IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

Radio Shack warrants for a period of 90 days from the date of delivery to customer that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if this unit's case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair. Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U K
280-316 VICTORIA ROAD	PARC INDUSTRIEL DE NANINNE	BILSTON ROAD WEDNESBURY
RYDALMERE, N.S.W 2116	5140 NANINNE	WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.

Written by James Farvour

Microsoft BASIC Decoded & Other Mysteries

Foreword by Harvard Pennington

Edited by Jim Perry

Graphics by John Teal
Cover by Harvard Pennington

TRS-80 Information Series Volume 2

Contents

Acknowledgments

This book has been a long time in its creation, without the help, advice and support of many people it would not have been possible. In particular thanks are due to Rosemary Montoya for her days of keyboarding, David Moore for hours of example testing, Jerry De Diemar, Mary and MG at Helens place for turning the Electric Pencil files into type and Al King for his 24 hour message service.

This book was produced with the aid of several TRS-80 computer systems, an NEC Spinterm printer, the Electric Pencil word processor with a special communications package to interface to an Itek Quadritek typesetter, plus lots of coffee and cigarettes.

Microsoft BASIC Decoded & Other Mysteries
ISBN 0 - 936200 - 01 - 4

The small print

First Edition
First Printing

January 1981

Published by

IJG Computer Services

1260 W Foothill Blvd
Upland, CA 91786 USA.

Foreword

A little over a year ago, I said to Jim Farvour, 'Jim, why don't you write a book about Microsoft BASIC and the TRS-80? You have the talent and the expertise and thousands of TRS-80 owners need help, especially me!'. Needless to say, he agreed. Now it's one thing to SAY you are going to write a book and quite another thing to actually do it.

Writing a book requires fantastic disipline, thorough knowledge of the subject matter, talent and the ability to communicate with the reader. Jim Farvour has all of the above.

This is no ordinary book. It is the most complete, clear, detailed explanation and documentation you will see on this or any similar subject.

There have been other books and pamphlets purporting to explain the TRS-80 BASIC interperter and operating system. They have had some value, but only to experienced machine language programers - and even then these books had many short-comings.

This book will delight both professional and beginner. Besides walking you through power-up and reset (with and without disk) there are detailed explanations of every single area of the software system's operation. Examples, tables, and flow-charts complement the most extensively commented listing you have ever seen. There are over 7000 comments to Microsoft's BASIC interperter and operating system.

These are not the usual machine language programmer's comments whose cryptic and obscure meanings leave more questions than answers. These are english comments that anyone can understand. Not only that, but when a comment needs more explanation, you will find it on the next page.

This book even has something for anyone running Microsoft BASIC on a Z-80 based computer. Microsoft, in its great wisdom, has a system that generates similar code for similar machines. Although you may find that the code is organized differently in your Heath or Sorceror the routines are, for the most part, identical!

Is this a great book? It's an incredible book! It may well be the most useful book you will ever own.

H.C. Pennington

November 1980

Chapter 1

Introduction

Level II consists of a rudimentary operating system and a BASIC language intrepreter. Taken together, they are called the Level II ROM System. There is a extension to the Level II system called the Disk Operating System DOS, and also an extension to the BASIC portion of Level II called Disk BASIC.

Both Level II and DOS are considered independent operating systems. How the two systems co-exist and co-operate is a partial subject of this book. The real purpose is to describe the fundamental operations of a Level II ROM so that assembly language programmers can make effective use of the system.

A computer without an operating system is of little use. The reason we need an operating system is to provide a means of communication between the computer and the user. This means getting it to 'listen' to the keyboard so that it will know what we want, and having it tell us what's going on by putting messages on the video. When we write programs, which tell the computer what to do, there has to be a program inside the machine that's listening to us. This program is called an operating system.

It is impossible to give an exact definition of an operating system. There are thousands of them, and each has slight variations that distinguish it from others. These variations are the result of providing specific user features or making use of hardware features unique to the machine that the operating system is designed for. In spite of the differences between operating systems, the fundamental internal routines on most are very similar - at least from a functional point of view.

The common components in a general purpose, single user system, such as Level II would consist of:

1. Drivers (programs) for all peripheral devices such as the keyboard, video, printer, and cassette.

2. A language processor capability (such as BASIC, COBOL, or FORTRAN) of some kind.

3. Supporting object time routines for any language provided. This would include math and arithmetic routines, which are implied by the presence of a language.

4. Ancillary support routines used by the language processor and its implied routines. These are usually invisible to the user. They manage resources such as memory and tables, and control access to peripheral devices.

5. A simple monitoring program that continually monitors the keyboard, or other system input device, looking for user input.

6. System utility commands. These vary considerably from system to system. Examples from Level II would be: EDIT, LIST, CLOAD, etc.

Remember that these definitions are very general. The exact definition of any individual component is specific to each operating system. In the case of the Level II ROMs we'll be exploring each of the components in more detail later on. First we will discuss how the operating system gets into the machine to begin with.

Generally, there are two ways an operating system can be loaded. The operating system can be permanently recorded in a special type of memory called Read Only Memory (ROM) supplied with the system. In this case the operating system is always present and needs only to be entered at its starting point, to initialize the system and begin accepting commands.

Another way of getting the operating system into the machine is to read it in from some external storage medium such as a disk or cassette. In this case however, we need a program to read the operating system into the machine. This program is called an Initial Program Loader (or IPL), and must be entered by hand or exist in ROM somewhere in the system. For the sake of simplicity, we'll assume that all machines have at least an IPL ROM or ROM based operating system.

In the TRS-80 Model I we have a combination of both ROM and disk based operating systems. A Level II machine has a ROM system which occupies the first 12K of addressable memory. When the Power On or Reset button is pressed control is unconditionally passed to location 0 or 66 respectively. Stored at these locations are JUMPS to another region of ROM which initializes the system and then prints the user prompt 'MEMORY SIZE?'.

In a Level II system with disks, the same ROM program still occupies the first 12K of memory, however during Power On or Reset processing another operating system is read from disk and loaded into memory. This Disk Operating System (DOS) occupies 5K of RAM starting at 16K. After being loaded control is then transfered to DOS which initializes itself and displays the prompt 'DOS READY'. So, even though a ROM operating system is always present, if the machine has disks another operating system is loaded also. In this case, the Level II ROM acts as an IPL ROM.

It should be emphasized that the DOS and ROM operating systems are complementary and co-operative. Each provides specific features that the other lacks. Elementary functions required by DOS are found in ROM, and DOS contains extensions to the ROM, as well as unique capabilities of its own.

Level II And DOS Overview

Level II is a stand alone operating system that can run by itself. It is always present, and contains the BASIC interpreter plus support routines necessary to execute BASIC programs. It also has the facility to load programs from cassette, or save them onto a cassette.

A Disk Operating System, (such as TRSDOS or NEWDOS) is an extension to Level II that is loaded from disk during the IPL sequence. It differs from Level II in several ways. First, it has no BASIC interpreter, in order to key-in BASIC statements control must be passed from DOS to Level II. This is done by typing the DOS command BASIC. As well as transfering control from DOS to Level II this command also performs important initialization operations which will be discussed later. Second, the commands recognized by DOS are usually disk utility programs not embedded routines - such as those in Level II. This means they must be loaded from disk before they can be used. In turn this means that there must be an area of RAM reserved for the loading and execution of these utilities.

Memory Utilization

From the description of DOS and Level II we can see that portions of RAM will be used differently depending on which operating system is being used. Immediately after IPL the memory is setup for each of the operating systems as shown in figure 1.1 below. Notice the position of the Central Processing Unit (CPU) in each part of the figure.

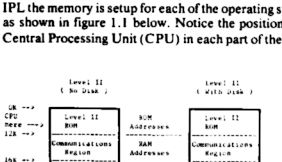


Figure 1.1: Memory organization after the Initial Program Load.

A Level II system with disks that has had a BASIC command executed would appear as in figure 1.2.

The first 16K of memory is dedicated to Level II and the Communications Region regardless of the operating system being used.

Starting at the end of the Communications Region or the Disk BASIC area, depending on the system being used, is the part of memory that will be used by Level II for storing a BASIC program and its variables. This part of memory can also be used by the programmer for keeping assembly language programs. A detailed description of this area for a Level II system without disks follows.

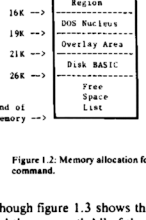


Figure 1.2: Memory allocation for a system with disks, after a BASIC command.

Although figure 1.3 shows the sub-divisions of RAM as fixed they are not! All of the areas may be moved up or down depending on what actions you perform. Inserting or deleting a line from a program, for example, causes the BASIC Program Table (called the Program Statement Table or PST) to increase or decrease in size. Likewise defining a new variable would increase the length of the variables list. Since the orgin of these tables may shift, their addresses are kept in fixed locations in the Communications Region. This allows the tables to be moved about as required, and provides a mechanism for letting other users know where they are.

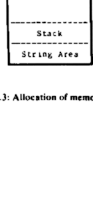


Figure 1.3: Allocation of memory in a Level II system without disks.

The Program Statement Table (PST) contains source statements for a BASIC program in a compressed format (reserved words have been replaced with tokens representing their meaning). The starting address for this table is fixed, but its ending address varies with the size of the program. As program statements are added or deleted, the end of the PST moves accordingly. A complete description of this table can be found in chapter 4 (page 44).

Following the PST is the Variable List Table (or VLT). This contains the names and values for all of the variables used in a BASIC program. It is partitioned into four subtables according to the following variable types: simple variables (non dimensioned); single dimensioned lists; doubley dimensioned lists and triple dimensioned lists. Variable names and their values are stored as they are encountered during the execution of a program. The variable table will change in size as new variables are added to a program, and removing variables will cause the table to shrink. After a variable is defined it remains in the table, until the system is reinitialized. For a full description of this table see chapter 4 (page 45).

Not shown in figure 1.3 is the Free Space List or FSL. It is a section of memory that initially extends from the end of the Communications Region to the lower boundery of the String Area. There are two parts to this list, the first is used to assign space for the PST and VLT. For these areas space is assigned from low to high memory. The second part of the FSL is used as the Stack area. This space is assigned in the opposite direction - beginning at the top of the String Area and working down towards Level II.

The stack area shown is a dynamic (changable) table. It is used by the Level II and DOS systems as a temporary storage area for subroutine return addresses and the hardware registers. Any CALL or RST instruction will unconditionally cause the address of the following instruction to be saved (PUSH'd) onto the stack, and the stack pointer is automatically decremented to the next lower sequential address. Execution of a RET instruction (used when exiting from a subroutine) removes two bytes from the stack (the equivalent of a POP instruction) and reduces the stack pointer by two.

Storage space in the stack area can be allocated by a program, but it requires carefull planning. Some BASIC subroutines such as the FOR-NEXT routine, save all values related to their operation on the stack. In the FOR-NEXT case an eighteen byte block (called a frame) is PUSH'd onto the stack and left there until the FOR-NEXT loop is completed.

Before space is assigned in either part of the FSL (except for Stack instructions such as CALL or PUSH) a test is made (via a ROM call) to insure there is enough room. If there is insufficient space an Out of Memory error is given (OM). See chapter 2 (page 31) for a description of the ROM calls used to return the amount of space available in the FSL.

