-232-C Output DCB).

I/O ROUTER ROUTINES (MODEL III ONLY, NOT IN MODEL 4)

006CH - Model III only. Change I/O device routing. On entry, memory locations 4222H-4223H are expected to contain the two-letter ASCII abbreviation for the source device, and 4220H-4221H are expected to contain the two-letter ASCII abbreviation for the destination device. Valid two-letter abbreviations are KI (Keyboard Input), DO (Display Output), RI (RS-232-C Input), RO (RS-232-C Output), and PR (PRINTER). Uses AF,DE.

3027H - Model III Vector to the I/O Router driver (jump to same address as is found in I/O Router DCB).

NOTE REGARDING THE MODEL III I/O ROUTER ROUTINES: The I/O router routine has been removed from the Model 4 ROM. If a CALL is made to either of the above routines on a Model 4, there will be a return to the calling routine with no change in device routing.

CHAPTER TWO - ARITHMETIC ROUTINES

NUMBERS AND THE TRS-80

One of the advantages of using the ROM subroutines found within the TRS-80 is that the programmer generally does not need to understand the Microsoft system of floating-point number storage. ROM input routines can be used to translate numbers from human-readable form to the proper integer or floating-point format so that they can be processed by the computer, and ROM output routines can translate the result of our computations back to ASCII format for display purposes.

However, it never hurts to understand the manner in which the BASIC interpreter goes about its tasks. The problem here is that most of the currently available texts do not succeed in explaining the Microsoft number storage systems in a way that can be easily understood (translation: after I read them I was more confused than before). I finally figured out the system, and it isn't really that difficult to understand if you approach it properly. The one assumption that I am making in the following discussion is that the reader understands the binary (base 2) number system. If you're not sure of yourself on that point, try to follow the text anyway and you'll probably figure it out. If you need further assistance in understanding binary numbers, it is highly recommended that you ask for assistance at your local school system (assuming that they teach the "new math"), or at a computer club meeting. Binary is not difficult to learn.

INTEGERS

Each "byte" of memory in the TRS-80 holds eight "bits" (BInary digits). Thus, the largest number that can be represented in one memory location of the TRS-80 is 11111111 binary, or 255 decimal. This is why arguments for the BASIC PEEK and POKE commands must be in the range of 0 to 255. These commands will only operate on one byte of memory at a time. Each individual register of the Z-80 microprocessor also holds only eight bits, or one byte, of memory.

Carrying this idea one step further, if we use two adjacent memory locations, or one of the register pairs of the Z-80 (such as the BC, DE, HL, IX, or IY register pair), we can store a number as large as 1111111111111111 binary (that's sixteen ones), or 65535 decimal. Indeed, this is how the TRS-80 represents BASIC line numbers. We'll call this the POSITIVE INTEGER format.

The difference between this format and the INTEGER PRECISION of the TRS-80 is the ability to represent negative numbers. You may know that if a BASIC variable is defined as an integer, the allowable range for that variable is from -32768 to +32767. We get negative numbers by using the leftmost bit as a sign bit. If the bit is set (a binary "1"), the remaining fifteen digits represent a negative number.

Otherwise, the rightmost fifteen binary digits represent a positive number. As an example, if we have the binary digits

0 1111111111111111

the sign bit indicates a positive number, and the remaining digits represent 32767 decimal. So, the number in decimal is +32767.

All well and good so far. Now, suppose we have the following group of sixteen bits:

1 1111111111111111

Our sign bit is set, indicating a negative number. The remaining digits represent 32767, so we have -32767, right? Sorry about that, it doesn't work that way. There are two ways that we can determine the correct value, and you may pick the one that is easiest for you to use.

The first method is as follows: If the sign bit is set, treat the number as a 16-bit positive integer and subtract 65536 from that value to get the true value. In other words, we already know that sixteen "1" bits represent 65535 decimal in POSITIVE INTEGER format. If we subtract 65536 from 65535, we are left with a total of -1, which is the INTEGER PRECISION value of the binary 1111111111111111.

The second method is this: If the sign bit is set, complement each of the 16 bits (replace each "1" with a "0" and each "0" with a "1"). Then add one to the result. The result of this action will be a positive expression of our negative number. To illustrate this point, consider that binary 1111111111111111 when complemented will equal 0000000000000000. If we then add one, we will have our correct value of (negative) one.

If you have more than 16K of memory in your TRS-80, you probably are already aware of the offset for negative numbers, if you ever do PEEKs or POKEs to high memory addresses. If the address to be referenced by the PEEK or POKE is greater than 32767, you must subtract 65536 from the actual memory address in order to make the PEEK or POKE function properly. This is an example of converting a number from POSITIVE INTEGER format to INTEGER PRECISION as used in the TRS-80.

Just in case this isn't complicated enough for you, there's one more "gotcha" you should be aware of, if you're not already. It is that numbers are stored in memory with the LEAST significant bytes FIRST. In other words, if a number is written in binary as

0111011110001000

and we instruct the TRS-80 to store this number in memory locations 5000H and 5001H, here's what each of these locations will contain:

5000H:	10001000
5001H:	01110111

You guessed it. Just the opposite of what you'd expect. The Z-80 processor was designed this way because it's much easier to work with a number when the least significant bits are accessed first.

SINGLE AND DOUBLE PRECISION NUMBERS

Integer numbers are relatively painless. Single and Double precision numbers are a little different. The first thing to remember is that the rules that apply to integer numbers do not apply to single or double precision numbers. The second thing to remember is that single precision numbers are stored in four bytes of memory (or sometimes two register pairs, such as BC and DE), while double precision numbers take eight bytes of memory. However, the formats for single and double precision are exactly the same, except that double precision numbers have four more bytes. Therefore, anything that is said about single precision numbers also applies equally to double precision, if you change the byte count accordingly. If we talk about "the remaining three bytes" of a single precision number, you can usually figure that the same thing will apply to "the remaining seven bytes" of a double precision number.

A single precision number is stored in memory in the following format:

LSB	2SB	MSB	EXP
-----	-----	-----	-----

A double precision number is stored in this manner:

LSB	6SB	5SB	4SB	3SB	2SB	MSB	EXP
-----	-----	-----	-----	-----	-----	-----	-----

In the above examples, MSB indicates the Most Significant Byte, 2SB is the 2nd most Significant Byte, and so on to the Least Significant Byte (LSB). EXP stands for the Exponent byte.

To decode a number in memory, start with the exponent byte. If this byte contains zero, the number is zero regardless of what the other bytes may contain. Otherwise, take the number found in the exponent byte and subtract 128 to get the actual exponent. Another way to think of this is that the leftmost bit of the exponent byte will be set if the exponent is positive, or will equal zero if the exponent is negative. The rightmost seven bits contain the actual exponent value, which can be read "as is" for a positive exponent. If the exponent is negative, the rightmost seven bits must be complemented, and then have a value of one added, to produce the proper negative exponent. It's easier to just subtract 128 decimal (80H) from the value found in the exponent byte. Note that if the exponent byte contains 128 decimal, our exponent will be zero (which does NOT mean that the number is zero - for that the exponent itself must be zero).

Next, look at the Most Significant Byte. If the leftmost bit of that byte is set, the number is negative, but if it's a zero, the number is positive (note that we are talking about the number itself here, whereas before we were dealing with the sign of the exponent). Now that you know the sign of the number, treat the leftmost bit as if it were set (a binary "1") and write down the bits of the MSB.

Continue to write out the bits of the remaining bytes, until you have written out the LSB. Place a decimal point to the left of the resulting binary number. At this point, your number should be written out in this format:

.lbbbbbbb bbbbbbbb bbbbbbbb (...continue if Double Precision)
 MSB 2SB LSB (3SB)

Again note that the first digit of the MSB (which contained the sign bit) has been changed to a one.

Next, take the exponent value and move the decimal point that many places. If the exponent was positive, move the decimal point that number of places to the right, while a negative exponent moves the decimal point to the left (in which case you'll have to add leading zeros as required).

Let's back up just a bit. Suppose we had found the value 84H in the exponent byte. After subtracting the 80H offset, we wind up with an exponent of +4. So, we would move the decimal point four places to the right, like this:

lbbb.bbbb bbbbbbbb bbbbbbbb (....)

If the value in the exponent byte had been 7BH, after subtracting 80H we would have an exponent of -5. So, our number would look like this:

.000001bb bbbbbbbb bbbbbbbb bbbb...

Now we have a binary number that we can convert to decimal. Oh, joy! How in the heck do we do that when there's a decimal point (excuse me, BINARY point) in the darn thing?

Simple enough. You probably already know how to convert the bits on the left side of the decimal point. As an example, let's take the binary number 10111001. If we write it out like this...

1	0	1	1	1	0	0	1
128	64	32	16	8	4	2	1

we can take the decimal values below each "1" in the binary number and add them to come up with the decimal number. In this case, $128+32+16+8+1=185$ decimal. The pattern is obvious - each time we move left one digit that digit is "worth" twice as much (if we had a nine-digit binary number, the leftmost digit would count for two times 128, or 256. A tenth digit would be the equivalent of 512 decimal, and so on). Notice that this also works in reverse - each digit to the right is "worth" half the value of its neighbor to the left. This gives us the key to decoding binary numbers that have a fractional part. Suppose we want to decode the binary number 10110.01101. Here's how it's done:

1	0	1	1	0	.	0	1	1	0	1
16	8	4	2	1		$1/2$	$1/4$	$1/8$	$1/16$	$1/32$
						.5	.25	.125	.0625	.03125

Note the pattern. Here we'd add $16+4+2+.25+.125+.03125$ for a result of 22.40625.

As an outlet to our masochistic tendencies, let's actually decode a single precision number from memory. Using VARPTR on a single-precision variable, we come up with a memory location. A PEEK at that and the next three memory locations yields the following values:

183	209	28	134
(LSB)	(2SB)	(MSB)	(EXP)

We know which byte is which (everything's in typical Z-80 reverse order), so we start with the exponent. We subtract 128 from the value of 134 which gives us an exponent of +6. So far, so good.

Next, we write out the MSB as a binary number. 28 decimal is 00011100 binary. That first (leftmost) bit is a zero, so we know the number is positive. We assign a 1 to that bit (we ALWAYS replace the sign bit with a "1", remember?), giving us an MSB of 10011100. Now, we convert the 2SB and LSB. 209 decimal = 11010001 binary and 183 decimal = 10110111 binary. Putting them all together, with the decimal point in front, gives us this:

.10011100 11010001 10110111

Now, let's see here. Our exponent was +6, so we'll have to move the decimal point six places to the right - which gives us this:

100111.00 11010001 10110111

On the left side of the (binary) point, we have 100111. That's easily converted to 39 decimal. How about our fractional part? If you followed the chart above, you can easily count off places to the right of the decimal point. The first ten would look like this:

0	0	1	1	0	1	0	0	0	1
$1/2$	$1/4$	$1/8$	$1/16$	$1/32$	$1/64$	$1/128$	$1/256$	$1/512$	$1/1024$

If we add the values for the first ten digits, we get a value of .204102, which when added to 39 gives us 39.2041 (approximately). The actual number is 39.2048, so you can see that the number of bits we use has a great effect on accuracy.

Well, that is a (relatively) painless explanation of the number formats of the TRS-80 (I hope!). You may not know how to convert numbers from one base to another by counting on your toes, but at least you should have some idea of how the TRS-80 allocates memory for variables. Usually, the math routines will take care of juggling the numbers for you, and you'll probably seldom use numbers other than integers in your assembly-language programs, anyway (and integers are downright easy to understand, at least when compared to the other formats). But as I said, it never hurts to know as much as possible about why the electrons chase around the innards of your computer in just a certain way.

STRING VECTORS

String vectors? What are strings doing in the arithmetic portion of this book? Well, in some respects the TRS-80 treats string variables in the same manner as numeric variables. So we need to know how strings are stored in memory.

The first thing we need to know is that the string itself may be stored in high memory (just below any protected memory), or it may be part of a BASIC program. That's not important right now. What IS important is how the TRS-80 finds these strings when needed. It's done by use of a three-byte STRING VECTOR, which is organized as follows:

First byte:	Length of string (number of bytes)
Second byte:	LSB of starting address of string
Third byte:	MSB of starting address of string

Note that the information contained in the string vector (the starting address of the string, and the length of the string) is sufficient to determine the exact position of the string in memory. When the BASIC VARPTR function is used on a string variable, the address returned is that of the first byte of the string vector (the string length byte). Note that because only one byte is allotted for the length, the maximum string length is limited to 255 characters.

The VARPTR address of a string is important in its own right. This two-byte pointer to the string vector is used by all of the string-handling functions. We'll cover these string functions in much greater detail in chapter three of this book. For now, just remember that for each string, there is a two-byte pointer (the VARPTR), which points to the three-byte STRING VECTOR, of which the last two bytes point to the string itself.

NUMBER TYPES

The TRS-80 uses a single digit as a number type indicator. This digit is the number of bytes required to store a given type of number (that is, 2 for an integer, 4 for a single-precision number, or 8 for a double-precision number), or in the case of a string, the number of bytes required for the string vector (3). The number type indicator digit figures quite prominently in all TRS-80 arithmetic operations. The number type of the number currently being processed by the TRS-80 can be found in the NUMBER TYPE FLAG (or NTF) which is located at 40AFH. This flag is generally associated with the number currently stored in the primary ACCUMULATOR. You will see many references to the NTF among the following routines, so keep in mind that NTF stands for NUMBER TYPE FLAG, and that it is located at 40AFH.

ARITHMETIC ACCUMULATORS

The two arithmetic accumulators in the TRS-80 memory are located from 411DH to 4124H, and from 4127H to 412EH. Numbers are stored in the accumulators during computations. The method of storage of a given

number depends on its number type, and on which accumulator is being used.

The primary accumulator - the one that is used in most calculations, hereinafter referred to as ACCUM - is the one located from 411DH to 4124H. Here is how the various number types are stored in ACCUM:

ADDRESS	INTEGER	SINGLE	DOUBLE	STRING
411DH			LSB	
411EH			6SB	
411FH			5SB	
4120H			4SB	
4121H	LSB	LSB	3SB	VARPTR LSB
4122H	MSB	2SB	2SB	VARPTR MSB
4123H		MSB	MSB	
4124H		EXP	EXP	

Note that storage for all number types except double-precision begins at 4121H.

The second accumulator (hereinafter referred to as ACCUM2) is the one located from 4127H to 412EH. This accumulator is mostly used in double-precision arithmetic, but regardless of the precision, when ACCUM2 is used the LSB of the number will be found at 4127H. The MSB of the number will be at 4128H for an integer, while the exponent byte will be found at 412AH or 412EH for a single or double precision number respectively.

The byte just prior to the beginning of each of the accumulators (411CH and 4126H) is used as an extension to the respective accumulator during certain arithmetic operations. The byte at 4125H is sometimes used to hold the sign of the result of certain arithmetic operations.

The two arithmetic accumulators are not the only places where the operands of a calculation can be stored. For example, an integer might be placed in either the HL or the DE register pairs during computations. A single precision number might have its four bytes placed in the BC and DE register pairs, or it might be temporarily placed on the stack. Multiplication and division operations may make use of a working accumulator located at 414AH - 4151H.

Some of the TRS-80 ROM arithmetic routines can directly access a single-precision number stored anywhere in memory - in this case the HL register pair must contain the address of the LSB of the number. This is indicated by the use of (HL) in the following routines, which actually means the contents of the four bytes beginning with the address pointed to by HL (that is, (HL), (HL+1), (HL+2), and (HL+3)).

ROM ROUTINES

Now that we've had some background regarding the various number formats and storage locations, we can examine the various ROM routines. One nice thing about the arithmetic routines is that ALL

arithmetic routines are found in the same locations on both the Model I and the Model III (and, of course, on the Model 4 when operated in the "Model III mode"). You can use any of the following routines with confidence, secure in the knowledge that the math routines seem to be firmly entrenched in their present memory locations, at least for the time being.

Please note that when a routine is said to work with "ANY" precision number, that means it will work with an integer, or a single- or double-precision number. It does NOT mean that it will work with a string.

BASIC ADDITION, SUBTRACTION, MULTIPLICATION, AND DIVISION ROUTINES

ADDRESS	OPERATION	PRECISION
0BD2H	ACCUM = DE + HL	INTEGER
0BC7H	ACCUM = DE - HL	INTEGER
0BF2H	ACCUM = DE * HL	INTEGER
2490H	ACCUM = DE / HL	INTEGER
0716H	ACCUM = BC:DE + ACCUM	SINGLE
0713H	ACCUM = BC:DE - ACCUM	SINGLE
0847H	ACCUM = BC:DE * ACCUM	SINGLE
08A2H	ACCUM = BC:DE / ACCUM	SINGLE
0C77H	ACCUM = ACCUM + ACCUM2	DOUBLE
0C70H	ACCUM = ACCUM - ACCUM2	DOUBLE
0DALH	ACCUM = ACCUM * ACCUM2	DOUBLE
0DE5H	ACCUM = ACCUM / ACCUM2	DOUBLE
0708H	ACCUM = ACCUM + 0.5	SINGLE
070BH	ACCUM = (HL) + ACCUM	SINGLE
0710H	ACCUM = (HL) - ACCUM	SINGLE
089DH	ACCUM = STACK / (HL)	SINGLE
08A0H	ACCUM = STACK / ACCUM	SINGLE (see note below)
093EH	ACCUM = ACCUM * 10	SINGLE
0E4DH	ACCUM = ACCUM * 10	DOUBLE
0F0AH	ACCUM = ACCUM * 10	SINGLE OR DOUBLE (see note below)
0897H	ACCUM = ACCUM / 10	SINGLE
0DDCH	ACCUM = ACCUM / 10	DOUBLE
0F18H	ACCUM = ACCUM / 10	SINGLE OR DOUBLE

The following two routines operate on the MANTISSA ONLY (exponent byte ignored):

0D33H	ACCUM = ACCUM + ACCUM2	DOUBLE
0D45H	ACCUM = ACCUM - ACCUM2	DOUBLE

The following register multiply routine jumps to the Bad Subscript error routine if the result of the multiplication is greater than FFFFH:

0BAAH DE = BC * DE POSITIVE INTEGER

NOTE: Calling the multiply-by-ten routine at 0F0AH results in an immediate RETurn (without performing the indicated multiplication) if the Z flag is set upon entry. If this is not desirable, use 0F0BH as the entry point for this routine. Also, prior to the use of any of the division routines (especially the one at 08A0H), be sure to read the note at the end of the following section (covering the BASIC arithmetic function routines).

BASIC ARITHMETIC FUNCTION ROUTINES

ADDRESS	OPERATION	PRECISION
0809H	ACCUM = LOG (ACCUM)	ANY (RESULT SINGLE)
0977H	ACCUM = ABS (ACCUM)	ANY
098AH	ACCUM (& HL) = SGN (ACCUM)	ANY (RESULT INTEGER)
0B26H	ACCUM = FIX (ACCUM)	ANY
0B37H	ACCUM = INT (ACCUM)	ANY
13E7H	ACCUM = SQR (ACCUM)	ANY (RESULT SINGLE)
1439H	ACCUM = EXP (ACCUM)	ANY (RESULT SINGLE)
14C9H	ACCUM = RND (ACCUM)	ANY (RESULT SINGLE - see note below)
1541H	ACCUM = COS (ACCUM)	ANY (RESULT SINGLE)
1547H	ACCUM = SIN (ACCUM)	ANY (RESULT SINGLE)
15A8H	ACCUM = TAN (ACCUM)	ANY (RESULT SINGLE)
15BDH	ACCUM = ATN (ACCUM)	ANY (RESULT SINGLE)

NOTE: Although the number in ACCUM may be any precision when using the RND function (at 14C9H), it is immediately converted to an integer upon entry. Therefore, an OV Error will result if the number is not in the range -32768 to 32767.

The following is a partial listing of other available functions and some alternate entry points for a few of the above functions:

01D3H	RANDOM (CHANGE RANDOM NUMBER SEQUENCE)	
0778H	ACCUM = 0 (ZERO EXPONENT)	SINGLE OR DOUBLE
0982H	ACCUM = -ACCUM	SINGLE
0B3DH	ACCUM = INT (ACCUM)	SINGLE
0B59H	ACCUM = INT (ACCUM)	DOUBLE
0C5BH	ACCUM = ABS (ACCUM)	INTEGER
13F2H	ACCUM = STACK ↑ ACCUM	SINGLE (ACCUM WILL BE CONVERTED TO SINGLE PRECISION-see note below)
13F5H	ACCUM = STACK ↑ ACCUM	SINGLE (ACCUM MUST BE SINGLE - see note below)
13F7H	ACCUM = BC:DE ↑ ACCUM	SINGLE
14CCH	ACCUM = RND (NUMBER IN HL)	SINGLE
14F0H	ACCUM = RND (0)	SINGLE

The following two routines perform logical operations on register pairs HL and DE. However, on exit the LSB of the result is stored in the L register and the MSB is stored in the A register. To obtain the

complete result in the HL register pair the user should immediately LD H,A upon return from the routine.

25F7H	A , L = HL OR DE	INTEGER
25FDH	A , L = HL AND DE	INTEGER

NOTE: When any routine is used that takes one of its values from the stack (such as the exponentiation routines at 13F2H or 13F5H or the division routine at 08A0H), the value stored on the stack is expected to be on the topmost two levels of the stack when the routine is entered, with the RET address below this value. For this reason, these routines cannot be simply CALLED in the same manner as most of the other routines mentioned here. Instead, if you wish to use one of these routines you must first PUSH the RET address onto the stack, then use the routine at 09A4H or some other method to get the required value onto the stack, and finally jump to the desired routine (don't CALL it). Another way to accomplish this would be to set up a subroutine, within which the proper value is placed onto the stack and which terminates with a jump to the desired routine (instead of a RET instruction). The following exponentiation subroutine illustrates this construct, and assumes that the base number is stored in the ACCUM in single precision format on entry:

```
EXPON    CALL    09A4H ;Move base number from ACCUM to stack
          (code to place exponent in ACCUM goes here)
          JP     13F2H ;(or 13F5H)
```

The above subroutine may be CALLED and will properly return to the caller most of the time. The one exception is something that all disk system users should be aware of when using ANY of the exponentiation routines, or any of the other math routines that make use of the non-integer division subroutine in the ROM. If your program runs under a disk operating system and BASIC has not been initialized at least once since power-up (or if BASIC's reserved RAM has been overwritten or otherwise mucked up), use of one of these routines will likely send your program off on a trip to a galaxy far, far away. The reason is that when BASIC is initialized, a subroutine is moved from the ROM to the reserved RAM at 4080H (see the above discussion on 4080H). If this subroutine is not present and the ROM attempts to make the CALL to 4080H (from the non-integer division subroutine), almost anything can happen. Unless you are sure that this subroutine at 4080H will always be present when your program is run, I suggest that you include within your program initialization the following segment of code, which does a block move of ROM locations 18F7H through 1904H to RAM locations 4080H through 408DH:

```
LD      HL,18F7H
LD      DE,4080H
LD      BC,000EH
LDIR
```

CHANGING PRECISIONS OF NUMBERS

The following routines are all used in the process of converting a number from one precision to another:

ADDRESS	OPERATION
=====	=====
0A8AH	ACCUM FROM SINGLE TO INTEGER (RESULT ALSO IN HL - see note below)
0AB9H	ACCUM FROM DOUBLE TO SINGLE
0ACCH	ACCUM FROM INTEGER TO SINGLE
0AE3H	ACCUM FROM SINGLE TO DOUBLE
0964H	INTEGER IN A TO SINGLE (RESULT IN ACCUM - see note below)
0ACFH	INTEGER IN HL TO SINGLE (RESULT IN ACCUM)
2B05H	NUMBER IN ACCUM TO INTEGER (RESULT ALSO IN DE, Z FLAG SET IF NUMBER IN RANGE 0 TO 0FFH).

NOTES: When the routine at 0964H is used to convert the number in the A register to single precision format, the most significant bit of the number is considered a sign bit. Therefore, if A contains 80H or greater, the result will be a negative number. Also, note that use of the single precision to integer conversion routine at 0A8AH normally generates an OV ERROR if the result is out of range. If you do not want to jump to the BASIC error message in the event of an OV error, follow this procedure: First, PUSH the desired return address (where the routine should return if there is NOT an error) onto the stack, then CALL 0A8EH. The instruction immediately following the CALL should be the first instruction of your error-handling routine (this is where the routine will return to if there IS an OV Error).

0A7FH	ACCUM = CINT (ACCUM) (Uses routine at 0A8AH - see note above)
0AB1H	ACCUM = CSNG (ACCUM)
0ADBH	ACCUM = CDBL (ACCUM)

NUMBER TYPE FLAG MANIPULATION & TESTING

ADDRESS	OPERATION
=====	=====
0020H	TEST NTF. RST 20H calls routine (jumps to 4009H which in turn jumps to 25D9H). Operates by loading A register with contents of NTF, subtracting three, and then executing an OR A instruction. Flags are set as you would normally expect from this operation, EXCEPT that the C flag is always set UNLESS the number is double precision (NTF=8).
0A9DH	SET NTF FOR INTEGER (NTF=2)
0AECH	SET NTF FOR DOUBLE PRECISION (NTF=8) USES BC.
0AEFH	SET NTF FOR SINGLE PRECISION (NTF=4)

ARITHMETIC COMPARE ROUTINES

NOTE: In the following routines, when a subtract operation is shown, it means that the flags in the Z-80 are set as if this subtraction has taken place. However, on return both numbers will be unchanged.

ADDRESS	OPERATION
=====	=====

CHECK ACCUM FOR SIGN ROUTINES: The following three routines return -1, 0, or +1 in the A register (flags set accordingly) depending on the sign of the number in ACCUM.

0994H	Check ACCUM for sign - ANY PRECISION (Requires NTF)	
099BH	Check ACCUM for sign - INTEGER ONLY	
0955H	Check ACCUM for sign - SINGLE OR DOUBLE PRECISION	
0018H	POSITIVE INTEGER compare	HL - DE (Z set if equal, C set if HL < DE). RST 18H calls routine (0018H jumps to 4006H which in turn jumps to 1C90H where routine is located). Used to compare BASIC line numbers, etc.
0A0CH	SINGLE PRECISION compare	ACCUM - BC:DE
0A39H	INTEGER compare	HL - DE
0A49H	DOUBLE PRECISION compare	ACCUM - (DE)
0A4FH	DOUBLE PRECISION compare	ACCUM - ACCUM2
0A78H	DOUBLE PRECISION compare	ACCUM2 - ACCUM
09DFH	COMPARE SIGN & SET SIGN BITS	ACCUM compared to BC:DE. Bit 7 of the MSB of both numbers is set (making both numbers negative). Then the original Bit 7 (sign) bits of both numbers are compared. If both numbers had the same sign on entry (both were positive or both were negative) the Z-80 sign flag will be set (M flag set), otherwise the sign flag will be reset (P flag condition).

ARITHMETIC MOVE ROUTINES

ADDRESS	OPERATION
=====	=====

09A4H PUSH ACCUM

Place the single precision value at 4121H on the stack (4121H-4122H PUSHed first, then 4123H-4124H).

09B1H ACCUM = (HL)

Copy the single precision value stored at the address pointed to by HL (four bytes) to 4121H.

09B4H ACCUM = BC:DE

Store the single precision value in BC:DE at 4121H (DE to 4121H-4122H, BC to 4123H-4124H). Destroys value in BC:DE.

09BFH BC:DE = ACCUM

Load the single precision value at 4121H into BC:DE (4121H-4122H to DE, 4123H-4124H to BC).

09C2H BC:DE = (HL)

Load the single precision value at the address pointed to by HL (four bytes) into BC:DE (E=(HL), D=(HL+1), C=(HL+2), B=(HL+3)).

09CBH (HL) = ACCUM

Copy the single precision value at 4121H to the address pointed to by HL (four bytes).

09CEH (HL) = (DE)

Copy the single precision value from the address pointed to by DE to the address pointed to by HL (four bytes).

09D2H (DE) = (HL)

Copy value from the address pointed to by HL to the address pointed to by DE (number of bytes moved = NTF).

09D3H (HL) = (DE)

Copy value from the address pointed to by DE to the address pointed to by HL (number of bytes moved = NTF).

09D6H (HL) = (DE) Can be used for BLOCK MOVES (see note below)

Copy value from the address pointed to by DE to the address pointed to by HL (number of bytes moved in A register).

09D7H (HL) = (DE) Can be used for BLOCK MOVES (see note below)

Copy value from the address pointed to by DE to the address pointed to by HL (number of bytes moved in B register).

09F4H ACCUM = ACCUM2

Copy any precision value from ACCUM2 to ACCUM. Precision determined by NTF.

09F7H ACCUM = (HL)

Copy any precision value from the address pointed to by HL to ACCUM (precision and number of bytes moved determined by NTF).

09FCH ACCUM2 = ACCUM

Copy any precision value from ACCUM to ACCUM2. Precision determined by NTF

09FFH (HL) = ACCUM

Copy any precision value from ACCUM to the address pointed to by HL (precision and number of bytes moved determined by NTF).

0A9AH ACCUM = HL

Copy the integer value in HL to 4121H-4122H and set NTF for integer (NTF=2)

0AFBH B,C,D,E = A IF A = 0

If (and only if) the A register contains zero, copy the value in A to the B, C, D, and E registers and return with Z flag set.

NOTES: Routines that copy any precision value will also copy string VARPTRs (NTF=3), however, three bytes will be copied even though only the first two are required. Also, the routines at 09D6H and 09D7H can be used as general-purpose block move subroutines to move up to 256 bytes from one part of memory to another, by simply placing the starting address of the block to be moved in DE, the destination address in HL, and the number of bytes to be moved in the A or B register (depending on which routine is used). While in most cases it

is easier and MUCH faster to use the Z-80 block move instructions (LDIR or LDDR), there may be an unusual circumstance that would warrant using one of the above routines for block move purposes.

NUMERIC STRINGS AND I/O ROUTINES

It is very fortunate that we do not have to enter numbers into the TRS-80 in the format that it uses for numeric computations. Imagine the difficulty of entering numbers in integer or single- or double-precision formats! Instead, we can give the computer a string of ASCII digits, and it will do the work of converting them to the proper format, doing the arithmetic, and converting the result back to an ASCII string so we can read it. The routines that follow are those used in the ASCII string conversion process.

0E65H LOAD DOUBLE PRECISION ASCII CONSTANT TO ACCUM

Same as following routine (at 0E6CH) except that number will always be returned in double precision format.

0E6CH LOAD ASCII CONSTANT TO ACCUM

Evaluate a numeric string that begins at the address pointed to by the HL register pair, store it in ACCUM and set the NTF. This routine stops as soon as it encounters a character that is not part of the number (it will return a value of zero if no valid numeric characters are found). It will accept signed values in Integer, Real or Scientific Notation. Number returned will be in integer format if possible, else single precision unless the string has over seven digits (not including exponent), in which case number will be returned as double precision.

0FA7H DISPLAY "IN " AND POSITIVE INTEGER STORED IN HL

Used by BASIC to display messages such as "BREAK IN nnnnn", where nnnnn is the BASIC line number stored in the HL register pair. Saves HL, calls routine at 28A7H to display "IN ", then restores HL and continues with routine at OFAFH (see details of that routine below, and note warning about use of routine at 28A7H).

0FAFH DISPLAY POSITIVE INTEGER STORED IN HL

Converts the positive integer (such as a BASIC line number) stored in HL to decimal ASCII character string, which is then output to device specified by byte at 409CH (0FFH=Tape, 0=Video, 1=Printer). Updates cursor position (if output to video). This routine jumps to the display routine at 28A7H (a description of that routine was given in chapter one of this book) and thus may require that the associated Disk BASIC links be "plugged".

0FBDH PREPARE NUMBER IN ACCUM FOR DISPLAY (NUMERIC EDIT)

Non-formatted numeric edit routine. Converts number in ACCUM to display format. On exit, ASCII string is stored in buffer located at 4130H-4149H, and HL register will point to last leading space (that is, the space just before the first non-space character of the created string). When positive integers greater than zero (such as BASIC line numbers) are converted, the area from 4130H to 4136H is used, and the string is stored right justified with leading spaces in locations 4130H to 4135H (4130H always contains a space character), while 4136H

always contains a zero byte to terminate the string. Other number types may be converted and stored differently within the buffer.

0FBEH FORMATTED NUMERIC EDIT

Formatted numeric edit routine (as used by PRINT USING). Entry requirements are as follows:

B register - must equal number of whole digits (digits to left of decimal point). Include commas (if desired) in count, but exclude space and sign characters.