The last area shown in the memory profile is the string

area. This is a fixed length table that starts at the end of memory and works toward low memory. The size of this area may be specified by the CLEAR command. Its default size is 50 bytes. String variables are stored in this area, however strings made equal to strings, String$ and quoted strings are stored in the PST.

Earlier it was mentioned that there are six general components that form an operating system. Because of the way Level II was put together the individual pieces for some components are scattered around in ROM, instead of being collected together in a single area. Figure 1.4 is an approximate memory map of addresses in Level II. For exact addresses and description of these regions see chapter 4.

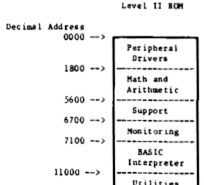


Figure 1.4: Approximate memory map of Level II addresses.

The Communications Region

The Communications Region is a scratch pad memory for the Level II ROMs. An example of addresses stored here are those for the PST and the variables list. Also BASIC supports variable types that require more space than the working registers can provide, and as a result certain arithmetic operations require temporary storage in this region.

Another important use of the Communications Region is to provide a link between Level II and DOS - for passing addresses, and data, back and forth. The DOS Exit addresses and Disk BASIC addresses are kept in this area. As mentioned earlier a Level II system, with disks, begins execution in the DOS system. Control is passed from DOS to Level II only after the command BASIC has been executed (which also updates the Communications Region by storing the DOS Exits and Disk BASIC addresses).

Because Level II is in ROM it is impractical to try and modify it. Yet, changes to an operating system are a practical necessity that must be considered. In order to solve this problem the Level II system was written with jumps to an area in RAM, so that future changes could be incorporated into the ROM system. Those jumps are called DOS Exits, and on a system without a DOS they simply return to Level II. When a DOS is present, the jump addresses are changed to addresses within Disk BASIC which allows changes to be made to the way Level II operates.

The Disk BASIC addresses are used by Level II when a Disk BASIC command such as GET or PUT is encountered. They are needed because the code that supports those operations is not present in Level II. It is a part of Disk BASIC that is loaded into RAM, and since it could be loaded anywhere Level II needs some way of locating it. The Disk BASIC exits are a group of fixed addresses, known to both Level II and Disk BASIC, which allows Level II to pass control to Disk BASIC for certain verb action routines.

Another interesting aspect of the Communications Region is that it contains a section of code called the Divide Support Routine. This code is called by the division subroutines, to perform subtraction and test operations. It is copied from Level II to the RAM Communications Region during the IPL sequence. When a DOS is present it is moved from ROM to RAM by the DOS utility program BASIC.

An assembly language program using the Level II division routine on a disk system which has not had the BASIC command executed will not work because the Divide Support Routine is not in memory. Either execute the BASIC utility or copy the suppport routine to RAM, when executing assembly language routines that make division calls.

Level II Operation

Earlier in this chapter there was a brief description of six components which are generally found in all operating systems. Using those components as a guideline, Level II can be divided into the following six parts:

Part 1 ... Input or scanner routine.
Part 2 ... Interpretation and execution routine.
Part 3 ... Verb action routines
Part 4 ... Arithmetic and math routines
Part 5 ... I/O driver routines.
Part 6 ... System function routines.

There is another part common to all systems which is not included in the above list. This part deals with system initialization (IPL or Reset processing), and it will be discussed separately. Continuing with the six parts of Level II, we will begin at the point where the system is ready to accept the first statement or command. This is called the Input Phase.

Part 1 - Input Phase

The Input Phase is a common part of all operating systems. Its function is to accept keyboard input and respond to the commands received. In the case of a Level II system it serves a dual purpose - both system commands and BASIC program statements are processed by this code.

Entry to the Input Scan routine is at. This is an initial entry point that is usually only called once. The message 'READY' is printed, and a DOS Exit (41AC) is taken before the main loop is entered. Systems without disks jump to this point automatically, at the end of IPL processing. For systems with disks, this code is entered by the DOS after program BASIC at the end of its processing. The Input or Scanner phase is summarized below.

1. Get next line of input from keyboard.
2. Replace reserved words with tokens
3. Test for a system command such as RUN, CLOAD, etc or a DIRECT STATEMENT (BASIC statement without a line number) and branch to 6 if true.
4. Store tokenized statement in program statement table.
5. Return to step 1.
6. Begin interpretation and execution

The Input Phase loop begins at 1A33. After printing the prompt >, or a line number if in the Auto Mode a CALL to 03612 is made to read the next line. Then the line number is converted from ASCII to binary with a CALL to 1E5A. The statement is scanned and reserved words are replaced by tokens (CALL 1BC0). Immediately after tokenization a DOS Exit to 41B2 is taken. Upon return a test for a line number is made. If none is found a System Command or Direct Statement is assumed, and control is passed to the Execution Driver at 1D5A. On systems without disks this test is made at 1AA4. On a disk system the test, and branch, is made at the DOS Exit 41B2 called from 1AA1.

If a line number is present the incoming line is added to the PST, the pointers linking each line are updated by the subroutine at 1AFC to 1B0E. If the line replaces an existing line, the subroutine at 2BE4 is called to move all of the following lines down over the line being replaced.

When in the Auto Mode the current line number is kept in 40E2 and 40E3 the increment between lines is stored at 40E4. The code from 1A3F to 1A73 prints and maintains the automatic line number value. Null lines (statements consisting of a line number only) are discarded. They are detected by a test at 1ABF.

Part 2 - Interpretation & Execution

Statement and command execution in a Level II system is by interpretation. This means that a routine dedicated to the statement type, or command, is called to interpret each line and perform the necessary operations. This is a common method for system command execution. With DOS, for example, seperate modules are loaded for commands such as FORMAT and COPY. In some systems, commands which are related may be combined into a single module, after the module has been loaded it decides which sub-function to execute by examining (interpreting) the name which called it.

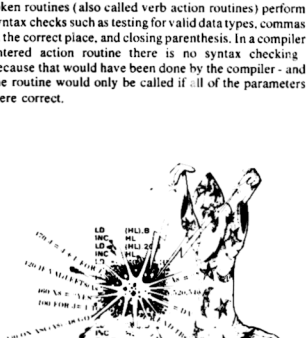
Program execution by interpretation is not common except on microcomputers, and even then only for selected languages such as BASIC and APL. The alternative to an interpreter is program compilation and execution, with the use of a compiler.

Compilers translate source statements into directly executable machine language code (called object code). The object code is then loaded into RAM as a seperate step using a utility program called a Loader. After loading the object code into RAM, control is passed to it and it executes almost independently of the operating system.

Not all source code is converted to object code by a compiler. Some statements such as READ and WRITE or functions such as SINE or COSINE may be recognized by the compiler, and rather than generate code for them, subroutine calls for the specific routines will be produced.

These routines are in object code form in a library file. When the loader loads the object code, for the compiled program, any subroutine calls are satisfied (the subroutines are loaded) from the library file. A loader that will take modules from a library is called a linking loader.

An interpreter operation is much simpler by comparison. Each source statement is scanned for reserved words such as FOR, IF, GOTO, etc.. Every reserved word is replaced by a unique numeric value called a token then the tokenized source statement is saved. In Level II it is saved in the Program Statement Table. When the program is run control goes to an execution driver which scans each statement looking for a token. When one is found control is given to a routine associated with that token. These token routines (also called verb action routines) perform syntax checks such as testing for valid data types, commas in the correct place, and closing parenthesis. In a compiler entered action routine there is no syntax checking because that would have been done by the compiler - and the routine would only be called if all of the parameters were correct.



In Level II the execution phase is entered when a statement without a line number has been accepted, or when a RUN command is given. This may be a system command or a single BASIC statement that is to be executed. When a RUN command is received an entire BASIC program is to be executed. The Execution driver loop starts at 1D5A and ends at 1DE1. These addresses are deceptive though, because portions of this code are shared with other routines.

The steps in this phase are summerized as follows. For more details see figure 1.5.

1. Get the first character from the current line in the PST. If the end of the PST has been reached then return to the Input Phase.
2. If the character is not a token, go to step 6.
3. If the token is greater than BC it must be exactly FA (MID$), otherwise a sytnax error is given.
4. If the token is less than BC, use it as an index into the verb action table.
5. Go to action routine and return to step 1.
6. Assignment section. Locate variable name, if it's not defined, then create it.
7. Call expression evaluation.
8. Return to step 1.

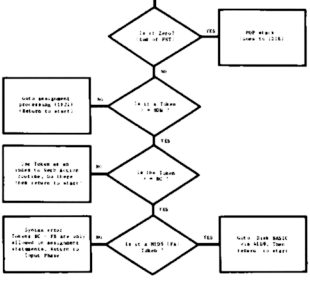


Figure 1.5: Flowchart of the execution driver routine.

The Execution driver begins by loading the first character from the current line in the PST. This character is tested to see if it is a token (80-FA) if not, the current line is assumed to be an assignment statement such as:

A = 1.

The assignment statement routine begins at 1F21. It is similiar to the other action routines, except that it is entered directly rather than through a table look up process. Before it is entered a return address of 1D1E in the execution driver is PUSH'd onto the stack, so it can exit as any other action routine.

The assignment routine assumes that the pointer for the current line is immediately to the left of the variable name to be assigned. It locates, or creates an entry for the variable name, tests for an equals ( ) after the name - and then CALLs 2337. The routine at this location evaluates the expression. The result is converted to the correct mode, and stored at the variable address.

Assuming that a good token was found as the first character, a second test is made to see if it is valid as the first token in a line. Valid tokens which can occur at the start of a line are 80 - BB. The tokens BC - F9 can only occur as part of an assignment statement or in a particular sequence such as 8F (IF) 'Expression' CA (then) XXXX. The MID$ token FA is the only exception to this rule. There is a test for it at 2AE7 where a direct jump to its Disk BASIC vector (41D9) is taken. If the token is between 80 and BB it is used as an index into a verb action routine table and the address of the the action routine, for that token is located. Control is then passed to that action routine which will do all syntax checking and perform the required function.

Parameters for the verb routines are the symbols in the statement following the token. Each routine knows what legitimate characters to expect, and scans the input string from left to right (starting just after the token) until the end of the parameters are reached. The end of the parameters must coincide with the end of the statement, or a syntax error is produced.

Symbols which terminate a parameter list vary for each action routine. Left parentheses '(' terminate all math and string functions. A byte of machine zeros (00) stops assignment statements, other routines may return to the execution phase after verifying the presence of the required value.

As each verb routine is completed control is returned to the Execution driver, where a test for end of statement (EOS) or a compound statement (:) is made. The EOS is one byte of machine zeros. If EOS is detected the next line from the Program Statement Table is fetched, and it becomes the current input line to the Execution driver.

When a System Command or a Direct Statement has been executed there is no pointer to the next statement, because they would have been executed from the Input Phase's input buffer. This is in a different area than the PST where BASIC program statements are stored. When the RUN command is executed, it makes the Execution driver get its input from the PST.

When the end of a BASIC program, or a system command, is reached, control is unconditionally passed to the END verb which will eventually return to the Input Phase. Any errors detected during the Execution and Interpre-

tion phase cause control to be returned to the Input Phase after printing an appropriate error code. An exception is the syntax error, which exits directly to the edit mode.

Part 3 - Verb Action

The verb action routines are where the real work gets done. There are action routines for all of the system commands such as CLOAD, SYSTEM, CLEAR, AUTO as well as the BASIC verbs such as FOR, IF, THEN, GOTO, etc. In addition there are action routines for all the math functions and the Editor sub-commands.

Verb action routines continue analyzing the input string beginning at the point where the Execution phase found the verb token. Like the Execution phase, they examine the string in a left to right order looking for special characters such as (, ,), or commas and tokens unique to the verb being executed. If a required character is missing, or if an illogical condition arises, a syntax error is generated.

The verb routines use a number of internal subroutines to assist them while executing program statements. These internal routines may be thought of as part of the verb action routines, even though they are used by many other parts of the Level II system.

A good example of an internal routine is the expression evaluation routine, which starts at 2337. Any verb routine that will allow, and has detected, an expression as one of its arguements may CALL this routine. Examples of verb action routines that allow expressions in their arguements are IF, FOR, and PRINT. In turn the expression evaluation routine will CALL other internal routines (such as 260D to find the addresses of variables in expressions being evaluated). Since subscripted variables can have expressions as their subscript, the find address routine may in turn CALL back to the expression evaluation routine!

This type of processing is called recursion, and may be forced by the following expression:

```
c0 = c(1a/bc(2d)/c(1*c0))
```

Other internal routines used by the verb action routines are : skip to end of statement 1F05; search Stack for a FOR frame 1936 and build a literal string pool entry 2865.

Any intermediate results, which may need to be carried forward, are stored Work Register Area 1 (WRA1) in the Communications Region. Some verbs such as FOR build a stack frame which can be searched for and recognized by another verb such as NEXT. All of the action routines except MID$ are entered with the registers set as shown in figure 1.6. A full list of verb action routines, and their entry points is given in chapter 4 (page 43).

Register		Contents
AF	-	Next element from code string Flag: Verb token, CARRY - if numeric NO CARRY - if alpha
BC	-	Address of the action routine
DE	-	Address of action token in code string
HL	-	Address of next element in code string

Figure 1.6. Register settings for verb action routine entry.

Part 4 - Arithmetic & Math

Before going into the Arithmetic and Math routines we should review the arithmetic capabilities of the Z-80 CPU and the BASIC interpreter.

The Z-80 supports 8 bit and 16 bit integer addition and subtraction. It does not support multiplication or division, nor does it support floating point operations. Its register set consists of seven pairs of 16 bit registers. All arithmetic operations must take place between these registers. Memory to register operations are not permitted. Also operations between registers are extremely restricted, especially with 16 bit quantities.

The BASIC interpreter supports all operations e.g., addition, subtraction, multiplication, and division for three types (Modes) of variables which are: integer, single precision and double precision. This support is provided by internal subroutines which do the equivalent of a hardware operation. Because of the complexity of the software, mixed mode operations, such as integer and single precision are not supported. Any attempt to mix variable types will give unpredictable results.

The sizes for the variable types supported by BASIC are as follows:

Integer ..... 16 bits (15 bits 1 sign bit)
Single Precision ... 32 bits (8 bit biased exponent plus 24 bit signed mantissa)
Double Precision .... 56 bits (8 bit biased exponent plus 48 bit signed mantissa)

From this it is clear that the registers are not large enough to hold two single or double precison values, even if floating point operations were supported by the hardware. Because the numbers may be too big for the registers, and because of the sub-steps the software must go through an area of RAM must be used to support these operations

Within the Communications Region two areas have been set aside to support these operations. These areas are labeled: Working Register Area 1 (WRA1) and Working Register Area 2 (WRA2). They occupy locations **411D** to **4124** and **4127** to **412E** respectively. They are used to hold one or two of the operands, depending on their type, and the final results for all single and double precision operations. A description of the Working Register Area follows.

Address	Integer	Single Precision	Double Precision
411D			LSB
411E			NMSB
411F			NMSB
4120			NMSB
4121	LSB	LSB	NMSB
4122	MSB	NMSB	NMSB
4123		MSB	MSB
4124		Exponent	Exponent

Where:
LSB = Least significent byte
NMSB = Next most significent byte
MSB = Most significent byte

WRA2 has an identical format.

Figure 1.7: Working Register Area layout

Integer

Destination Register	Operation	Source Registers
HL	Addition	HL + DE
HL	Subtraction	HL - DE
HL	Multiplication	HL * DE
WRA1	Division	DE / HL

Single Precision

Destination Register	Operation	Source Registers
WRA1	Addition	WRA1 + (BCDE)
WRA1	Subtraction	WRA1 - (BCDE)
WRA1	Multiplication	WRA1 * (BCDE)
WRA1	Division	WRA1 / (BCDE)

Double Precision

Destination Register	Operation	Source Registers
WRA1	Addition	WRA1 + WRA2
WRA1	Subtraction	WRA1 - WRA2
WRA1	Multiplication	WRA1 * WRA2
WRA1	Division	WRA1 / WRA2

Figure 1.8: Register arrangements usesd by arithmetic routines.

Because mixed mode operations are not supported integer operations can only take place between integers, the same being true for single and double precision values. Since there are four arithemetic operations (+, -, *, and /), and three types of values, there must be twelve arithmetic routines. Each of these routines knows what type of values it can operate on, and expects those values to be loaded into the appropriate hardware or working registers before being called. Figure 1.8 shows the register assignments used by the arithmetic routines. These assignments are not valid for the Math routines because they operate on a single value, which is always assumed to be in WRA1.

The math routines have a problem in that they must perform arithmetic operations, but they do not know the data type of the argument they were given. To overcome this another byte in the Communications Region has been reserved to indicate the data type (Mode) of the variable in WRA1. This location is called the Type flag. Its address is **40AF** and contains a code indicating the data type of the current contents of WRA1. Its codes are:

CODE	DATA TYPE (MODE)
02	Integer
03	String
04	Single precision
08	Double precision

The math routines do not usually require that an argument be a particular data type, but there are some exceptions (see chapter 2, page xx, for details).

Part 5 - I/O Drivers

Drivers provide the elementary functional capabilities necessary to operate a specific device. Level II ROM contains Input/Output (I/O) drivers for the keyboard, video, parallel printer, and the cassette. The disk drivers are part of the DOS system and consequently will not be discussed.

All devices supported by Level II, with the exception of the cassette, require a Device Control Block (DCB). The drivers use the DCB's to keep track of perishable information, such as the cursor position on the video and the line count on the printer. The DCB's for the video, keyboard, and printer are part of the Level II ROM. Since information must be stored into them, they are moved from ROM to fixed addresses in RAM (within the Communications Region) during IPL.

The Level II drivers must be called for each character that is to be transmitted. The drivers cannot cope with the concept of records or files, all record blocking and de-blocking is left to the user. Level II has no general purpose record management utilities. For BASIC programs you must use routines such as PRINT and INPUT to block off each record.

When writting to a cassette, for example, the PRINT routine produces a header of 256 zeroes, followed by an A5. After the header has been written each individual variable is written as an ASCII string, with a blank space between each variable, finally terminating with a carriage return. Non string variables are coonverted to their ASCII equivalent.

INPUT operation begins with a search for the 256 byte header. Then the A5 is skipped and all variables are read into the line buffer until the carriage return is detected. When the INPUT is completed all variables are converted to their correct form and moved to the VLT.

The keyboard, video and line printer drivers can be entered directly or through a general purpose driver entry point at 03C2. Specific calling sequences for each of these drivers are given in chapter 2

The cassette driver is different from the other drivers in several respects. It does its I/O in a serial bit mode whereas all of the other drivers work in a byte (or character) mode. This means that the cassette driver must transmit data on a bit-by-bit basis. The transmission of each bit is quite complex and involves many steps. Because of the timing involved, cassette I/O in a disk based system, must be done with the clock off (interrupts inhibited). For more details on cassette I/O see chapter 4

Part 6 - System Utilities

System utilities in Level II ROM are the Direct Commands:
AUTO, CLEAR, CSAVE, CLOAD, CLEAR, CONT, DELETE, EDIT, LIST, NEW, RUN, SYSTEM, TROFF and TRON. These commands may be intermixed with BASIC program statements. However, they are executed immediately rather than being stored in the program statement table (PST). After executing a Direct Command, control returns to the Input Phase.

After an entire BASIC program has been entered (either through the keyboard or via CLOAD or LOAD, on a disk system), it must be executed by using the RUN command. This command is no different from the other system commands except that it causes the BASIC program in the PST to be executed (the Execution Phase is entered). As with other system commands, when the BASIC program terminates, control is returned to the Input Phase.

System Flow During IPL

The IPL sequence has already been discussed in general terms. A complete description of the procedure follows. The description is divided into seperate sections for disk and non-disk systems.

Reset Processing (non-disk)

Operations for this state begin at absolute location zero when the Reset button is pressed. From there control is passed to 0674 where the following takes place.
00UFC
A) Ports FF (255 decimal) to 80 (128 decimal) are initialized to zero. This clears the cassette and selects 64 characters per line on the video.

B) The code from 06D2 to 0707 is moved to 4000 - 4035 .This initializes addresses for the restart vectors at 8, 10, 18 and 20 (hex) to jump to their normal locations in Level II. Locations 400C and 400F are initialized to RETURNs.

If a disk system is being IPL'd 400C and 400F will be modified to JUMP instructions with appropriate addresses by SYS0 during the disk part of IPL. The keyboard, video, and line printer DCB's are moved from ROM to RAM beginning at address' 4015 to 402C after moving the DCB's locations 402D, 4030, 4032 and 4033 are initialized for non-disk usage. They will be updated by SYS0 if a disk system is being IPL'd.

C) Memory from 4036 to 4062 is set to machine zeros.(00)

After memory is zeroed, control is passed to location 0075 where the following takes place:
00UFC
A) The division support routine is moved from @FT218F7-191B to 4080-40A6 .This range also includes address pointers for the program statement table. Location 41E5 is initialized to:

LD A, (2C00)

B) The input buffer address for the scanner routine is set to 41E8 .This will be the buffer area used to store each line received during the Input Phase.

C) The Disk BASIC entry vectors 4152-41A5 are initialized to a JMP to 012D .This will cause an L3 ERROR if any Disk BASIC features are used by the program. Next, locations 41A6-41E2 (DOS exits) are set to returns (RETs). 41E8 is set to zero and the current stack pointer (CSP) is set to 41F8 .(We need a stack at this point because CALL statements will be executed during the rest of the IPL sequence and they require a stack to save the return address).

D) A subroutine at 1B8F is called. It resets the stack to 434C and initializes 40E8 to 404A . It then initializes the literal string pool table as empty, sets the current output device to the video, flushes the print buffer and turns off the cassette. The FOR statement flag is set to zero, a zero is stored as the first value on the stack and control is returned to 00B2

E) The screen is cleared, and the message 'MEMORY SIZE' is printed. Following that, the response is accepted

and tested, then stored in 40B1. Fifty words of memory are alloted for the string area and its lower boundry address is stored in 40A0.

F) Another subroutine at 1B4D is called to turn Trace off, initialize the starting address of the simple variables (40F9), and the program statement table (40A4). The variable type table 411A is set to single precision for all variables, and a RESTORE is done. Eventually control is returned to 00FC .