C register - must equal number of digits desired on right of decimal point plus one (for the decimal point itself).

A register - flags. Each bit has a meaning if set, as follows:

BIT	MEANING (IF SET)
7	perform edit
6	include commas
5	asterisk fill
4	precede by dollar sign
3	force sign (+ or -)
2	trailing sign
0	scientific notation

On exit, HL points to the leading space character just before the start of the resulting string, and if the number was single- or double-precision, DE points to end of resulting string plus one.

NOTE: The routines at 0FBDH and 0FBEH are the same routine. The instruction at 0FBDH clears the A register so no formatting is done.

1E46H LOAD POSITIVE INTEGER EXPRESSION IN DE

Same as following routine (at 1E5AH) except that HL may also point to any valid BASIC variable or expression, which must evaluate to a number in range 0 to 32767 decimal (used to get argument of BASIC CLEAR command). Note that this routine assumes that BASIC is operational. Results may be unpredictable if this is not the case, particularly if a BASIC variable name (or something that could be interpreted as such) is part of the string.

1E5AH LOAD POSITIVE INTEGER CONSTANT IN DE

Evaluate the string at the address pointed to by the HL register pair for a positive integer value (such as a BASIC line number), stopping at the first non-numeric character. The result is returned in the DE register pair. A value of zero is returned if no numeric value is found. Maximum allowable number is 65529 decimal. NOTE: If this routine is being used to obtain a BASIC line number, it may be CALLED at 1E4FH. In this case, a period (2EH) character at (HL) will cause the routine to return with the "current" BASIC line number (as stored at 40ECH-40EDH).

21E3H ASSIGN STRING(S) TO BASIC VARIABLE(S)

This routine is part of the BASIC INPUT command routine, and can be used to process input obtained from CALLing one of the keyboard input routines at 1BB3H, 0361H, etc. On entry, BC must point to the first character of a string that contains the variable name(s) (if more than one variable name is used the names must be separated by commas, and a

zero byte or colon must be placed after the last variable name). HL must point to the byte just prior to the beginning of the input string (this is where it is placed by the above-mentioned input routines - note that this byte is altered by this routine), and the string may contain input for more than one variable (items must be separated by commas). String must be terminated with zero byte. On exit, the numbers or strings in the input will be assigned to the variables listed in the string containing the variable names. Numeric or string variables may be used, but only valid characters for numeric input may be used with numeric variables (depending on how certain flags are set, a "?REDO" message or an error will occur if this rule is violated. To force the "?REDO" message, make sure that memory location 40DEH contains zero, and that location 40A9H contains a non-zero value. To determine if the "?REDO" message has occurred, prior to calling the routine load memory locations 40E6H-40E7H with zero. If the HL register pair contains zero on exit, then an error has occurred and the "?REDO" message has been printed). NOTE: For proper operation, the input string should be in the BASIC input buffer (as it will be if one of the input routines mentioned above is used) when this routine is CALLED. See chapter one of this book for information on the INPUT routine located at 21C9H.

2337H EVALUATE EXPRESSION AT (HL)

Evaluate BASIC string expression (may include constants, BASIC variables, BASIC functions, operators, etc.) and place result in ACCUM (also set NTF). On entry, HL must point to first character of the string to be evaluated. On exit, HL will point to the string delimiter, which must be an acceptable BASIC expression terminator (such as a zero byte, a colon, a right parenthesis, etc.). NOTE: This routine may be entered at 2335H, in which case the HL register pair MUST point to a left parenthesis (which precedes the expression to be evaluated) or a BASIC syntax error will result. This routine assumes that BASIC is operational - result may be unpredictable if this is not the case.

252CH EVALUATE PARENTHESIZED EXPRESSION

Evaluate any valid BASIC expression enclosed in parenthesis. Calls previous routine (at 2335H to check for right parenthesis). A BASIC syntax error will result if expression is not terminated with a right parenthesis.

2540H LOAD ACCUM WITH VALUE OF BASIC VARIABLE

Get value of BASIC variable and put in ACCUM (also put precision of variable in NTF). On entry, HL must point to first character of variable name. On exit, HL will point to first character following variable name.

260DH LOCATE OR CREATE A BASIC VARIABLE

This routine will locate the storage area in memory for an existing BASIC variable, or will assign a storage area for the specified variable if one does not presently exist. On entry, the HL register pair must point to the first character of the variable name. On exit, HL will point to the next character following the variable name, and DE will contain the address of the variable storage area (same as would be returned in BASIC VARPTR function). NOTE: If the variable name does not contain a type declaration character (!, %, #, or \$ as

the final character of the variable name), the variable will be set to the precision as defined by the variable type declaration table at 4101H through 411AH. This table is organized so that location 4101H contains a the variable type flag for variables that begin with the letter "A", 4102H contains the flag for variables that start with "B", 4103H contains the flag for variables starting with "C", and so on. The flag digits are defined in the same manner as the NTF (integer = 2, string = 3, single = 4, double = 8). All table locations are set to 4 (single precision) on power-up and when the BASIC RUN or CLEAR commands are executed. The table values are altered through use of the BASIC DEF commands (such as DEFINT, DEFSNG, DEFDBL, or DEFSTR).

2B02H LOAD INTEGER EXPRESSION IN DE

Same as routine at 1E46H (described above) except allows negative argument. Number must evaluate within range -32768 to 32767.

CHAPTER THREE - STRINGS AND STRING-HANDLING ROUTINES

STRING STORAGE IN THE TRS-80

Most of us at one time or another have given at least a passing glance to the appendix in the Level II BASIC reference manual entitled "LEVEL II TRS-80 MEMORY MAP". This chart shows how memory is utilized in the TRS-80. A portion of the map that is of particular interest to us, at least for the purpose of understanding string storage in the TRS-80, is reprinted (and expanded upon) here:

START OF USER MEMORY (The default addresses in non-disk systems are as follows: Model I: 17129 decimal, 42E9 hex. Model III: 17383 decimal, 43E9 hex.)

PROGRAM TEXT - where your BASIC program resides in memory

SIMPLE VARIABLES - storage for non-array variables

ARRAYS - storage for array variables

***** FREE MEMORY (Unused by BASIC) *****

STACK - location of the stack while running BASIC

STRING SPACE - for storage of strings

MEMORY RESERVED AT POWER UP (by MEMORY SIZE? answer)

END OF ACTUAL MEMORY (that's all, folks!)

Now, what is interesting about the way memory is allocated is that everything builds toward the free space. You load your BASIC program, then the variables are placed above it. If you have a large array and define a new "simple" variable, ALL of the array elements must be moved up to accomodate the addition (which will be placed at the end of the present list of "simple" variables). Should a new array be defined, it will be placed at the end of the array variable list (in other words, in what is now "free memory").

If we start at the top of memory and work our way down, we have a similar situation. First we find any "protected" memory that might have been reserved in answer to the MEMORY SIZE? question, then just below that is our string space. 50 bytes of string space are reserved on power-up, and we may change that amount at any time by using a "CLEAR n" statement, where n is the number of bytes of string space we want to reserve. Finally we come to our stack, which is located just below the string space and expands as necessary into "free memory". Note that if we change the reserved string space by using a CLEAR statement, the stack is also relocated. Should the top end of the variables get too close to the bottom of the stack (or vice-versa), the BASIC umpire cries "foul!" and we get an Out of Memory Error (OM ERROR) message.

Whenever a "string" is created, it is normally stored in one of two places. Usually, it is stored in the string space area near the top of memory. The exception to this is when a string constant is defined in a BASIC statement, either through a LET type of statement (such as A\$="STRING"), or through a READ statement where the string is contained in a DATA statement. In either case, the string vector will point to the portion of the BASIC program that contains the string, thus avoiding unnecessary duplication of strings in memory.

Strings that are not part of the BASIC program are stored in the string space area, beginning as close to the top of memory as possible. The last character of the first string to be defined will be stored in the highest unprotected memory location. Each new string to be created will be stored just below the last, until string space is full, at which point a "garbage collection" will be performed in order to get rid of unneeded strings. If this operation is unable to free enough string space for the next string, an Out of String space Error (OS ERROR) results.

I hear you asking "But why should there be unneeded strings in the string space?" The answer is that when a new value is assigned to an existing string variable, the old string is not erased from memory - it is simply abandoned by BASIC. Any new strings will still be placed below the abandoned string, until a "garbage collection" is performed.

Note that a string may be created without the programmer realizing it or intending for it to happen. For example, the PRINT statement actually creates a string of the items to be printed, then outputs the string. BASIC string functions often create intermediate strings, and the programmer may wish to alter his programming techniques to avoid this. Here are two ways to achieve the same result - namely, a count of the number of characters in A\$ minus two:

```
PRINT LEN(MID$(A$,3))      PRINT LEN(A$)-2
```

Both statements would print the same result, but the first would create an extra string in the process (a string consisting of all but the first two characters of A\$).

If you have a BASIC program that uses a lot of string space (string arrays, for example), you will want to try to avoid the "garbage collection", or at least forestall it as long as possible. The reason is that a "garbage collection" can take as long as several MINUTES, during which the computer will appear to be "dead" (it will not even respond to the BREAK key). Note that using the FRE(X\$) function automatically performs the "garbage collection" (X\$ is the "dummy" variable name and may be replaced by any valid STRING variable name).

This "garbage collection" is one of the major factors that make string sorts in BASIC seem unbearably slow. Consider the following BASIC statements, which are commonly used to exchange two array variables during a sort:

```
Z$=A$(X):A$(X)=A$(X+1):A$(X+1)=Z$
```

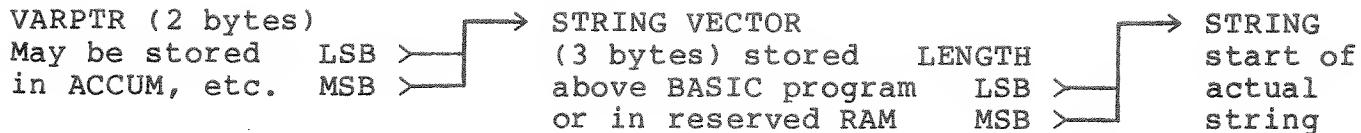
This simple exchange creates THREE new strings - first, a new string (a duplicate of A\$(X)) is created for Z\$. Then, a duplicate of A\$(X+1) is created for A\$(X). And finally, a duplicate of Z\$ is created for A\$(X+1). Note also that the original strings for each of these three variables are abandoned during the exchange. At this rate (three abandoned strings for each exchange!), it's easy to see why it takes seemingly forever to sort even a few strings in BASIC. Smart programmers have been known to write routines to switch the string

pointers (instead of the strings themselves) in order to avoid this problem.

If all of these strings are sitting up there in string space, how does BASIC find the right one when you want the value of a certain one (say X\$, for example)? To answer this, remember that simple variables are stored just above the BASIC program text. In the case of an integer, single-precision, or double-precision variable, the two, four, or eight bytes of the variable itself are stored here, but in the case of a string, the three-byte STRING VECTOR (discussed in chapter two of this book) is stored here. The last two bytes of the string vector point to the string itself. Temporary string vectors are also created for strings created by BASIC, and these are stored in reserved memory, in the string work area starting at 40B5H.

The BASIC VARPTR function points to the first byte of the string vector. The system uses a temporary VARPTR for each temporary vector - this two-byte pointer points to the three-byte string vector. You may find this temporary VARPTR used in the ACCUM (4121H-4122H), among other places.

This may seem a bit confusing, as we have a pointer that points to a string vector, which is actually another pointer that points to the string itself. Here's how it all fits together:



In the remainder of this chapter, when we refer to the VARPTR we will be talking about the two-byte pointer to the string vector, as shown above. This string vector may not always be connected with an actual BASIC string variable, but instead may be associated with a "temporary" string that is not assigned to any BASIC variable. Thus, in one sense, the use of "VARPTR" may be misleading. However, since this VARPTR points to the first byte of the string vector, just as the BASIC VARPTR function does for BASIC string variables, we will continue to use the term VARPTR in this manner.

USING THE ROM STRING FUNCTIONS

You may find very limited use for the ROM string function calls, for three reasons. First, unless you are writing a "hybrid" program (part BASIC and part machine language), you probably will not have string space reserved and properly maintained by the system. Since the BASIC functions require that this string space be available, you may have some difficulty getting the function calls to work properly (don't be afraid to experiment, though). Second, in order to use the ROM calls your strings will usually have to have the three-byte string vectors (in the proper format) stored at some location, and you will have to supply the starting address of these vectors (in other words, the VARPTR) to the ROM. And third, it is not as easy to use most of the ROM string functions as it is the arithmetic functions. In many cases you can't just CALL the routine, but instead you must set up the

stack just so and then jump to the proper location. However, the effort may be worth it in many cases, so we will show you how to use the ROM string functions and let you decide.

When it comes to string handling (or any other type of programming, for that matter), avoid using the ROM as a "crutch". Think through what you are trying to do and decide if it is really worth the effort to use the ROM. These routines are presented for your information, but that does not mean that you can or should use them in any given situation. Think - is there an easier way?

THE "FUDGE-IT" METHOD OF USING THE ROM

If you really want to make things easy on yourself, you may want to fudge a bit and insert a little bit of BASIC into your machine-language program. This method is particularly useful when you are writing machine language code to be used with a BASIC program.

Here's how it works. Suppose, for example, that your machine language program needs the numeric value of the first three digits of the number in A\$. If you were writing in BASIC, you might use a statement like this:

```
X=VAL(LEFT$(A$,3))
```

Well, you can do almost the same thing in machine language. Here's how: First, turn on your computer (or type NEW). Then enter ONLY the portion of the statement to be evaluated (the part that would normally be placed at the right side of the equals sign) as the first BASIC line. For example, the above statement would be entered as

```
10 VAL(LEFT$(A$,3))
```

Note that this would normally cause a syntax error, since we aren't assigning the result to a variable (we want the result available to our machine language program, not BASIC). Now, enter the following from the keyboard:

```
X=PEEK(16548)+PEEK(16549)*256+4 :FOR Y=X TO X+100 :PRINT PEEK(Y);  
:NEXT
```

What you will see are the BASIC encoded bytes of your statement. You are only interested in the bytes up to the first zero byte. For the above example, they would look like this:

```
245 40 248 40 65 36 44 51 41 41 0
```

Carefully copy down these bytes, then insert them as a string somewhere in your machine language program. You will probably have to use several DEFB statements. Remember that the numbers are in DECIMAL, not hex, and be sure to include the ending zero byte.

When it is time to evaluate this expression, just point HL to the first byte of the string and CALL 2337H, the handy all-purpose expression evaluator mentioned in chapter two of this book. On return

from this routine, the result of the evaluation will be in the ACCUM, with the NTF set accordingly. Note that as long as the syntax is correct (meaning that you could put X= in front of the expression, or X\$= if the result is a string, and BASIC would accept it as a valid statement), the result may be either numeric or string.

Those who need maximum execution speed are warned that the "fudge-it" method is SLOW (in comparison to other methods) and probably should not be used in repetitive loops if another method is available.

STRING ROUTINES

The following routines are all available in both the Model I and the Model III ROM, except for the Model III time and date routines. Also, unless otherwise specified, on exit from these routines the result is stored in the ACCUM and the NTF is set accordingly. If the result is a string, the ACCUM will hold the VARPTR at locations 4121H and 4122H.

For many of these routines, the entry specifications will state that certain values must be PUSHed onto the stack. These must be placed on the stack in the order specified. In other words, you may see something like this: PUSH the return address on the stack, load R1 with argument value, PUSH RR onto stack, then jump to nnnn. Here's what you would do:

```

LD rr,nnnn      ;Load a register pair with return address
PUSH rr          ;PUSH the return address onto the stack
LD r,n           ;Load A, B, D, or H with argument
PUSH rr          ;PUSH argument (AF, BC, DE, or HL) onto stack
JP nnnn          ;Jump to routine entry point

```

Note that if an R1 register is specified, you may use A, B, D, or H, while if R2 is specified you may use C, E, or L. You must then PUSH the appropriate register pair.

Many of these routines save the HL register pair when called, and restore HL upon return. In addition, the A register may contain the contents of the memory location pointed to by HL upon return, that is, A=(HL). This is done primarily for the BASIC interpreter as HL usually points to the next byte of the BASIC program to be executed, but it may save you a few bytes. Contents of other registers should be presumed destroyed by the routines. It is suggested that the user test the action of specific routines using a monitor and/or single-step debugging program (such as TASMON), both to ascertain the entry and exit requirements of the routine, and to make sure that the stack is being properly prepared prior to use of the routine.

STRING NTF (NUMBER TYPE FLAG) TEST

0AF4H Checks the contents of the NTF for a value of three (indicating a string). Generates a Type Mismatch error if the NTF is not set properly. Uses AF.

STRING COMPARE ROUTINES

258CH Compare two strings. On entry, the VARPTR for string number 2 must be in the ACCUM, and the NTF=3. The stack must be set up as follows: PUSH the return address on the stack, PUSH the VARPTR for string number 1 on the stack. Then jump to 258CH (don't CALL). On exit, the A register will contain FFH, 0, or 1 (zero and carry flags set accordingly) depending on result of string number 1 compared to string number 2. NOTE: This routine may also be used to obtain a True-False result from a specific compare in the ACCUM. Follow the above instructions, EXCEPT set up the stack as follows: PUSH the return address on the stack, then PUSH the BC register pair (BC is saved). Load R1 (first half any register pair) with an integer from one through six that represents the desired compare, as follows:

```

1 - string 1 > string 2
2 - string 1 = string 2
3 - string 1 >= string 2
4 - string 1 < string 2
5 - string 1 <> string 2
6 - string 1 <= string 2

```

Then PUSH RR. Next, PUSH 25B8H onto the stack (load any register pair with 25B8H and then push that register pair). Finally, PUSH the VARPTR for string number 1 onto the stack. You may then jump to 258CH. On exit, the result of the compare (-1 if True, 0 if False) will be stored in single-precision format (NTF=4) in the ACCUM.

25A1H Compare two strings in memory. On entry, HL must point to string number 1, while BC must point to string number two. The D and E registers must contain the lengths (number of bytes) of strings 1 and 2 respectively. Then CALL 25A1H. On exit, the A register will contain FFH, 0, or 1 (zero and carry flags set accordingly) depending on result of string number 1 compared to string number 2.

STRING CONCATENATION

299CH Concatenate two strings. On entry, the stack must be prepared as follows: PUSH the return address on the stack, PUSH BC, and PUSH HL (BC and HL will be restored on exit from the routine). The HL register pair must contain the VARPTR for string number 1, and the ACCUM must contain the VARPTR for string number 2 (NTF must contain value of 3). Then jump to 299CH (don't CALL). On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3.

29C6H Move string to (DE) - use to concatenate strings. On entry, DE must point to the location that the string(s) will be moved to, and the VARPTR of the string to be moved must be PUSHed onto the stack. Then CALL 29C6H. Note that several strings may be concatenated by PUSHing the VARPTRs onto the stack in reverse order, then CALLing 29C6H once for each string to be concatenated. As an example, to concatenate three strings, follow this sequence:

LOAD DE with the starting address of the new (concatenated) string
 PUSH VARPTR for string 3 onto the stack
 PUSH VARPTR for string 2 onto the stack
 PUSH VARPTR for string 1 onto the stack
 CALL 29C6H - do this THREE times

The resulting string will be the equivalent of string 1 + string 2 + string 3. Note that the VARPTRs must not be POPped back off of the stack, as the routine at 29C6H takes care of this automatically.

STRING MOVE ROUTINE

29C8H Move string to (DE). On entry, HL must contain the VARPTR of the string to be moved, and DE must point to the location that the string will be moved to. ALTERNATE ENTRY POINTS: To use the alternate entry points, BC must point to the first character of the string to be moved and DE must point to the location that the string will be moved to. If the A register contains the length of the string to be moved you may CALL 29CDH. If the L register contains the string length you may instead CALL 29CEH. Note that strings may also be moved using the arithmetic move routines at 09D6H or 09D7H (see chapter two of this book), or by using the Z-80 block move commands LDIR or LDDR (this is the fastest method).

BASIC STRING FUNCTIONS

019EH INKEY\$ Create one-character string from keyboard input.
 CALL 019EH to use. Routine will create a one-character string using the contents of memory location 4099H. However, if 4099H contains zero, the keyboard will be scanned and if a key is depressed, a string will be created using the character input from the keyboard. If no key is depressed, a null string (a string zero bytes in length) will be created. On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3. NOTE: This routine may be CALLed at 019DH, in which case a RST 10H will be the first instruction executed (see chapter four of this book for details on RST 10H). Also, note that the keyboard is normally scanned during BASIC program execution for the presence of the BREAK or shift-@ keys. If a character other than one of those two is pressed, the character thus entered is stored in location 4099H. Thus, a keystroke received between successive INKEY\$ calls is held "in the buffer" at 4099H, to be used at the next execution of INKEY\$.

27DFH FRE(x\$) Get number of bytes free string space to ACCUM.
 CALL 27DFH to use. On exit, the ACCUM will contain the number of bytes of free string space remaining in the string storage area. This value will be stored in single-precision format (NTF=4).

2836H STR\$ Convert number in ACCUM to string.
 On entry, the number to be converted to a string must be in the ACCUM with the NTF set appropriately. PUSH the return address onto the stack, PUSH HL, and PUSH BC (HL and BC are restored on exit from the routine). Then jump to 2836H (don't CALL). On exit, the VARPTR to

the string created by this routine will be found in the ACCUM, with the NTF set to 3.

2A03H LEN Get length (number of bytes) of a string to ACCUM. On entry, the VARPTR of the string must be in the ACCUM, with the NTF set to 3. You may then CALL 2A03H. On exit, the length of the string will be stored (in integer precision) in the ACCUM, with the NTF set to 2. NOTE: When programming in Assembly Language, it is generally unnecessary to use this routine (unless you intend to perform further calculations with the resulting value, etc.), because the string length is contained in the byte pointed to by the VARPTR.

2A0FH ASC Get ASCII value of first character of string to ACCUM.

On entry, the VARPTR of the string must be in the ACCUM, with the NTF set to 3. You may then CALL 2A0FH. On exit, the ASCII value of the first character of the string will be stored (in integer precision) in the ACCUM, with the NTF set to 2. NOTE: Keep in mind that the two bytes following the byte pointed to by the VARPTR in turn point to the beginning of the string. Therefore, use of this routine would be considered highly inefficient programming, except when the resulting value is to be further processed by the ROM arithmetic routines.

2A1FH CHR\$ Make a one-character string - ASCII value in ACCUM. On entry, the ASCII value of the single character string must be stored in the ACCUM in single-precision format (NTF must be set to 4). PUSH the return address onto the stack, PUSH HL, and PUSH BC (HL and BC are restored on exit from the routine). Then jump to 2A1FH (don't CALL). On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3.

2A3DH STRING\$ Make a string of multiple bytes of one character. On entry, the ASCII value of the character used in the string must be stored in the ACCUM in integer format (NTF must be set to 2). PUSH the return address onto the stack, then load R2 (last half of a register pair) with the desired length of the string and PUSH RR. Then jump to 2A3DH (don't CALL). On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3. ALTERNATE ENTRY: PUSH the return address, then PUSH HL. If the desired string length is already the second half of a register pair other than HL, PUSH that register pair and jump to 2A3FH, otherwise place the string length in the L register and jump to 2A3EH (2A3EH contains a PUSH HL instruction).

2A64H LEFT\$ Make a string of the leftmost n characters of string.

To use this routine, PUSH the return address onto the stack, then PUSH the VARPTR of the string onto the stack. Load the B register with the numeric argument (the number of characters to be retained at the left side of the string), then jump to 2A64H (don't CALL). On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3. See NOTE following RIGHT\$ routine (below).

2A94H RIGHT\$ Make a string of the rightmost n characters of string.

To use this routine, PUSH the return address onto the stack, then PUSH the VARPTR of the string onto the stack. Load the B register with the numeric argument (the number of characters to be retained at the right side of the string), then jump to 2A94H (don't CALL). On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3.

NOTE: 2A61H and 2A91H are the "standard" entry points for LEFT\$ and RIGHT\$, respectively. However, those entry points require extra effort during setup. Specifically, the numeric argument must be PUSHed onto the stack (from the last half of a register pair), and the DE register pair must point to a left parenthesis (29H) character. See also the routine at 2A68H (described below).

2AB3H MID\$ Make a string from a portion of another string.

On entry to this routine, the A register must contain the position of the first character to be retained (same as the first numeric argument of the MID\$ function), and the E register must contain the maximum length of the new string (same as the optional second numeric argument of MID\$) - use FFH if you do not wish to limit the length of the string. The stack must be prepared as follows: PUSH the return address onto the stack, then PUSH the VARPTR of the string on the stack. Then jump to 2AB3H. On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3.

NOTE: The "standard" entry point to MID\$ is 2A9AH, however that entry point requires extra setup procedures. Also, an ALTERNATE ENTRY for this routine may be used in this manner: After PUSHing the return address, then PUSH HL, load HL with the VARPTR of the string, and jump to 2AB4H. See also the routine at 2A68H (described below).

2AC5H VAL Get value of numeric string to ACCUM.

On entry, the VARPTR of the string must be in the ACCUM, with the NTF set to 3. You may then CALL 2AC5H. On exit, the value of the number represented in the string will be stored (in double-precision format) in the ACCUM, with the NTF set to 8.

3030H TIME\$ (Model III only) Make a DATE + TIME string.

Creates a seventeen character date/time string of the format MO/DA/YR HR:MN:SS. Executes a RST 10H instruction on entry to advance the BASIC statement pointer (the HL register pair - see chapter four of this book for details on RST 10H). Note that the entry point at 3030H contains a jump to the actual start of the routine, which is found at different locations depending on which version of the ROM is being used. Therefore, if it is necessary to skip the RST 10H instruction, one of the following subroutines may be used:

TIME	LD	HL,(3031H)	TIME	PUSH	HL
	INC	HL		LD	HL,(3031H)
	JP	(HL)		INC	HL
(This routine destroys HL)				INC	HL
				JP	(HL)

The second code segment is two bytes longer but will preserve the contents of the HL register pair, while the first segment destroys the

contents of HL. The desired code segment should be set up as a subroutine and CALLED from the main portion of the program. However, if the execution of the RST 10H statement IS desired, a simple CALL to 3030H will suffice. On exit, the VARPTR to the string created by this routine will be found in the ACCUM, with the NTF set to 3. Also see "MODEL III TIME & DATE ROUTINES" below.

CREATING A STRING VECTOR

2865H CREATE A STRING VECTOR.

This routine will create a string vector for any string stored in memory. Several options are available, depending on how the routine is entered. Once a string vector has been created, the string may be processed further through use of other string-handling routines.

One use of this routine is to create a string vector for a number that has been converted to display format through use of the routines at 0FBDH or 0FBEH. To convert a number in the ACCUM to a string, first CALL 0FBDH or 0FBEH (see chapter two of this book for detailed instructions for using those routines), then CALL 2865H. The VARPTR to the resulting string vector will be found in the ACCUM (at 4121H-4122H).

To create a string vector for any other string in memory, this routine may be CALLED at 2865H PROVIDED that the following conditions are met: First, the HL register pair must point to the first character of the string. And second, the string MUST be terminated with either a zero byte or a quotation mark (0 or 22H).

If the routine is called at 2866H or beyond, the HL register pair must point to the memory location just BEFORE the first character of the string. This is because 2865H contains a DEC HL instruction. This entry point may be used to create a string vector for a string that has been input from the keyboard using the routine at 0361H.

If you do not want the string to be terminated by a quotation mark, or if you wish to define your own terminators, you may effect these changes by using alternate entry points and by manipulating the B or D registers, as follows:

Entry at 2868H: define your own terminator to replace the quotation mark by loading its ASCII code into the B register. Or, load the B register with zero if you do not wish the string to terminate with a quotation mark but do not wish to define your own terminator.

Entry at 2869H: define your own terminator while still allowing a quotation mark to function as a terminator by loading the ASCII value of your terminator into either the B or D register. Or, you may define two terminators of your own choosing by loading their ASCII codes into both the B and D registers.

Note that whatever entry point is used, a zero byte will always function as a terminator. Once you have loaded the appropriate register(s) (if required), CALL the routine at the entry point of your

choice. On exit from this routine, the VARPTR to the resulting string vector will be found in the ACCUM at 4121H-4122H.

STORING STRINGS IN THE BASIC STRING STORAGE AREA

In the event that the assembly-language programmer wishes to use the BASIC string storage area (located near the top of memory) to hold a newly-created string, the following routines may prove useful.

28BFH MAKE ROOM FOR A STRING IN STRING STORAGE AREA.

This routine will make room for a string in the string storage area if possible (will go to an Out of String space Error if not possible). On entry, the A register must contain the length of the string to be placed in storage. On exit, the DE register pair will point to the location where the string should be stored (this routine does NOT actually move the string, it simply indicates where it should be placed). The pointer at 40D6H will be equal to DE minus one (this pointer indicates the next usable string space location).

2857H CALLS 28BFH AND THEN CREATES STRING VECTOR.

This routine CALLs the above routine (at 28BFH) and then creates a string vector for the new string. All of the entry and exit conditions for the above routine apply, but in addition, on exit a three-byte string vector for the new string will be stored at 40D3H-40D5H, and the HL register pair will serve as a VARPTR (it will contain 40D3H). It may be desirable to move this vector to another location in memory (easily done by using the routine at 09D2H, which is described in chapter two of this book), in order to protect it for future use.

PROGRAMMING HINT: Keep in mind that the routine at 260DH (described in chapter two of this book) can be used to locate or create a BASIC string variable. If you were to call the above routine (at 2857H), then save HL on the stack, move your string to the space created, and use the routine at 260DH to locate or create a variable, and then POP HL and CALL 09D2H to move the string vector, you would effectively have created or reassigned a BASIC string variable. This is a neat way to pass your strings back to BASIC.

2A68H CREATE OR DUPLICATE STRING OR SUBSTRING.

This routine is used by LEFT\$, MID\$, and RIGHT\$ to create new substrings. It can also be used to duplicate existing strings in memory or to create a permanent storage area for a temporary string. On entry to this routine, the HL register pair must contain the string VARPTR. To duplicate a string or move a temporary string to permanent storage, load the BC register pair with FF00H. Otherwise, load the B register with the maximum length of the string, and load the C register with the number of characters to ignore at the start of the line. Note that the new string will not exceed the current string length or the length placed in the B register, whichever is less. However, if the C register contains a value other than zero, it is possible that "garbage" characters may be included in the new string unless precautions are taken to avoid this. For example, if the B register contains a value of 25 decimal but the current string is only 10 characters long, the maximum string length will be 10 characters.

If a value of 5 is placed in the C register, the resulting string will contain the last five characters of the original string plus the five bytes following the original string in memory (which probably contain "garbage" or parts of other strings). On exit from this routine, the VARPTR for the resulting string is stored in the ACCUM, with the NTF set to 3.

PROGRAMMING HINT: As with the previous routine (at 2857H), it would be possible to use other ROM routines in conjunction with this routine to create a BASIC variable. One suggested method would be to use the routine at 260DH to locate or create the variable first, then PUSH DE and use this routine (2A68H) to create the desired string. Then LD HL,(4121H) to get the VARPTR in HL and POP DE to get the VARPTR for the new variable. Then CALL 09D3H to move the string vector from (HL) to (DE). This will assign the new string to a BASIC variable.

MODEL III TIME & DATE ROUTINES

In the Model III, the time and date are stored in a seven-byte buffer starting at 4216H. A similar buffer is provided in the Model I starting at 4040H, but it is not properly updated unless an Expansion Interface is connected, and there are no time or date routines in the Model I ROM (they are provided by the Model I DOS). In both models the time is updated by an interrupt service routine, therefore, if interrupts are disabled (such as during cassette or disk access) the clock locations will not be properly updated, and will lose time. The format of the time storage locations are as follows:

MODEL III	MODEL I	STORAGE FOR:
4216H	4040H	25 MS. INTERRUPT COUNTER
4217H	4041H	SECONDS COUNTER
4218H	4042H	MINUTES COUNTER
4219H	4043H	HOURS COUNTER
421AH	4044H	YEARS COUNTER
421BH	4045H	DAYS COUNTER
421CH	4046H	MONTHS COUNTER

The Model III TIME\$ routine at 3030H has been previously discussed under "BASIC STRING FUNCTIONS". In addition, the following routines are available in the Model III:

3033H CREATE DATE STRING IN EIGHT-BYTE BUFFER.