G) At 00FC the message 'RADIO SHACK Level II BASIC' is printed and control is passed to the Input Phase.

Reset Processing (disk systems)

Operations for this state begin at location 0000 and jump immediately to 0674 . The code described in paragraphs A, B, and C for RESET processing (non-disk systems on page xx) is common to both IPL sequences. After the procedure described in paragraph C has taken place a test is made to determine if there are disks in the system. If there are no disk drives attached, control goes to 0075, otherwise. . .
00UFC
A) Disk drive zero is selected and positioned to track 0 sector 0. From this position the sector loader (BOOT/SYS) is read into RAM locations 4200 - 4455. Because the sector loader is written in absolute form it can be executed as soon as the READ is finished.

After the READ finishes, control is passed to the sector loader which positions the disk to track 11 sector 4. This sector is then read into an internal buffer at 4D00. The sector read contains the directory entry for SYS0 in the first 32 bytes. Using this data the sector loader computes the track and sector address for SYS0 and reads the first sector of it into 4D00.

B) Following the READ, the binary data is unpacked and moved to its specified address in RAM. Note that SYS0 is not written in absolute format so it cannot be read directly into memory and executed. It must be decoded and moved by the sector loader. Once this is done control is passed to SYS0 beginning
at address 4200.

C) The following description for SYS0 applies to NEWDOS systems only. It begins by determining the amount of RAM memory and storing its own keyboard driver address in the keyboard DCB at 4015. The clock interrupt vector address (4012) is initialized to a CALL 4518. Next, more addresses are initialized and the NEWDOS header message is written.

D) After writing the header, a test for a carriage return on the keyboard is made. If one is found, the test for an AUTO procedure is skipped and control passes immediately to 4400 were the DOS Input SCANNER phase is initiated.

Assuming a carriage return was not detected the Granule Allocation Table (GAT) sector (track 11 sector 0) is read and the E0 byte is tested for a carriage return value. Again, if one is found (the default case) control goes to 4400, otherwise a 20 byte message starting at byte E0 of the GAT sector is printed. Then control is passed to 4405 where the AUTO procedure is started. Following execution of the AUTO procedure control will be passed to the DOS Input Phase which starts at 4400.

Disk BASIC

One of the DOS commands is a utility program called BASIC. In addition to providing a means of transfering control from DOS to Level II, it contains the interpretation and execution code for the following Disk BASIC statements:

```
TRSDOS and NEWDOS
CVI      CVS      CVD      MKI$     MKS$     MKD$     DEFFN    DEFUSR
TIME$    CLOSE    FIELD    GET      PUT      AS       LOAD     SAVE
KILL     MERGE    NAME     LSET     RSET     INSTR    LINE     AH
A0       CMD"S"   CMD"T"   CMD"R"   CMD"D"   CMD"A"   USR0-USR9
MID$ (left side of equation)       OPEN"R"  OPEN"O"  OPEN"I"

NEWDOS only
OPEN"E" RENUM    REF      CMD"E"   CMD"DOS command"

An additional command peculiar to TRSDOS only is:
                CMD"X", <ENTER>  - Version 2.1
                CMD"#", <ENTER>  - Version 2.2 & 2.3
```

These hidden, and undocumented commands display a 'secret' copyright notice by Microsoft. Also undocumented is CMD'A' which performs the same function as CMD'S'.

Disk BASIC runs as an extension to Level II. After being loaded, it initializes the following section of the Communications Region:
00UFC
1. DOS exits at 41A6 - 41E2 are changed from RETURN's to jumps to locations within the Disk BASIC utility.
2. The Disk BASIC exits at 4152 - 41A3 are changed from JP 12D L3 syntax error jumps to addresses of verb action routines within Disk BASIC.

Following the initialization of the Communications Region, DCBs and sector buffers for three disk files are allocated at the end of Disk BASIC's code. Control is then given to the Input Scanner in Level II (1A19).

Disk BASIC will be re-entered to execute any Disk BASIC statement, or whenever a DOS Exit is taken from Level II. The Disk BASIC entry points are entered as though they are verb action routines. When finished control returns to the execution driver.

Note: Disk BASIC occupies locations 5200 - 5BAD (NEWDOS system). Each file reserved will require an additional (32 256 decimal) bytes of storage. Assembly programs should take care not to disturb this region when running in conjunction with a BASIC program.

Chapter 2

Subroutines

Level II has many useful subroutines which can be used by assembly language programs. This chapter describes a good number of the entry points to these subroutines. However there are many more routines than those described here. Using the addresses provided as a guide, all of the Level II routines dealing with a particular function may be easily located.

Before using the math or arithmetic calls study the working register concept and the mode flag (see chapter 1 page 14). Also, remember that the Division Support Routine (see chapter 1 page 10) is loaded automatically only when IPL'ing a non-disk system. On disk systems it is loaded by the Disk BASIC utility. If you are using a disk system and executing an assembly language program, which uses the any of the math or arithmetic routines that require division, you must enter BASIC first or load the Division Support Routine from within your program.

The I/O calling sequences described are for Level II only. The TRSDOS and Disk BASIC Reference Manual contains the DOS calling sequences for disk I/O.

The SYSTEM calls and BASIC functions are somewhat specialized, consequently they may not always be useful for an application written entirely in assembly language. However if you want to combine assembly and BASIC you will find these routines very useful.

I/O Calling Sequences

Input and Output (I/O) operations on a Model I machine are straight forward, being either memory mapped or port addressable. There are no DMA (direct memory access) commands and interrupt processing is not used for I/O operations.

The selection of entry points presented here is not exhaustive. It covers the more general ones and will point the reader in the right direction to find more specialized entry points, if needed.

In memory mapped operations, storing or fetching a byte from a memory location, causes the data to be transfered between the CPU register and the target device. Examples of memory mapped devices are the video, the keyboard, and the disk. Programmed I/O (via ports) is a direct transfer of data between a register and a device. The only device using port I/O is the cassette.

Keyboard Input

The keyboard is memory mapped into addresses **3800 - 3BFF**. It is mapped as follows:

BIT	<---------- Keyboard Addresses ---------->							
	3801	3802	3804	3808	3810	3820	3840	3880
0	@	H	P	X	0	8	ENTER	SHIFT
1	A	I	Q	Y	1	9	CLEAR	
2	B	J	R	Z	2	:	BREAK	
3	C	K	S		3	;	UP ARW	
4	D	L	T		4	,	DN ARW	
5	E	M	U		5	-	LT ARW	
6	F	N	V		6	.	RT ARW	
7	G	O	W		7	/	SP BAR	

When a key is depressed, a bit in the corresponding position in the appropriate byte, is set, also bits set by a previous key are cleared. You will notice that only eight bytes (3801 - 3880) are shown in the table as having any signifigance. This might lead one to believe that the bytes in between could be used. Unfortunately this is not the case as the byte for any active row is repeated in all of the unused bytes. Thus all bytes are used.

CALL 002B — Scan Keyboard

Performs an instantaneous scan of the keyboard. If no key is depressed control is returned to the caller with the A-register and status register set to zero. If any key (except the BREAK key) is active the ASCII value for that character is returned in the A-register. If the BREAK key is active, a RST 28 with a system request code of 01 is executed. The RST instruction results in a JUMP to the

DOS Exit 400C On non-disk systems the Exit returns, on disk systems control is passed to SYS0 where the request code will be inspected and ignored, because system request codes must have bit 8 on. After inspection of the code, control is returned to the caller of **002B**. Characters detected at **002B** are not displayed. Uses DE, status, and A register

```
;
;  SCAN KEYBOARD AND TEST FOR BREAK OR ASTERISK
;
      PUSH    DE              ; SAVE DE
      PUSH    IY              ; SAVE IY
      CALL    2BH             ; TEST FOR ANY KEY ACTIVE
      DEC     A               ; KEY ACTIVE, WAS IT A BREAK
      JR      M,NO            ; GO IF NO KEY HIT
      JR      Z,BRK           ; ZERO IF BREAK KEY ACTIVE
      INC     A               ; <A> BACK TO ORIGINAL VALUE
      CP      2AH             ; NO, TEST FOR * KEY ACTIVE
      JR      Z,AST           ; ZERO IF *
          .
          .
          .
```

CALL 0049 Wait For Keyboard Input

Returns as soon as any key on keyboard is pressed. ASCII value for character entered is returned in A-register. Uses A, status, and DE registers.

```
;
;   WAIT FOR NEXT CHAR FROM KEYBOARD AND TEST FOR ALPHA
;
      PUSH    DE              ; SAVE DE
      PUSH    IY              ; AND  IY
      CALL    49H             ; WAIT TILL NEXT CHAR. ENTERED
      CP      41H             ; TEST FOR LOWER THAN "A"
      JR      NC,ALPHA        ; JMP IF HIGHER THAN NUMERIC
          .
          .
          .
```

CALL 05D9 Wait For Next Line

Accepts keyboard input and stores each character in a buffer supplied by caller. Input continues until either a carriage return or a BREAK is typed, or until the buffer is full. All edit control codes are recognized, e.g. TAB, BACKSPACE, etc. The calling sequence is:
On exit the registers contain:

```
;
;  GET NEXT LINE FROM KEYBOARD, EXIT IF BREAK STRUCK.
;  LINE CANNOT EXCEED 25 CHARACTERS
;
SIZE  EQU     25              ; MAX LINE SIZE ALLOWED
      LD      HL,BUFF         ; BUFFER ADDRESS
      LD      B,SIZE          ; BUFFER SIZE
      CALL    5D9H            ; READ NEXT LINE FROM KEYBOARD
      JR      C,BREAK         ; JMP IF BREAK TYPED
          .
          .
          .
BUFF  DEFS    SIZE            ; LINE BUFFER
          .
          .
          .
```

HL Buffer address
B Number of characters transmitted excluding last.
C Orginal buffer size
A Last character received if a carriage return or BREAK is typed.
Carry Set if break key was terminator, reset otherwise.

If the buffer is full, the A register will contain the buffer size.

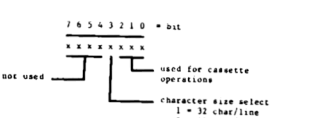
Video Output

Video I/O is another example of memory mapped I/O. It uses addresses 3C00 thru 3FFF where 3C00 represents the upper left hand corner of the video screen and 3FFF represents the lower right hand corner of the screen.

Screen control codes such as TAB, CURSON ON/OFF, BACKSPACE and such are processed by the video driver routine. The video device itself does not recognize any control codes. Codes recognized by the driver and their respective actions are:

Code (hex.)	Action
08	backspace and erase character.
0E	turn on cursor.
0F	turn off cursor.
17	select line size of 32 char/line.
18	backspace one character (left arrow)
19	skip forward one character (right arrow)
1A	skip down one line (down arrow).
1B	skip up one line (up arrow).
1C	home cursor. select 64 char/line.
1D	position cursor to start of current line
1E	erase from cursor to end of line
1F	erase from cursor to end of frame

Character and line size (32/64 characters per line) is selected by addressing the video controller on port FF, and sending it a function byte specifying character size. The format of that byte is:



CALL 0033 Video Display

Displays the character in the A-register on the video. Control codes are permitted. All registers are used.

```
;
;    DISPLAY MESSAGE ON VIDEO
;
      LD      HL,LIST         ; MESSAGE ADDRESS
LOOP  LD      A,(HL)          ; GET NEXT CHARACTER
      OR      A               ; TEST FOR END OF MESSAGE
      JR      Z,DONE          ; JMP IF END OF MESSAGE (DONE)
      PUSH    HL              ; NT END, PRESERVE HL
      CALL    33H             ; AND PRINT CHARACTER
      POP     HL              ; RESTORE HL
      INC     HL              ; BUMP TO NEXT CHARACTER
      JR      LOOP            ; LOOP TILL ALL PRINTED
DONE      .
          .
          .
LIST  DEFM    'THIS IS A TEST'
      DEFB    0DH             ; CARRIAGE RETURN
      DEFB    0               ; END OF MESSAGE INDICATOR
```

CALL 01C9 — Clear Screen

Clears the screen, selects 64 characters and homes the cursor. All registers are used.

```
;
; CLEAR SCREEN, HOME CURSOR, SELECT 32 CHAR/LINE
; SKIP 4 LINES
;
      CALL    01C9H          ; CLEAR SCREEN
      LD      A,17H          ; SELECT 32 CHAR/LINE
      CALL    0033H          ; SEND CHAR SIZE TO VIDEO
      LD      B,4            ; NO. OF LINES TO SKIP
      LD      A,1AH          ; CODE TO SKIP ONE LINE
LOOP  PUSH    BC             ; SAVE BC
      CALL    33H            ; SKIP 1 LINE
      POP     BC             ; GET COUNT
      DJNZ    LOOP           ; LOOP TILL FOUR LINES DONE
```

CALL 022C — Blink Asterisk

Alternately displays and clears an asterisk in the upper right hand corner. Uses all registers.

```
;
;  BLINK ASTERISK THREE TIMES
;       LD   B,3        ; NO. OF TIMES TO BLINK
 LOOP  PUSH BC          ; SAVE COUNT
       CALL 022CH       ; BLINK ASTERISK ONCE
       POP  BC          ; GET COUNT
       DJNZ LOOP        ; COUNT 1 BLINK
 DONE    .
         .
         .
```

Printer Output

The printer is another example of a memory mapped device. Its address is 37E8H. Storing an ASCII character at that address sends it to the printer. Loading from that address returns the printer status. The status is returned as a zero status if the printer is available and a non-zero status if the printer is busy.

CALL003B — Print Character

The character contained in the C-register is sent to the printer. A line count is maintained by the driver in the DCB. When a full page has been printed (66 lines), the line count is reset and the status register returned to the caller is set to zero. Control codes recognized by the printer driver are:

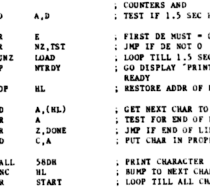
CODE	ACTION
00	Returns the printer status in the upper two bits of the A-register and sets the status as zero if not busy, and non-zero if busy.
0B	Unconditionally skips to the top of the next page.
0C	Resets the line count (DCB 4) and compares its previous value to the lines per page (DCB 3) value. If the line count was zero, no action is taken. If the line count was non-zero then a skip to the top form is performed.
0D	Line terminator. Causes line count to be incremented and tested for full page. Usually causes the printer to begin printing.

```
;
; WRITE MESSAGE ON PRINTER. IF NOT READY WITHIN 1.5 SECONDS
; DISPLAY ERROR MESSAGE ON VIDEO
;
       LD     HL,LIST        ; ADDR OF LINE TO PRINT
START  LD     B,5            ; PREPARE TO TEST FOR PRINTER
                              READY
LOAD   LD     DE,10H         ; LOAD DELAY COUNTERS
TST    CALL   05D1H          ; GET PRINTER STATUS
       JR     Z,ADY          ; JP IF PRINTER READY
       DEC    DE             ; NOT READY, DECREMENT
                              ; COUNTERS AND
       LD     A,D            ; TEST IF 1.5 SEC HAS ELAPSED
       OR     E              ; FIRST DE MUST = 0
       JR     NZ,TST         ; JMP IF DE NOT 0
       DJNZ   LOAD           ; LOOP TILL 1.5 SEC PASSED
       JP     NTRDY          ; GO DISPLAY 'PRINTER NOT
                              READY
RDY    POP    HL             ; RESTORE ADDR OF PRINT LINE
       LD     A,(HL)         ; GET NEXT CHAR TO PRINT
       OR     A              ; TEST FOR END OF LINE
       JR     Z,DONE         ; JMP IF END OF LINE
       LD     C,A            ; PUT CHAR IN PROPER REGISTER
       CALL   58DH           ; PRINT CHARACTER
       INC    HL             ; BUMP TO NEXT CHAR
       JR     START          ; LOOP TILL ALL CHARS PRINTED
NTRDY  LD     HL,NTRDM       ; HL = ADDR OF NOT READY MSG
       CALL   VIDEO*         ; PRINT MSG
DONE   .                      ; LINE PRINTED ON PRINTER
       .
       .
LIST   DEFM   'THIS IS A TST'
       DEFB   0DH             ; CR MAY BE REQUIRED TO START
                              PRINTER
       DEFB   0               ; END OF MSG FLAG
NTRDM  DEFM   'PRINTER NOT READY'
       DEFB   0               ; TERMINATE PRINTED MSG
       .                      .
       .                      .
```

CALL 05D1 — Get Printer Status

Returns the status of the line printer in the status register as zero if the printer is ready, and non-zero if not ready.

Other status bits returned are as shown:



The out of paper and busy bits are optional on some printers.

```
;
; MONITOR PRINTER STATUS ACCORDING TO STATUS BITS ABOVE
; AND PRINT APPROPRIATE ERROR MESSAGE.
;
       LD     BC,10          ; TIMER COUNT FOR PRINTER
START  CALL   05D1H          ; GET PRINTER STATUS
       JR     Z,OK           ; JUMP IF READY
       BIT    7,A            ; IS IT STILL PRINTING?
       JR     Z,TIME         ; YES IF NZ, GO TIME IT
       BIT    4,A            ; NOT PRINTING, IS IT SELECTED
       JR     Z,NS           ; ZERO IF NOT SELECTED
                             ; WE HAVE   HARDWARE PROBLEM
       BIT    5,A            ; UNIT IS SELECTED AND NOT BUSY
       JR     Z,NR           ; ZERO IF NOT READY
```

```
; UNIT IS SELECTED, READY, AND NOT BUSY. ASSUME OUT OF PAPER
;
OP   LD    HL,OPM        ; DISPLAY OUT OF PAPER MSG
       .
       .
     JP    WAIT          ; GO WAIT FOR OPERATOR REPLY
                         ; AND RETRY OR ABORT
NR   BIT   6,A           ; UNIT IS NOT READY, TEST FOR OUT
     JR    NZ,OP         ; OF PAPER ALSO. JMP IF OUT OF PAPER
       .
     LD    HL,NRM        ; DISPLAY NOT READY MSG
       .
       .
     JP    WAIT          ; GO WAIT FOR OPERATOR REPLY
                         ; AND RETRY OR ABORT
NS   LD    HL,NSM        ; GET DISPLAY NOT SELECTED MSG
       .
       .
     JP    WAIT          ; GO WAIT FOR OPERATOR REPLY
                         ; AND RETRY OR ABORT
TIME POP   BC            ; GET TIME COUNTER
     DEC   BC            ; COUNT 1 LOOP
     PUSH  BC            ; SAVE NEW VALUE
     LD    A,B           ; IF ITS GONE TO ZERO
     OR    C             ; WE HAVE TIMED OUT
     JR    NZ,START      ; LOOP TILL OP FINISHED OR TIME-OUT
       .
     LD    HL,TOM        ; DISPLAY TIMEOUT MSG
       .
       .
     JP    WAIT          ; GET OPERATOR REPLY AND RETRY
                         ; OR ABORT
       .
```

Cassette I/O

Cassette I/O is not memory mapped. Cassettes are addressed via port FF after selecting the proper unit, and I/O is done a bit at a time whereas all other devices do I/O on a byte basis (except for the RS-232-C)

Because of the bit-by-bit transfer of data, timing is extremely critical. When any of the following calls are used, the interrupt system should be disabled to guarantee that no interruptions will occur and therefore disturb the critical timing of the output.

CALL 0212 — Turn On Motor

Selects unit specified in A-register and starts motor. Units are numbered from one. All registers are used.

```
LD      A,1             ; CODE TO SELECT CASSETTE 1
CALL    0212H           ; SELECT UNIT 1, TURN ON MOTOR
      .
      .
      .
```

CALL 0284 — Write Leader

Writes a Level II leader on currently selected unit. The leader consists of 256 (decimal) binary zeros followed by a hex A5. Uses the B and A registers.

```
LD      A,1             ; CODE TO SELECT UNIT 1
CALL    212H            ; SELECT UNIT, TURN ON MOTOR
CALL    284H            ; WRITE HEADER
      .
      .
      .
```

CALL 0296 — Read Leader

Reads the currently selected unit until an end of leader (A5) is found. An asterisk is displayed in the upper right hand corner of the video display when the end is found. Uses the A-register.

```
LD      A,1             ; CODE FOR UNIT 1
CALL    0212H           ; SELECT UNIT 1,TURN ON MOTOR
CALL    0296H           ; READ HEADER. RTN WHEN A5 ENCOUNTERED
      .
      .
```

CALL 0235 — Read One Byte

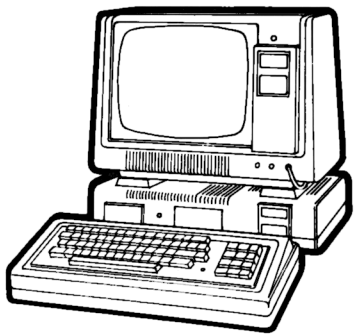
Reads one byte from the currently selected unit. The byte read is returned in the A-register. All other registers are preserved.

```
LD      A,1             ; UNIT TO SELECT
CALL    0212H           ; SELECT UNIT TURN ON MOTOR
CALL    0296H           ; SKIP OVER HEADER
CALL    0235H           ; READ FOLLOWING BYTE
CP      41H             ; TEST FOR OUR FILE NAME (A)
JR      Z,YES           ; JMP IF FILE A
      .
      .
      .
```

CALL 0264 — Write One Byte

Writes the byte in the A-register to the currently selected unit. Preserves all register.

```
LD      A,1             ; UNIT NO. MASK.
CALL    0212H           ; SELECT UNIT, START MOTOR
CALL    0284H           ; WRITE HEADER (256 ZEROS AND A5)
LD      A,41H           ; WRITE FILE NAME (OURS IS A)
CALL    0264H           ; WRITE A AFTER HEADER
      .
      .
      .
```



Conversion Routines

These entry points are used for converting binary values from one data type or mode to another, such as integer to floating point, and for conversions between ASCII and binary representation. These conversion routines assume the value to be converted is in WRA1 and that the mode flag (40AF) reflects the current data type. The result will be left in WRA1 and the mode flag will be updated.