On entry, HL must point to an eight-byte buffer which will hold the resulting string. On exit, HL will point to first location past the end of the buffer, and the buffer itself will contain the date string in the format MO/DA/YR. Uses AF,BC,DE,HL.

3036H CREATE TIME STRING IN EIGHT-BYTE BUFFER.

On entry, HL must point to an eight-byte buffer which will hold the resulting string. On exit, HL will point to first location past the end of the buffer, and the buffer itself will contain the time string in the format HR:MN:SS. Uses AF,BC,DE,HL.

CHAPTER FOUR - MISCELLANEOUS ROM ROUTINES

In the previous three chapters of this book, we have covered the I/O routines, the arithmetic routines, and the string-handling routines. These routines occupy most of the ROM, but there is yet another category that we should not overlook. Most of the routines in this category have to do with system "housekeeping" - that is, the running of BASIC programs, making sure that computer memory is properly allocated, etc. There are also a few routines that don't fit well into any category, so they wind up here. A few of these routines are specific to only one Model of the TRS-80 (I or III), and those will be indicated as such (all other routines are available on either Model).

Many of the routines in this chapter will be useful only to the programmer who is writing a program designed to interact in some way with BASIC. Examples of this might include a machine-language routine called by the USR function, or a utility program that is intended to simplify the creation or editing of a BASIC program. However, you may find it profitable to study these routines even if you are not writing this type of program, as they will give you many clues as to how the computer manages memory, interprets BASIC programs, etc.

So, without further ado, here are the ROM routines that couldn't be categorized, but which are unique and perhaps even useful.

THE RST (ReStart) INSTRUCTIONS

There is a special type of CALL instruction available in Z-80 machine language. Whereas a normal CALL instruction requires three bytes (one for the instruction itself and two for the address), the RST instruction requires only one byte. The catch is that there are only eight address in all of memory that can be called using the RST instruction. These addresses are all in the lowest part of memory, deeply embedded in the ROM. Fortunately, the designers of the TRS-80 decided to make the RST instructions somewhat more flexible, with the result being that each of the RST addresses (with the exception of RST 0) contains a jump to user RAM. Starting at 4000H is a series of three byte jump vectors that are normally set to jump to some routine back in ROM. So, when you use a RST instruction, what happens is that the program calls one of the RST locations in low ROM, then jumps to RAM in the area of 4000H, then jumps back to ROM where the desired routine is actually located. With this setup, the user could change the vectors for any of the RST instructions, and thereby have the use of a one-byte CALL for any machine language subroutine. The problem in doing this is that BASIC makes heavy use of some of these instructions, so you would do well to avoid changing these vectors unless you are sure of what you are doing.

Some of the RST instructions have been previously described in earlier chapters of this book. Here is a list of the eight RST instructions, and the action taken by the TRS-80 when each is executed (unless otherwise indicated, all of the routines that follow are the same on both the Model I and the Model III):

RST 0H POWER-UP

Executing a RST 0 instruction will cause the computer to re-initialize itself, in the same manner as if the electricity had just been applied to the keyboard. Same effect as a jump to 0000H. Executes a DI (Disable Interrupts) and XOR A instruction, then jumps to 0674H in the Model I or 3015H in the Model III, which continues with the cold start initialization routine.

RST 8H JUMP TO 4000H, then JUMP TO 1C96H

1C96H - This routine is used by BASIC to check for an expected character - if that character is not present, a jump is taken to the syntax error routine. An expected character is one that must be present to preserve proper syntax. An example of this would be the semicolon in the following BASIC statement:

```
INPUT "WHAT IS YOUR NAME"; A$
```

After BASIC has processed (in other words, output to video) the prompt string, the next character following MUST be a semicolon. The code in ROM that performs this check looks like this:

21D3	CF	RST	08H
21D4	3B	DEFB	';'
21D5	(continue with INPUT subroutine)	

This is what happens when the RST 8 instruction is executed: If the byte pointed to by the HL register pair (which points to the next byte of the BASIC program) is NOT the same as the byte that follows the RST 8 instruction (a semicolon in the above example), a syntax error results. However, if the bytes match, the return address (at the top of the stack, which presently points to the DEFB instruction) is incremented by one (so that the subroutine will return to the instruction FOLLOWING the DEFB instruction). A jump is then taken to 1D78H, which is the location of the routine called by the RST 10H instruction, in order to advance the BASIC pointer (the HL register pair) to the next character of the BASIC program, after skipping space characters as explained below. In the above example, at the completion of the call to RST 8, return will be to 21D5H, and the HL register pair will point to the "A" in the variable name "A\$". Uses AF,HL.

RST 10H JUMP TO 4003H, then JUMP TO 1D78H

1D78H - Advances HL register pair to point to the next character of a BASIC program. During a run of a BASIC program, the HL register pair normally points to the next byte of the BASIC program to be executed. The RST 10H instruction causes HL to be incremented, and if HL then points to a space character (20H) or a linefeed character (0AH), the process is repeated until HL points to some character other than a space or linefeed. On return, if HL points to a colon (3AH) or a zero byte (either of which are a valid BASIC statement terminator), the Z flag will be set. If the character is numeric (0 through 9, represented in ASCII by 30H through 39H), the C flag will be set. If, on the other hand, the character pointed to by HL is NOT numeric, a colon, or a zero byte, then the C and Z flags will both be reset. Uses AF,HL.

RST 18H JUMP TO 4006H, then JUMP TO 1C90H
 Compare HL register pair to DE register pair. Both register pairs are assumed to hold 16 bit unsigned (positive) integers (in the range 0 - 65535 decimal). Used by BASIC to compare program line numbers. On exit, the Z flag is set if HL is equal to DE, while the C flag is set if DE is greater than HL. Uses AF.

RST 20H JUMP TO 4009H, then JUMP TO 25D9H
 Test the NTF (Number Type Flag) at 40AFH. On return, Z flag is set if string, M set if integer, P and C set if single precision, P set and C reset (NC) if double precision. Uses AF.

RST 28H JUMP TO 400CH

Break key vector. In non-disk systems 400CH contains a RET instruction. In a disk system this vector is used for DOS overlay requests (Bit 7 of the A register is set on entry if DOS overlay). In addition, the following code is found at the end of the ROM keyboard driver routine:

MODEL I:	MODEL III:	MODEL 4:	
0453H	3100H or 344AH	33FCH FE01	CP 01H ; l is BREAK key
0455H	3102H or 344CH	33FEH C0	RET NZ ; Return if not BREAK
0456H	3103H or 344DH	33FFH EF	RST 28H ; Call 0028H
0457H	3104H or 344EH	3400H C9	RET ; Then return

The above code allows the DOS to intercept the BREAK key as long as the ROM keyboard driver is in use. It also permits the programmer to disable the BREAK key completely, by modifying the contents of this vector. For example, the BASIC statements

POKE 16396, 175 : POKE 16397, 201

will disable the BREAK key. To re-enable, simply restore the original contents of this vector (under non-disk BASIC, use the statement POKE 16396, 201).

RST 30H JUMP TO 400FH

DEBUG entry point under DOS. In non-disk systems 400FH contains a RET instruction.

RST 38H JUMP TO 4012H

Under a non-disk system on the Model I, 4012H contains an EI (Enable Interrupts) instruction, followed by a RET instruction at 4013H. Under a Model I disk system, 4012H contains a jump to the DOS Interrupt Service Routine. On the Model III, 4012H contains a JUMP TO 3018H, the Maskable Interrupt Handler routine, which jumps to various interrupt service routines depending on the bits set at input port E0H. Port E0H is the Interrupt Request Mask on the Model III, and an input from this port indicates (according to which bits are set) the last device to generate an interrupt. Jumps are then directed as follows:

BIT SET:	JUMPS TO:	REMARKS:
0	3365H	Part of 1500 baud cassette read routine
1	3369H	Part of 1500 baud cassette read routine
2	4046H	4046H = JP 3529H Cursor blink routine
3	403DH	403DH = JP 35FAH "RET" instruction
4	4206H	4206H = JP 35FAH "RET" instruction
5	4209H	4209H = JP 35FAH "RET" instruction
6	4040H	4040H = JP 35FAH "RET" instruction
7	4043H	4043H = JP 35FAH "RET" instruction

If more than one bit is set, the lowest bit number takes priority. If one of bits 2 - 7 are set, a jump is made to reserved RAM and then from there back to ROM. Prior to doing that, ALL register pairs (AF,BC,DE,HL,IX,IY) are PUSHed onto the stack, and are restored when the interrupt routine RETurns. The jumps from the reserved .M vectors shown above (to 3529H or 35FAH) may be changed on a disk-based system by the DOS.

MISCELLANEOUS ROM ROUTINES

Although it would be handy if the following list included all of the ROM routines that have not yet been covered in this series, it's probably safe to assume that I've missed a few. If you know of any that have been missed, why not write and let me know about them? Perhaps they will appear in a future edition of this book.

The routines that follow are varied, so scan through the list - you just may find the one you need. Note that all routines are the same on both the Model I and the Model III, unless otherwise specified.

000BH RESOLVE RELOCATION ADDRESS

POP HL followed by JP (HL). Same as RET instruction except leaves return address in HL. Thus, a relocatable program can CALL 000BH, and upon return, the HL register pair will contain the current address of the Program Counter. In this way, a program can "find" itself in memory.

000DH DISK BOOTSTRAP LOADER

Model I: JUMP TO 069FH. Model III: JUMP TO 3012H. Does a "warm" system reboot, in that it reloads and executes the Disk Operating System but does not re-initialize all system pointers, etc. first.

0060H TIME DELAY LOOP

On entry, the value in the BC register pair determines the amount of delay. Shortest delay occurs when BC=1, longest delay (about one second) occurs when BC=0 (in this case 65536 delay loops are performed by the routine. The next longest delay would occur when using a value of FFFFH). Note that the delay is very slightly longer in the Model III - this is due to the addition of an extra instruction in the delay loop, for the purpose of (slightly over-) compensating for the faster clock speed of the Model III. In the Model I, each count of BC produces approximately a 14.6555 microsecond delay, while in the Model III the time for each count is 14.7964 microseconds. To put it in

more simple terms, for any given delay value in BC, the Model III will take about 1% more time to complete the loop. For very short delays, keep in mind that a minimal amount of time is also required to CALL the routine. Uses AF,BC.

0066H NMI RESET

Non-Maskable Interrupt routine. Jumps here when the RESET button is pressed, or when a HALT instruction is encountered (provided that Z-80 microprocessor is set to Interrupt Mode 0). Different actions are taken by this routine in the different Models. In the Model I, a test is made to determine if a disk controller is present, and if so a jump is taken to the power-up sequence at 0000H, otherwise the routine jumps to the BASIC re-entry point at 06CCH (see description of that routine below). In the Model III a jump is taken to 3039H, where input port E4H is tested to see if bit 5 is set (interrupt from disk system present), and if not a jump is taken to a RAM vector at 4049H, which normally contains a RST 0 instruction. If a disk interrupt is present (bit 5 is set), the program loops until bit 5 of port E4H is reset and then jumps to 0000H. Note that if the vector at 4049H were changed it would be possible to direct the Model III to jump to something other than the power-up sequence when the RESET button is pressed in a non-disk system. Since the memory locations from 404AH to 407FH are unused in a non-disk system, it would even be possible to insert a short program starting at 4049H to handle the RESET button. For example, if the instructions LD BC,1A18H followed by JP 19AEH were placed in this area, depressing RESET would return the non-disk user to the BASIC "READY" prompt (as in the Model I)!

0072H JUMP TO 06CCH

See routine at 06CCH for further details.

0075H (ALMOST) COLD START FOR NON-DISK SYSTEMS

A JUMP to this location will bring up the power-up sequence for Level II/Model III (non-disk) BASIC, regardless of whether a disk system is connected. However, not all system vectors are reset from this routine, so caution is in order if attempting to use this entry point.

06CCH MODEL I RETURN TO BASIC "READY"

This is the best re-entry point to BASIC on the Model I, but unfortunately, on the Model III 06CCH is part of the BASIC LIST command. See the next two routines (at 19AEH and 1A19H) for more information.

19AEH ALTERNATE ENTRY TO BASIC "READY"

To use this routine, load the BC register pair with 1A18H, then JUMP to 19AEH. This will work with both the Model I and the Model III, and does not return an invalid "Out of Memory Error" message (see next routine at 1A19H).

1A19H "OFFICIAL" RETURN TO BASIC "READY"

This is the entry point sanctioned by Radio Shack for a return the the BASIC "READY" prompt. However, the disadvantage of using this entry is that it will often return an "Out of Memory Error" message in response to the next command typed in from the keyboard, even though an "out of memory" condition does not exist. Model I programmers have

often substituted 06CCH or 0072H as alternate entry points, but as mentioned above, these will not work with the Model III. The use of the routine at 19AEH is recommended (see above for details on this routine).

1A7EH PROCESS INPUT FROM "READY" PROMPT

This is the beginning of the section of ROM that acts upon keyboard input from the "READY" prompt. At this entry point, it is assumed that the HL register pair points to the memory location just prior to the beginning of the buffer containing the input to be processed, and that if the C flag is set, the computer should return to the READY prompt without acting upon the input (these assumptions are based on the fact that the ROM has just CALLED the keyboard input routine at 0361H, and if the C flag is set it indicates that the BREAK key was pressed). The input will be compressed (BASIC reserved words will be compressed to one-byte tokens), and if the input began with a line number, the line will be inserted into the resident BASIC program (otherwise the input will be directly executed). There are several possible entry points to this routine, as presented in this disassembly of a short portion of the ROM:

1A7B	CD6103	CALL 361H	;This gets keyboard input
1A7E	DA331A	JP C,1A33H	;If "BREAK" pressed, go to "READY"
1A81	D7	RST 10H	;Now HL points to start of input
1A82	3C	INC A	; and C flag is set if input
1A83	3D	DEC A	; started with line number
1A84	CA331A	JP Z,1A33H	;If first byte = 0, go to "READY"
1A87	F5	PUSH AF	;Save AF on stack
1A88	CD5A1E	CALL 1E5AH	;Get line number in DE
			(code continues at 1A8BH)
...			

Note that by anticipating the various conditions expected by the ROM, we can JUMP to this routine at any of several points. For example, if DE contained a line number, HL pointed to the start of some text we wanted to put into a line, and the C flag was set and AF pushed onto the stack, we could jump to 1A8BH and the line would be entered into our BASIC program. There are a couple of problems to be aware of, however. First, if you do make use of the entry point at 1A8BH, you should be aware that if there is more than one space character (20H) immediately prior to the text pointed to by HL, all but one of these space characters will be tacked onto the beginning of the text. To avoid this, jump to 1A98H instead of 1A8BH. The other problem is that this routine always returns to the BASIC ">" prompt (the "READY" state), thus it cannot be used as a subroutine that is CALLED from your program - unless you are willing to resort to some "tricky" programming. Assuming that a new line was entered into the resident BASIC program, the last five instructions of this routine look like this:

1AE9	CDFC1A	CALL 1AFCH	;Adjust BASIC line pointers
1AEC	CDB541	CALL 41B5H	;DOS vector
1AEF	CD5D1B	CALL 1B5DH	;"CLEAR" (adjusts pointers)
1AF2	CDB841	CALL 41B8H	;DOS vector
1AF5	C3331A	JP 1A33H	Back to READY (">" prompt)

Note that there are two DOS vectors that could be temporarily intercepted. If you had CALLED the routine, and one of the vectors contained a POP rr instruction (to get rid of the return address in ROM), followed by a RET instruction, you could return to your program (assuming that the return address to your program was up next on the stack). Just be sure to restore the DOS vector before you exit your program, and remember that this won't work if the line is interpreted as a direct statement, instead of as a line to be added to the BASIC program. This is one of the key routines of the BASIC interpreter, and certainly merits close examination by anyone attempting to understand how BASIC operates.

1AF8H ADJUST BASIC POINTERS

As stored in memory, the first two bytes of each line of a BASIC program point to the first byte of the following BASIC program line. When a program has been CLOADed, or after editing, these pointers may not be correct. This routine adjusts all of these forward pointers so that they correctly point to the beginning of the next line, starting with the first line of the program. Uses AF,DE,HL.

1AFCH ADJUST BASIC POINTERS STARTING AT (DE)

Same as above routine (at 1AF8H), except begins adjusting pointers at line pointed to by the DE register pair (DE must point to the first byte of a BASIC line). Used when a line has been edited, and it is known that all pointers prior to the beginning of that line are still correct.

1B2CH SEARCH FOR A MATCHING BASIC LINE NUMBER

On entry to this routine, the DE register pair must contain the line number for which the search is to be made. On exit, the C flag will be set if a match was found. The Z flag will be set if a match was found OR the end of the program was reached without finding a line number larger than the one being searched for. If a line with a matching line number OR a line number larger than the one being searched for was found, the BC register pair will point to the start of that line, and the HL register pair will point to the start of the next line following. If all of the line numbers in the program were smaller than the one being searched for, then both the HL and BC registers will point to the byte immediately following the last byte of the last line of the program (the second of the three zero bytes at the end of the program, which is the location where a new line should begin if you are performing this search in order to add a line to the BASIC program). Uses AF,BC,HL (DE still holds line number on exit).

1B4DH NEW

This routine will wipe out the BASIC program currently in memory by resetting the pointers associated with it. Alternate entry points are 1B4AH (also clears video display) and 1B49H (same as 1B4AH except that the Z flag must be set upon entry, otherwise a return is made to the calling program immediately, without resetting any of the pointers). This following routine (at 1B61H) is a part of this routine, and the WARNING it carries in regard to the Stack Pointer also applies to this routine.

1B61H CLEAR ALL VARIABLES

This routine is CALLED whenever the length of the BASIC program is changed in any way, such as by adding, deleting, or editing program lines. All variables are cleared, the associated pointers are reset, and various other flags and pointers are reset. Entry at 1B61H saves the HL register pair, but if the routine is entered at 1B5DH, the HL register pair will point to the start of the BASIC program minus one on exit (also, on the Model III the 1B5DH entry point will unprotect the video display). WARNING: This routine resets the Stack Pointer origin to the address pointed to by the contents of 40A0H-40A1H (the "start of string space" pointer) minus two. The RET address is preserved but all other items on the stack will be lost unless steps are taken in order to preserve the stack.

1BC0H COMPRESS BASIC LINE

BASIC stores all reserved words as one-byte tokens. This routine will take a line of text and compress it so that all BASIC reserved words are tokenized. On entry, HL must point to the first byte of the text to be encoded (text must be terminated with a zero byte). On exit, the resulting encoded statement will be stored in the input buffer, beginning at the location pointed to by 40A7H-40A8H minus two. The HL register pair will point to the start of the encoded statement minus one, and this location will contain a colon (3AH) character. Three zero bytes will be placed at the end of the encoded statement, and the DE register pair will point to the last of these zero bytes. The BC register pair will contain the actual length of the encoded statement (not counting the beginning colon or ending zero bytes), plus five (this is to allow room for a forward line pointer, a line number, and an ending zero byte if the line is to be inserted into a BASIC program). In addition to compressing BASIC reserved words, this routine converts all other alphabetic characters in the line to uppercase, except for those within literal strings that are enclosed by quotation marks. Uses AF,BC,DE,HL.

1D91H RESTORE

Resets the DATA pointer (at 40FFH) to point to the start of the BASIC program, so that the next time a BASIC READ statement is executed, the first DATA item in the program will be read. Uses DE.

1DF7H TRON

Enables the BASIC trace function by setting the trace flag at 411BH to a non-zero value. Uses AF.

1DF8H TROFF

Disables the BASIC trace function by setting the trace flag at 411BH to zero. Uses AF.

1E3DH CHECK FOR UPPERCASE ALPHABETIC CHARACTER AT (HL)

If the byte pointed to by the HL register pair contains the ASCII code for an uppercase letter of the alphabet (A-Z), the C flag will not be set, otherwise the C flag will be set on return. The reason that this routine is not designed to recognize lowercase characters is that any lowercase letters found in a BASIC line that are important to the BASIC interpreter (such as variable names) are converted to uppercase by the compression routine at 1BC0H. Uses AF.

1E83H CLEAR n

Reserves a number of bytes of memory for string storage (resets the "start of string space" pointer at 40A0H), then jumps to the CLEAR routine at 1B61H (see above for details). On entry, the DE register pair must contain the number of bytes of string space to be reserved.

1F07H INCREMENT HL (IF NECESSARY) UNTIL (HL)=0

Used by BASIC to advance the program pointer (HL register pair) to the end of the BASIC line (to skip REM statements or ELSE clauses that are not executed). On entry, if HL points to a zero byte no action is taken, otherwise HL is incremented until it does point to a zero byte. Uses AF,BC,DE,HL.

1F21H LET

On entry, HL must point to the first character of a string which consists of a valid variable name, followed by a D5H byte (the BASIC = function), followed by a valid BASIC expression (see the details for the "Fudge-It" method in part three of this series for details on how to encode the expression - note that the method used there to encode the expression could also be used here to encode the entire LET string). On exit, the expression will have been evaluated and the result assigned to the indicated BASIC variable. ALTERNATE ENTRY POINTS: If the variable that is to receive the new value has previously been located (through a CALL 260DH instruction, for example) you may wish to use one of the following alternate entry points (expected entry conditions are shown for each):

1F24H DE=address of variable, HL --> D5H byte ("=").

1F26H DE=address of variable, HL --> first byte of expression.

1F27H HL=address of variable, DE --> first byte of expression.

2587H JUMP TO ((HL))

Loads the HL register pair with the address pointed to by HL on entry, then executes a JP (HL) instruction.

2608H DIM

Dimension one or more BASIC variables. On entry, HL must point to the first character of a string which contains one or more variable names and desired dimensions (same syntax as would be required immediately following a BASIC DIM statement). String must be terminated with a zero byte or colon.

27D4H GET AMOUNT OF FREE MEMORY TO ACCUM

Used by MEM and FRE(x) functions. On entry, the NTF at 40AFH must NOT be set to 3 (string), but it may be set to any other precision or may contain zero. On exit, the number of bytes of remaining free memory will be stored in the ACCUM, in single-precision format (NTF=4). Uses AF,DE,HL.

27F5H POS

Gets the position of the cursor on the current line being output to video from location 40A6H (value will be in range 0 - 63), and stores in the ACCUM in integer format (NTF=2). Uses AF,HL.

27FEH USR

Yes, it's true - the USR function can be CALLED from a machine language program, just like any other BASIC function. Of course, it would be a highly inefficient programming method, but it could be done. There's even an alternate entry point for non-disk users that will save HL: 2802H. While this information may be next to useless in all but the strangest circumstances, it serves to point out that the USR function is, after all, treated like any other BASIC function by the BASIC interpreter. This is important, because it shows how we can pass arguments to and from BASIC. For one thing, whatever you place within the parenthesis as the argument is evaluated by the routine at 252CH prior to exiting to your machine-language routine. This means that the argument can be numeric (any precision) or string, and that the result of the evaluated expression will be stored in the ACCUM, with the NTF set appropriately. Radio Shack says that you must CALL 0A7FH to recover the argument, but as you know from reading part two of this series, all that the routine at 0A7FH does is to change the number in the ACCUM to an integer (don't use it on string arguments!). Similarly, R.S. says that we must jump to 0A9AH on return if we wish to pass a value back to basic. But, as you probably realize by now, that's only true if the number you wish to pass back to BASIC is stored in the HL register pair. If the value you wish to pass back is already in the ACCUM, with the NTF properly set, you need only RETurn. Note that nothing prevents taking in a number from BASIC and passing back a string, or vise-versa, provided that you use the proper variable types at each end. It should be clear from the above that you are NOT limited to passing integers only. So, use your USR call more effectively! PROGRAMMING NOTE: When using a string variable as the argument of a USR function call, note that when control is passed to the USR routine the ACCUM will contain the VARPTR to the original variable (or to a temporary variable if a complex string expression was used as an argument). It is suggested that the programmer immediately CALL 0AF4H to check the NTF for a string, then LD BC,FF00H and CALL 2A68H to duplicate and/or reserve string space for the string before attempting to manipulate it. See the routine at 2A68H (in the previous chapter of this book) for more information.

2828H CHECK FOR ILLEGAL DIRECT ERROR

Checks the current BASIC line number (stored at 40A2H-40A3H) for a value of FFFFH, which indicates that commands are being executed from the direct mode (in which case the BASIC statements currently being executed are stored in the BASIC input buffer, so it is not available to receive input). If this is the case an Illegal Direct error exists and a jump to the error routine is taken, otherwise returns to caller. Uses AF.

2B7EH EXPAND COMPRESSED BASIC LINE

After a BASIC line has been compressed, this routine may be used to expand it back into a line of text, so that it can be LISTed, EDITed, etc. On entry, HL must point to the first byte of compressed text in the line (that is, the first byte following the forward pointer and line number). The line will be decoded and the result placed in the BASIC text buffer until a zero byte is encountered in the line (the starting address of the buffer is stored at 40A7H-40A8H). The expanded line will be terminated with a zero byte (routines at 2B75H or 28A7H may be used to output the line). Note: This routine has been

expanded in the Model III, so that when graphics characters are found between a pair of quotation marks they are left unchanged rather than being converted to BASIC reserved words (as happens in the Model I). Uses AF,BC,DE.

2BE4H DELETE BASIC LINE OR LINES

On entry, the BC register pair must point to the start of the first BASIC line to be deleted, and the HL register pair must point to the start of the following line to be retained (note that the routine at 1B2CH may be used to find the start of the desired lines). On exit, the lines are deleted, but the forward line pointers are not corrected by this routine. Therefore, suggested entry to this routine is to PUSH BC prior to the CALL to 2BE4H. Then, after the return from this routine, you can POP DE and CALL 1AFCH to adjust the forward pointers. It is also permissible to eliminate the PUSH/POP sequence by simply making the CALL to 2BE4H, then a CALL to 1AF8H to adjust the pointers. However, the latter method suffers from excessively long execution times, especially for BASIC programs that have many lines (however, if the lines to be deleted are located near the beginning of the BASIC program, the difference in execution times for the two methods will be slight). ALTERNATE ENTRY POINT: On entry to this routine, if the pointer to the first line to be retained is in DE (instead of HL), you may enter at 2BE5H. Uses AF,BC,DE,HL.

BASIC ERROR ROUTINES

Occasionally, it may be desirable to use the BASIC error-handling routine, especially with machine language subroutines of BASIC programs. The advantages of using the BASIC error routine include these: The Stack Pointer is reset and a graceful re-entry is made to BASIC. If error trapping is in effect (through use of an ON ERROR GOTO statement), the program will branch to the BASIC error handling routine, otherwise the selected error message will be printed, and the computer will return to the "READY" prompt. Using the built-in error routines saves memory. The main disadvantages are that there may not be a built-in error message that is appropriate to the error encountered, and that control will not normally be returned to your program, but will instead be passed to BASIC. This latter problem is one you may wish to deal with anyway, since many of the ROM routines documented in this series and elsewhere will jump to the BASIC error routine if an error condition is encountered.

To start with first things first, there are two methods of using the error-handling routine. The first is to load the E register with the error code number, and then executing a JUMP to 19A2H. This error code number is not the same as the error code numbers listed in your TRS-80 BASIC manual, but rather is the same as the error code numbers returned by the BASIC ERR function. Specifically, the error code number may be any EVEN number from zero through FEH (254 decimal), although codes above 2CH (44 decimal) are not recognized by ROM BASIC and normally return an Unprintable Error (in Disk BASIC some of these codes are used to provide additional error messages).

The other method, which may be used to advantage for some types of errors, is to simply JUMP to the proper location in ROM for that error. Usually, the code at that location will load the E register

with the proper error code and then JUMP to 19A2H. However, in some cases this jump is conditional on the status of one of the flags. When this is the case, it probably will not be possible to use that routine unless you know that the required flag will be properly set. It may still be possible to save one byte in this case, since it takes only one byte to set the carry flag (SCF) or clear the zero flag (XOR A), while it takes two bytes to load the E register with a value.

The following is a list of each of the error codes recognized by BASIC, the type of error generated by that code, the JUMP address to the specific routine for that error, and the flag condition required (if any) to use that routine:

CODE	ERROR	ADDRESS	FLAG
00H	NF NEXT without FOR	199DH	
02H	SN Syntax	1997H	
04H	RG RETURN without GOSUB	1EEAH	NZ
06H	OD Out of DATA	2212H	Z
		or 22A0H	Z
08H	FC Illegal Function Call	1E4AH	
0AH	OV Overflow	07B2H	
0CH	OM Out of Memory	197AH	
0EH	UL Undefined Line	1ED9H	
10H	BS Subscript out of range	273DH	
12H	DD Redimensioned array	2733H	NZ
14H	/0 Division by zero	199AH	
16H	ID Illegal Direct	2831H	
18H	TM Type Mismatch	0AF6H	
1AH	OS Out of String space	28DBH	Z
1CH	LS String too Long	29A3H	C
1EH	ST String formula too complex	28A1H	
20H	CN Can't CONTinue	1DE9H	Z
22H	NR No RESUME	198AH	NZ
24H	RW RESUME Without error	19A0H	
26H	UE Unprintable Error	2003H	
28H	MO Missing Operand	24A0H	Z
2AH	FD Bad File Data	218AH	Z
2CH	L3 Disk BASIC only	012DH	

Of course, there is always the possibility that you may want to manipulate the error-handling routine a bit from within your machine language program. Here are some facts you may find useful for that purpose:

First of all, if you want to set up an ON ERROR GOTO condition, or cancel a previous one, you should know that the line number for the error trap is stored at 40EAH-40EBH. If you wish to set up an error trap, simply store the starting line number of the trap at those locations. To cancel an existing error trap, load those two locations with zero.

Another way to disable an error trap is to load memory location 40F2H with FFH. This makes BASIC think it has found an error within an error trap routine, and it will jump to the error message. To restore normal operation, load 40F2H with zero (be sure to do this

prior to returning to BASIC, or BASIC will think it is in an error trap routine).

If you are using ROM routines in a machine language program, you may not wish to be inadvertently thrown into an error routine. Unfortunately, there isn't too much you can do about it, but you can catch up with the runaway program before too much damage is done. This is accomplished by intercepting the vector at 41A6H, which is normally used under DOS to provide long error messages for disk BASIC. You can insert a jump to an error-handling routine in your program at this point. Unfortunately, one bit of serious damage has already been done at this exit, and that is that the Stack Pointer has been reset to the BASIC Stack Pointer Origin, which is stored at 40E8H-40E9H. Depending on whether you are running a stand-alone program or a program that works with BASIC, you will have to decide how to recover your stack after an error. The best plan is to try to avoid the possibility of errors occurring within ROM routines. If that is not possible for a given routine, then prior to calling that routine you may wish to save the Stack Pointer in the IX or IY register pair, or in a pair of memory locations you have reserved for the purpose. Should the program go into the error-handling routine, it will get back to your program through 41A6H, and you can then reload the Stack Pointer from wherever it was stored (the error code will be found in the E register). For this to work successfully, you will need to do three things at the beginning of your program:

1 - Load the three-byte vector at 41A6H with a jump to your error-handling routine.

2 - Disable the ON ERROR GOTO trap using one of the methods in the above paragraphs.

3 - Load 40E8H-40E9H with the address of an alternate stack (will only have two register pairs PUSHed on it at once) that is out of the way of your program (one suggestion might be 40D2H, which is at the top of the string work area and is very unlikely to interfere with the running of your program).

If you are interfacing with BASIC through the USR command and decide to use the above method, remember that you will have to restore the changed memory locations to their original values before returning to BASIC.

CHAPTER FIVE - RESERVED RAM LOCATIONS

The ROM routines of the TRS-80 utilize a special area of memory in order to store the various flags, pointers, and vectors needed by the BASIC interpreter. This area is in RAM - Random Access Memory - that's the memory that you can write to as well as read from. However, we refer to this area as "reserved RAM" for the simple reason that this area is reserved for the use of the system - user programs are not normally stored in this area. In this chapter I will attempt to describe the various reserved RAM locations used by Level II/Model III BASIC.