Data Type Conversions

CALL 0A7F — Floating Point Integer

The contents of WRA1 are converted from single or double precision to integer. No rounding is performed. All registers are used.

```
;
; CONVERT SINGLE PRECISION VALUE TO INTEGER AND MOVE THE RESULT
;  TO IVAL
;
      LD      HL,4121H        ; ADDR OF LSB IN WRA1
      LD      DE,VALUE        ; ADDR OF LSB OF SP NO.
      LD      BC,4            ; NO OF BYTES TO MOVE
      LDIR                    ; MOVE VALUE TO WRA1
      LD      A,4             ; TYPE CODE FOR SP
      LD      (40AFH),A       ; SET TYPE TO SP
      CALL    0A7FH           ; CONVERT SP VALUE TO INTEGER
      LD      A,(4121H)       ; LSB OF INTEGER EQUIVALENT
      LD      (IVAL),A        ; SAVE IN INTEGER LOCATION
      LD      A,(4122H)       ; MSB OF INTEGER EQUIVALENT
      LD      (IVAL+1),A      ; SAVE IN INTEGER LOCATION
              .
              .
              .
VALUE DEFB    0EH             ; LSB OF 502.778 (SP)
      DEFB    86H             ; NLSB
      DEFB    00H             ; MSB
      DEFB    88H             ; EXPONENT
IVAL  DEFB    0               ; WILL HOLD INTEGER EQUIVALENT OF
      DEFB    0               ; SP 502.778
              .
              .
```

CALL 0AB1 — Integer To Single

The contents of WRA1 are converted from integer or double precision to single precision. All registers are used.

```
;
; CONVER INTEGER VALUE TO SINGLE PRECISION AND MOVE TO
; LOCAL AREA
;
      LD      A,59H
      LD      (4121H),A       ; LSB OF INTEGER 26457 (10)
      LD      A,67H
      LD      (4122H),A       ; MSB OF INTEGER 26457 (10)
      LD      A,2             ; TYPE CODE FOR INTEGER
      LD      (40AFH),A       ; SET TYPE TO INTEGER
      CALL    0AB1H           ; CONVERR INTEGER TO SP
      LD      HL,VALUE        ; ADDR. OF AREA FOR SP EQUIVALENT
      CALL    09CBH           ; MOVE SP VALUE FROM WRA1 TO VALUE
              .
              .
              .
VALUE DEFS    4               ; WILL HOLD 26457 IN SP FORMAT
              .
              .
              .
```

CALL 0ADB — Integer To Double

Contents of WRA1 are converted from integer or single precision to double presicion. All registers are used.

```
;
;
;
      LD      A,59H
      LD      (4121H),A       ; LSB OF 26457 (10)
      LD      A,67H
      LD      (4122H),A       ; MSB OF 26457 (10)
      LD      A,2             ; TYPE CODE FOR INTEGER
      LD      (40AFH),A       ; SET TYPE TO INTEGER
      CALL    0ADBH           ; CONVER INTEGER TO DP
      LD      DE,VALUE        ; NOW, MOVE DP VALUE
      LD      HL,4121H        ; FROM WRA1 TO LOCAL AREA
      LD      BC,8            ; NO. OF BYTES TO MOVE
      LDIR                    ; MOVE VALUE
              .
              .
              .
VALUE DEFS    8               ; HOLDS DP EQUIVALENT OF 26457
              .
              .
```

ASCII To Numeric Representation

The following entry points are used to convert between binary and ASCII. When converting from ASCII to binary the HL register pair is assumed to contain the address of the ASCII string. The result will be left in WRA1 or the DE register pair and the mode flag will be updated accordingly.

CALL 1E5A — ASCII To Integer

Converts the ASCII string pointed to by HL to its integer equivalent. The result is left in the DE register pair. Conversion will cease when the first non-numeric character is found.

```
;
;
;
      LD      HL,AVAL         ; HL = ADDR. OF ASCII NUMBER
      CALL    1E5AH           ; CONVERT IT TO BINARY
      LD      (BVAL),DE       ; SAVE BINARY VALUE
              .
              .
              .
AVAL  DEFM    '26457'         ; ASCII VALUE 26457
      DEFB    0               ; NON-NUMERIC STOP BYTE
BVAL  DEFW    2               ; HOLDS BINARY VALUE 26457
              .
              .
```

CALL 0E6C — ASCII To Binary

Converts the ASCII string pointed to by HL to binary. If the value is less than 2**16 and does not contain a decimal point or an E or D descriptor (exponent), the string will be converted to its integer equivalent. If the string contains a decimal point or an E, or D descriptor or if it exceeds 2**16 it will be converted to single or double precision. The binary value will be left in WRA1 and the mode flag will be to the proper value.

```
        LD      HL,AVAL         ; ASCII NUMBER
        CALL    0E6CH           ; CONVERT ASCII TO BINARY
                .
                .
                .
AVAL    DEFM    '26457'         ; ASCII VALUE TO BE CONVERTED
        DEFB    0               ; NON-NUMERIC STOP BYTE
                .
                .
```

CALL 0E65 — ASCII To Double

Converts the ASCII string pointed to by HL to its double precision equivalent. All registers are used. The result is left in WRA1.

```
        LD      HL,AVAL         ; ADDR OF ASCII VALUE TO CONVERT
        CALL    0E65H           ; CONVERT VALUE TO DP
        LD      DE,BVAL         ; THEN MOVE VALUE FROM
        LD      HL,411DH        ; WRA1 TO A LOCAL AREA
        LD      BC,8            ; NO. OF BYTES TO MOVE
        LDIR                    ; MOVE DP VALUE TO LOCAL AREA
                .
                .
                .
AVAL    DEFM    '26457'         ; ASCII VALUE TO BE CONVERTED
        DEFB    0               ; NONNUMERIC STOP BYTE
BVAL    DEFS    8               ; LOCAL AREA THAT HOLDS BINARY
                                  EQUIVALENT
                .
                .
```

Binary To ASCII Representation

The next set of entry points are used to convert from binary to ASCII.

CALL 0FAF — HL To ASCII

Converts the value in the HL register pair (assumed to be an integer) to ASCII and displays it at the current cursor position on the video. All registers are used.

```
        LD      HL,6488H        ; HL = 25784 (10)
        CALL    0FAFH           ; CONVERT TO ASCII AND DISPLAY
                .
                .
```

CALL 132F — Integer To ASCII

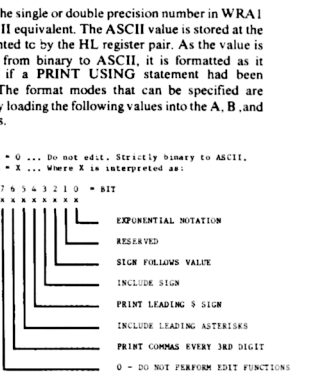
Converts the integer in WRA1 to ASCII and stores the ASCII string in the buffer pointed to by the HL register pair. On entry, both the B and C registers should contain a 5 to avoid any commas or decimal points in the ASCII string. All registers are preserved.

```
        LD      HL,500
        LD      (411BH),HL      ; 500 (10) TO WRA1
        LD      BC,905H         ; SUPRESS COMMAS OR DEC. PTS.
        LD      HL,BUFF         ; BUFFER ADDR FOR ASCII STRING
        CALL    132FH           ; CONVERT VALUE IN WRA1 TO ASCII
                .               ; AND STORE IN BUFF.
                .
BUFF    DEFS    5               ; BUFFER FOR ASCII VALUE
                .
                .
```

CALL 0FBE — Floating to ASCII

Converts the single or double precision number in WRA1 to its ASCII equivalent. The ASCII value is stored at the buffer pointed tc by the HL register pair. As the value is converted from binary to ASCII, it is formatted as it would be if a PRINT USING statement had been invoked. The format modes that can be specified are selected by loading the following values into the A, B ,and C registers.

```
REGISTER  A = 0 ... Do not edit. Strictly binary to ASCII.
REGISTER  A = X ... Where X is interpreted as:

                7 6 5 4 3 2 1 0  = BIT
                X X X X X X X X
```




```
REGISTER B = The number of digits to the left of the
             decimal point.
REGISTER C = The number of digits after the decimal point
```

```
        LD      HL,AVAL1        ; ASCII VALUE TO CONVERT
        CALL    0E6CH           ; CONVERT ASCII TO BINARY
        LD      HL,AVAL2        ; BUFFER ADDR. FOR CONVERTED VALUE
        LD      A,0             ; SIGNAL NO EDITING
        CALL    0FBE2H          ; CONVERT SP VALUE BACK TO ASCII
                .
                .
                .
AVAL1   DEFM    '1103.25'       ; ORGINAL ASCII VALUE
        DEFB    0               ; NON-NUMERIC STOP BYTE
AVAL2   DEFS    7               ; WILL HOLD RECONVERTED VALUE
                .
                .
```

Arithmetic Routines

These subroutines perform arithmetic operations between two operands of the same type. They assume that the operands are loaded into the correct hardware or Working Register Area, and that the data type or mode is set to the correct value. Some of these routines may require the Divide Support Routine (See Chapter 1 for details.)

Integer Routines

The following routines perform arithmetic operations between integer values in the DE and HL register pairs. The orginal contents of DE is always preserved and the result of the operations is always left in the HL register pair.

CALL 0BD2 Integer Add

Adds the integer value in DE to the integer in HL. The sum is left in HL and the orginal contents of DE are preserved. If overflow occurs (sum exceeds 2**15), both values are converted to single precision and then added. The result would be left in WRA1 and the mode flag would be updated.

```
        LD      A,2             ; TYPE CODE FOR INTEGER
        LD      (40AFH),A       ; SET TYPE TO INTEGER
        LD      HL,(VAL1)       ; LOAD FIRST VALUE
        LD      DE,(VAL2)       ; LOAD SECOND VALUE
        CALL    0BD2H           ; ADD SO THAT HL = HL + DE
        LD      A,(40AFH)       ; TEST FOR OVERFLOW
        CP      2               ; IF TYPE IS NOT INTEGER
        JR      NZ,...          ; NZ IF SUM IS SINGLE PRECISION
                                ; ELSE SUM IS INTEGER
          .
          .
          .
VAL1    DEFW    25
VAL2    DEFW    20
          .
          .
          .
```

CALL 0BC7 Integer Subtraction

Subtracts the value in DE from the value in HL. The difference is left in the HL register pair. DE is preserved. In the event of underflow, both values are converted to single precision and the subtraction is repeated. The result is left in WRA1 and the mode flag is updated accordingly.

```
        LD      A,2             ; TYPE CODE FOR INTEGER
        LD      (40AFH),A       ; SET TYPE TO INTEGER
        LD      HL,(VAL1)       ; VALUE 1
        LD      DE,(VAL2)       ; VALUE 2
        CALL    0BC7H           ; SUBTRACT DE FROM HL
        LD      A,(40AFH)       ; GET MODE FLAG
        CP      2               ; TEST FOR UNDERFLOW
        JR      NZ,...          ; NZ IF UNDERFLOW
          .
          .
          .
VAL1    DEFW    25
VAL2    DEFW    20
          .
          .
          .
```