You may recall the "MEMORY MAP" from your Level II BASIC Reference Manual. It looks something like this:

0000H - 2FFFH Level II BASIC ROM (0000H - 37FFH in Model III)

NOTE: Locations 3000H-37FFH as shown below apply to Model I only:

3000H - 37DDH Unused in Model I

37DEH	Communication status address
37DFH	Communication data address
37E0H	Interrupt latch address
37E1H	Disk drive select latch address
37E4H	Cassette select latch address
37E8H	Line printer address
37ECH	Disk controller command/status register address
37EDH	Disk track position register address
37EEH	Disk sector register address
37EFH	Disk data register address

3800H - 3BFFFH Memory-mapped keyboard

3C00H - 3FFFH Video display memory

4000H - 42E8H Level II BASIC fixed RAM (4000H - 43E8H in Model III)

It is this latter section of memory which will be covered in this chapter, and within that section only those addresses accessed by the Level II/Model III ROM will be documented. Furthermore, in a few instances portions of this area have already been documented in earlier chapters. When that is the case, the reader will be referred back to the appropriate section in order to avoid duplication of material which has already been presented in this book.

Before we do go on, however, it might be helpful to focus briefly on a few addresses from the above list. Model I, Level II BASIC uses these three addresses for its own purposes (Model III uses input/output ports instead, except as noted for 37E8H below):

37E4H - Prior to turning on the cassette motor, BASIC writes a "0" or "1" into this location to select cassette drive number one or two, respectively.

37E8H - If a parallel line printer is in use, it is mapped to this address. Writing a byte to this address sends it to the printer, while reading this address determines printer status as follows:

```
Bit 7 = 0 if NOT BUSY
Bit 6 = 0 if NOT OUT OF PAPER
Bit 5 = 1 if DEVICE SELECTED
Bit 4 = 1 if NO PRINTER FAULT
Bits 0 - 3 are not used
```

On the Model III this address is valid only for reading printer status (output is accomplished by writing to output port F8H, and printer status can be obtained through an input from port F8H as well). On either Model, the following line of BASIC code can be used to determine if the printer is ready:

```
IF (PEEK (14312) AND 240) = 48 THEN ... (printer is available)
```

37ECH - This location is tested during the power-up sequence to determine if a floppy disk controller is online, and if so the TRS-80 will jump to the disk bootstrap routine (unless the BREAK key is being depressed). If either zero or FFH is found at this address, then no disk controller is found and control goes to Level II BASIC.

So much for memory-mapped addresses. We'll skip past the memory-mapped keyboard (whole articles have been presented on this subject alone, including Dennis Kitsz's fine article in *The Alternate Source*, Volume 1, Number 5, which has been reprinted on pages 53-56 of "The Custom TRS-80 & Other Mysteries", published by IJG Inc.). We'll also skip the video memory, and get right into the reserved RAM starting at 4000H. From here on, everything applies equally to the Model I and the Model III unless otherwise noted.

RESERVED RAM ADDRESSES

4000H - 4014H Vectors for RST 0 through RST 38H. See the complete description of these vectors that appeared in chapter four of this book.

4015H - 402CH Device Control Blocks for Keyboard, Video, and Printer. See the complete description of the DCBs in chapter one of this book.

402DH - 402FH Normal Disk Operating System re-entry vector.

4030H - 4032H Abnormal DOS re-entry

4033H - 4035H DOS Device Vectoring Routine

4036H - 403CH Storage locations for keyboard scan routine. The contents of keyboard "rows" are stored here so that when the keyboard routine is called, it can determine whether the key has just been depressed (or whether that slow human hasn't taken his finger off the dadburned key yet!). These storage locations are organized as follows:

Storage address:	Keyboard row address:	Bit set if key pressed:							
		0	1	2	3	4	5	6	7
4036H	3801H	@	A	B	C	D	E	F	G
4037H	3802H	H	I	J	K	L	M	N	O
4038H	3804H	P	Q	R	S	T	U	V	W
4039H	3808H	X	Y	Z					
403AH	3810H	0	1!	2"	3#	4\$	5%	6&	7*
403BH	3820H	8(9)	:*	;+	,<	--	.>	/?
403CH	3840H	ENT	CLR	BRK	UP	DWN	LFT	RGT	SPC

Programmers wishing to detect keys outside of the normal keyboard scan can utilize these locations. For example, when the Model I keyboard scan routine is called, if either the SHIFT-0 or the spacebar are depressed a space (20H) character will be returned. However, if the spacebar was pressed location 403CH will have bit 7 set, while if the zero key was pressed location 403AH will have bit 0 set. Also, a zero can be POKEd into any or all of these locations to make the computer "forget" what keys were held down on a previous key scan (repeating keys, anyone?).

403DH - 407FH (Model III ONLY)-Used primarily by the Disk Operating System, vectors at 403DH, 4040H, and 4043H service interrupt vectors 3, 6, and 7 respectively (under non-disk systems all contain a jump to 35FAH, which in turn contains a "RET" instruction). 4046H is an interrupt vector servicing the clock (interrupt vector 2) and contains a jump to 35A9H in a non-disk system. 4049H is the Non-Maskable Interrupt vector and normally contains a RST 0 instruction. The remainder of this area is unused in non-disk systems.

403DH (Model I ONLY)-A flag byte used by the BASIC interpreter, this location contains the CURRENT PORT 0FFH OUTPUT BITS, which are organized as follows:

BIT 3	Select video 32 character mode if set
BIT 2	Turns on cassette tape relay if set
BITS 1 & 0	Are set for positive and negative audio pulses to the cassette "AUX" plug

Programmers that attempt to output to port FFH should be aware that at various points in the BASIC interpreter (such as when returning to BASIC "READY", doing tape I/O, etc.) some or all of the bits stored here may be output to port FFH, thereby canceling the previous status of port FFH.

403EH - 407FH (Model I ONLY)-Unused in non-disk systems. The real-time clock storage locations are located from 4040H to 4046H (see 4216H - 421CH on the Model III for details).

407DH - Stack pointer for DOS (set by ROM bootstrap routine).

4080H - 408DH Subroutine CALLED from 08CAH, used in non-integer precision division routine. On power-up this routine is moved from 18F7H - 1904H in the ROM to this location. Prior to calling the routine it is modified by instructions at 08B1H, 08B6H, 08BBH, 08C4H, 08D2H, and 08F4H.

408EH - 408FH USR routine entry point address.

4090H - 4092H Mantissa of constant used in generating random numbers.

4093H - 4095H Port input routine. Contains instructions IN A,(n) followed by RET, where the value of n is placed into the routine (at 4094H) prior to calling it.

4096H - 4098H Port output routine. Contains instructions OUT (n),A followed by RET, where the value of n is placed into the routine (at 4097H) prior to calling it.

4099H - INKEY\$ buffer (see information on the INKEY\$ routine in chapter three of this book). From BASIC (but not from machine language) it is possible to poke a "default" response into the INKEY\$ buffer. For example, consider the following BASIC line:

```
POKE 16537, 65 : A$ = INKEY$
```

If a key is being held down when this line is executed, A\$ will contain the character associated with that key, otherwise it will contain the letter A (which has an ASCII code of 65 decimal). This works in BASIC because between the execution of the POKE statement and the execution of the assignment (A\$=INKEY\$) statement, BASIC does a keyboard scan to check for the BREAK or SHIFT-@ keys, but if any other key is found to be depressed its ASCII character code will be stored at 4099H, replacing the value POKEd there. The INKEY\$ function itself checks first for a non-zero value at 4099H, and if none is found it does a keyboard scan, thus from a machine language program the keyboard would be ignored if a character is stored at 4099H. Note also that a POKE 16537,0 statement can be used to clear the INKEY\$ buffer, without creating a string or using string space to store an unwanted character.

409AH - Current Error code (returned by ERR function).

409BH - Current TAB position of line printer. This byte contains a count of the number of characters already printed on the current line, and may be used to simulate a TAB function that works correctly past the 63 or 127 character limit of the BASIC TAB function. For example, instead of this:

```
LPRINT TAB(n) "SOMETHING TO PRINT"
```

Which will not work properly if n is greater than 63 (or 127 on Model III or new-ROM Model I TRS-80's), use this:

```
LPRINT STRING$(n-PEEK(16539),32) "SOMETHING TO PRINT"
```

Remember that the above TAB function substitute could be made a defined function under Disk BASIC if desired:

```
DEFFNA$(X%) = STRING$(X%-PEEK(16539),32)
```

The above LPRINT statement would then be written as:

```
LPRINT FNA$(n) "SOMETHING TO PRINT"
```

409CH - Output device type flag. Directs output of many ROM routines to video display (if zero), printer (if +1), or cassette (if -1, or FFH). Normally contains zero (output to video).

409DH - Maximum number of characters on video display line. Used by PRINT command routine when printing numeric variables or constants so that a number does not overflow the end of a video display line. This location is set to 64 at power-up and is not changed by the BASIC interpreter once set. However, a POKE to this location may be used to prevent the PRINTing of numerics beyond a certain point on a line. Also, if you are operating in 32-character display mode it may be necessary to POKE 16541, 32 to prevent numeric printouts from overflowing the end of the line. The setting of this location does not affect the printout of strings (variables or constants).

409EH - The value in this location specifies the maximum number of 16-character print zones on a line (used when items in a PRINT statement are separated by commas). Value stored here is decoded as follows: 0 = one print zone, 16 = two print zones, 32 = three print zones, and 48 = four print zones per line (when a comma separator is found in a PRINT statement, BASIC checks the cursor line position byte at 40A6H and if it is greater than or equal to the value stored here, a carriage return is output to the video display driver). This location is set to 48 (four print zones) at power-up and is not changed by the BASIC interpreter once set.

409FH - Unused in non-disk systems.

40A0H - 40A1H Start of string storage area, set to 50 less than top of memory pointer (at 40B1H) at power-up to provide 50 bytes of string storage area. Changed by use of CLEAR n command (with argument).

40A2H - 40A3H Line number of BASIC line currently being executed, or FFFFH if in direct mode.

40A4H - 40A5H Start of BASIC program storage area. This may be changed by the user for various reasons (for example, to temporarily load a program in higher memory while retaining a program in lower memory), but BASIC will not operate properly unless the memory location just below the one pointed to contains a zero byte (in other words, if 40A4H-40A5H points to 7000H and a BASIC program is loaded, it will begin loading at 7000H, but in order to execute it memory location 6FFFH must contain a zero byte).

40A6H - Current cursor position on video display line (as returned by the POS function).

40A7H - 40A8H Start of keyboard buffer area, also used by BASIC while encoding and decoding BASIC lines (as during LIST, etc.)

40A9H - Used by INPUT command routine, contains zero byte if (and only if) input is from cassette. A bug in the first release of the Model I

Level II BASIC ROM required that this location be POKEd with a non-zero value prior to reading program DATA statements. This bug was corrected soon after Level II BASIC was released, so that this POKE is required only for older TRS-80 Model I machines.

40AAH - 40ACH Seed for next random number. Byte at 40ABH is loaded with current value of Refresh register whenever RANDOM statement is executed.

40ADH - Unused in non-disk systems.

40AEH - This flag is used by the BASIC locate or create variable routine. If location contains zero, variable is to be created or located (if already created). If byte is not zero, variable is to be created only and an error exists if the variable is found to be already created. The latter situation exists when an array is being created using the DIM statement.

40AFH - Number Type Flag (NTF) for ACCUM. Indicates type of number currently stored in ACCUM (2=integer, 3=string, 4=single precision, 8=double precision). See chapter two of this book for more information.

40B0H - Used to flag DATA statements while encoding BASIC lines, and to store operator number during expression evaluation.

40B1H - 40B2H Top of memory pointer. Last usable location for BASIC. This pointer is set by one of the following: If the user answers Memory Size? question with a value, it will be used. However, if only ENTER is pressed in response to Memory Size?, then on a non-disk system the highest available memory location will be used, but under Disk BASIC the DOS top of memory pointer will be copied into this location. If one is programming in BASIC and attempting to change this pointer by POKEing in new values, a CLEAR n statement (with a mandatory numeric argument) should be used immediately following the POKEs, in order to adjust other system pointers to the new memory size. Also, temporarily changing this pointer will allow a short BASIC program to CLEAR more than 32767 bytes of string space on a 48K system, provided that enough free memory is available (BASIC normally does not permit this). As an example, the following BASIC line will actually clear 16384 bytes more than is specified in the CLEAR statement (the equivalent of a "CLEAR 35000" statement in this case):

POKE 16562, PEEK (16562)-64: CLEAR 18616: POKE 16562, PEEK (16562)+64

40B3H - 40B4H Pointer to the next available location for storage of a three-byte string variable VARPTR in the string VARPTR storage area that begins at 40B5H.

40B5H - 40D2H String variable VARPTR storage area. Holds three-byte string descriptors (first byte contains length, second and third bytes contain address of string) for strings currently being used in BASIC string operations (such as "temporary" strings).

40D3H - 40D5H VARPTR storage area for string currently being created by BASIC (first byte contains length, second and third bytes contain address of string).

40D6H - 40D7H Pointer to next free byte in string storage area. Strings build downward from the top of memory, therefore at power-up (or when a CLEAR command is executed) this pointer will contain the same address as the top of memory pointer (40B1H-40B2H). If a ten byte long string is then created, this pointer will point to the top of memory minus ten, and so on. When there is not enough room left to insert a new string (the difference between this pointer and the one at 40A0H-40A1H is less than the length of a string to be stored), a "garbage collection" is performed, and if that does not free enough string space an Out of String Space error occurs. Under some circumstances, we may be able to manipulate this pointer in order to forestall a "garbage collection" (which may appear to "lock up" the computer for as much as several MINUTES). For example, if we are performing some operation that will create many "temporary" strings (but none that we want to save after a certain point), we could PEEK at the values in these pointer locations and store them in numeric variables. At the completion of our string operation, we could then re-POKE the original pointer values, thus abandoning the unneeded strings without performing a "garbage collection". Under certain circumstances this technique could cut program execution time considerably.

40D8H - 40D9H This pair of locations is used a temporary storage location by more than one routine. Uses include program pointer during expression evaluation, pointer to data while processing DIM statement, pointer to end of array while packing strings, and within the PRINT USING routine the byte at 40D8H is used to temporarily store the PRINT USING format flag bits.

40DAH - 40DBH Line number of last DATA item read. Line number is saved so that in the event of a syntax error in a DATA statement, the proper error line number will be displayed and the EDIT mode will function on the correct line.

40DCH - If this byte contains 64 a FOR-NEXT loop is being processed, otherwise byte will contain zero. Used to prevent an array variable from being used as the index counter in a FOR-NEXT loop (for example, the statement FOR X(0) = 1 TO 10 will cause a syntax error, because X(0) is an array variable and cannot be used as the counter variable).

40DDH - Flag indicates whether inputting text. Used by RETURN and RESUME NEXT commands.

40DEH - Used for two different purposes: Flags whether READ or INPUT statement when processing those commands (zero represents INPUT). Also used to store delimiter character during processing of PRINT USING statement.

40DFH - 40E0H Used by several routines, as a pointer to variable receiving new value during evaluation of a "LET" expression, execution address of BASIC program, etc. Also, after a SYSTEM tape is loaded

these locations contain the entry point address of the program (which is used if only a "/" is typed in response to the next SYSTEM command prompt).

40E1H - AUTO input flag. Zero byte here means AUTO is not on.

40E2H - 40E3H Current input line number used by AUTO input function.

40E4H - 40E5H Line increment used by AUTO input function.

40E6H - 40E7H Used by several routines for temporary storage of various pointers to the BASIC program (saved position in program text, buffer position during text encoding, etc.).

40E8H - 40E9H Temporary storage of the BASIC stack pointer origin.

40EAH - 40EBH Line number of last error, as returned by ERL function.

40ECH - 40EDH Current or error line number (used when a period is used to replace a BASIC line number, and when syntax error occurs in program).

40EEH - 40EFH Pointer to last byte executed when error occurred. Used by RESUME command.

40F0H - 40F1H Address of error trap line (points to first statement of line specified in ON ERROR GOTO statement), or zero if no error trap set.

40F2H - Flag to indicate whether an error has occurred. Set to -1 (FFH) on error, otherwise set to zero (such as after RESUME statement has been executed). POKEing the appropriate values into this location will allow you to do many things that BASIC normally does not permit, such as exiting an error trap without using a RESUME statement, or redefining the error trap (by using an ON ERROR GOTO statement) from within a previously defined error trap.

40F3H - 40F4H Used to store position of expressions being evaluated, as a decimal point location pointer, etc.

40F5H - 40F6H Last line number executed prior to execution of STOP or END statement or termination using the BREAK key.

40F7H - 40F8H Pointer to end of last statement executed (points to the colon or zero byte terminator). Used by CONT command.

40F9H - 40FAH Pointer to start of simple variables, or one memory location higher than the last of the three zero bytes marking the end of the BASIC program. Used to determine end of BASIC program when saving the program. Note that by taking the address stored here, subtracting 2, and POKEing the resulting address into the start of BASIC program pointer at 40A4H-40A5H, then loading a program with higher line numbers than the one presently in memory, and finally re-POKEing the original values of 40A4H-40A5H back into those

locations, it is possible to append a program to a program already in memory. From the BASIC command mode, this is done as follows:

```
>? PEEK (16548), PEEK (16549)  'write down the results
>POKE 16548, PEEK (16633)-2 : POKE 16549, PEEK (16634)
>CLOAD  'or LOAD program to be appended
>POKE 16548, n1 : POKE 16549, n2 'replace n1 and n2 with values
you wrote down in first step
```

Note: If PEEK (16633) is less than two, the second step above will bomb with an error message. If that happens, replace it with the following line:

```
>POKE 16548, PEEK (16633)+254 : POKE 16549, PEEK (16634)-1
```

40FBH - 40FCH Pointer to start of array variables.

40FDH - 40FEH Pointer to end of array variables (start of "free" memory).

40FFH - 4100H Pointer to end of last DATA item read. Points to comma or other terminator at end of last item read - search for next DATA item to be read will begin here. RESTORE changes this pointer to point to one byte prior to beginning of BASIC program. A selective RESTORE to some data item past the first can be accomplished by manipulating this pointer. For example, suppose that at some point in your program you will want to RESTORE to the 51st DATA item. At the start of your program you could insert a line similar to this:

```
FOR X=1 TO 50: READ X$: NEXT: R1=PEEK(16639): R2=PEEK(16640): RESTORE
```

Then, whenever you wanted to RESTORE to the 51st DATA item, you would simply execute the following statements:

```
POKE 16639,R1: POKE 16640,R2
```

The next READ instruction would then get the 51st DATA item.

4101H - 411AH Variable default type declaration table. Each memory location in this table applies to variables beginning with a particular letter of the alphabet - the location at 4101H applies to variables starting with the letter A, 4102H applies to variables starting with B, 4103H to variables starting with C and so on up to 411AH, which contains the default type for all variables starting with the letter Z. Each location contains one of the following values: 2 (integer), 3 (string), 4 (single precision), or 8 (double precision). When a variable name is used in a BASIC program and does not have a specific type declaration character (%\$, !, or #) following, its type defaults to the type specified in this table. All defaults are set to single-precision when a program is RUN, and may be changed through use of the DEFINT, DEFSTR, DEFSNG, or DEFDBL statements.

411BH - Trace flag. If zero, trace is off, while AFH indicates that trace is on (usually controlled by TRON and TROFF commands).

411CH - 412EH This area contains the arithmetic accumulators (ACCUM and ACCUM2) and some associated locations. These are fully detailed in chapter two of this book.

412FH - Apparently unused in non-disk systems.

4130H - 4149H Buffer used to store result of conversion of numbers to displayable ASCII characters. When positive integers (BASIC line numbers, etc.) are converted, the area from 4130H to 4136H is used, and the string is stored right justified with leading spaces in locations 4130H to 4135H (4130H always contains a space character), while 4136H always contains a zero byte to terminate the string. Other routines (such as PRINT USING) may use more of this area.

414AH - 4151H Temporary ACCUM sometimes used by arithmetic routines (such as to hold divisor used by double precision division routine).

4152H - 41A5H contain three-byte vectors to Disk BASIC commands and functions. Under non-disk BASIC, all of these vectors (with the exception of TIMES\$ on the Model III only) contain a jump to 012DH, the "L3 Error" routine (Model III TIMES\$ normally contains JP 3030H). Under Disk BASIC these vectors are modified, so that the jumps are to the various Disk BASIC routines that support these commands and functions. Non-disk users may place jumps to their own machine language routines in these vectors, so that the routine can be called by using the corresponding BASIC keyword. The addresses of these vectors (and corresponding keywords) are as follows:

4152H	CVI
4155H	FN
4158H	CVS
415BH	DEF
415EH	CVD
4161H	EOF
4164H	LOC
4167H	LOF
416AH	MKI\$
416DH	MKS\$
4170H	MKD\$
4173H	CMD
4176H	TIME\$
4179H	OPEN
417CH	FIELD
417FH	GET
4182H	PUT
4185H	CLOSE
4188H	LOAD
418BH	MERGE
418EH	NAME
4191H	KILL
4194H	&
4197H	LSET
419AH	RSET
419DH	INSTR
41A0H	SAVE
41A3H	LINE

41A6H - 41E4H contain three-byte Disk BASIC links which are used to extend the capabilities of various BASIC routines when a Disk BASIC is present. Under non-disk systems, the first byte of each of these vectors is a machine language RET instruction. A short description of each of these links follows:

41A6H - Called at 19ECH, from error-handling routine. Used to provide long error messages.

41A9H - Called at 27FEH, from entrance to USR function routine. Used to expand function to provide up to ten USR calls. If a program running under Disk BASIC contained the statement POKE 16809,201 (placing a RET instruction at 41A9H), the expanded USR function would be defeated and the single USR call of non-disk BASIC would be restored. Such a program could run correctly under either non-disk or Disk BASIC, provided that the normal Level II/Model III BASIC requirement for POKEing the USR routine starting address into memory locations 408EH-408FH is adhered to (rather than the use of the DEFUSR command that Disk BASIC normally requires).

41A1CH - Called at 1A1CH, from BASIC re-entry ("READY").

41AFH - Called at 0368H, from near start of routine to input a line from the keyboard into the I/O buffer (zeroes INKEY\$ buffer and video tab position indicator prior to call).

41B2H - Called at 1AA1H, after BASIC line has been tokenized (HL points to start of tokenized line).

41B5H - Called at 1AECH, after insertion or replacement of BASIC line

41B8H - Called at 1AF2H, after above call followed by call to 1B5DH (CLEAR command, leaves HL pointing to start of BASIC program -1).

41BBH - Called at 1B8CH (from NEW command) and at 1DB0H (from END command). Used to CLOSE any files remaining open.

41BEH - Called at 2174H, at termination of PRINT statement (used to terminate output to disk using PRINT # statement).

41C1H - Called at 032CH, from byte output routine that starts at 032AH.

41C4H - Called at 0358H, from start of routine to scan keyboard and input keystroke (if any).

41C7H - Called at 1EA6H, when RUN command has an argument (filename) following.

41CAH - Called at 206FH, from start of PRINT command (check for output to disk using PRINT #).

41CDH - Called at 20C6H, from PRINT routine after a numeric item has been converted to an ASCII string in preparation for printing.

41D0H - Called at 2103H, from PRINT routine after code which outputs a carriage return (to prevent numeric items being printed from overflowing the end of a line).

41D3H - Called at 2108H (from PRINT command when a comma is used to separate items to be printed) and at 2141H (from PRINT command routine when the TAB function is used, after the TAB argument has been evaluated and placed in the E register. It is worth noting that the original argument (if within the range 0-255) is still stored as an integer in ACCUM, with the NTF set to 2, indicating an integer (when this vector is called from the comma separator routine the NTF will normally be set to 3, indicating a string). Thus, if it is desired to be able to use TABs in the range 0-255 (instead of 0-63 or 0-127 as allowed by the ROM), this vector may be altered to jump to the following patch routine which will permit the longer TABs:

```

E7      RST      20H      ;Check Number Type Flag
FC052B  CALL     M,2B05H ;DE=original TAB if NTF=integer
C9      RET      ;Cassette BASIC only

```

Disk BASIC users should use a jump to the routine formerly called from this vector in place of the RET instruction).

41D6H - Called at 219EH, from start of INPUT command (check for input from disk using INPUT #).

41D9H - Jump to this location at 2AECH, to Disk BASIC routine that permits MID\$ on left side of "=" (assignment statement).

41DCH - Called at 222DH. Part of READ/INPUT command routine, called just prior to assigning data that has been READ or INPUT to variable.

41DFH - Called at 2278H (after data from INPUT command has been assigned to variable, just prior to test for "extra" data that would cause an "?EXTRA IGNORED" message to be printed) and at 2B44H (from LIST command after start and end addresses of program have been found and a check for end of program has been made)

41E2H - Called at 02B2H. This call is located at the entrance to the SYSTEM command and is executed each time the "*?" prompt is about to be displayed. If a SYSTEM tape is being loaded and it places a jump to its entry point address in this vector, the program will start automatically. The same thing can be achieved by placing a JP (HL) instruction (an E9H byte) at 41E2H, provided that the entry point address on the tape is correct. Note, however, that the SYSTEM command will be unusable afterwards unless the program reloads a RET instruction (a C9H byte) to 41E2H after execution begins.

41E5H - 42E8H (42E5H - 43E8H on Model III) Addresses 41E5H-41E7H (42E5H-42E7H on Model III) are initialized by ROM code at 0080H-0089H to contain the bytes 3AH, 0, and 2CH (a colon, zero byte, and comma). The BASIC input/output buffer (also used while encoding and decoding BASIC lines) begins one byte higher (41E8H or 42E8H) and is defined by a continuation of the above mentioned ROM code (008AH - 008DH), which initializes the buffer pointer at 40A7H-40A8H. The buffer is exactly 256 (100H) bytes long, and is immediately followed by a byte that is always zero, and marks the end of reserved RAM (42E8H or 43E8H).

During power-up, the system stack briefly resides within this buffer (the Stack Pointer is initialized at 41F8H or 42F8H by the instruction at 00A8H), and on the Model I, the SYSTEM command places its Stack Pointer within this buffer, at 4288H.

ADDITIONAL LOCATIONS USED IN MODEL III

The following locations apply to the Model III (and Model 4 when used in the Model III mode) ONLY:

41E5H - 41FCH Device Control Blocks for RS-232-C Input, Output, and Initialization. See the complete description of the DCBs in chapter one of this book. Model 4 users should also see ADDITIONAL LOCATIONS USED IN MODEL 4 below.

41FDH - Used by repeating key routine. Saves LSB of keyboard buffer pointer (buffer at 4036H-403CH) when key is found depressed.

41FEH - Used by repeating key routine. Saves contents of current keyboard "row" when key is found depressed.

41FFH - 4200H Used by repeating key routine. Holds maximum repeat delay count (1500 decimal or 5DCH for first repeat delay, 150 decimal or 96H for subsequent repeats of same character).

4201H - 4202H Used by repeating key routine. Holds current repeat delay count, which is compared with value at 41FFH-4200H to determine if it is time to repeat the character.

4203H - 4205H Three byte BREAK key vector used when BREAK is pressed during cassette tape or RS-232-C operations. Initialized to contain jump to 022EH, which in turn contains EI and JP 1A19H instructions.

4206H - 4208H Three byte vector services interrupt # 4. Under non-disk systems contains a jump to 35FAH, which in turn contains a "RET" instruction.

4209H - 420BH Three byte vector services interrupt # 5. Under non-disk systems contains a jump to 35FAH, which in turn contains a "RET" instruction.

420CH - 420DH Pointer to "write byte to cassette" routine. Initialized by "write cassette leader" routine, normally contains address of either 500 baud (3241H) or 1500 baud (32BAH) byte output routine.

420EH - 420FH Pointer to "read byte from cassette" routine. Initialized by "read cassette leader" routine, normally contains address of either 500 baud (3203H) or 1500 baud (32CAH) byte input routine.

4210H - A flag byte used by the BASIC interpreter, this location contains the CURRENT PORT 0ECH OUTPUT BITS, which are organized as follows:

BIT 6	Enables fast clock speed if set on Model 4 ONLY
BIT 5	Disables video wait states if set (not used on Model 4).
BIT 4	Enables I/O bus if set
BIT 3	Japanese Kana character set used as "special" characters if set
BIT 2	Select video 32 character mode if set
BIT 1	Turns on cassette tape relay if set
BIT 0	Enables clock display on video if set

Programmers that attempt to output to port ECH should be aware that at various points in the BASIC interpreter (such as when returning to BASIC "READY", doing tape I/O, etc.) some or all of the bits stored here may be output to port ECH, thereby canceling the previous status of port ECH. Also, it must once again be noted that in all current editions of the Model III ROM, an error exists in that the test for double-width characters at 0348H has not been changed to test the flag at 4210H rather than the flag at 403DH. This can cause serious problems when attempting to make use of the 32-character mode on the Models III and 4.

4211H - Cassette speed flag. If zero, 500 baud speed is used for tape input/output, otherwise 1500 baud speed is used.

4212H - Asterisk blink counter. Each time a byte is read from tape, this counter is incremented and ANDed with 5FH. If the result is zero, the status of the asterisk in the upper right corner of the video display is reversed (turned on if it was off, or off if it was on).

4213H - Default interrupt vector setting. This byte is output to port E0H (the maskable interrupt latch) whenever the "turn off cassette" routine is executed.

4214H - Number of lines to protect at top of video display. Only the lowest three bits are used (byte is ANDed with 7 to determine number of protected lines).

4215H - Unused in non-disk systems.

4216H - 421CH Clock storage locations. The corresponding addresses in the Model I are also given in the following table:

Model III	Model I	
Address:	Address:	Storage for:
-----	-----	-----
4216H	4040H	Heartbeat Counter
4217H	4041H	Seconds
4218H	4042H	Minutes
4219H	4043H	Hours
421AH	4044H	Years
421BH	4045H	Days
421CH	4046H	Months

421DH - 4223H Device Control Block for Input/Output Router. See the complete description of the DCBs in chapter one of this book.

4224H - Control key flag used by keyboard driver routine. Contains 1FH if control key sequence (left SHIFT and down-arrow keys) was pressed, else contains FFH. Can be tested to determine how certain control codes were produced (for example, whether a carriage return character was produced by pressing the ENTER key, or through the use of the Control-M sequence).

4225H - 42E4H Presently undefined in non-disk systems, except that the SYSTEM command places its Stack Pointer within this area, at 4288H

42E5H - 43E8H See above under "41E5H - 42E8H" (equivalent Model I addresses).

ADDITIONAL LOCATIONS USED IN MODEL 4

The following locations apply to the Model 4 when used in the Model III mode ONLY. Although they are technically inside the boundaries of the RS-232-C Input and Output Device Control Blocks, they are actually referenced only by the Model 4 Keyboard Driver routine, and as such merit further discussion here:

41EBH - ASCII character that will be returned when "F1" key is depressed. Normally set to 60H (shift-@) during BASIC initialization, but the character stored at this location may be changed, so that any desired character may be returned when "F1" is pressed.

41ECH - ASCII character that will be returned when "F2" key is depressed. Normally set to 1BH (shift-up arrow) during BASIC initialization, but the character stored at this location may be changed, so that any desired character may be returned when "F2" is pressed.

41F3H - ASCII character that will be returned when "F3" key is depressed. Normally set to 08H (backspace) during BASIC initialization, but the character stored at this location may be changed, so that any desired character may be returned when "F3" is pressed.

41F4H - Storage location for keyboard scan routine. The contents of the keyboard "row" at 3880H is stored here (see the discussion on memory locations 4036H through 403CH for more information on the use of the keyboard row storage locations). This storage location is organized as follows:

Bit set if key pressed:							
0	1	2	3	4	5	6	7
<hr/>							
LSHFT	RSHFT	CTRL	CAPS	F1	F2	F3	(unused)

APPENDIX I

THE BASIC DIFFERENCES
Model I & III ROM BASIC Compared

(This appendix is a revision of an article that originally appeared in The Alternate Source, Volume 3, Number 2; Issue 14; March, 1982. It details the differences between the Model I and the Model III ROM. References to the Model 4 have been added near the end of the article, but otherwise it remains relatively unchanged from its original form.)

I have this love/hate thing about the Model III TRS-80. For example, I loved it when I found out that turning the power on or off with a disk still in the drive did not necessarily sprinkle garbage all over the disk (I still wouldn't recommend making a habit of doing that, though). I loved the sharper video display with built-in lowercase and better graphics.

I've decided to hold on to my Model I for a while longer, though, because I hate the way they destroyed the beauty of the BASIC interpreter. Let me explain. Model I BASIC was a masterpiece of assembly language programming. The code was efficient and tight, with hardly a wasted byte. Its flaws were few, and for the most part easily corrected in later editions.