CALL 0BF2 Integer Multiplication

Multiplies HL by DE. The product is left in HL and DE is preserved. If overflow occurs, both values are converted to single precision and the operation is restarted. The product would be left in WRA1.

```
        LD      A,2             ; TYPE CODE FOR INTEGER
        LD      (40AFH),A       ; SET TYPE TO INTEGER
        LD      HL,(VAL1)       ; LOAD FIRST VALUE
        LD      DE,(VAL2)       ; LOAD SECOND VALUE
        CALL    0BF2H           ; HL = HL * DE
        LD      A,(40AFH)       ; GET MODE FLAG
        CP      2               ; TEST FOR OVERFLOW
        JR      NZ,...          ; NO IF VALUE HAS OVERFLOWED
          .
          .
          .
VAL1    DEFW    25
VAL2    DEFW    20
          .
          .
          .
```

CALL 2490 Integer Division

Divides DE by HL. Both values are converted to single precision before the division is started. The quotient is left in WRA1; the mode flag is updated. The orginal contents of the DE and HL register sets are lost.

```
        LD      DE,(VAL1)       ; LOAD VALUE 1
        LD      HL,(VAL2)       ; LOAD VALUE 2
        CALL    2490H           ; DIVIDE DE BY HL. QUOTIENT TO WRA1
          .
          .
          .
VAL1    DEFW    50
VAL2    DEFW    2
```

CALL 0A39 Integer Comparison

Algebraically compares two integer values in DE and HL. The contents of DE and HL are left intact. The result of the comparison is left in the A register and status register as:

OPERATION	A REGISTER
DE > HL	A = -1
DE < HL	A = +1
DE = HL	A = 0

```
;
;
;
        LD      DE,(VAL1)       ; DE AND HL ARE VALUES
        LD      HL,(VAL2)       ; TO BE COMPARED
        CALL    0A39H           ; COMPARE DE TO HL
        JR      Z,...           ; Z IF DE = HL
        JP      P,...           ; POSITIVE IF DE < HL
          .
          .
```

Single Precision Routines

The next set of entry points are used for single precision operations. These routines expect one argument in the BC/DE registers and the other argument in WRA1.

CALL 0716 Single Precision Add

Add the single precision value in (BC/DE) to the single precision value in WRA1. The sum is left in WRA1

```
        LD      HL,VAL1         ; ADDR. OF ONE SP VALUE
        CALL    9B1H            ; MOVE IT TO WRA1
        LD      HL,VAL2         ; ADDR. OF 2ND SP VALUE
        CALL    9C2H            ; LOAD IT INTO BC/DE REGISTER
        CALL    716H            ; ADD VALUE 1 TO VALUE 2
                .               ; SUM IN WRA1
                .
                .
VAL1    DEFS    4               ; HOLDS A SP VALUE
VAL2    DEFS    4               ; HOLDS A SP VALUE
                .
                .
```

CALL 0713 Single Precision Subtract

Subtracts the single precision value in (BC/DE) from the single precision value in WRA1. The difference is left in WRA1.

```
        LD      HL,VAL1         ; ADDR OF ONE SP. VALUE
        CALL    9B1H            ; MOVE IT TO WRA1
        LD      HL,VAL2         ; ADDR OF 2ND SP VALUE
        CALL    9C2H            ; LOAD IT INTO BC/DE
        CALL    713H            ; SUBTRACT DE FROM WRA1
                .               ; DIFFERENCE LEFT IN WRA1
                .
                .
VAL1    DEFS    4               ; HOLDS A SP VALUE
VAL2    DEFS    4               ; HOLDS A SP VALUE
                .
                .
```

CALL 0847 Single Precision Multiply

Multiplies the current value in WRA1 by the value in (BC/DE). the product is left in WRA1.

```
        LD      HL,VAL1         ; ADDR OF ONE SP VALUE
        CALL    9B1H            ; MOVE IT TO WRA1
        LD      HL,VAL2         ; ADDR OF 2ND SP VALUE
        CALL    9C2H            ; LOAD 2ND VALUE INTO BC/DE
        CALL    847H            ; MULTIPLY
                .               ; PRODUCT LEFT IN WRA1
                .
                .
VAL1    DEFS    4               ; HOLDS A SP VALUE
VAL2    DEFS    4               ; HOLDS A SP VALUE
                .
                .
```

CALL 2490 Single Precision Divide

Divides the single precision value in (BC/DE) by the single precision value in WRA1. The quotient is left in WRA1.

```
        LD      HL,VAL1         ; ADDR OF DIVISOR
        CALL    9B1H            ; MOVE IT TO WRA1
        LD      HL,VAL2         ; ADDR. OF DIVIDEND
        CALL    9C2H            ; LOAD BC/DE WITH DIVIDEND
        CALL    2490H           ; DIVIDE BC/DE BY WRA1
                .               ; QUOTIENT IN WRA1
                .
                .
VAL1    DEFS    4               ; HOLDS DIVISOR
VAL2    DEFS    4               ; HOLDS DIVIDEND
                .
                .
```

CALL 0A0C Single Precision Comparison

Algebraically compares the single precision value in (BC/DE) to the single precision value WRA1. The result of the comparison is returned in the A and status as:

OPERATION	A REGISTER
(BC/DE) > WRA1	A = -1
(BC/DE) < WRA1	A = +1
(BC/DE) = WRA1	A = 0

```
        LD      HL,VAL1         ; ADDR OF ONE VALUE TO BE COMPARED
        CALL    9B1H            ; MOVE IT TO WRA1
        LD      HL,VAL2         ; ADDR OF 2ND VALUE TO COMPARE
        CALL    9C2H            ; LOAD 2ND VALUE INTO BC/DE
        CALL    0A0CH           ; COMPARE BC/DE TO WRA1
        JR      Z,...           ; ZERO IF (BC/DE) = WRA1
        JP      P,...           ; POSITIVE IF (BC/DE) < WRA1
                .
                .
                .
VAL1    DEFS    4               ; HOLDS A SP VALUE
VAL2    DEFS    4               ; HOLDS A SP VALUE
                .
                .
```

Double Precision Routines

The next set of routines perform operations between two double precision operands. One operand is assumed to be in WRA1 while the other is assumed to be in WRA2 (4127-412E). The result is always left in WRA1.

CALL 0C77 Double Precision Add

Adds the double precision value in WRA2 to the value in WRA1. Sum is left in WRA1.

```
        LD      A,8             ; TYPE CODE FOR DP
        LD      (40AFH),A       ; SET TYPE TO DP
        LD      DE,VAL1         ; ADDR OF 1ST DP VALUE
        LD      HL,411DH        ; ADDR OF WRA1
        CALL    9D3H            ; MOVE 1ST DP VALUE TO WRA1
        LD      DE,VAL2         ; ADDR OF 2ND DP VALUE
        LD      HL,4127H        ; ADDR OF WRA2
        CALL    9D3H            ; MOVE 2ND VALUE TO WRA2
        CALL    0C77H           ; ADD WRA2 TO WRA1. SUM IN WRA1
                .
                .
                .
VAL1    DEFS    8               ; HOLDS A DP VALUE
VAL2    DEFS    8               ; HOLDS A DP VALUE
                .
                .
```

CALL 0C70 Double Precision Subtraction

Subtracts the double precision value in WRA2 from the value in WRA1. The difference is left in WRA1.

```
        LD      A,8             ; TYPE CODE FOR DP
        LD      (40AFH),A       ; SET TYPE TO DP
        LD      DE,VAL1         ; ADDR OF 1ST DP VALUE
        LD      HL,411DH        ; ADDR OF WRA1
        CALL    9D3H            ; MOVE 1ST DP VALUE TO WRA1
        LD      DE,VAL2         ; ADDR OF 2ND DP VALUE
        LD      HL,4127H        ; ADDR OF WRA2
        CALL    9D3H            ; MOVE 2ND VALUE TO WRA2
        CALL    0C70H           ; SUBTRACT WRA2 FROM WRA1
                .               ; DIFFERENCE IN WRA1
                .
                .
VAL1    DEFS    8               ; HOLDS A DP VALUE
VAL2    DEFS    8               ; HOLDS A DP VALUE
```

CALL 0DA1 Double Precision Multiply

Multiplies the double precision value in WRA1 by the value in WRA2. The product is left in WRA1.

```
        LD      A,8             ; TYPE CODE FOR DP
        LD      (40AFH),A       ; SET TYPE TO DP
        LD      DE,VAL1         ; ADDR OF 1ST DP VALUE
        LD      HL,411DH        ; ADDR OF WRA1
        CALL    9D3H            ; MOVE 1ST DP VALUE TO WRA1
        LD      DE,VAL2         ; ADDR OF 2ND DP VALUE
        LD      HL,4127H        ; ADDR OF WRA2
        CALL    9D3H            ; MOVE 2ND VALUE TO WRA2
        CALL    0DA1H           ; MULTIPLY WRA1 BY WRA2
                .               ; PRODUCT IN WRA1
                .
                .
VAL1    DEFS    8               ; HOLDS A DP VALUE
VAL2    DEFS    8               ; HOLDS A DP VALUE
```

CALL 0DE5 Double Precision Divide

Divides the double precision value in WRA1 by the value in WRA2. The quotient is left in WRA1.

```
        LD      A,8             ; TYPE CODE FOR DP
        LD      (40AFH),A       ; SET TYPE TO DP
        LD      DE,VAL1         ; ADDR OF 1ST DP VALUE
        LD      HL,411DH        ; ADDR OF WRA1
        CALL    9D3H            ; MOVE 1ST DP VALUE TO WRA1
        LD      DE,VAL2         ; ADDR OF 2ND DP VALUE
        LD      HL,4127H        ; ADDR OF WRA2
        CALL    9D3H            ; MOVE 2ND VALUE TO WRA2
        CALL    0DE5H           ; DIVIDE WRA1 BY WRA2
                .               ; QUOTIENT LEFT IN WRA1
                .
                .
VAL1    DEFS    8               ; HOLDS A DP VALUE
VAL2    DEFS    8               ; HOLDS A DP VALUE
```

CALL 0A78 Double Precision Compare

Compares the double precision value in WRA1 to the value in WRA2. Both register areas are left intact. The result of the comparison is left in the A and status registers as:

OPERATION	A REGISTER
WRA1 > WRA2	A = -1
WRA1 < WRA2	A = +1
WRA1 = WRA2	A = 0

```
        LD      A,8             ; TYPE CODE FOR DP
        LD      (40AFH),A       ; SET TYPE FLAG TO DP
        LD      DE,VAL1         ; ADDR OF 1ST DP VALUE
        LD      HL,411DH        ; ADDR OF WRA1
        CALL    9D3H            ; MOVE 1ST VALUE TO WRA1
        LD      DE,VAL2         ; ADDR OF 2ND DP VALUE
        LD      HL,4127H        ; ADDR OF WRA2
        CALL    9D3H            ; MOVE 2ND VALUE TO WRA2
        CALL    0A78H           ; COMPARE WRA1 TO WRA2
        JR      Z,...           ; ZERO IF THEY ARE EQUAL
        JP      P,...           ; POSITIVE IF WRA1 < WRA2
                .
                .
```

Math Routines

All of the following subroutines assume that location 40AF contains a code indicating the data type or mode of the variable e.g., integer, single precision, or double precision, and that the variable itself is in Working Register Area 1 (WRA1). Also, the floating point Division Support Routine must be loaded at 4080.