The Model III ROM, however, is something like an old art masterpiece that has been enlarged and touched up by an art student. The ROM additions are clumsy by comparison. Much space is wasted, both by the routines themselves, and by scattered areas throughout the ROM where a few bytes here or there are "dead" -- leftover pieces of old code, etc. that is not used by the new ROM. To some extent, this may have been unavoidable, but it seems that more care could have been taken to make the new code interface well with the old code. More new features could have been placed in the added 2K of ROM, or more of the Model I code could have been retained in order to lessen program compatibility problems, or both.

In any event, the bytes of Model III BASIC have already been etched in silicon, so the purpose of this article will be to explore the differences between the Level II BASIC so familiar to Model I users, and the new version designed for the Model III.

A word about the listings below - ALL differences between the Model I and Model III ROMs are shown. The Model I ROM version 1.3 (the latest version of the so-called "old" ROM) was compared to the latest known version of the Model III ROM (at the time this article was originally written). Differences are shown in the format nnnnH - nnnnH, where all addresses falling within the range shown are different in the two ROM versions. At certain locations, the two versions may by coincidence be the same for a byte or two even though the routines found in those areas are completely different. This explains why the start or end of some routines may not exactly coincide with the start or end of the "different" portion of ROM, or why a routine may have more than one block of "different" code.

If an address falls within one of the ranges shown below, you can be sure that the byte stored there was different in the Model I ROM 1.3, and in the latest Model III ROM. Otherwise, the byte was the same in the two ROM versions. By checking the text nearest a ROM address of interest, you should be able to determine if the routine you wish to work with has been relocated or modified. Be sure to read the text carefully if a memory location near the one you are interested in has been changed, because as mentioned, sometimes bytes are the same due to coincidence.

THE ROM DIFFERENCES

0003H - 0004H Part of JP instruction that begins at 0002H. The jump is to the power-up routine, located at 0674H in the Model I but jump is to 3015H in the Model III (3015H is the location of a vector to the start of the actual routine).

000EH - 000FH Part of JP instruction that begins at 000DH. The jump is to the disk bootstrap loader routine, located at 069FH in the Model I but jump is to 3012H in the Model III (3012H is the location of a vector to the start of the actual routine).

0047H - 0048H Part of JP instruction that begins at 0046H. The jump is to the I/O driver entry routine, located at 03C2H in the Model I but relocated to 0674H in the Model III.

0050H - 0062H In the Model I, locations 0050H through 005FH are a lookup table for special characters associated with the keyboard scan routine. These same locations in the Model III contain the entry points for routines associated with the RS-232-C interface (Receive character at 0050H, Transmit character at 0055H, Initialize RS-232-C at 005AH). In both models, 0060H is the start of a time-delay routine, but in the Model III the three bytes starting at 0060H have been changed to a JP 01FBH instruction. The actual time delay routine (same as in the Model I except that an extra instruction is added to compensate for the faster clock speed of the Model III) has been moved to 01FBH in the Model III. Note that entry at 0063H has the same effect in both models, that is, the time delay will occur only if the Z flag is reset (NZ).

0066H - 0070H Pressing the RESET button causes a non-maskable interrupt, which in turn forces a jump to 0066H. In the Model I, a portion of the code that handles non-maskable interrupts is located from 0066H to 0071H. In the Model III, 0066H contains a jump to 3039H (the non-maskable interrupt routine in the Model III), 0069H contains a jump to 0452H (routine to initialize all I/O drivers), and 006CH is the beginning of the routine to route I/O (loads DE with 421DH and then jumps to 001BH).

0082H - 0082H Instruction starting at 0080H loads HL with 41E5H in the Model I, 42E5H in the Model III. Following this, three memory locations (starting with the one pointed to by HL) are loaded with the values 3AH, 00H, and 2CH respectively. HL is then incremented once more and the result (41E8H or 42E8H) points to the start of the input buffer (and is stored at 40A7H).

00AAH - 00AAH Instruction starting at 00A8H loads HL with 42E8H in the Model I, 43E8H in the Model III. This memory location is then zeroed (BASIC programs begin at the FOLLOWING memory location in non-disk systems).

00AEH - 00AEH Instruction starting at 00ACH loads the Stack Pointer with 41F8H in the Model I, 42F8H in the Model III.

00B2H - 00B4H In the Model I, a CALL to 01C9H (the "clear screen" routine) is located here (just prior to printing the "MEMORY SIZE?" prompt on the video display). In the Model III this has been replaced by three zero bytes (NOPs).

00C6H - 00C6H Instruction starting at 00C4H loads HL with 434CH in the Model I, 444CH in the Model III. If only "ENTER" was pressed in response to the "MEMORY SIZE?" prompt, a memory test is initiated starting at the location pointed to by HL, and continuing upward until the end of memory (or a bad memory location) is reached.

00EAH - 00EAH Instruction starting at 00E8H loads DE with 4414H in the Model I, 4514H in the Model III. This is the minimum "MEMORY SIZE" that can be specified by the user.

00FFH - 0101H The instruction previous to this loads HL with the starting address of the opening message ("RADIO SHACK..."). In the Model I, a CALL to 28A7H (the "display message" routine) is stored here, while in the Model III this has been changed to a jump to 37EBH.

0106H - 010AH Part of "MEMORY SIZE" prompt (changed from "EMORY" to "emory").

010DH - 010FH Part of "MEMORY SIZE" prompt (changed from "IZE" to "ize").

0112H - 0115H Part of "RADIO SHACK" (changed from "ADIO" to "adio").

0118H - 011BH Part of "RADIO SHACK" (changed from "HACK" to "hack").

011DH - 0121H Changed from "LEVEL" to "Model".

0125H - 012CH Part of "LEVEL II BASIC" or "Model III Basic" (changed from "BASIC" followed by carriage return and zero byte to "I Basic" followed by carriage return only).

01DAH - 01EFH The routine to print the contents of the screen on the line printer is located from 01D9H to 01F4H on the Model III. On the Model I, 01D9H - 01F7H contains the routine to output one bit to the cassette.

01F1H - 01F1H 01F0H contains CALL 0221H instruction on Model I, CALL 0214H instruction on Model III. See above.

01F3H - 01F4H Contains LD B,5CH instruction on Model I, JR 01DCH instruction on Model III. See above.

01F8H - 020FH In the Model I, the routine to turn off the cassette recorder is located from 01F8H to 0211H. In the Model III, 01F8H contains a jump to 300CH (the location of a vector to the "turn off cassette" routine in the Model III). 01FBH through 0201H contain the time delay routine (see notes on 0060H), and 0202H through 020FH contains the text "(c) '80 Tandy" and a carriage return.

0212H - 0231H In the Model I, routines to define cassette drive (0212H - 021DH), reset the cassette input port FFH (021EH - 022BH), and to blink the asterisk while reading a cassette (022CH - 0234H). In the Model III, a routine to insure compatibility with programs that define the cassette drive (XOR A followed by RET, located at 0212H & 0213H), a subroutine used by the routine that begins at 01D9H (0214H - 0227H), a couple of cassette-related segments (0228H - 022DH), and an EI instruction followed by JP 1A19H (enable interrupts and return to BASIC "READY", located at 022EH - 0231H).

0235H - 0245H In the Model I, 0235H - 0240H contains the routine to read one byte from the cassette, and 0241H - 0260H contains the routine to get one bit from the cassette. In the Model III, 0235H - 023CH contain the start of the Model III routine to read one byte from cassette, 023DH - 0242H is part of the routine that begins at 0287H (writes cassette leader and sync byte), 0243H - 024CH is the actual start of the routine to search for the cassette leader and sync byte, 024DH - 0252H is the actual start of the routine to write a byte to tape, and 0253H - 025EH is a subroutine used by the system to select 500 or 1500 baud tape speed.

0247H - 025EH See above.

0264H - 0282H In the Model I, 0264H - 0283H contains the routine to output one byte to the cassette. In the Model III, 0264H - 0266H contains a jump to 024DH (the start of the Model III routine to output one byte to cassette), followed by time data (60 seconds, 60 minutes, 24 hours) at 0266H - 0268H, followed by twelve bytes which contain the length of each of the twelve months (0264H - 0274H). This is followed by two NOPs, then starting at 0277H is a 1DH byte, a 1EH byte, the message "Diskette?", and finally a 03H byte (at 0282H).

0284H - 02A7H In the Model I, this area contains several cassette I/O routines, including turn on cassette, write leader and sync byte (0284H); write leader and sync byte (0287H); turn on cassette, search for leader and sync byte (0293H); search for leader and sync byte (0296H), put 2 asterisks in upper right corner of video (part of previous routines, begins at 029FH). In the Model III, 0284H contains a JP 0287H instruction (faster than three NOPs), while 0287H is the start of the routine to turn on the cassette, write leader and sync byte. 028DH - 0292H contains the fast routine to check if BREAK is depressed. 0293H contains a JP 0243H instruction, while 0296H contains a JR 0243H (0243H is the actual start of the routine to turn on the cassette, search for leader and sync byte). 0298H - 02A0H is the machine language routine to turn on the built-in clock display (in the upper right hand corner of the video display), while 02A1H - 02A8H

is the location of the corresponding routine to turn the clock display back off.

02E2H - 02E4H This is part of the SYSTEM tape load routine. In the Model I, these locations contain a JR NZ,02D1H instruction followed by an INC HL instruction. In the Model III, these instructions have been in effect reversed (INC HL followed by JR NZ,02D1H). Since this area is not referenced by any other part of the ROM, I can only guess that this change may have been implemented in order to correct some problem that occurred while reading the filename on a SYSTEM tape.

03C2H - 03E9H Start of I/O driver area which has been totally rearranged in the Model III. Specifically, in the Model I the area from 03C2H through 05D0H was arranged as follows: 03C2H - 03E2H is the I/O driver entry routine, 03E3H - 0457H is the keyboard driver routine, 0458H - 058CH is the video driver routine, and 058DH - 05D0H is the line printer driver routine. In the Model III, 03C2H - 0451H is the line printer driver routine, 0452H - 0468H is the actual location of the routine to initialize all I/O drivers, 046BH - 0472H is a routine used by the RUN/EDIT/NEW commands to unprotect the video display and to load HL with the start of BASIC program pointer at 40A4H - 40A5H, and 0473H - 05D0H is the video driver routine (just in case you're wondering, the keyboard driver begins at 3024H in the Model III).

03EBH - 0468H	See above.
046BH - 0494H	See above.
0496H - 049EH	See above.
04A0H - 04B7H	See above.
04B9H - 050CH	See above.
050EH - 0532H	See above.
0534H - 05CFH	See above.

05D1H - 05D3H This change appears to be malicious destruction of the "printer ready" test routine found at 05D1H - 05D8H in the Model I. In the Model III, the first three bytes of the routine are changed from a LD A,(37E8H) instruction to 52H, 4FH, and 4EH (LD D,D; LD C,A; and LD C,(HL); or the ASCII characters "RON"). The entire routine from 05D1H - 05D8H is unused in the Model III, so I can't understand why Microsoft didn't just leave these three bytes unchanged - the routine would have still worked and it would have been one less area of incompatibility between the two Models. An equivalent routine that does work (and which would also have fit nicely into this space) resides from 044BH to 0451H in the Model III (part of the printer driver routine).

0674H - 0707H In the Model I, this is part of the power-up routine and includes the disk bootstrap routine (06A1H - 06CBH), the preferred BASIC re-entry routine (06CCH - 06D1H), and the RST vectors and I/O Device Control Blocks which are relocated to RAM on power-up (06D2H - 0707H, relocated to 4000H - 4035H). In the Model III, this area contains the I/O driver entry routine (0674H - 0699H) and additional code used by the LIST and LLIST commands (069AH - 0707H). Note that a jump to 06CCH will no longer get you back into BASIC in the Model III. This might be considered a major blunder in the design

of the Model III ROM, both because so many available programs use this re-entry point to BASIC, and because the ROM itself has two jumps to 06CCH (one at 0072H that appears to be unused, and one at 02C3H, which is used if the BREAK key is pressed under the SYSTEM command - a bona fide BUG in the Model III ROM). A substitute re-entry to BASIC that will work with either model is to LD BC,1A18H and then JP 19AEH (this is the code presently found at 06CCH in the Model I).

124CH - 124DH This change first appeared in the "new" ROMs for the Model I. The order of two instructions (OR A and POP DE) have been reversed for no apparent reason.

1918H - 1918H 1917H - 1918H contains 434CH in the Model I, 444CH in the Model III. This value is loaded to 40A0H (the start of string space pointer) during power-up.

191CH - 191CH 191BH - 191CH contains 42E9H in the Model I, 43E9H in the Model III. This value is loaded to 40A4H (the start of BASIC program pointer) during power-up.

1B5DH - 1B5FH Part of the RUN, NEW, EDIT, etc. commands; in the Model I these three bytes contain a LD HL,(40A4H) instruction (resets HL to point to the start of the BASIC program). In the Model III, this has been replaced by a CALL 046BH instruction, which unprotects the video display in addition to resetting HL to the start of the BASIC program.

206DH - 206DH All of the changes from 206DH - 20F7H first appeared in the "new" ROMs of the Model I. The changes were made to allow the use of multiple @'s in a PRINT statement.

2073H - 2073H	See above.
2075H - 2075H	See above.
2077H - 20B8H	See above.
20BCH - 20BCH	See above.
20F7H - 20F7H	See above.

213BH - 213BH This also first appeared in the "new" ROMs of the Model I, as part of an AND 7FH instruction located at 213AH. In the original version the instruction was AND 3FH. The operand of the instruction sets the maximum argument for the TAB function, thus the early TRS-80's could only handle a maximum TAB (63) while the latest Models can go as high as TAB (127).

2167H - 2167H Another "new" ROM change, this corrects a jump back into the revised PRINT command routine.

2B85H - 2B88H This change is to the LIST command. JP 069AH in the Model III replaces LD D,0FFH followed by JR 2B8CH in the Model I.

2B8CH - 2B8EH Another change to LIST. In the Model I three instructions occupied this area: LD A,(HL); OR A; INC HL. In the Model III the INC HL instruction is moved to the beginning of the three.

2B91H - 2B93H The final change to LIST - a JP P,2B89H instruction in the Model I is changed to a JP 302DH instruction in the Model III.

2C1FH - 2C42H Another change that originated in the "new" ROM Model I modified the CLOAD command (changed the order of portions of the CLOAD routine, disallowed CLOAD from cassette drive #2, etc.).

2C7AH - 2C7FH Also part of CLOAD, this change turns off the tape before printing READY on the video display. In the Model I, the code from 2C7AH - 2C82H consisted of the instructions LD HL,1929H; CALL 28A7H, CALL 01F8H. In the Model III the CALL to 01F8H has been moved to the beginning of these instructions.

2C81H - 2C82H See above.

2C8AH - 2C8CH This is part of the CLOAD? routine used when a bad byte has been read from the tape. In the Model I, a LD HL,2CA5H instruction is found here (loads HL with the starting address of the "BAD" message), while the Model III uses a CALL 31BDH instruction, which does some housekeeping in addition to pointing HL to the "BAD" news.

3000H - 37FFH This portion of memory is non-existent in the Model I (except that in the Model I, some hardware devices are "memory mapped" to the area from 37E0H - 37FFH). However, some words about this area are in order. First of all, in the Model III the area from 3000H through 3044H contains jump vectors to routines located in the remainder of the top 2K of ROM. Apparently "routines got legs", that is, they are subject to being moved around in different revisions of the ROM. At least three different versions of this added 2K of ROM have appeared, and the locations of a few of the routines have been moved slightly. So, you are advised to enter routines through the vectors where possible. Here is a list of the routines available through these vectors:

- 3000H Write 500 baud cassette leader and sync byte.
- 3003H Write 1500 baud cassette leader and sync byte.
- 3006H Search for 500 baud cassette leader and sync byte.
- 3009H Search for 1500 baud cassette leader and sync byte.
- 300CH Turn off cassette.
- 300FH Turn on cassette.
- 3012H Disk bootstrap routine.
- 3015H Power-up routine (jump here from 0002H).
- 3018H Maskable interrupt handler.
- 301BH RS-232-C initialization driver routine.
- 301EH RS-232-C input driver routine.
- 3021H RS-232-C output driver routine.
- 3024H Keyboard driver routine.
- 3027H I/O Route driver routine (not available on Model 4).
- 302AH Part of routine to search for cassette leader and sync byte (jumps here from 0229H).
- 302DH Part of LIST command (jumps here from 2B91H).
- 3030H Basic TIME\$ function routine.
- 3033H \$DATE routine (date to 8-character buffer pointed to by HL).
- 3036H \$TIME routine (time to 8-character buffer pointed to by HL).
- 3039H Non-maskable interrupt handler.
- 3042H \$SETCAS routine (prompt user to set cassette baud rate).

The upper 2K of ROM also contains lookup tables for the keyboard and line printer drivers. This strikes me as being quite wasteful of memory that could be put to better use, particularly since the Model I did not rely on the use of such tables. Not only that, but there are separate tables for use with the keyboard driver, depending on whether or not the SHIFT key is pressed, and on whether or not the "CAPS LOCK" feature is enabled. The printer lookup table is particularly wasteful of memory, since this table occupies the area from 3145H to 31A4H, yet only one character is changed through the use of the table (a 60H SHIFT-@ character is changed to a 40H unshifted @ character before being output to the printer - and in the Model 4, even this one-character change has been eliminated, yet the table remains)!

One change that has NOT been made in current editions of the Model III ROM, but which I had said would surely appear in any future revisions, is to change the code at memory locations 0348H - 034CH. This code is a holdover from the Model I, and is used to test whether or not the video display is currently in the 32-character mode. Presently it is LD A,(403DH) followed by AND 08H, but unfortunately, in the Model III memory location 403DH no longer holds the 32-character flag. To perform the test properly, this code will have to be changed to LD A,(4210H) followed by AND 04H. This is another BUG in the Model III ROM, and is responsible for the improper operation of the TAB function while the video display is in the 32 character mode. I had thought that surely Tandy would fix this bug as soon as possible, but apparently I overestimated their dedication, because we're now up to the Model 4 and the bug still resides right there in the ROM, as if glued to the silicon!

The information presented above should help you determine what areas of the Model III ROM may cause problems with programs designed to run on a Model I.

In preparing this article, I found a book entitled "MOD III ROM COMMENTED" by Soft Sector Marketing quite helpful. As the title implies, it is nothing more or less than a disassembled listing of the Model III ROM, with almost every instruction commented. Although I noticed a few minor errors (incorrect digits crept into a few hex addresses), for the most part the book is accurate. If you are converting Model I programs to run on the Model III, you would find the going much easier with a copy of this book handy. In my opinion, the price was a bit steep (\$22.50), but if time is money to you, you'd probably find the time saved by having the commented code is worth the price of the book. Unfortunately, due to the untimely demise of Soft Sector Marketing, "MOD III ROM COMMENTED" is no longer in print. If, by chance, you come across someone who has a copy and who is willing to part with it, by all means buy it if you can!

APPENDIX II

ROM DIFFERENCES BETWEEN THE MODELS III AND 4

The Model 4 was designed to be fully compatible with software developed for the Model 3, and generally speaking, it meets that goal. The Model 4 uses stock Model 3 ROMs for the first 12K of memory - that is, there is NO difference in the code at memory locations 0000H through 2FFFH between the Models III and 4. However, the upper 2K of ROM (memory locations 3000H through 37FFH) has been somewhat modified. The most significant changes in that area of memory are as follows:

1. The I/O router routine is no longer present in the ROM.
2. The keyboard scan routine has been significantly altered, in order to make the additional keys on the Model 4 keyboard fully functional.
3. Some of the other routines within the Model 4 ROM have been moved around, or changed slightly.

Note that with the exception of the re-router routine (entry point at 3027H) and a redundant relative jump to the bootstrap routine (at 3040H), all of the routine vectors at 3000H through 3044H still work normally. This means that as long as entry is made to the routines in the upper 2K of memory only through the use of these vectors, a program that runs properly on the Model III should need no modification to run on the Model 4 (in the Model III mode, of course).

It should not be considered good practice to jump into the area of memory from 3045H - 37FFH, as this area is subject to change at any time! This was true of the Model III ROMs, as routines in the upper 2K of memory were moved about slightly with each new release of the ROM. Some routines seem to be fairly stable (the cassette routines, for example), but where a vector is available to enter a routine, use it! You will save yourself much grief at a later date.

If it should become necessary to have a user program determine which Model computer it is running on (III or 4), I suggest that the easiest and safest test would be to check location 3029H. This location will always contain a zero byte on a Model 4, but will contain 37H on the Model III. Since this is part of the vector to the I/O re-router routine, which has been eliminated in the Model 4, I doubt that the contents of this location will be changed in any future release of the Model 4 ROM. Suggested assembly language code for this test would be as follows:

```
LD      A,(3029H)
OR      A
(Z flag set if running on Model 4)
```

Under BASIC, a statement similar to the following would do the trick:

```
IF PEEK (12329) = 0 THEN ... (running on Model 4)
```

In the list of changes below, general differences between the Model III and Model 4 ROMs are shown. Note that since we are comparing one version of the Model III ROM with one version of the Model 4 ROM (either of which could be available in other versions), I am not attempting a "byte by byte" comparison. Rather, in cases where the code is mostly different but one or two consecutive bytes are by coincidence the same, I have simply included those bytes within the block of "changed" code. Even then, between any two versions of the Model III and Model 4 ROMs there may exist differences not noted here. The main purpose of this appendix is to give the reader a general idea of which routines have been changed and/or moved, but should not be considered an absolute guide to those changes.

ROM CHANGES IN THE MODEL 4:

3013H - 3013H Part of JP instruction that begins at 3012H (start of warm bootstrap routine changed to 3461H).

3016H - 3016H Part of JP instruction that begins at 3015H (start of cold bootstrap routine changed to 3401H).

3027H - 3029H JP to I/O re-router routine has been changed to RET instruction followed by two NOPs. I/O re-router no longer exists in Model 4 ROM.

302EH - 302EH Part of JP instruction that begins at 302DH (start of patch to LIST command routine changed to 37A4H).

3031H - 3031H Part of JP instruction that begins at 3030H (start of TIME\$ function routine changed to 37C2H).

303EH - 3041H Part of JP instruction that begins at 303DH (entry point to this portion of Non-Maskable Interrupt routine changed to 34CEH), followed by two unused bytes (formerly JR 3015H instruction, now changed to two XOR D instructions).

3043H - 3044H Part of JP instruction that begins at 3042H (start of \$SETCAS routine changed to 310BH).

3060H - 3064H Formerly part of keyboard scan routine, now replaced by five NOP instructions.

307DH - 3084H Formerly part of keyboard scan routine, now replaced by eight NOP instructions.

30A0H - 30A4H Formerly part of keyboard scan routine, now replaced by five NOP instructions.

30BDH - 30C4H Formerly part of keyboard scan routine, now replaced by eight NOP instructions.

30E0H - 30E4H Formerly part of keyboard scan routine (CALLED screen print routine at 01D9H, then did an XOR A followed by a RET instruction in order to return a null character), now replaced by five NOP instructions.

30FDH - 3144H Formerly part of the keyboard scan routine (included three subroutines and two lookup tables), now replaced by eight NOP instructions (30FDH - 3104H), a portion of the non-disk bootstrap routine that prompts the user to set the cassette baud rate (3105H - 310AH), the \$SETCAS routine which prompts the user to set the cassette baud rate (310BH - 313AH), and a portion of the keyboard scan routine which CALLs the screen print routine at 34FDH, then executes XOR A and RET instructions to return a null character (313BH - 313FH). The final five bytes (3140H through 3144H) are now NOP instructions.

3185H - 3185H Part of the totally unnecessary printer lookup table. This table formerly took up 96 bytes to convert one character before sending it to the printer (an ASCII 60H shift-@ character was converted to an ASCII 40H normal @ character). Guess what? It doesn't even do that anymore! The byte at 3185H has been changed from 40H (normal @) to 60H (shift-@). Now, all characters are "converted" to their original values before being output to the printer. That doesn't make sense to you? Good! It shouldn't! Apparently it made sense to someone at Microsoft, though, because it still appears in the Model 4 ROM. As Dave McGlumphy of the Chattanooga Microcomputer Club once said, "You don't have to be sane to be a programmer. In fact, it doesn't even help".

338EH - 3528H The largest block of changed code. This portion of memory formerly included most of the keyboard scan routine (338EH - 3454H), the bootstrap routine (3455H - 3517H), a short time-delay routine used by the bootstrap routine (3518H - 351BH), the Non-Maskable Interrupt handler routine (351CH - 3527H), and an unused FFH byte (3528H). This area now contains part of the keyboard scan routine (338EH - 3400H), the bootstrap routine (3401H - 34CDH), the Non-Maskable Interrupt handler routine (34CEH - 34D9H), more of the keyboard scan routine (37DAH - 34FCH), a new screen print routine used when the control and asterisk keys are pressed (34FDH - 351EH), and ten NOPs (zero bytes at 351FH through 3528H).

36D6H - 36D6H This byte is moved to reserved RAM location 402CH (part of the printer Device Control Block) during power up. This location formerly contained the letter "R" (probably leftover garbage from the Model I "PR" designator) but now contains a zero byte.

36E0H - 36E6H These seven bytes are moved to reserved RAM locations 4036H - 403CH during power up. These locations are used by the keyboard scan routine to store an "image" of the first seven rows of the keyboard matrix. They are all initialized to contain AAH bytes in the Model III (for some unknown reason), but in the Model 4 they are correctly initialized with zero bytes.

36FFH - 3700H Two bytes that are moved to RS-232 Input Device Control Block (locations 41EBH & 41ECH) during power up. These bytes formerly contained the ASCII characters "RI" (designator for "RS-232 input"), but now contain the ASCII character codes that will be returned when the "F1" or "F2" keys are pressed. 36FFH contains 60H (a shift-@ character), which will be loaded into 41EBH as the character returned when the "F1" key is pressed, while 3700H contains 1BH (a shift-up arrow character), which will be loaded into 41ECH as the character returned when the "F2" key is pressed.

3707H - 3708H Two bytes that are moved to RS-232 Output Device Control Block (locations 41F3H & 41F4H) during power up. These bytes formerly contained the ASCII characters "RO" (designator for "RS-232 Output"), but now contain the ASCII character code that will be returned when the "F3" key is pressed and the "image" for the eighth keyboard row (the row mapped to 3880H). 3707H contains 08H (a shift-@ character), which will be loaded into 41F3H as the character returned when the "F3" key is pressed, while 3708H contains a zero byte, which will be loaded into 41F4H to initialize the storage location used to store the "image" (current status) of the CAPS, CTRL, and function keys (F1, F2, & F3).

370FH - 3710H Two bytes that are moved to RS-232 Initialization Device Control Block (locations 41FBH & 41FCH) during power up. These bytes formerly contained the ASCII characters "RN" (designator for "RS-232 iNitalization"), but now contain two zero (00H) bytes.

3731H - 3733H Three bytes that are moved to 421DH - 421FH during power up. These bytes formerly contained the device type flag and the driver address, and were part of the I/O re-router Device Control Block. As previously mentioned, the I/O re-router routine has been eliminated from the Model 4 ROM.

3739H - 37E7H This area of memory formerly contained the I/O re-router routine (3739H - 377AH), a patch to the LIST command routine (377BH - 3798H), the TIME\$ function routine (3799H - 37AEH), a portion of the non-disk bootstrap routine that prompted the user to set the cassette baud rate (37AFH - 37B4H), the \$SETCAS routine (37B5H - 37DBH), another portion of the non-disk bootstrap routine, which put the address of the TIME\$ routine into the vector at 4177H (37DCH - 37E4H), and three unused AAH bytes (37E5H - 37E7H). In the Model 4, this area now contains a good portion of the keyboard scan routine (3739H - 37A3H), a patch to the LIST command routine (37A4H - 37C1H), the TIME\$ function routine (37C2H - 37D7H), more of the keyboard scan routine (37D8H - 37E0H), a short time-delay routine used by the disk bootstrap routine (37E1H - 37E4H), and three unused zero bytes (37E5H - 37E7H).

37EAH - 37EAH Computer version flag in Model III, unused (zero byte) in Model 4.

37F4H - 37FFH Part of the power up routine. In the Model III, 37F4H - 37F5H contains a JR 37DCH instruction, followed by the text for the "Cass? " prompt (37F6H - 37FDH) and two unused AAH bytes (37FEH - 37FFH). In the Model 4, the code that used to begin at 37DCH (a portion of the non-disk bootstrap routine, which puts the address of the TIME\$ routine into the vector at 4177H) has been moved down to 37F4H - 37FCH, so the JR instruction is no longer needed. 37FDH - 37FFH appear to contain unused code in the Model 4 (two PUSH HL instructions followed by RST 38H).

APPENDIX III
MODEL I ROM CHANGES

There are four known versions of the Model I ROM. Each new version was released to fix some bug that appeared in earlier versions, or to enhance the capabilities of the computer. In any event, all known changes between the earliest Model I ROMs and the last edition to appear (the so-called "new" ROMs) are shown below. This information may prove useful when writing a program intended to be run on any Model I TRS-80.

FIRST EDITION MODEL I ROM:

```
0059      ORG    0059H
0059 1A    DEFB   1AH
```

LAST EDITION MODEL I ROM:

```
0059      ORG    0059H
0059 00    DEFB   00H
```

This byte is located in a table used by the keyboard scan routine. It defines the ASCII character returned when the SHIFT-down arrow keys are depressed. In new ROM Model I's, this combination is used to produce control characters, thus the SHIFT-down arrow combination by itself does not return an ASCII character when depressed.

00FC ORG 00FCH	00FC ORG 00FCH
00FC 211101 LD HL,0111H	00FC 210E01 LD HL,010EH

Here HL is pointed to the start of the "RADIO SHACK LEVEL II BASIC" message. However, in new ROM units this message has been shortened to "R/S L2 BASIC" and begins in a different location.

0105 ORG 0105H	0105 ORG 0105H
0105 4D DEFM 'MEMORY SIZE'	0105 4D DEFM 'MEM SIZE'
45 4D 4F 52 59 20 53 49	45 4D 20 53 49 5A 45
5A 45	010D 00 DEFB 00H
0110 00 DEFB 00H	010E 52 DEFM
0111 52 DEFM 'RADIO SHACK LEVEL II BASIC'	'R/S L2 BASIC'
41 44 49 4F 20 53 48 41	2F 53 20 4C 32 20 42 41
43 4B 20 4C 45 56 45 4C	53 49 43
20 49 49 20 42 41 53 49	011A 0D DEFB 0DH
43	011B 00 DEFB 00H
012B 0D DEFB 0DH	011C C5 PUSH
012C 00 DEFB 00H	BC
	011D 010005 LD BC,0500H
	0120 CD6000 CALL 0060H
	0123 C1 POP
	BC
	0124 0A LD A,(BC)
	0125 A3 AND
	E
	0126 C8 RET
	Z
	0127 7A LD A,D
	0128 07 RLCA
	0129 07 RLCA
	012A C3FE03 JP 03FEH

The above area has been changed to make room for the keyboard debounce routine of the new ROMs. In the old ROMs, the messages "MEMORY SIZE?" and "RADIO SHACK LEVEL II BASIC" are in this area, while in the new ROMs this has been changed to "MEM SIZE?" and "R/S L2 BASIC", which is followed by the debounce routine.

0248	ORG	0248H	0248	ORG	0248H
0248 0641	LD	B,41H	0248 0660	LD	B,60H
024F	ORG	024FH	024F	ORG	024FH
024F 0676	LD	B,76H	024F 0685	LD	B,85H

The above two changes correct the timing of the "read one bit from cassette" routine, in order to make loading data from cassette easier and more reliable.

02E2	ORG	02E2H	02E2	ORG	02E2H
02E2 23	INC	HL	02E2 20ED	JR	NZ,02D1H
02E3 20EC	JR	NZ,02D1H	02E4 23	INC	HL

This is a change in the routine that reads a filename from a SYSTEM tape. It is interesting to note that the Model III ROM forsakes the "new ROM" code, and returns to the original coding of the oldest Model I ROMs.

03FB	ORG	03FBH	03FB	ORG	03FBH
03FB 7A	LD	A,D	03FB C31C01	JP	011CH
03FC 07	RLCA				
03FD 07	RLCA				

Part of the keyboard scan routine. In the new ROM a jump to the new keyboard debounce routine is inserted here.

0683	ORG	0683H	0683	ORG	0683H
0683 20EF	JR	NZ,0674H	0683 20F1	JR	NZ,0676H

This is part of the power-up sequence. Some initialization data is copied from ROM into reserved RAM 128 times, in order to make sure that the memory chips are stable enough to receive and store the data properly. Unfortunately, on the earliest ROMs this jump (which was executed 128 times) went back one instruction too far, to an OUT (0FFH),A instruction. This caused the cassette relay to buzz during power-up.

1226	ORG	1226H	1226	ORG	1226H
1226 EA3412	JP	PE,1234H	1226 300B	JR	NC,1233H
1229 014391	LD	BC,9143H	1228 014391	LD	BC,9143H
122C 11F94F	LD	DE,4FF9H	122B 11F94F	LD	DE,4FF9H
122F CD0C0A	CALL	0A0CH	122E CD0C0A	CALL	0A0CH
1232 1806	JR	123AH	1231 1806	JR	1239H
1234 116C13	LD	DE,136CH	1233 116C13	LD	DE,136CH

1237 CD490A	CALL	0A49H	1236 CD490A	CALL	0A49H
123A F24C12	JP	P,124CH	1239 F24B12	JP	P,124BH
123D F1	POP	AF	123C F1	POP	AF
123E CD0B0F	CALL	0F0BH	123D CD0B0F	CALL	0F0BH
1241 F5	PUSH	AF	1240 F5	PUSH	AF
1242 18E1	JR	1225H	1241 18E2	JR	1225H
1244 F1	POP	AF	1243 F1	POP	AF
1245 CD180F	CALL	0F18H	1244 CD180F	CALL	0F18H
1248 F5	PUSH	AF	1247 F5	PUSH	AF
1249 CD4F12	CALL	124FH	1248 CD4F12	CALL	124FH
124C F1	POP	AF	124B F1	POP	AF
124D D1	POP	DE	124C D1	POP	DE
			124D B7	OR	A

An invalid test for a double precision number was corrected by this fix. Most instructions are the same, but are just moved down one address in the new ROMs.

1265	ORG	1265H	1265	ORG	1265H
1265 F24412	JP	P,1244H	1265 F24312	JP	P,1243H

This corrects a jump back into the part of ROM that was moved down in the new ROMs.

206C	ORG	206CH	206C	ORG	206CH
206C C39B20	JP	209BH	206C C37C20	JP	207CH
206F CDCA41	CALL	41CAH	206F CDCA41	CALL	41CAH
2072 FE40	CP	40H	2072 FE23	CP	23H
2074 2019	JR	NZ,208FH	2074 2006	JR	NZ,207CH
2076 CD012B	CALL	2B01H	2076 CD8402	CALL	0284H
2079 FE04	CP	04H	2079 329C40	LD	(409CH),A
207B D24ALE	JP	NC,1E4AH	207C 2B	DEC	HL
207E E5	PUSH	HL	207D D7	RST	10H
207F 21003C	LD	HL,3C00H	207E CCFE20	CALL	Z,20FEH
2082 19	ADD	HL,DE	2081 CA6921	JP	Z,2169H
2083 222040	LD	(4020H),HL	2084 F620	OR	20H
2086 7B	LD	A,E	2086 FE60	CP	60H
2087 E63F	AND	3FH	2088 201B	JR	NZ,20A5H
2089 32A640	LD	(40A6H),A	208A CD012B	CALL	2B01H
208C E1	POP	HL	208D FE04	CP	04H
208D CF	RST	08H	208F D24ALE	JP	NC,1E4AH
208E 2C	INC	L	2092 E5	PUSH	HL
208F FE23	CP	23H	2093 21003C	LD	HL,3C00H
2091 2008	JR	NZ,209BH	2096 19	ADD	HL,DE
2093 CD8402	CALL	0284H	2097 222040	LD	(4020H),HL
2096 3E80	LD	A,80H	209A 7B	LD	A,E
2098 329C40	LD	(409CH),A	209B E63F	AND	3FH
209B 2B	DEC	HL	209D 32A640	LD	(40A6H),A
209C D7	RST	10H	20A0 E1	POP	HL
209D CCFE20	CALL	Z,20FEH	20A1 CF	RST	08H
20A0 CA6921	JP	Z,2169H	20A2 2C	INC	L
20A3 FEBF	CP	0BFH	20A3 18C7	JR	206CH
20A5 CABD2C	JP	Z,2CBDH	20A5 7E	LD	A,(HL)
20A8 FEBC	CP	0BCH	20A6 FEBF	CP	0BFH
20AA CA3721	JP	Z,2137H	20A8 CABD2C	JP	Z,2CBDH
20AD E5	PUSH	HL	20AB FEBC	CP	0BCH

20AE FE2C	CP	2CH	20AD CA3721	JP	Z,2137H
20B0 CA0821	JP	Z,2108H	20B0 E5	PUSH	HL
20B3 FE3B	CP	3BH	20B1 FE2C	CP	2CH
20B5 CA6421	JP	Z,2164H	20B3 2853	JR	Z,2108H
20B8 C1	POP	BC	20B5 FE3B	CP	3BH
20B9 CD3723	CALL	2337H	20B7 285E	JR	Z,2117H
20BC E5	PUSH	HL	20B9 CD3723	CALL	2337H
			20BC E3	EX	(SP),HL

The above changes affect the PRINT routine, and allow the @ character (of a PRINT @ statement) to be placed in the PRINT line at places other than immediately following the PRINT command. Also, multiple @ specifiers may be used within a single PRINT statement line.

20F6	ORG	20F6H	20F6	ORG	20F6H
20F6 C39B20	JP	209BH	20F6 C37C20	JP	207CH

This corrects a jump back into the revised portion of the PRINT routine.

213A	ORG	213AH	213A	ORG	213AH
213A E63F	AND	3FH	213A E67F	AND	7FH

This instruction defines the maximum valid argument for the TAB function (63 decimal in the old ROM, 127 decimal in the new ROM).

2166	ORG	2166H	2166	ORG	2166H
2166 C3A020	JP	20A0H	2166 C38120	JP	2081H

Another jump back into the changed portion of the PRINT routine that had to be fixed in the new ROM.

226A	ORG	226AH	226A	ORG	226AH
226A 3AA940	LD	A,(40A9H)	226A 00	NOP	
226D B7	OR	A	226B 00	NOP	
226E C8	RET	Z	226C 00	NOP	
			226D 00	NOP	
			226E 00	NOP	

This was the source of the (in)famous POKE 16553, 255 fix sometimes required on early TRS-80's. My guess is that the routine originally placed here may have been written to suppress printing of an "EXTRA IGNORED" message when input came from the cassette rather than the keyboard. Due to the undesirable side effects, the test of location 40A9H was eliminated in later versions of the ROM.

2C1F	ORG	2C1FH	2C1F	ORG	2C1FH
2C1F CD9302	CALL	0293H	2C1F D6B2	SUB	0B2H
2C22 7E	LD	A,(HL)	2C21 2802	JR	Z,2C25H
2C23 D6B2	SUB	0B2H	2C23 AF	XOR	A
2C25 2802	JR	Z,2C29H	2C24 01	DEFB	01H

2C27 AF	XOR	A	2C25 2F	CPL	
2C28 01	DEFB	01H	2C26 23	INC	HL
2C29 2F	CPL		2C27 F5	PUSH	AF
2C2A 23	INC	HL	2C28 7E	LD	A,(HL)
2C2B F5	PUSH	AF	2C29 B7	OR	A
2C2C 2B	DEC	HL	2C2A 2807	JR	Z,2C33H
2C2D D7	RST	10H	2C2C CD3723	CALL	2337H
2C2E 3E00	LD	A,00H	2C2F CD132A	CALL	2A13H
2C30 2807	JR	Z,2C39H	2C32 1A	LD	A,(DE)
2C32 CD3723	CALL	2337H	2C33 6F	LD	L,A
2C35 CD132A	CALL	2A13H	2C34 F1	POP	AF
2C38 1A	LD	A,(DE)	2C35 B7	OR	A
2C39 6F	LD	L,A	2C36 67	LD	H,A
2C3A F1	POP	AF	2C37 222141	LD	(4121H),HL
2C3B B7	OR	A	2C3A CC4D1B	CALL	Z,1B4DH
2C3C 67	LD	H,A	2C3D 210000	LD	HL,0000H
2C3D 222141	LD	(4121H),HL	2C40 CD9302	CALL	0293H
2C40 CC4D1B	CALL	Z,1B4D			

Changes to the CLOAD command under the new ROM permit the user to CLOAD from tape drive # 1 only, but allow a filename to be specified when CLOADing while operating under Disk BASIC.

2FFB	ORG	2FFBH	2FFB	ORG	2FFBH
2FFB 00	NOP		2FFB DEC3	SBC	A,0C3H
2FFC 00	NOP		2FFD C344B2	JP	0B244H
2FFD 00	NOP				
2FFE 00	NOP				
2FFF 00	NOP				

These changes appear to be nothing more than leftover garbage that was inadvertently programmed into the newer ROMs. These final five bytes are not accessed by any other ROM code.

APPENDIX IV

TRS-80 vs. PMC-80 / VIDEO GENIE / DICK SMITH SYSTEM 80 / TRZ-80

PMC-80, VIDEO GENIE, DICK SMITH SYSTEM 80, and TRZ-80 are all names used in various parts of the world to refer to the same microcomputer, a TRS-80 "workalike" made by ECCA International of Hong Kong. Although most programs written for the TRS-80 Model I can be used directly on the Hong Kong copy without modification, there are a few differences between the two machines. These differences will mostly affect machine language programs that contain their own printer driver, and BASIC or machine language programs that use the right arrow key or the double width character mode. The following is a summary of the major differences affecting user software between the two machines. Please keep in mind that the discussion which follows is applicable to the original edition of the ECCA computer, but may not be fully applicable to later models.

HARDWARE DIFFERENCES

Two keys that are standard on the TRS-80 Model I are omitted on the copy - the right arrow key and the CLEAR key. A hardware-oriented user can easily add these two keys to the copied unit, but they are not normally included. Because the right arrow is omitted, the method of switching to the 32 character per line mode has been changed to a switch on the back of the unit. Switching between the 32- and 64-character per line modes under software control is not possible, unless another minor hardware change is made. If a BASIC program attempts to switch to the 32-character mode (using CHR\$(23), etc.) and the modification is not in place, BASIC will nevertheless think that the screen is in the 32-character mode and will ignore every other screen memory location. For many programs using the double width character mode, this simply means that the screen will be formatted normally except that the characters will be normal width, and there will be a space between each character.

The Hong Kong unit has a built in cassette deck which unfortunately has no volume control. There is a provision to use an external cassette recorder, but it is mapped to port FEH instead of port FFH. Selecting cassette drive # -2 (from BASIC or machine language) will automatically use the external drive. Once again, a hardware modification can be made to allow the external recorder to function as cassette drive # -1.

The character generator IC in the Far East machine is uppercase only, and is not pin compatible with the equivalent IC used in the TRS-80. This simply means that most users of the foreign micro will probably not have a lowercase modification installed, although a dedicated hardware hacker could probably accomplish the conversion without much difficulty. The printer and cassette select circuitry is mapped to ports rather than memory locations as will be described below. There are other minor hardware differences as well (including the labeling of certain keys), but they will mostly not affect software compatibility.

ROM SOFTWARE DIFFERENCES

The ROM used in the Hong Kong machine appears to be an almost exact copy of the version 1.3 Model I TRS-80 ROM (the latest version of the so-called "old" ROM). The differences between the Model I version 1.3 ROM and the ROM found in the Hong Kong copy are described below. Even a program that does not make ROM calls may be affected by some of the hardware changes that made the ROM changes below necessary, so any programmer writing a program intended for use on the Far East copy should review the few ROM changes below to see if similar changes are required in their program.

0105H - 0110H First of two power-up messages changed. In the TRS-80 contains "MEMORY SIZE" followed by a zero byte, while the copy contains "READY" followed by a space and six zero bytes. The net effect of this is that the "READY?" prompt that appears on the screen when Hong Kong unit is first powered-up is actually a "MEMORY SIZE?" prompt, while all subsequent "READY" prompts really are the BASIC "READY" prompt that we are familiar with. Score one for confusion!

0111H - 012CH Second power-up message changed. In one machine (guess which one) this area contains "RADIO SHACK LEVEL II BASIC" followed by a carriage return and zero byte, while the copy contains twenty-seven carriage returns followed by the zero byte. The effect here is to scroll the first "READY" prompt (the one that should be "MEMORY SIZE?") clear off the screen before proceeding (and to eliminate Radio Shack's name from the ROM).

0212H - 0214H Part of cassette drive select routine. When this code is executed, the A register contains zero if cassette drive # -1 is to be selected, or 1 if drive # -2 is to be used. In the Model I contains a LD (37E4H),A instruction, while in the copy contains OUT (FEH),A followed by a zero byte (NOP).

05ADH - 05AFH Part of line printer output routine. When this code is executed, the A register contains a character to be sent to the printer. In the Model I contains a LD (37E8H),A instruction, while in the copy contains a zero byte (NOP) followed by an OUT (FDH),A instruction.

05BBH - 05BDH Also part of line printer output routine. Once again, when this code is executed the A register contains a character to be sent to the printer. In the Model I contains a LD (37E8H),A instruction, while in the copy contains a zero byte (NOP) followed by an OUT (FDH),A instruction.

05D1H - 05D3H Part of line printer status test. After this code has been executed, the A register will contain a bit pattern which indicates whether or not the printer is ready to receive a character. In the Model I contains a LD A,(37E8H) instruction, while in the copy contains a zero byte (NOP) followed by an IN A,(FDH) instruction.

18F5H - 18F6H Part of table of two-character strings used for error messages under non-Disk BASIC. Here the string L3 found in the TRS-80 has been changed to SN in the copy. Thus, the use of a Disk

BASIC reserved word in a non-disk system is reported as an L3 ERROR on the TRS-80, but simply as a SN ERROR (syntax error) in the imported unit.

MEMORY LOCATION / PORT DIFFERENCES

In summary, the following points should be kept in mind: When defining the cassette drive, memory location 37E4H is used on the TRS-80 Model I, while port FEH is used on the Hong Kong machine. When accessing the printer, memory location 37E8H is used by the TRS-80 Model I, and port FDH is used by the foreign copy (in contrast, the Model III uses port F8H to access the printer, and does not have cassette select circuitry). If you have commercial software designed for the TRS-80 and are trying to use it on the foreign machine, and the printer or cassette select functions do not work properly, try searching for the following instructions and making the changes shown:

SEARCH FOR:			CHANGE TO:			
-----			-----			
32 E4 37	LD	(37E4H),A	D3 FE 00	OUT NOP	(FEH),A	
32 E8 37	LD	(37E8H),A	00 D3 FD	NOP OUT	(FDH),A	
3A E8 37	LD	A,(37E8H)	00 DB FD	NOP IN	A,(FDH)	

APPENDIX V

RELOCATABLE PROGRAM SAMPLE

Back in issue six of The Alternate Source, an article appeared explaining how to make your machine language programs relocatable. What's a relocatable machine language program? It's one that loads into low memory, then moves itself up to the highest available unprotected memory and protects that memory. The advantages of that are that several relocatable programs can co-exist in memory without conflict and without wasting memory space, and that the user does not have to preset the MEMORY SIZE for the program. When I wrote that first article on the subject, I didn't have the process fully perfected, but even so that article has provided the basis for the relocation routines that appear in several programs. Since that article has appeared, I have figured out how to make the assembler do most of the work, so that the programmer has to do little more than append the relocation routine to the front of his assembly language programs.

What follows is an example of a program using the relocation routine. The program itself is nothing special - it's simply a Model III style line printer driver that can be used with the Model I or the Model III. What is important about the program is the method of relocation used. This program can be loaded under the BASIC SYSTEM command, or from the Model I or Model III DOS, and it will still relocate itself and protect itself in high memory.

The code is self-modifying, in that it is set up for use with cassette BASIC, but if it is loaded from disk it detects this fact and modifies some instructions within the relocation routine to account for the difference. Specifically, it must change where it looks for the top-of-memory pointer stored by the system, which is found at 40B1H under BASIC, 4049H under Model I DOS, and 4411H under the Model III DOS. The final exit from the relocation routine must also be changed.

A relocation routine is different from other programs in that it has only one task to do, and when that task is finished the routine is discarded by the system. Therefore, we can take some liberties while writing the routine that we might be hesitant to take otherwise. One such liberty taken in this routine is that of the code modifying itself - for some strange reason, that is considered a sort of taboo among programmers, presumably because someday they might want to encapsulate their program in a ROM (how many programs have YOU burned into a ROM this week?). Well, I never much cared for that old chestnut, and in this case there's a good argument against it - if the program were going to be placed in ROM it wouldn't need a relocation routine, right? So, the relocator modifies itself. Critics are welcome to rewrite the routine as they see fit.

In most cases you will want to modify the relocation routine to serve your own purposes. That's OK - this is just a jumping-off point to demonstrate how it can be done. But you need to know the basics in order to get going, so here they are:

First of all, it's a good idea to start out with a normal (that is, non-relocatable) version of the program and make sure that all the bugs are out of it before you attempt to make it relocatable. Once you know that it works properly, go through the program and find every absolute JP or CALL instruction in the program that references another location within the program itself. Add -OFFSET to each such reference as is done for the CALLs in the program below. You will also have to do this where you load a register pair with an address within the program, or where you use a DEFW pseudo-op to reference an address within the program. DON'T add the -OFFSET to labels referencing locations OUTSIDE of the program (such as calls or jumps to the ROM or DOS), and DON'T add the -OFFSET label to relative jump instructions (such as JR or DJNZ). You must then make sure that each of these instructions (the ones with -OFFSET added) are themselves labeled, so that they can be placed in the relocator table.

You must also make sure that the lines labeled OFFSET and END are in their proper locations. OFFSET goes just before the first instruction of the main program, and END goes at the very end of the program. And, if the program contains an initialization segment that is used only once, you'll probably want to move that segment to the exit of the relocator routine.

As shown in the example program, a table must be provided that contains pointers to all of the addresses that will need to be changed after relocation. This table must be placed prior to the start of the main program (you don't want to save the table with the program to be relocated, do you?). When building the relocation table, note that the table entries must point to the first byte to actually be changed. This means that either no offset, or an offset of zero or one may have to be added to each label. Here's an example of what the table entry might look like for various instructions:

Instruction:	Relocation table entry:
GOTHER JP ELSWHR-OFFSET	GOTHER+1
GOSUB CALL BELL-OFFSET	GOSUB+1
STRDAT LD DE,(STORIT-OFFSET)	STRDAT+2
WORD DEFW PRGLOC-OFFSET	WORD

Note that the DEFW instruction is not offset because the label address is also the first byte to be changed. Conversely, the LD DE instruction is four bytes long, so we must add two to the label address in order to get the first byte to be changed. JP and CALL instructions always have +1 added. The final entry in the relocation table must be a DEFW 0 instruction to mark the end of the table.

One important portion of the relocation routine remains to be dealt with, and that is the exit routine. At line 520 a jump is taken if the end of the relocation program has not been reached. Thus, what follows (lines 530 to 660 in this case) is the exit routine for the relocator. This is probably the only portion of the relocation routine that cannot simply be copied from program to program. Rather, it must be customized to the requirements of the program being relocated. On entry to this portion of the program, the DE register pair contains the address of the first byte of the relocated program,

which will usually be the entry point address. If that isn't the case, the entry point can be calculated as follows: First, label the entry point (we'll use the label ENTRY in this example). Then simply use the following code to calculate the relocated entry point address:

```
LD      HL,ENTRY-OFFSET ;# bytes from pgrm start
ADD    HL,DE           ;HL=relocated entry addr
```

After the above segment is executed HL will contain the relocated entry point address.

In any event, once the entry point address is known, it can be used as desired. In this case we are patching it into the printer Device Control Block. Depending on the intended use of the program, you may wish to patch it into a vector used by BASIC (however, note that the contents of most of these are destroyed when Disk BASIC is entered), or you may wish to simply convert the entry address to ASCII and display it on the video display, so that the user can access the program as he sees fit. You may also do any other required initialization of the program prior to actually exiting the routine, as has been done here by resetting other counters located in the printer DCB (doing one-shot initialization from within the relocator program saves memory space, since the initialization routine is then discarded after use along with the relocator). The code after that (LD BC,1A18H followed by JP 19AEH) is the best return to BASIC that will work on both the Model I and the Model III, but if the routine was entered from DOS then some previously executed code will have modified this JP instruction to JP 402DH, the normal return to DOS READY.

A couple of other quick points: The routine is ORGed at 7000H, which is usually high enough to clear Disk BASIC. Also, the main program can overlap itself while being moved to higher memory, since it is moved starting at the end of the program and working backwards (using an LDDR instruction).

The relocation routine can be simplified somewhat if the main program is not intended to have universal application (for example, the DOS entry tests could be removed in a utility designed to work with cassette BASIC only). It invites customization by the programmer. If you are thinking about marketing a program, you should seriously consider using a relocation routine if the program normally resides in high memory, as this will make your program more salable.

The sample assembly language program listing follows:

	00100	;	*****	INITIALIZATION & RELOCATION SEGMENT BEGINS	*****
	00110				
7000	00120	ORG	7000H		
7000 AF	00130	INTLZE	XOR	A	;Clear Carry flag
7001 3D	00140		DEC	A	;Set A to OFFH
7002 112D40	00150		LD	DE,402DH	;RET address if DOS
7005 E1	00160		POP	HL	;Get actual RET address
7006 E5	00170		PUSH	HL	;Re-save it as well
7007 ED52	00180		SBC	HL,DE	;Result zero if under DOS
7009 2019	00190		JR	NZ,NOTDOS	;If not under DOS
700B 214940	00200		LD	HL,4049H	;Mod I DOS top-of-memory
700E 3A5400	00210		LD	A,(54H)	;Get byte from ROM
7011 3D	00220		DEC	A	;Determine if Mod I or 3
7012 2803	00230		JR	Z,RSTMEM	;Go if Model I DOS
7014 211144	00240		LD	HL,4411H	;Mod 3 DOS top-of-memory
7017 222970	00250	RSTMEM	LD	(MEMSIZ+2),HL	;Self-modify program
701A 223270	00260		LD	(STRMEM+2),HL	;memory pointers & put
701D 224C70	00270		LD	(GETMEM+2),HL	;DOS addr in exit of
7020 ED536E70	00280		LD	(IEXIT+1),DE	;initialization routine
7024 210471	00290	NOTDOS	LD	HL,END	;End of unrelocated pgrm
7027 ED5BB140	00300	MEMSIZ	LD	DE,(40B1H)	;End of unprotected mem
702B 017500	00310		LD	BC,END-OFFSET+1	;Length of main program
702E EDB8	00320		LDDR		;Move the program
7030 ED53B140	00330	STRMEM	LD	(40B1H),DE	;Save new memory size
7034 3C	00340		INC	A	;A=0 if under BASIC
7035 2017	00350		JR	NZ,SKPCLR	;Skip CLEAR if under DOS
7037 310670	00360		LD	SP,INTLZE+6	;Don't crash moved code
703A 21BB41	00370		LD	HL,41BBH	;HL=1st byte DOS vector
703D 7E	00380		LD	A,(HL)	;Save DOS vector in
703E 08	00390		EX	AF,AF'	; alternate A register
703F 3EC9	00400		LD	A,OC9H	;Plug DOS vector with
7041 77	00410		LD	(HL),A	; a RET instruction
7042 113200	00420		LD	DE,32H	; "CLEAR 50"
7045 CD831E	00430		CALL	1E83H	;& reset other pointers
7048 08	00440		EX	AF,AF'	;Get original DOS vector
7049 77	00450		LD	(HL),A	;Restore original vector
704A ED5BB140	00460	GETMEM	LD	DE,(40B1H)	;Get new pgrm location
704E 13	00470	SKPCLR	INC	DE	;DE=Start relocated pgrm
704F 218470	00480		LD	HL, TABLE	;Get start of reloc table
7052 7E	00490	NXTLOC	LD	A,(HL)	;Get first address byte
7053 47	00500		LD	B,A	;Save first byte in B
7054 23	00510		INC	HL	;Point @ 2nd address byte
7055 B6	00520		OR	(HL)	;Do both bytes equal 0?
7056 2018	00530		JR	NZ,REPLCE	;No - replace address
7058 212640	00540		LD	HL,4026H	;Printer Device Cntrl Blk
705B 73	00550		LD	(HL),E	;Put in LSB of pgm START
705C 23	00560		INC	HL	;Point to next DCB loctn
705D 72	00570		LD	(HL),D	;Put in MSB of pgm START
705E 23	00580		INC	HL	;Point to next DCB loctn
705F 3643	00590		LD	(HL),43H	;Set lines/page
7061 23	00600		INC	HL	;Point to next DCB loctn
7062 3601	00610		LD	(HL),1	;Set current line counter
7064 23	00620		INC	HL	;Point to next DCB loctn
7065 3600	00630		LD	(HL),0	;Set current char counter
7067 23	00640		INC	HL	;Point to next DCB loctn
7068 36FF	00650		LD	(HL),OFFH	;Set char/line (no max)

706A 01181A	00660	LD	BC,1A18H	;Best return to BASIC on
706D C3AE19	00670	IEXIT	JP 19AEH	; both Models I & III
7070 E5	00680	REPLCE	PUSH HL	;Save reloc table pointer
7071 66	00690	LD	H,(HL)	;H=MSB of addr from table
7072 68	00700	LD	L,B	;L=LSB of addr from table
7073 19	00710	ADD	HL,DE	;Get address of label
7074 4E	00720	LD	C,(HL)	;Get byte displacement
7075 23	00730	INC	HL	; from program and
7076 46	00740	LD	B,(HL)	; put in BC
7077 2B	00750	DEC	HL	;HL=address of label
7078 D5	00760	PUSH	DE	;Save new START address
7079 EB	00770	EX	DE,HL	;HL=new START address
707A 09	00780	ADD	HL,BC	;HL=new address for label
707B EB	00790	EX	DE,HL	; now put in DE
707C 73	00800	LD	(HL),E	;New address calculated -
707D 23	00810	INC	HL	; now write it into
707E 72	00820	LD	(HL),D	; label (address field)
707F D1	00830	POP	DE	;Restore new START addr
7080 E1	00840	POP	HL	;Restore reloc table pntr
7081 23	00850	INC	HL	;Bump pointer to nxt addr
7082 18CE	00860	JR	NXTLOC	;Process next table entry
	00870			
	00880	***** RELOCATION TABLE *****		
	00890	;Must be located prior to "START" label of main program!		
	00900			
7084 0D00	00910	TABLE	DEFW FFLOOP+1-OFFSET	
7086 2900	00920	DEFW	LINEFD+1-OFFSET	
7088 5000	00930	DEFW	REL01+1-OFFSET	
708A 5500	00940	DEFW	REL02+1-OFFSET	
708C 5800	00950	DEFW	SKPCNT+1-OFFSET	
708E 0000	00960	DEFW	0	
	00970			
	00980			
	00990	***** MAIN PROGRAM (WILL BE RELOCATED) BEGINS *****		
	01000			
7090	01010	OFFSET	EQU \$;Used by relocator
	01020			
7090 79	01030	LPDVR	LD A,C	;Get character to print
7091 FE0C	01040	CP	OCH	;Is it a form feed?
7093 2013	01050	JR	NZ,NOTFF	;Go if not form feed
7095 DD7E03	01060	LD	A,(IX+3)	;Get # lines per page
7098 DD9604	01070	SUB	(IX+4)	;Subtract lines already
709B 47	01080	LD	B,A	; printed & put in B
709C CD6400	01090	FFLOOP	CALL PTRTST-OFFSET	;Wait for printer ready
709F 3E0A	01100	LD	A,0AH	;Linefeed character in A
70A1 32E837	01110	LD	(37E8H),A	;Output it to printer
70A4 10F6	01120	DJNZ	FFLOOP	;Repeat to top next form
70A6 1820	01130	JR	RSTCT	;Reset line & char counts
70A8 FE0A	01140	NOTFF	CP OAH	;Is character a linefeed?
70AA 280C	01150	JR	Z,LINEFD	;Go if linefeed
70AC FE0D	01160	CP	ODH	;Is it a carriage return?
70AE 2024	01170	JR	NZ,NTSPCL	;Go if not special char
70B0 DD7E05	01180	LD	A,(IX+5)	;If any characters have
70B3 B7	01190	OR	A	; been printed on line
70B4 2002	01200	JR	NZ,LINEFD	; then print the <CR>,
70B6 0EOA	01210	LD	C,0AH	; else change to linefd

70B8 CD6400	01220	LINEFD	CALL	PTRTST-OFFSET	;Wait for printer ready
70BB 79	01230		LD	A,C	;Get character to print
70BC 32E837	01240	LNFD2	LD	(37E8H),A	;Output it to printer
70BF DD3404	01250		INC	(IX+4)	;Increment line count
70C2 DD7E04	01260		LD	A,(IX+4)	;Get line count in A
70C5 DDBE03	01270		CP	(IX+3)	;Max line count reached?
70C8 DD360500	01280	RSTCT	LD	(IX+5),0	;Reset char count to 0
70CC 2023	01290		JR	NZ,ENDRT	;If not yet max line cnt
70CE DD360401	01300		LD	(IX+4),1	;Reset line count
70D2 181D	01310		JR	ENDRT	;Go to routine end
70D4 DD7E06	01320	NTSPCL	LD	A,(IX+6)	;Get max char per line
70D7 3C	01330		INC	A	;If max char byte = 255
70D8 280D	01340		JR	Z,SKPCNT	; skip char count test
70DA DDBE05	01350		CP	(IX+5)	;If # chars already prntd
70DD 3008	01360		JR	NC,SKPCNT	; <maximum don't do <CR>
70DF CD6400	01370	RELO1	CALL	PTRTST-OFFSET	;Wait for printer ready
70E2 3E0D	01380		LD	A,ODH	;Carriage return in A
70E4 CD2C00	01390	RELO2	CALL	LNFD2-OFFSET	;Outpt char, reset counts
70E7 CD6400	01400	SKPCNT	CALL	PTRTST-OFFSET	;Wait for printer ready
70EA 79	01410		LD	A,C	;Get character to print
70EB 32E837	01420		LD	(37E8H),A	;Output it to printer
70EE DD3405	01430		INC	(IX+5)	;Increment char count
70F1 AF	01440	ENDRT	XOR	A	;Set Z flag
70F2 79	01450		LD	A,C	;Character restored to A
70F3 C9	01460		RET		;Return to calling program
70F4 3AE837	01470	PTRTST	LD	A,(37E8H)	;Get printer status addr
70F7 E6F0	01480		AND	OF0H	;Mask off lowest 4 bits
70F9 FE30	01490		CP	30H	;Check for printer ready
70FB C8	01500		RET	Z	;Return if printer ready
70FC 3A4038	01510		LD	A,(3840H)	;Check to see if "BREAK"
70FF E604	01520		AND	04H	; key depressed
7101 28F1	01530		JR	Z,PTRTST	;If "BREAK" not pressed
7103 F1	01540		POP	AF	;Get rid of RET address
7104 C9	01550		RET		;To calling program
	01560				
7104	01570	END	EQU	\$-1	;Used by relocator
	01580				
7000	01590		END	INTLZE	

00000 TOTAL ERRORS

END	7104	ENDRT	70F1	FFLOOP	709C	GETMEM	704A	IEXIT	706D
INTLZE	7000	LINEFD	70B8	LNFD2	70BC	LPDVR	7090	MEMSIZ	7027
NOTDOS	7024	NOTFF	70A8	NTSPCL	70D4	NXTLOC	7052	OFFSET	7090
PTRTST	70F4	RELO1	70DF	RELO2	70E4	REPLCE	7070	RSTCT	70C8
RSTMEM	7017	SKPCLR	704E	SKPCNT	70E7	STRMEM	7030	TABLE	7084

APPENDIX VI

IMPROVED AMPERSAND FUNCTION

(The following article originally appeared in The Alternate Source; Volume III, Number 1; Issue 13. It has been revised to include the decimal-to-hexadecimal conversion routine. It is reprinted here as an example of accessing a machine language subroutine through an unused DOS exit, and also to illustrate the use of ROM routines within a user program.)

In Disk BASIC there is a function called by the "&" character, which lets you use a hexadecimal or octal constant as part of a BASIC statement. As written, it seems to me to be a fairly useless function, except that it is often more convenient for the programmer to use a hexadecimal address in a PEEK or POKE statement, and this function allows it. But, if you don't have the function available, you need only convert the number to decimal first. The "&" function works with constants only -- you can't use it with variables to convert them to decimal.

Can we improve upon this situation? Of course we can -- and you can even use the program with Level II BASIC! The program can also be patched into Disk BASIC, and will remain completely compatible with the existing "&" function. For example:

X=&H7FFF	(convert hex to decimal, X will equal 32767)
X=&O1777	(convert octal to decimal, X will equal 1023)
X=&l777	(octal conversion assumed if not specified)

However, the following additional functions are now made available:

X=&B11011100	(convert binary to decimal, X will equal 220)
X=&H(A\$)	(convert hex string contained in string variable A\$ to decimal, and store in X. Routine stops at first non-hex character)
X=&O(A\$) or X=&(A\$)	(same as above except A\$ assumed to be octal string)
X=&B(A\$)	(same as above except A\$ assumed to be binary)
X=&PEEK(16548)	(two byte peek -- returns integer value of the two memory locations at and immediately following the specified location. This particular statement would be the equivalent of this line of BASIC: X=PEEK(16548)+PEEK(16549)*256)
X\$=&FN(32767)	(decimal to hex conversion --- returns a four-byte long STRING containing the hexadecimal equivalent of the specified integer expression. X\$ will equal "7FFF" in this example)

Note that the ability to convert a string containing a hex, octal or binary number is probably the most useful part of this routine. You may think of this as a VAL function for strings in another base. As an example of the power of this routine, let's consider a short program that asks the user for an address in hexadecimal, then prints the contents of the two-byte pointer at that address:

```
10 INPUT "ADDRESS IN HEX"; A$  
20 PRINT &PEEK(&H(A$))
```

That's the whole program! Note that the number returned by the "&" function must be an integer in the range -32768 to 32767 (the same as the Level II PEEK and POKE functions). Should you require an absolute decimal number, use a statement similar to this:

```
X=&H8000 : X=X-(X<0)*65536      (X will equal 32768)
```

Also note that "&" functions can be nested if necessary. For example, suppose you wanted the above two-byte pointer returned in hexadecimal format. You could change line 20 as follows:

```
20 PRINT &FN(&PEEK(&H(A$)))
```

During base conversions (except for integer decimal to hexadecimal), any valid string expression may be used within the parenthesis. Any of the following might be used:

```
&H("7FFF")      &H(Z$(0))      &B("1111"+X$).
```

Now let's track down an apparent bug. Try entering these two statements from the keyboard:

```
?&H("1DEF")  
?&H1DEF
```

You SHOULD get the same result - but the second statement bombs! (Disk BASIC users, don't laugh until you try the second statement on your system). The reason for this is that when DEF is seen by the BASIC interpreter as part of a command line, it is changed to the token for the DEF function, so the characters DEF are not there for the "&" function to decode. However, if the characters are part of a literal string or string variable, they are not encoded by BASIC, so the function works. The easiest way to avoid this problem in a BASIC program is, if you are using a hex constant with the characters DEF, place a space between the D and E or between the E and F. This breaks up the reserved word, and BASIC will not encode it. Try entering:

```
?&H1D EF
```

It works. Remember, this problem only affects hex constants -- it does not affect string variables enclosed within parentheses.

When using the integer to hexadecimal function, keep in mind that if the result is to be assigned to a variable, it must be a STRING variable, and also that the expression within the parenthesis must evaluate to an integer expression. This is in contrast to the other

"&" functions, all of which return an integer as a result, and some of which accept a string argument.

This routine makes several calls to ROM. One of the not-so-well documented ones is at 252CH. This is the parenthesized expression evaluator (sounds like a futuristic machine, right?). On entry to this routine, HL must point to the left parenthesis. The expression will be evaluated up to the corresponding right parenthesis, and the result (or pointer to the string vector, if a string expression) will be stored in the math accumulator, with the number type flag set appropriately. The math accumulator begins at 4121H, except for double precision numbers -- in that instance, it begins at 411DH. The number type flag is located at 40AFH. If you wish to evaluate an expression that is not in parentheses you may call 2337H; on entry to this routine, HL must point to the first character of the string to be evaluated. This time, the expression will be evaluated up to a zero byte, colon, etc. You may be able to make use of these routines in your assembly language programs, also.

In order to use this routine you must protect the memory it resides in by setting the MEMORY SIZE to 32595. If you have assembled the program to tape, simply load it in using the SYSTEM command. If you have assembled the program to disk (hopefully you changed the ORG address in line 170 to something more appropriate for your system), go into BASIC (don't forget to protect memory!) and from BASIC READY do a CMD"filename" where "filename" is the name of the assembled object code (under TRSDOS use CMD"I","filename"). This will (should) get the program into high memory and patched into BASIC. The program listing follows:

	00100	;	LINK TO & FUNCTION VECTOR		
	00110				
4194	00120	ORG	4194H	;VECTOR FROM "&" FUNCTION	
4194 C3537F	00130	JP	START	;JUMP TO START OF ROUTINE	
	00140				
	00150	;	MAIN PROGRAM BEGINS HERE		
	00160				
7F53	00170	ORG	7F53H	;MAIN PRGM-MAY RELOCATE	
7F53 D7	00180	START	RST	10H	;GET NEXT CHARACTER
7F54 CB7F	00190		BIT	7,A	;CHECK FOR TOKEN
7F56 2844	00200		JR	Z,NOTTKN	;GO IF NOT BASIC TOKEN
7F58 F5	00210		PUSH	AF	;SAVE TOKEN
7F59 D7	00220		RST	10H	;GET NEXT CHARACTER
7F5A CD2C25	00230		CALL	252CH	;EVALUATE EXPRESSION
7F5D E3	00240		EX	(SP),HL	;SAVE BASIC POINTER
7F5E E5	00250		PUSH	HL	;RE-SAVE TOKEN
7F5F CD7F0A	00260		CALL	0A7FH	;CHANGE TO INTEGER
7F62 F1	00270		POP	AF	;GET TOKEN
7F63 FEE5	00280		CP	0E5H	;2-BYTE PEEK?
7F65 2009	00290		JR	NZ,NOTPK	;IF NOT 2-BYTE PEEK
7F67 5E	00300		LD	E,(HL)	;GET CONTENTS OF ADDRESS
7F68 23	00310		INC	HL	;POINTED TO BY HL AND
7F69 56	00320		LD	D,(HL)	;PUT IN DE REGISTERS
7F6A ED532141	00330		LD	(4121H),DE	;AND MATH BUFFER

7F6E E1	00340	POP	HL	; RESTORE BASIC POINTER
7F6F C9	00350	RET		; NUMBER IN BUFFER & DE
7F70 FEBE	00360	NOTPK	CP OBEH	; CHECK FOR HEX CONVERSION
7F72 C29719	00370		JP NZ,1997H	; SN ERROR IF NOT HEX CONV
7F75 EB	00380		EX DE, HL	; PUT VALUE IN DE
7F76 CD807F	00390	CALL	CONV	; CONVERT TO STRING
7F79 CD9310	00400	CALL	1093H	; MARK END OF STRING
7F7C C5	00410	PUSH	BC	; FOR ROM TO DISCARD
7F7D C33928	00420	JP	2839H	; RETURN THRU STR\$ ROUTINE
7F80 213041	00430	CONV	LD HL, 4130H	; ASCII CONV. WORKSPACE
7F83 7A	00440	LD	A,D	; CONVERT D REGISTER
7F84 CD887F	00450	CALL	CONV2	; TO ASCII (HEX)
7F87 7B	00460	LD	A,E	; CONVERT E REGISTER
7F88 F5	00470	CONV2	PUSH AF	; SAVE BYTE TO CONVERT
7F89 0F	00480	RRCA		; ROTATE HIGH NYBBLE
7F8A 0F	00490	RRCA		; DOWN TO LOWER
7F8B 0F	00500	RRCA		; FOUR BITS
7F8C 0F	00510	RRCA		
7F8D CD917F	00520	CALL	CONV3	; CONVERT TO ASCII (HEX)
7F90 F1	00530	POP	AF	; RESTORE BYTE TO CONV
7F91 E60F	00540	CONV3	AND 0FH	; USE LOWER FOUR BITS ONLY
7F93 C690	00550	ADD	A, 90H	; THIS ROUTINE CONVERTS
7F95 27	00560	DAA		; HEX VALUE IN RANGE
7F96 CE40	00570	ADC	A, 40H	; 00H-0FH TO ASCII CHAR
7F98 27	00580	DAA		; 0-9 OR A-F
7F99 77	00590	LD	(HL), A	; STORE RESULT ASCII CHAR
7F9A 23	00600	INC	HL	; BUMP WORKSPACE POINTER
7F9B C9	00610	RET		; (THIS ROUTINE RECURSIVE)
7F9C 010201	00620	NOTTKN	LD BC, 102H	; SET UP BINARY PARAMETERS
7F9F FE42	00630	CP	'B'	; IS IT BINARY?
7FA1 280F	00640	JR	Z, CONT	; GO IF BINARY
7FA3 011004	00650	LD	BC, 410H	; SET UP HEX PARAMETERS
7FA6 FE48	00660	CP	'H'	; IS IT HEXADECIMAL?
7FA8 2808	00670	JR	Z, CONT	; GO IF HEXADECIMAL
7FAA 010803	00680	LD	BC, 308H	; SET UP OCTAL PARAMETERS
7FAD FE4F	00690	CP	'O'	; IS IT OCTAL?
7FAF 2801	00700	JR	Z, CONT	; GO IF OCTAL
7FB1 2B	00710	DEC	HL	; OCTAL ASSUMED
7FB2 D7	00720	CONT	RST 10H	; GET NEXT CHARACTER
7FB3 FE28	00730	CP	'('	; IS CHARACTER A "("?
7FB5 2018	00740	JR	NZ, NOTEXP	; GO IF NOT A "("
7FB7 C5	00750	PUSH	BC	; SAVE COUNTER & MAX DIGIT
7FB8 CD2C25	00760	CALL	252CH	; EVALUATE EXPRESSION
7FBB E3	00770	EX	(SP), HL	; SAVE BASIC BYTE POINTER
7FBC E5	00780	PUSH	HL	; RE-SAVE CNTR & MAX DIGIT
7FBD CD072A	00790	CALL	2A07H	; CLEAN UP WORKSPACE
7FC0 C1	00800	POP	BC	; RESTORE CNTR & MAX DIGIT
7FC1 5F	00810	LD	E,A	; GET STRING LENGTH IN E
7FC2 23	00820	INC	HL	; GET ADDRESS OF FIRST
7FC3 7E	00830	LD	A,(HL)	; CHARACTER OF STRING
7FC4 23	00840	INC	HL	; AND STORE IN HL
7FC5 66	00850	LD	H,(HL)	; REGISTERS
7FC6 6F	00860	LD	L,A	
7FC7 EB	00870	EX	DE, HL	; HL=STRING LNGTH, DE=STRT
7FC8 19	00880	ADD	HL, DE	; HL=STRING END + 1
7FC9 EB	00890	EX	DE, HL	; HL=START, DE=END + 1

7FCA CDD27F	00900	CALL	CONT2	; EVAL. STRING EXPRESSION
7FCD E1	00910	POP	HL	; RESTORE BASIC POINTER
7FC E9	00920	RET		; NUMBER IN MATH BUFFER
7FC F1FFFF	00930	NOTEEXP	LD DE,0FFFFH	; CHECK TO NONHEX CHAR.
7FD2 2B	00940	CONT2	DEC HL	; BACK UP BASIC POINTER
7FD3 E5	00950	PUSH	HL	; SAVE BASIC POINTER
7FD4 210000	00960	LD	HL,0	; ZERO MATH ACCUMULATOR
7FD7 CD9A0A	00970	CALL	0A9AH	; & SET TYPE FLG TO INT.
7FDA E3	00980	EX	(SP),HL	; HL=BASIC POINTER, (SP)=0
7FDB DDE1	00990	NXTDGT	POP IX	; STORE # SO FAR IN IX
7FDD D7	01000	RST	10H	; GET NEXT CHAR. (DIGIT)
7FDE 08	01010	EX	AF,AF'	; SAVE CHARACTER & FLAGS
7FDF DF	01020	RST	18H	; CHECK FOR END OF STRING
7FE0 C8	01030	RET	Z	; RETURN IF END OF STRING
7FE1 08	01040	EX	AF,AF'	; RESTORE CHAR. & FLAGS
7FE2 3805	01050	JR	C,DIGIT	; IF CHAR. IN RANGE 0 TO 9
7FE4 FE41	01060	CP	41H	; IS CHAR BELOW ASCII "A"?
7FE6 D8	01070	RET	C	; END OF NUMBER IF < "A"
7FE7 D607	01080	SUB	7	; OFFSET FOR ALPHA CHARS.
7FE9 D630	01090	DIGIT	SUB 30H	; A=0 TO 15 FOR 0-9 OR A-F
7FEB B9	01100	CP	C	; C=2, 8, OR 10H MAX DIGIT
7FEC D0	01110	RET	NC	; END OF NUMBER IF >= MAX.
7FED DDE5	01120	PUSH	IX	; PUT # SO FAR ON STACK
7FEF E3	01130	EX	(SP),HL	; HL=# SO FAR, (SP)=PNTR
7FF0 C5	01140	PUSH	BC	; SAVE COUNTER & MAX DIGIT
7FF1 29	01150	TIMES2	ADD HL,HL	; MULTIPLY HL TIMES 2
7FF2 DAB207	01160	JP	C,7B2H	; ERROR IF OVERFLOW >FFFFH
7FF5 10FA	01170	DJNZ	TIMES2	; REPEAT MULTIPLY 'TIL B=0
7FF7 4F	01180	LD	C,A	; ADD VALUE OF LATEST
7FF8 09	01190	ADD	HL,BC	; DIGIT FETCHED TO HL
7FF9 222141	01200	LD	(4121H),HL	; CURRENT HL TO BUFFER
7FFC C1	01210	POP	BC	; RESTORE CNTR & MAX DIGIT
7FFD E3	01220	EX	(SP),HL	; HL=BASIC PNTR, (SP)=#
7FFE 18DB	01230	JR	NXTDGT	; GET NEXT DIGIT (IF ANY)
	01240			
06CC	01250	END	06CCH	; USE 1A19H FOR MODEL III
	01260			; OR 402DH FOR DOS

00000 TOTAL ERRORS

CONT	7FB2	CONT2	7FD2	CONV	7F80	CONV2	7F88
CONV3	7F91	DIGIT	7FE9	NOTEEXP	7FCF	NOTPK	7F70
NOTTKN	7F9C	NXTDGT	7FDB	START	7F53	TIMES2	7FF1

APPENDIX VII

LINE INPUT ROUTINES FOR LEVEL II BASIC
(or, making BASIC behave the way you'd like it to)

(At the time of writing this book, this article had been accepted for publication by The Alternate Source Programmer's Journal, which unfortunately ceased publication before the article was printed. It is printed here to demonstrate interfacing a machine language routine with BASIC, and the use of ROM routines in a user program).

The ideal situation is that all computer programs be user-oriented (translation: dunce-proof). Unfortunately, in order to achieve that ideal you might spend quite a bit of time trying to figure out every possible way that Sam Scatterbrain can make the program bomb (through lack of knowledge about how the program works, deliberate attempt, or otherwise). The biggest need for bomb-proofed programs is in business applications - you can always reload and restart a game program without any loss (other than in time and frustration), but if you can bomb a business program you may manage to foul up a data base that is going to cost someone quite a bit of time and/or money to fix or replace. The problem is usually compounded by the fact that the operators of business programs are often secretaries who barely understand the difference between a computer and an adding machine, and who tend to prove Murphy's Law whenever confronted with the unexpected -- that is to say, whenever the computer does something strange, the secretary will attempt to correct the condition by doing the one thing that will cause a major disaster ("None of the buttons on the keyboard seem to work, maybe if I try this button they have hidden in the back corner...").

The problem in making user-oriented programs is that BASIC won't always cooperate. One of the best examples of this is the INPUT command. Use of the INPUT command is almost an open invitation to disaster, especially if your application requires that a nice, formatted video display be maintained. Assuming you have had the foresight to disable the BREAK key, there are still several easy ways to accidentally garble or destroy the current screen contents:

1. Hit the CLEAR key.
2. Hit the DOWN ARROW (linefeed) key.
3. Hit the SHIFT and RIGHT ARROW key together (32 character mode).
4. Enter a comma or colon as part of input ("EXTRA IGNORED" message).
5. Type an overlength line.

I'm sure there are other ways also, but these five are sufficient to cause terminal frustration.

Try to correct the situation using the INKEY\$ function, and listen to the touch typists complain - "It doesn't keep up with my typing!" Is there a better way? If you have Disk BASIC, you can use the LINE INPUT command. If you don't have disk, you can buy Level III BASIC (and use up a good portion of your available memory). Or, you can use one of the two programs shown below.

Why two programs? The difference is that the first uses much less memory, while the second is more versatile. Both implement the LINE INPUT command in Level II BASIC. You should also be able to use these with a disk-based system, but they would disable the LINE INPUT command already in Disk BASIC. Don't go away, disk users, these programs make use of some ROM routines that you might find useful during your next attempts at assembly language programming.

Using the first (shorter) program, the LINE INPUT statement can be used exactly like the regular INPUT statement, with three differences:

1. The input variable must be a string variable or a Type Mismatch (TM) error will occur.
2. The ? (question mark) prompt is not automatically supplied. If a prompt string is used, user input begins immediately at the end of the prompt string. If the question mark and/or a trailing space are desired, they can be inserted at the end of the prompt string.
3. Any commas, colons, or quotation marks which are typed in by the user are accepted as part of the input string. The ability to input commas is probably the handiest feature of the LINE INPUT command, at least insofar as the end user is concerned.

Acceptable formats for the line input statement include:

```
LINEINPUT A$  
LINEINPUT "Enter city and State: "; A$
```

Using the second (longer) program, all of the features of the first program are present. In addition, the following additional features may be optionally implemented by placing an exclamation mark (!) just before the variable name:

1. All control keys are ignored except ENTER, LEFT ARROW (backspace), and SHIFT LEFT ARROW (backspace to start of input). This makes it impossible to accidentally clear the screen or to drop down a line on the video display because you inadvertently hit the down-arrow (linefeed) key.

2. When ENTER is pressed at the end of input, it is NOT echoed to the screen. Thus, it is possible to have several inputs on the same line on the screen if desired. When a linefeed is needed, simply execute a PRINT statement immediately following the LINE INPUT statement.

Another feature of the second program is that it is permissible to insert an argument which defines the maximum input length. If the user attempts to type beyond that maximum, the additional input is ignored. The argument may specify a maximum length of zero to 240 characters (if zero is specified, only the ENTER or BREAK keys will be recognized, while 240 characters is the normal maximum for the INPUT statement and is the default if an argument is not specified). If the argument is used, it may be an expression (such as $2*A+Z$). This feature can be used to prevent the user from typing in an overlength line that would otherwise overflow a data field or spoil the video display.

Acceptable formats for the line input statement using this second program include:

```
LINEINPUT A$  
LINEINPUT! A$  
LINEINPUT "Enter City and State: "; A$  
LINEINPUT "Enter City and State: "; !A$  
LINEINPUT #20,A$  
LINEINPUT #20,!A$  
LINEINPUT #20,"Enter City and State: "; A$  
LINEINPUT #20,"Enter City and State: "; !A$  
A=2:B=5:LINEINPUT #A*B*2,!A$
```

Note that the last five statements would all limit the user input to 20 characters. The argument must be preceded by a # (number sign) and followed by a comma, and must be placed before the prompt string if one is used.

If you want to give your secretary (or whomever) some indication of how many characters s/he may type in response to an input request, try this:

```
10 PRINT@518,STRING$(25,95);:PRINT@512,,:LINEINPUT#25,"Name: ";A$
```

Assuming that you have the second LINE INPUT routine in place, try entering this BASIC program segment into your computer and RUN it - it may get you to thinking about ways to make your prompts more effective. Be especially careful of the punctuation - note that a semicolon must follow the STRING\$ statement and that both a comma and a semicolon (as well as a colon to separate the statements) follows the second PRINT@. Also note that a space is the last character of the prompt string (this makes the prompt and the user response more readable).

HOW THE PROGRAM WORKS

Let's take a look at the first (short) program. It begins by initializing the vector from the LINE command in BASIC at 41A3H to jump to the start of the line input program whenever BASIC sees the LINE command. At the beginning of the program, we must first check to make sure that the token for "INPUT" follows the LINE command. The method by which we do this is to execute an RST 8. Note that the byte immediately following the RST 8 is defined as 89 hexadecimal. This is the token for the reserved word INPUT. To understand how this works, you should first know that the HL register pair normally points to the next byte to be executed in a BASIC program. In other words, when BASIC calls the LINE routine HL is already pointing to the INPUT statement, assuming that the syntax is correct. This is true even if there were spaces between the LINE and INPUT commands.

RST 8 is a "shortcut" for calling a frequently used subroutine. In this case, the actual location of the subroutine is at 1C96H, so RST 8 is the same as CALL 1C96H in the TRS-80, except that we save two bytes by calling RST 8. When we call the subroutine, the return address (the next byte of our program, which is DEFB 89H) is pushed

onto the stack. The subroutine gets this location from the stack, and compares the byte stored there with the byte stored at the location pointed to by HL (the next statement in our BASIC program). The return address is then incremented once (to skip the 89H byte and point to the next "real" Z-80 instruction) and is then placed back on the stack. If the compared bytes were the same (meaning that we have a valid LINE INPUT statement) a jump is taken to 1D78H, which is the location of the routine called by RST 10H (we'll get back to this one). If they are not the same (something other than INPUT followed the LINE statement), a jump is made to 1997H - the location of the syntax error routine (SN ERROR).

For those who would like to follow the operation of this routine, here is an assembly language listing:

LC96 7E	LD	A,(HL)	;HL=BASIC program pointer
LC97 E3	EX	(SP),HL	;Return address in HL
LC98 BE	CP	(HL)	;Compare bytes
LC99 23	INC	HL	;Increment return address
LC9A E3	EX	(SP),HL	;Put return address back
LC9B CA781D	JP	Z,1D78H	;If compared bytes same
LC9E C39719	JP	1997H	;Syntax error if not same

The routine at 1D78H is normally called by executing an RST 10 instruction. However, since we are already in a subroutine, it is easier just to jump to the next subroutine. The subroutine at 1D78H increments HL (our BASIC program pointer) so that it points to the next character of our BASIC program (skipping any spaces or linefeeds along the way). At the end of this subroutine, HL will point to the next character following the INPUT token that isn't a space or linefeed. If the character is numeric (it shouldn't be in this case) the carry flag will be set. If the character is a colon or zero byte (end of BASIC statement) the zero flag will be set (this shouldn't happen now, either). In any event, the character pointed to by HL will be stored in the A register upon return.

In other words, if our statement was:

LINE INPUT Q\$

upon return from the routine at 1D78H, the HL register pair will point to the character "Q" (of "Q\$"), the A register will contain 51H (ASCII code for "Q"), and neither the carry or zero flags will be set.

This is the code for the subroutine at 1D78H:

1D78 23	INC	HL	;Bump BASIC program pntr
1D79 7E	LD	A,(HL)	;Get char at pointer loc
1D7A FE3A	CP	3AH	;Return (no flags set) if
1D7C D0	RET	NC	; char greater than "9"
1D7D FE20	CP	20H	;If char is space, bump
1D7F CA781D	JP	Z,1D78H	; ptr again (start over)
1D82 FE0B	CP	0BH	;Skip next test if char
1D84 3005	JR	NC,1D8BH	; greater than linefed
1D86 FE09	CP	09H	;If chr is linefeed, bump
1D88 D2781D	JP	NC,1D78H	; ptr again (start over)

1D8B FE30	CP	30H	;Set C flag if below "0"
1D8D 3F	CCF		;Complement C (set if 0-9)
1D8E 3C	INC	A	;If character zero byte
1D8F 3D	DEC	A	; this sets Z flag
1D90 C9	RET		;Return to sender

Anyone examining this bit of code might conclude that Microsoft had some space to kill, since two bytes could have been saved by substituting relative jump (JR) instructions in place of the absolute jump (JP) instructions. However, this is probably the most frequently called subroutine in the entire ROM, so it stands to reason that this routine would be optimized for speed of execution, not for conservation of space.

Back to the program at hand. Our line input program next calls 2828H to check for an Illegal Direct (ID) error. The reason for this is that the direct mode of BASIC uses the input buffer for storage of the BASIC statements being executed, so it cannot simultaneously be used to hold the user response from an INPUT statement.

The program next makes a call to 21CDH, in the middle of a routine used by BASIC's INPUT routine, to display the prompt string (if one is used). Because the routine is not entered at the beginning, we must first load the A register with the contents of the location pointed to by HL (our BASIC pointer) prior to making the call.

The rest of the calls into ROM are fairly straightforward (in this program, anyway). Here's a short description of what happens at each of them:

0361H actually gets our input from the keyboard and puts it into the input buffer. The location of the buffer is specified by a pointer located at 40A7H-40A8H. A zero byte is placed in the buffer at the end of the input, and HL points to the starting address of the buffer minus one on exit from the routine.

1DBEH is where the program jumps if BREAK was typed to end the input. This routine sets up the necessary pointers for the CONTINUE command before going back to BASIC "READY".

260DH is the routine that finds (or creates) a variable in memory. On entry, HL must point to the first character of the variable name. After return from this routine, HL will point to the next character following the variable, DE will contain the address of the variable in memory, and the number type flag at 40AFH will be set according to the type of variable (in this case it should contain a value of 3 to indicate a string).

0AF4H checks the Number Type Flag to see if it does in fact contain a value of 3. If it does not (the variable name did not indicate a string variable), a Type Mismatch error will be generated.

2868H creates a string VARPTR for the contents of the input buffer, so that BASIC can properly process the input as a string. The

two byte VARPTR will be stored at 4121H-4122H on return from this routine.

1F32H is part of the LET command routine. This routine assigns the string to the variable name and moves it up to the string storage area in high memory.

The second (longer) program uses some additional routines of interest. It starts out the same as the first program, but then loads the E register with a default maximum line length in the event that the user hasn't specified one. It then checks for the "#" character, indicating a user specified line length. Disk BASIC uses the "#" character to indicate that a line is to be input from a disk data file, so if you are attempting to patch this routine into Disk BASIC you will want to choose some other character to indicate the user specified line length in order to maintain compatibility with existing Disk BASIC programs.

If the "#" character is not found, the next portion of the program is skipped. Otherwise, the argument following the "#" character must be evaluated. Here's how that evaluation is accomplished: First a RST 10H instruction is executed, which (as explained later) gets the next valid character. Then the routine at 1E46H is called. This routine evaluates the expression beginning at the address pointed to by HL and returns the result in the DE register pair. An error message will be generated if the expression does not evaluate to a number between zero and 32767 decimal. BASIC uses this routine to evaluate the argument for the CLEAR command.

While we're on the subject, it may be worth mentioning that the routine at 1E46H is part of a group of related routines that work almost the same way. The routine at 2B02H operates in exactly the same manner except that the expression may be in the range -32768 to +32767 (used by PEEK and POKE). And the routine at 1E5AH won't accept an expression, just a string of digits (such as a line number) which must be in the range from zero to 65529. All of these routines return their result in DE (or zero if no expression or string of digits is present).

Once the expression is evaluated, it must be checked to make sure it's within limits. If it's not, we ship off to 1E4AH, the function call (FC) error routine. Otherwise, an RST 8 instruction is used to make sure that the comma follows the argument as it should (the comma is needed to separate the argument from the input variable name, so that BASIC doesn't get confused).

At this point, the E register contains the maximum line length. This will be needed later, so for now it will be saved on the stack. The next step is to check for an Illegal Direct (ID) error, and to display the prompt string (if any). Then a check is made look to see if the special "!" character has been included just prior to the variable name. The Z flag gives the result of that test, but for now the only special action taken is to bump HL up to point to the first character of the variable name if it isn't there already. Now the maximum line length is fetched from the stack and placed into both the B and C registers (B keeps a running count while C continues to hold

the maximum line length). Finally, it's time to save the BASIC pointer on the stack. Remember that the Z flag is still set if the "!" character was found - otherwise it is reset.

Now comes an interesting section. In the first program, we made a CALL to 0361H to get the user input, but that doesn't give us the control that we require in this program. So, in order to make things work the way we want them to, a portion of the ROM must be re-written. In order to save memory, the program will be jumping into the ROM later, but at a point where BASIC expects certain things to be on the stack. So we take this opportunity to load the stack the way that BASIC will expect it to be set up. First we place the return address to our program onto the stack, followed by the BC register pair (it's not needed, but BASIC expects something to be there). Next onto the stack is the return address for the part of ROM that the routine would have been called from if our program hadn't come in through the window, so to speak. Finally, HL is loaded with the address of the beginning of the input buffer. If the Z flag was not set (no "!" character found), the program can jump into the ROM now, otherwise the HL register pair must also be PUSHed onto the stack. Now the stack is set up properly.

Assuming that the "!" was present, we must continue to bypass the ROM for a while. So, our program loads the A register with 0EH and CALLs 33H. 33H prints the character stored in the A register on the video display, unless that character is a control character, in which case it performs the proper control code functions. Since 0EH is the control code for "turn on the cursor", that is just what the routine does.

After loading the B register with the maximum line length (B keeps a running count, while C continues to hold the maximum), a CALL is made to 49H. This is the famous "get a keystroke" routine - it waits until a key is depressed (pick a key, any key...) and then returns with whatever character was input stored in the A register. Various tests must now be run on the character to determine what should be done with it, if anything. Note that if a control character was typed, the program drops to a routine that deals with control characters. If the control character is one that is allowed to be input by the user, a jump is taken to the portion of ROM that handles that character, always making sure the proper return address is stored on the stack. In the case of the ENTER key, them jump is taken into the middle of the routine, since we don't want to output the carriage return to the video display (the character that turns off the cursor must still be output, however). Since the ROM expects AF to be pushed onto the stack at this point, our program must make sure that AF is indeed on the stack if it is going to take the plunge into the ENTER key routine. Using the ROM would be much simpler if we didn't have to take care that the stack is always adjusted correctly (but then, so would programming in general, come to think of it)!

Despite the problems, you can see that making full use of the ROMs results in programs that can do more while using less memory, and that (usually) much less effort is required of the programmer than if the stored code is not used. One thing that you will need in order to make the fullest use of the ROM routines is a disassembled listing of

the ROMs. If you have a disassembler (or a disassembling monitor program such as TASMON) and a printer, you can make your own disassembled listing (be sure that you're not low on paper when you start the printout, however)! There are a few books currently available that contain disassembled listings of the ROM, and in my opinion the best of these (at the time of this writing) is still MOD III ROM COMMENTED by Soft Sector Marketing. Although it covers the Model III only, the book contains a commented disassembly of almost the entire Model III ROM, with relevant comments for almost every instruction. Since most of the Model I ROM contains the same code as the Model III ROM, Model I users would also find this book very useful. It's presently out of print, but you may still be able to find a copy in your neighborhood computer store.

In closing, I would like to acknowledge that some of the information necessary to make the above routines work properly was obtained from a column authored by Mr. Larry Rosen, which appeared in the June, 1980 issue of the Chicatrug (Chicago TRS-80 Users Group) News. Thanks are in order to Mr. Rosen and to others like him who have helped to unravel the mysteries of the TRS-80 ROM routines.

The program listing for the SHORT line input command routine follows:

	00100 ;	LINK TO LINE COMMAND VECTOR		
	00110			
41A3	00120	ORG	41A3H	;VECTOR FROM "LINE"
41A3 C3DA7F	00130	JP	LINE	
	00140			
	00150 ;	MAIN PROGRAM BEGINS HERE		
	00160 ;	IF ORG ADDRESS NOT CHANGED		
	00170 ;	USE MEMORY SIZE = 32729		
	00180			
7FDA	00190	ORG	7FDAH	;MAY BE RELOCATED
7FDA CF	00200	LINE	RST	8 ;"INPUT" MUST FOLLOW
7FDB 89	00210		DEFB	89H ;"LINE" - ELSE SN ERROR
7FDC CD2828	00220		CALL	2828H ;CHECK FOR ID ERROR
7FDF 7E	00230		LD	A,(HL) ;GET NEXT CHARACTER
7FE0 CDCD21	00240		CALL	21CDH ;DISPLAY PROMPT IF ANY
7FE3 E5	00250		PUSH	HL ;SAVE BASIC POINTER
7FE4 CD6103	00260		CALL	0361H ;GET KEYBOARD INPUT
7FE7 C1	00270		POP	BC ;BC=BASIC POINTER
7FE8 DABE1D	00280		JP	C,1DBEH ;IF "BREAK" WAS PRESSED
7FEB C5	00290		PUSH	BC ;SAVE BASIC POINTER
7FEC E3	00300		EX	(SP),HL ;SAVE INPUT PTR-GET BASIC
7FED CD0D26	00310		CALL	260DH ;FIND OR CREATE VARIABLE
7FF0 CDF40A	00320		CALL	0AF4H ;TM ERROR IF NOT STRING
7FF3 EB	00330		EX	DE,HL ;SWITCH BASIC & ADR PTRS
7FF4 E3	00340		EX	(SP),HL ;SWITCH INP, VAR ADR PTRS
7FF5 D5	00350		PUSH	DE ;SAVE BASIC POINTER
7FF6 0600	00360		LD	B,0 ;TERMINATING BYTE=0
7FF8 CD6828	00370		CALL	2868H ;CREATE STRING
7FFB E1	00380		POP	HL ;RESTORE BASIC POINTER
7FFC AF	00390		XOR	A ;ZERO ACCUMULATOR

7FFD C3321F 00400	JP	1F32H	;JUMP TO ROM
06CC 00410	END	06CCH	;MODEL III USE 1A19H

00000 TOTAL ERRORS

LINE 7FDA

The program listing for the LONG line input command routine follows:

	00100 ;	LINK TO LINE COMMAND VECTOR	
	00110		
41A3	00120	ORG 41A3H	;VECTOR FROM "LINE"
41A3 C3767F	00130	JP LINE	
	00140		
	00150 ;	MAIN PROGRAM BEGINS HERE	
	00160 ;	IF ORG ADDRESS NOT CHANGED	
	00170 ;	USE MEMORY SIZE = 32629	
	00180		
7F76	00190	ORG 7F76H	;MAY BE RELOCATED
7F76 CF	00200	RST 8	;"INPUT" MUST FOLLOW
7F77 89	00210	DEFB 89H	; "LINE" - ELSE SN ERROR
7F78 1EF0	00220	LD E,0FOH	;DEFAULT MAX. LINE LENGTH
7F7A FE23	00230	CP '#'	;CHECK FOR ARGUMENT
7F7C 2012	00240	JR NZ,SVLGTH	;IF NO ARGUMENT
7F7E D7	00250	RST 10H	;POINT HL TO NEXT CHAR
7F7F CD461E	00260	CALL 1E46H	;GET ARGUMENT IN DE
7F82 7A	00270	LD A,D	;CHECK ARGUMENT FOR
7F83 B7	00280	OR A	; VALUE OVER OFOH
7F84 2005	00290	JR NZ,FCERR	; (FC ERROR IF OVER)
7F86 7B	00300	LD A,E	
7F87 FEF1	00310	CP 0FLH	
7F89 3803	00320	JR C,ENDCHK	;IF LENGTH IS OK
7F8B C34A1E	00330	FCERR JP 1E4AH	;FC ERROR ROUTINE
7F8E CF	00340	ENDCHK RST 8	;CHECK FOR END COMMA
7F8F 2C	00350	DEFB ',',	; (SN ERROR IF NOT)
7F90 D5	00360	SVLGTH PUSH DE	;SAVE MAXIMUM LINE LENGTH
7F91 CD2828	00370	CALL 2828H	;CHECK FOR ID ERROR
7F94 7E	00380	LD A,(HL)	;GET NEXT CHARACTER
7F95 CDCD21	00390	CALL 21CDH	;DISPLAY PROMPT IF ANY
7F98 7E	00400	LD A,(HL)	;GET NEXT CHARACTER
7F99 FE21	00410	CP '!'!	;IS IT "!"?
7F9B 2003	00420	JR NZ,NOEXP	;DON'T BUMP PNTR IF NOT !
7F9D F5	00430	PUSH AF	;SAVE "Z" FLAG
7F9E D7	00440	RST 10H	;POINT HL TO NEXT CHAR
7F9F F1	00450	POP AF	;RESTORE "Z" FLAG
7FA0 C1	00460	NOEXP POP BC	;RESTORE MAX LINE LENGTH
7FA1 41	00470	LD B,C	;B=C (MAXIMUM LINE LNGTH)
7FA2 E5	00480	PUSH HL	;SAVE BASIC POINTER
7FA3 21E77F	00490	LD HL,ENDINP	;SAVE RETURN ADDRESS
7FA6 E5	00500	PUSH HL	
7FA7 C5	00510	PUSH BC	

7FA8 217403 00520		LD	HL, 0374H	; SAVE RETURN ADDR
7FAB E5 00530		PUSH	HL	
7FAC 2AA740 00540		LD	HL, (40A7H)	; GET START OF BUFFER
7FAF C2D905 00550		JP	NZ, 05D9H	; IF THERE WAS NO "!" CHAR
7FB2 E5 00560		PUSH	HL	; SAVE START OF BUFFER
7FB3 3E0E 00570		LD	A, 0EH	; TURN ON CURSOR
7FB5 CD3300 00580		CALL	0033H	
7FB8 CD4900 00590	KEYSCN	CALL	0049H	; GET KEYBOARD CHARACTER
7FBB FE20 00600		CP	20H	; CHECK FOR CONTROL CHAR
7FBD 380D 00610		JR	C, CNTRL	; IF CONTROL CHARACTER
7FBF 77 00620		LD	(HL), A	; SAVE CHARACTER AT (HL)
7FC0 78 00630		LD	A, B	; MAXIMUM LENGTH REACHED?
7FC1 B7 00640		OR	A	
7FC2 28F4 00650		JR	Z, KEYSCN	; IF MAXIMUM LENGTH
7FC4 7E 00660		LD	A, (HL)	; GET CHARACTER
7FC5 23 00670		INC	HL	; ADVANCE BUFFER POINTER
7FC6 CD3300 00680		CALL	0033H	; DISPLAY CHARACTER
7FC9 05 00690		DEC	B	; DECREMENT CHAR COUNT
7FCA 18EC 00700		JR	KEYSCN	; GET NEXT CHARACTER
7FCC FE0D 00710	CNTRL	CP	ODH	; WAS <ENTER> PRESSED?
7FCE F5 00720		PUSH	AF	; MAY BE NEEDED
7FCF CA6906 00730		JP	Z, 0669H	; IF <ENTER>
7FD2 F1 00740		POP	AF	; NOT NEEDED-RESTORE
7FD3 FE01 00750		CP	1	; WAS <BREAK> PRESSED?
7FD5 CA6106 00760		JP	Z, 0661H	; IF <BREAK>
7FD8 11B87F 00770		LD	DE, KEYSCN	; SAVE RET ADDRESS
7FDB D5 00780		PUSH	DE	; ON STACK
7FDC FE08 00790		CP	8	; WAS BACKSPACE PRESSED?
7FDE CA3006 00800		JP	Z, 0630H	; IF BACKSPACE
7FE1 FE18 00810		CP	18H	; SHIFT-BACKSPACE PRESSED?
7FE3 CA2B06 00820		JP	Z, 062BH	; IF SHIFT-BACKSPACE
7FE6 C9 00830		RET		; NEXT KEYS CAN
7FE7 C1 00840	ENDINP	POP	BC	; BC=BASIC POINTER
7FE8 DABE1D 00850		JP	C, 1DBEH	; IF "BREAK" WAS PRESSED
7FEB C5 00860		PUSH	BC	; SAVE BASIC POINTER
7FEC E3 00870		EX	(SP), HL	; SAVE INPUT PTR-GET BASIC
7FED CD0D26 00880		CALL	260DH	; FIND OR CREATE VARIABLE
7FF0 CDF40A 00890		CALL	0AF4H	; TM ERROR IF NOT STRING
7FF3 EB 00900		EX	DE, HL	; SWITCH BASIC & ADR PTRS
7FF4 E3 00910		EX	(SP), HL	; SWITCH INP, VAR ADR PTRS
7FF5 D5 00920		PUSH	DE	; SAVE BASIC POINTER
7FF6 0600 00930		LD	B, 0	; TERMINATING BYTE=0
7FF8 CD6828 00940		CALL	2868H	; CREATE STRING
7FFB E1 00950		POP	HL	; RESTORE BASIC POINTER
7FFC AF 00960		XOR	A	; ZERO ACCUMULATOR
7FFD C3321F 00970		JP	1F32H	; JUMP TO ROM
06CC 00980		END	6CCH	; MODEL III USE 1A19H

00000 TOTAL ERRORS

CNTRL	7FCC	ENDCHK	7F8E	ENDINP	7FE7	FCERR	7F8B
KEYSCN	7FB8	LINE	7F76	NOEXP	7FA0	SVLGTH	7F90

APPENDIX VIII
BASIC TOKENS AND ENTRY POINTS

This table of BASIC entry points is provided for the benefit of those that wish to disassemble the ROM routines for the various BASIC statements and functions. It also shows the single-byte token used to represent each of the BASIC reserved words. Use of a single byte to represent each BASIC statement, function, or operator conserves memory and allows the BASIC interpreter to distinguish between these and other parts of the program (such as ASCII text, variable names, etc.). The entry address is the one used by BASIC and may not necessarily be the best entry point for use with other machine language routines. When no entry address is shown, it usually means that the token is referenced only to modify the action of another routine (for example, the token USING means something only within the PRINT routine).

'	FBH	
*	CFH	
+	CDH	
-	CEH	
/	D0H	
<	D6H	
=	D5H	
>	D4H	
ABS	D9H	0977H
AND	D2H	
ASC	F6H	2A0FH
ATN	E4H	15BDH
AUTO	B7H	2008H
CDBL	F1H	0ADBH
CHR\$	F7H	2A1FH
CINT	EFH	0A7FH
CLEAR	B8H	1E7AH
CLOAD	B9H	2C1FH
CLOSE	A6H	4185H
CLS	84H	01C9H
CMD	85H	4173H
CONT	B3H	1DE4H
COS	E1H	1541H
CSAVE	BAH	2BF5H
CSNG	F0H	0AB1H
CVD	E8H	415EH
CVI	E6H	4152H
CVS	E7H	4158H
DATA	88H	1F05H
DEF	B0H	415BH
DEFDBL	9BH	1E09H
DEFINT	99H	1E03H
DEFSNG	9AH	1E06H
DEFSTR	98H	1E00H
DELETE	B6H	2BC6H
DIM	8AH	2608H
EDIT	9DH	2E60H

ELSE	95H	1F07H
END	80H	1DAEH
EOF	E9H	4161H
ERL	C2H	
ERR	C3H	
ERROR	9EH	1FF4H
EXP	E0H	1439H
FIELD	A3H	417CH
FIX	F2H	0B26H
FN	BEH	
FOR	81H	1CALH
FRE	DAH	27D4H
GET	A4H	417FH
GOSUB	91H	1EB1H
GOTO	8DH	1EC2H
IF	8FH	2039H
INKEY\$	C9H	
INP	DBH	2AEFH
INPUT	89H	219AH
INSTR	C5H	
INT	D8H	0B37H
KILL	AAH	4191H
LEFT\$	F8H	2A61H
LEN	F3H	2A03H
LET	8CH	1F21H
LINE	9CH	41A3H
LIST	B4H	2B2EH
LLIST	B5H	2B29H
LOAD	A7H	4188H
LOC	EAH	4164H
LOF	EBH	4167H
LOG	DFH	0809H
LPRINT	AFH	2067H
LSET	ABH	4197H
MEM	C8H	
MERGE	A8H	418BH
MID\$	FAH	2A9AH
MKD\$	EEH	4170H
MKI\$	ECH	416AH
MKS\$	EDH	416DH
NAME	A9H	418EH
NEW	BBH	1B49H
NEXT	87H	22B6H
NOT	CBH	
ON	A1H	1F6CH
OPEN	A2H	4179H
OR	D3H	
OUT	A0H	2AFBH
PEEK	E5H	2CAAH
POINT	C6H	
POKE	B1H	2CBLH
POS	DCH	27F5H
PRINT	B2H	206FH
PUT	A5H	4182H
RANDOM	86H	01D3H
READ	8BH	21EFH

REM	93H	1F07H
RESET	82H	0138H
RESTORE	90H	1D91H
RESUME	9FH	1FAFH
RETURN	92H	1EDEH
RIGHT\$	F9H	2A91H
RND	DEH	14C9H
RSET	ACH	419AH
RUN	8EH	1EA3H
SAVE	ADH	41A0H
SET	83H	0135H
SGN	D7H	098AH
SIN	E2H	1547H
SQR	DDH	13E7H
STEP	CCH	
STOP	94H	1DA9H
STR\$	F4H	2836H
STRING\$	C4H	
SYSTEM	AEH	02B2H
TAB(BCH	
TAN	E3H	15A8H
THEN	CAH	
TIME\$	C7H	4176H
TO	BDH	
TROFF	97H	1DF8H
TRON	96H	1DF7H
USING	BFH	
USR	C1H	
VAL	F5H	2AC5H
VARPTR	C0H	
↑	D1H	

HEXADECIMAL ADDRESS CROSS-REFERENCE

The purpose of this cross-reference is to aid the reader in finding all relevant information about a given ROM routine or memory location. The format is as follows: A hexadecimal address is given, followed by the page numbers of all pages on which the address appears. Generally speaking, this cross-reference covers only addresses that fall within the ROM and reserved RAM areas of memory. References to RST instructions are also included under the appropriate address (for example, references to the RST 10H routine appear under "0010"). The fact that an address appears in this cross-reference indicates only that the address appears somewhere on the pages mentioned, and does not indicate that the ROM routine or memory location is actually discussed in any detail on those pages.

0000	50,51,54,63,64, 65,86,94	00AE	80	0213	81	02B2	74,121
0002	79,84	00B2	80	0214	16,80,81,96	02C3	83
0003	79	00B4	80	0215	17	02D1	82,91
0004	79	00C4	80	021B	15	02E2	82,91
0008	3,51,92,111,114, 116,117	00C6	80	021D	81	02E4	82
		00E8	80	021E	17,81	0314	17
		00EA	80	0221	80	032A	10,11,73
000B	53	00FC	90	0227	81	032C	73
000D	53,79	00FF	80	0228	81	033A	15
000E	30,79	0101	80	0229	84	0348	12,15,76,85
000F	79	0105	90,96	022B	81	034C	85
0010	3,44,46,47,51,55, 92,94,106,107, 108,112,114,117	0106	80	022C	17,81	0358	12,13,14,73
		010A	80	022D	81	035B	13
		010D	80	022E	18,75,81	0361	13,14,35,47,55, 113,115,116
0013	10	010E	90	0231	81		
0018	3,32,52,108	010F	80	0234	81	0368	73
001B	10,79	0110	96	0235	17,81	0374	118
0020	3,31,52,74	0111	90,96	023C	81	0384	13
0028	3,12,52	0112	80	023D	81	038B	11
002B	12,13	0115	80	0240	81	0394	16
0030	52	0118	80	0241	17,81	039C	16
0033	14,15,115,118	011B	80	0242	81	03C2	16,79,82
0038	52,64,89	011C	91	0243	17,81	03E2	82
003B	15,16	011D	80	0245	81	03E3	13,82
0040	13	0121	80	0247	81	03E9	82
0046	79	0125	80	0248	91	03EB	82
0047	79	012C	80,96	024C	81	03FB	91
0048	79	012D	61,72	024D	81	03FE	90
0049	13,115,118	0135	121	024F	91	0440	16
0050	19,79	0138	121	0252	81	044B	16,82
0055	19,79	0150	15	0253	81	0451	82
0059	90	019D	44	025E	81	0452	79,82
005A	19,79	019E	44	0260	81	0453	52
005F	79	01C9	15,80,119	0264	17,81	0455	52
0060	53,79,81,90	01D3	29,120	0266	81	0456	52
0062	79	01D9	16,80,81,87	0268	81	0457	52,82
0063	79	01DA	80	0274	81	0458	15,82
0066	54,79	01DC	81	0277	81	0468	82
0069	79	01EF	80	0282	81	046B	82,83
006C	19,79	01F0	80	0283	81	0472	82
0070	79	01F1	80	0284	17,81,92	0473	15,82
0071	79	01F3	81	0287	17,81	0494	82
0072	54,55,83	01F4	80,81	028D	12,13,81	0496	82
0075	54	01F7	80	0292	81	049E	82
0080	74,79	01F8	16,18,81,84	0293	17,81,93,94	04A0	82
0082	79	01FB	79,81	0296	17,81	04B7	82
0089	74	01FE	16,17	0298	15,81	04B9	82
008A	74	0201	81	029F	81	0500	90
008D	74	0202	81	02A0	81	050C	82
00A8	80	020F	81	02A1	15,81	050E	82
00AA	80	0211	81	02A7	81	0532	82
00AC	75,80	0212	16,17,81,96	02A8	81	0534	82

058C	82	09FC	33	14F0	29	1EA3	121
058D	16,82	09FF	33	1541	29,119	1EA6	73
05AD	96	0A0C	32,91	1547	29,121	1EB1	120
05AF	96	0A39	32	15A8	29,121	1EC2	120
05BB	96	0A49	32,92	15BD	29,119	1ED9	61
05BD	96	0A4F	32	18F5	96	1EDE	121
05CF	82	0A78	32	18F6	96	1EEA	61
05D0	82	0A7F	31,59,106,119	18F7	30,65	1F05	119
05D1	16,82,96	0A8A	31	1904	30,65	1F07	58,120,121
05D3	82,96	0A8E	31	1917	83	1F21	58,120
05D4	16	0A9A	33,59,108	1918	83	1F24	58
05D8	82	0A9D	31	191B	83	1F26	58
05D9	13,118	0AB1	31,119	191C	83	1F27	58
062B	118	0AB9	31	1929	84	1F32	114,117,118
0630	118	0ACC	31	197A	61	1F6C	120
0661	118	0ACF	31	198A	61	1FAF	121
0669	118	0ADB	31,119	1997	61,107,112	1FF4	120
0674	51,79,82,91	0AE3	31	199A	61	2003	61
0676	91	0AEC	31	199D	61	2008	119
0683	91	0AEF	31	19A0	61	2039	120
0699	82	0AF4	42,59,107,113,	19A2	60,61	2067	120
069A	82,83		116,118	19AE	54,55,83,100,102	206C	92
069F	53,79	0AF6	61	19EC	73	206D	83
06A1	82	0AFB	33	1A18	54,83,100,102	206F	73,120
06CB	82	OB26	29,120	1A19	18,54,75,81,108,	2073	83
06CC	54,55,82,83,	OB37	29,120		117,118	2075	83
	108,117	OB3D	29	1A1C	73	2077	83
06D1	82	OB59	29	1A33	55	207C	92,93
06D2	82	OBAA	29	1A7E	55	2081	93
0707	82	0BC7	28	1A8B	55	208F	92
0708	28	0BD2	28	1A98	55	209B	92,93
070B	28	0BF2	28	1AA1	73	20A0	93
0710	28	0C5B	29	1AEC	73	20A5	92
0713	28	0C70	28	1AF2	73	20B8	83
0716	28	0C77	28	1AF8	56,60	20BC	83
0778	29	OD33	28	1AFC	55,56,60	20C6	73
07B2	61	0D45	28	1B2C	56,60	20F6	93
0809	29,120	0DA1	28	1B49	56,120	20F7	83
0847	28	0DDC	28	1B4A	56	20F9	11,15
0897	28	ODE5	28	1B4D	56,94	20FE	11,15,92
089D	28	0E4D	28	1B5D	55,57,73,83	2103	74
08A0	28,29,30	0E65	34	1B5F	83	2108	74,93
08A2	28	0E6C	34	1B61	56,57,58	2117	93
08B1	65	0FOA	28,29	1B8C	73	2137	92,93
08B6	65	0FOB	29,92	1BB3	14,35	213A	83,93
08BB	65	OF18	28,92	1BC0	57	213B	83
08C4	65	0FA7	34	1C90	32,52	213F	11
08CA	65	0FAF	34	1C96	51,111	2141	74
08D2	65	0FB0	34,35,47	1CA1	120	2164	93
08F4	65	0FBE	35,47	1D78	51,112	2166	93
093E	28	1093	107	1D8B	112	2167	83
0955	32	1225	92	1D91	57,121	2169	11,92
0964	31	1226	91	1D9B	14	2174	73
0977	29,119	1233	91	1DA9	121	218A	61
0982	29	1234	91	1DAE	120	219A	120
098A	29,121	1239	91	1DB0	73	219E	74
0994	32	123A	91	1DBE	113,116,118	21C9	14,36
099B	32	1243	92	1DE4	119	21CD	14,113,116,117
09A4	30,32	1244	92	1DE9	61	21D5	51
09B1	32	124B	92	1DF7	57,121	21E3	14,35
09B4	32	124C	83,92	1DF8	57,121	21EF	120
09BF	32	124D	83	1E00	119	2212	61
09C2	32	124F	92	1E03	119	222D	74
09CB	33	1265	92	1E06	119	226A	93
09CE	33	136C	91	1E09	119	2278	74
09D2	33,48	13E7	29,121	1E3D	57	22A0	61
09D3	33,49	13F2	29,30	1E46	35,37,114,117	22B6	120
09D6	33,44	13F5	29,30	1E4A	61,92,114,117	2335	36
09D7	33,44	13F7	29	1E4F	35	2337	36,41,93,94,106
09DF	32	1439	29,120	1E5A	35,55,114	2490	28
09F4	33	14C9	29,121	1E7A	119	24A0	61
09F7	33	14CC	29	1E83	58,101	252C	36,59,106,107

2540	36	2BC6	119	310A	88	37DA	88
2587	58	2BE4	60	310B	87, 88	37DB	89
258C	43	2BE5	60	313A	88	37DC	89
25A1	43	2BF5	17, 119	313B	88	37DD	63
25B8	43	2C1F	18, 84, 93, 119	313F	88	37DE	63
25D9	31, 52	2C25	93	3140	88	37DF	63
25F7	30	2C29	93	3144	88	37E0	63, 84, 89
25FD	30	2C33	94	3145	85	37E1	63, 89
2608	58, 119	2C39	94	3185	88	37E4	63, 89, 96, 97
260D	36, 48, 49, 58, 113, 116, 118	2C42	84	31A4	85	37E5	89
2733	61	2C7A	84	31BD	84	37E7	89
273D	61	2C7F	84	3203	75	37E8	15, 16, 63, 64, 82, 96, 97, 102, 103
27D4	58, 120	2C81	84	3241	75	37EA	89
27DF	44	2C82	84	32BA	75	37EB	80
27F5	58, 120	2C8A	84	32CA	75	37EC	63, 64
27FE	59, 73	2C8C	84	3365	53	37ED	63
2802	59	2CAA	120	338E	88	37EE	63
2828	59, 113, 116, 117	2CBD	120	33FC	52	37EF	63
2831	61	2C8D	92	33FE	52	37F4	89
2836	44, 121	2E60	119	33FF	52	37F5	89
2839	107	2FFB	94	3400	52, 88 87, 88	37F6	89
2857	48, 49	2FFF	63, 86	3401	52	37FC	89
2865	47	3000	18, 63, 84, 86	344A	52	37FD	89
2866	47	3003	18, 84	344C	52	37FE	89
2868	47, 113, 116, 118	3006	18, 84	344D	52	37FF	63, 84, 86, 89
2869	47	3009	18, 84	344E	52	3800	63
28A1	61	300C	18, 81, 84	3454	88	3801	65
28A6	11	300F	18, 84	3455	88	3802	65
28A7	11, 15, 34, 60, 80, 84	3012	53, 79, 84, 87	3461	87	3804	65
28BF	48	3013	87	34CD	88	3808	65
28DB	61	3015	51, 79, 84, 87	34CE	87, 88	3810	65
299C	43	3016	87	34D9	88	3820	65
29A3	61	3018	52, 84	34FC	88	3840	13, 65, 103 9, 77, 89
29C6	43, 44	301B	19, 84	34FD	88	3880	63
29C8	44	301E	19, 84	3517	88	3BFF	8, 12, 63, 92 8, 12, 63
29CD	44	3021	19, 84	3518	88	3C00	3, 50, 51, 63, 64, 82
29CE	44	3024	14, 82, 84	351B	88	3FFF	51
2A03	45, 120	3027	20, 84, 86, 87	351C	88	4003	32, 52
2A0F	45, 119	3029	86, 87	351E	88	4006	31, 52
2A13	94	302A	84	351F	88	4009	3, 12, 52
2A1F	45, 119	302D	83, 84, 87	3527	88	400C	52
2A3D	45	302E	87	3528	88	400F	3, 52
2A3E	45	3030	46, 47, 49, 72, 84, 87	3529	53	4012	64
2A3F	45	3031	46, 87	35A9	65	4013	8, 64
2A61	46, 120	3033	49, 84	35FA	53, 65, 75	4014	8, 10
2A64	45	3036	49, 84	36D6	88	4015	8, 14
2A68	46, 48, 49, 59	3039	54, 79, 84	36E0	88	4016	8, 14, 92
2A91	46, 121	303D	87	36E6	88	4017	8
2A94	46	303E	87	36FF	88	4018	8
2A9A	46, 120	3040	86	3700	88	4019	8
2AB3	46	3041	87	3707	89	401A	8
2AB4	46	3042	18, 84, 87	3708	89	401B	8
2AC5	46, 121	3043	87	370F	89	401C	8
2AEC	74	3044	84, 86, 87	3710	89	401D	8
2AEF	120	3045	86	3731	89	401E	8
2AFB	120	3060	87	3733	89	401F	8
2B01	92	3064	87	3739	89	4020	8, 14, 92
2B02	37, 114	307D	87	377A	89	4021	8, 14
2B05	31, 74	3084	87	377B	89	4022	8
2B29	120	30A0	87	3798	89	4023	8
2B2E	120	30A4	87	3799	89	4024	8
2B44	74	30BD	87	37A3	89	4025	8
2B75	11, 59	30C4	87	37A4	87, 89	4026	8, 101
2B7E	59	30E0	87	37AE	89	4027	8
2B85	83	30E4	87	37AF	89	4028	8, 9
2B88	83	30FD	88	37B4	89	4029	8, 9
2B89	83	3100	52	37B5	89	402A	8
2B8C	83	3102	52	37C1	87, 89	402B	8
2B8E	83	3103	52	37C2	89	402C	8, 64, 88
2B91	83, 84	3104	52, 88	37D7	89	402D	64, 100, 101, 108
2B93	83	3105	88	37D8	89		

402F	64	40D6	48,69	4151	27,72	41F9	9,19
4030	64	40D7	69	4152	2,3,72,119	41FA	9,19
4032	64	40D8	69	4155	72	41FB	89
4033	64	40D9	69	4158	72,119	41FC	75,89
4035	64,82	40DA	69	415B	72,119	41FD	75
4036	64,65,75,77,88	40DB	69	415E	72,119	41FE	75
4037	65	40DC	69	4161	72,120	41FF	75
4038	65	40DD	69	4164	72,120	4200	75
4039	65	40DE	36,69	4167	72,120	4201	75
403A	65	40DF	69	416A	72,120	4202	75
403B	65	40E0	69	416D	72,120	4203	18,75
403C	64,65,75,77,88	40E1	70	4170	72,120	4205	75
403D	11,12,53,65,76,85	40E2	70	4173	72,119	4206	53,75
403E	65	40E3	70	4176	72,121	4208	75
4040	49,53,65,76	40E4	70	4177	89	4209	53,75
4041	49,76	40E5	70	4179	72,120	420B	75
4042	49,76	40E6	36,70	417C	72,120	420C	18,75
4043	49,53,65,76	40E7	36,70	417F	72,120	420D	75
4044	49,76	40E8	62,70	4182	72,120	420E	18,75
4045	49,76	40E9	62,70	4185	72,119	420F	75
4046	49,53,65,76	40EA	61,70	4188	72,120	4210	12,15,75,76,85
4049	54,65,98,101	40EB	61,70	418B	72,120	4211	18,76
404A	54	40EC	35,70	418E	3,72,120	4212	76
407D	65	40ED	35,70	4191	72,120	4213	76
407F	54,65	40EE	70	4194	72,106	4214	76
4080	30,65	40EF	70	4197	72,120	4215	76
408D	30,65	40FO	70	419A	72,121	4216	49,65,76
408E	66,73	40F1	70	419D	72	4217	49,76
408F	66,73	40F2	61,70	41A0	72,121	4218	49,76
4090	66	40F3	70	41A3	2,3,72,111,116, 117,120	4219	49,76
4092	66	40F4	70			421A	49,76
4093	66	40F5	70	41A5	72	421B	49,76
4094	66	40F6	70	41A6	3,4,62,73	421C	49,65,76
4095	66	40F7	70	41A9	73	421D	9,76,79,89
4096	66	40F8	70	41AC	73	421E	9
4097	66	40F9	70	41AF	13,73	421F	9,89
4098	66	40FA	70	41B2	73	4220	9,19
4099	44,66	40FB	71	41B5	55,73	4221	19
409A	66	40FC	71	41B8	55,73	4222	19
409B	11,16,66	40FD	71	41BB	73,101	4223	19,76
409C	10,11,34,67,92	40FE	71	41BE	11,73	4224	9,77
409D	67	40FF	57,71	41C1	10,73	4225	77
409E	67	4100	71	41C4	13,73	4288	75,77
409F	67	4101	37,71	41C7	73	42E4	77
40A0	57,58,67,69,83	4102	37,71	41CA	73,92	42E5	74,77,79
40A1	57,67,69	4103	37,71	41CD	73	42E7	74
40A2	59,67	411A	37,71	41D0	11,74	42E8	63,74,77,79,80
40A3	59,67	411B	57,71	41D3	74	42E9	38,83
40A4	67,70,82,83	411C	27,72	41D6	74	42F8	75,80
40A5	67,70,82	411D	26,27,106	41D9	74	434C	80,83
40A6	11,15,58,67,92	411E	27	41DC	74	43E8	63,74,77,80
40A7	5,13,57,59,67,74, 79,113,118	411F	27	41DF	74	43E9	38,83
40A8	5,13,57,59,67, 74,113	4120	27	41E2	3,4,74	4411	98,101
40A9	36,67,93	4121	9,27,32,33,40,42, 47,48,49,94,106,	41E4	73	4414	80
			107,108,114	41E5	8,74,75,77,79	444C	80,83
				41E6	8	4514	80
40AA	68	4122	9,27,32,33,40,42, 47,48,114	41E7	8,74		
40AB	68			41E8	8,19,74,79		
40AC	68	4123	9,27,32	41E9	8		
40AD	68	4124	26,27,32	41EB	8,77,88		
40AE	68	4125	27	41EC	8,77,88		
40AF	11,26,52,58,68, 106,113	4126	27	41ED	9		
40B0	68	4127	26,27	41EE	9		
40B1	67,68,69,98,101	412A	27	41EF	9		
40B2	68,69	412E	26,27,72	41F0	9,19		
40B3	68	412F	72	41F1	9		
40B4	68	4130	34,72,107	41F3	9,77,89		
40B5	40,68	4135	34,72	41F4	9,77,89		
40D2	62,68	4136	34,72	41F5	9		
40D3	48,69	4149	34,72	41F6	9		
40D5	48,69	414A	27,72	41F7	9		
				41F8	9,19,75,80		

Afterword
by Charley Butler

I am very proud that Jack Decker has chosen us for his publisher. Jack has always freely shared his technical expertise with TAS readers. He has contributed to the well-being of many programs that are published by TAS: Some without reward or recognition. For that, and this book, we extend our most cordial: Thank You, Jack.

Mastering assembly language is difficult, especially if you don't have the "big picture". The element most frequently missing is a general understanding of your machine, what it does, and how it works. Until you have the understanding of some rather unusual concepts, like "vectors", "device control blocks" and "number type flags" (and a few other things) you cannot become a capable machine language programmer. But assembly can be mastered with lots of patience and information. Jack provides the last part -- you've got to provide the first.

Don't be afraid to read this book twice. There's too much to grasp in a single sitting.

And don't be afraid to report errata, missing entries, and suggestions for future volumes. Who knows, maybe Jack will want to show us how to "tweak" BASIC and eliminate little used functions and add needed functions. We need your feedback. Part of the price you pay for this book is to help cover the expense of monitoring and acting on that feedback -- and ultimately providing a better product for all.

Address all correspondence to:

The Alternate Source
704 North Pennsylvania Avenue
Lansing, Michigan 48906

Thank you for your kind support.