CALL 0977 Absolute Value ABS (N)

Converts the value in Working Register Area 1 (WRA1) to its positive equivalent. The result is left in WRA1. If a negative integer greater than 2**15 is encountered, it is converted to a single precision value. The data type or mode flag (40AF) will be updated to reflect any change in mode.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,VAL1         ; ADDR OF SP VALUE TO ABS
        CALL    09B1H           ; MOVE SP VALUE TO WRA1
        CALL    0977H           ; FIND ABS VALUE
                .
                .
                .
VAL1    DEFB    58H             ; SP 81.6022(10)
        DEFB    36H
        DEFB    23H
        DEFB    87H
                .
                .
```

CALL 0B37 Return Integer INT (N)

Returns the integer portion of a floating point number. If the value is positive, the integer portion is returned. If the value is negative with a fractional part, it is rounded up before truncation. The integer portion is left in WRA1. The mode flag is updated.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SINGLE PREC.
        LD      HL,VAL1         ; ADDR OF SP VALUE
        CALL    09B1H           ; MOVE SP VALUE TO WRA1
        CALL    0B37H           ; ISOLATE INTEGER PART OF SP VALUE
        LD      DE,4121H        ; ADDR OF WRA1 (INTEGER PART OF SP
VALUE
        LD      HL,VAL2         ; LOCAL ADDR FOR INTEGERIZED VALUE
        CALL    09D3H           ; MOVE INTEGERIZED SP VALUE TO LOCAL
AREA
                .
                .
                .
VAL1    DEFB    0E0H            ; SP -41.3418
        DEFB    05DH
        DEFB    0A3H
        DEFB    086H
VAL2    DEFS    4               ; HOLDS INTEGER PORTION OF
                                ;       -41.3418
                .
                .
                .
```

CALL 15BD — Arctangent ATN (N)

Returns the angle in radians, for the floating point tangent value in WRA1. The angle will be left as a single precision value in WRA1.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),        ; SET TYPE TO SP
        LD      HL,TAN          ; ADDR OF VALUE FOR TANGENT
        CALL    09B1H           ; MOVE TAN TO WRA1
        CALL    15BDH           ; FIND ANGLE IN RADS
        LD      HL,ANGL         ; ADDR OF LOCAL STORAGE FOR ANGLE
        LD      DE,4121H        ; ADDR OF WRA1
        CALL    09D3H           ; MOVE ANGLE FROM WRA1 TO LOCAL AREA
                .
                .
                .
TAN     DEFB    9AH             ; TANGENT OF 30 DEG.
        DEFB    0C4H
        DEFB    13H
        DEFB    80H             ; EXPONENT
ANGL    DEFS    4               ; WILL HOLD  30 DEG. IN RADS (.5235)
```

CALL 1541 — Cosine COS (N)

Computes the cosine for an angle given in radians. The angle must be a floating point value; the cosine will be returned in WRA1 as a floating point value.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,ANGL         ; ADDR OF ANGLE VALUE
        CALL    09B1H           ; MOVE ANGLE TO WRA1
        CALL    1541H           ; COMPUTE COSINE
        LD      HL,CANGL        ; LOCAL ADDR FOR COSINE
        LD      DE,4121H        ; ADDR OF WRA1
        CALL    09D3H           ; MOVE COSINE FROM WRA1 TO LOCAL AREA
                .
                .
                .
ANGL    DEFB    18H             ; 30 DEG. IN RADS. (.5235)
        DEFB    04H
        DEFB    06H
        DEFB    80H             ; EXPONENT
CANGL   DEFS    4               ; WILL HOLD COSINE OF 30 DEG.
                .
                .
                .
```

CALL 1439 — Raise Natural Base EXP (N)

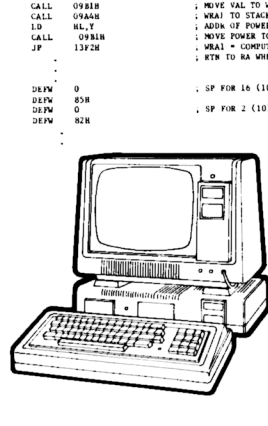
Raises E (natural base) to the value in WRA1 which must be a single precision value. The result will be returned in WRA1 as a single precision number.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,EXP          ; ADDR OF EXPONENT
        CALL    09B1H           ; MOVE EXPONENT TO WRA1
        CALL    1439H           ; FIND E ** 1.5708
        LD      DE,4121H        ; ADDR OF WRA1
        LD      HL,POW          ; ADDR OF LOCAL STORAGE
        CALL    09D3H           ; MOVE POWER TO LOCAL AREA
                .
                .
                .
EXP     DEFB    0DBH            ; SP  1.5708(10)
        DEFB    0FH
        DEFB    049H
        DEFB    081H
POW     DEFS    4               ; HOLDS E**1.5708
                .
                .
                .
```

CALL 13F2 — Raise X to the Y Power X**Y

Raises the single precision value which has been saved on the STACK to the power specified in WRA1. The result will be returned in WRA1.

```
;
; COMPUTE 16**2
;
        LD      BC,RETADD       ; RTN ADDR FOLLOWING
        PUSH    BC              ; RAISING X TO Y
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP FOR X
        LD      HL,X            ; ADDR OF VAL TO BE RAISED
        CALL    09B1H           ; MOVE VAL TO WRA1
        CALL    09AAH           ; WRA1 TO STACK
        LD      HL,Y            ; ADDR OF POWER
        CALL    09B1H           ; MOVE POWER TO WRA1
        JP      13F2H           ; WRA1 = COMPUTE X**Y
RA              .               ; RTN TO RA WHEN DONE
                .
                .
X       DEFW    0               ; SP FOR 16 (10)
        DEFW    85H
Y       DEFW    0               ; SP FOR 2 (10)
        DEFW    82H
                .
                .
```



CALL 0809 — Natural Log LOG (N)

Computes the natural log (base E) of the single precision value in WRA1. The result is returned as a single precision value in WRA1.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,POW          ; ADDR OF POWER
        CALL    0981H           ; MOVE POWER TO WRA1
        CALL    0809H           ; FIND NAT.LOG. OF POWER
        LD      DE,4121H        ; ADDR OF WRA1
        LD      HL,NLOG         ; ADDR OF LOCAL STORAGE AREA
        CALL    0983H           ; MOVE LOG FROM WRA1 TO LOCAL AREA
                .
                .
                .
POW     DEFB    00              ; FLOATING POINT 3 (LSB)
        DEFB    00
        DEFB    04H
        DEFB    82HH            ; EXPONENT FOR 3.0
NLOG    DEFS    4               ; WILL HOLD NAT. LOG OF 3
                .
                .
                .
```

CALL 0B26 — Floating To Integer FIX (N)

Unconditionally truncates the fractional part of a floating point number in WRA1. The result is stored in WRA1 and the type flag is set to integer.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,FLPT         ; ADDR OF FLOATING POINT VALUE
        CALL    0981H           ; MOVE FLT.PT. VALUE TO WRA1
        CALL    0B26H           ; TRUNCATE AND CONVERT TO INTEGER
        LD      HL,(4121H)      ; LOAD INTEGER PORTION FROM WRA1
        LD      (INTG),HL       ; AND STORE IN LOCAL AREA
                .
                .
                .
FLPT    DEFB    0BAH            ; SP 39.7107(10)
        DEFB    0DFH
        DEFB    01EH
        DEFB    086H
INTG    DEFS    2               ; HOLDS INTEGER PORTION OF
                                ;       39.7107
                .
                .
                .
```

CALL 01D3 — Reseed Random Seed RANDOM

Reseeds the random number seed (location 40AB) with the current contents of the refresh register.

```
        CALL    01D3H           ; RESEED RANDOM NUMBER SEED
                .
                .
                .
```

CALL 14C9 — Random Number RND (N)

Generates a random number between 0 and 1, or 1 and n depending on the parameter passed in WRA1. The random value is returned in WRA1 as an integer with the mode flag set. The parameter passed will determine the range of the random number returned. A parameter of 0 will return an interger between 0 and 1. A parameter greater than 0 will have any fraction portion truncated and will cause a value between 1 and the integer portion of the parameter to be returned.

```
        LD      A,2             ; TYPE CODE FOR INTEGER
        LD      (40AFH),A       ; SET TYPE TO INTEGER
        LD      A,50
        LD      (4121H),A       ; PUT AN INTEGER 50 INTO WRA1
        CALL    14C9H           ; GET A RANDOM NO. BETWEEN 1 AND 50
        LD      HL,(4121H)      ; LOAD RANDOM NO. INTO HL
        LD      (RVAL),HL       ; AND MOVE IT TO LOCAL AREA
                .
                .
                .
RVAL    DEFW    0               ; HOLDS RANDOM NUMBER (INTEGER)
                .
                .
                .
```

CALL 1547 — Sine SIN (N)

Returns the sine as a single precision value in WRA1. The sine must be given in radians in WRA1.

```
        LD      A,4             ; TYPE CODE FOR INTEGER
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,ANGL         ; ADDR. OF ANGLE IN RADIANS
        CALL    0981H           ; MOVE ANGLE TO WRA1
        CALL    1547H           ; COMPUTE SINE OF ANGLE
        LD      DE,4121H        ; ADDR OF SINE IN WRA1
        LD      HL,SANGL        ; ADDR OF LOCAL AREA FOR SIN
        CALL    0983H           ; MOVE SINE TO LOCAL AREA
                .
                .
                .
ANGL    DEFB    18H             ; 30 DEGS. IN RADS. (.5235)
        DEFB    04H
        DEFB    06H
        DEFB    80H             ; EXPONENT
SANGL   DEFS    4               ; WILL HOLD SINE OF 30 DEG.
                .
                .
                .
```

CALL 13E7 — Square Root SQR (N)

Computes the square root of any value in WRA1. The root is left in WRA1 as a single precision value.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,VAL1         ; VALUE TO TAKE ROOT OF
        CALL    0981H           ; MOVE BE IN WRA1
        CALL    13E7H           ; TAKE ROOT OF VALUE
        LD      DE,4121H        ; ADDR OF ROOT IN WRA1
        LD      HL,ROOT         ; ADDR OF LOCAL AREA
        CALL    0983H           ; MOVE ROOT TO LOCAL AREA
                .
                .
                .
VAL1    DEFB    00H             ; SP 4
        DEFB    00H
        DEFB    00H
        DEFB    83H             ; EXPONENT OF FLOATING POINT 4
ROOT    DEFS    4               ; HOLDS ROOT OF 4
                .
                .
                .
```

CALL 15A8 — Tangent TAN (N)

Computes the tangent of an angle in radians. The angle must be specified as a single precision value in WRA1. The tangent will be left in WRA1.

```
        LD      A,4             ; TYPE CODE FOR SP
        LD      (40AFH),A       ; SET TYPE TO SP
        LD      HL,ANGL         ; ADDR OF ANGLE IN RADIANS
        CALL    0951H           ; MOVE ANGLE TO WRA1
        CALL    15A8H           ; FIND TAN OF ANGLE
        LD      DE,4121H        ; ADDR OF WRA1
        LD      HL,TANGL        ; ADDR OF LOCAL STORAGE FOR TAN
        CALL    09D3H           ; MOVE TAN FROM WRA1 TO LOCAL AREA
            .
            .
            .
ANGL    DEFB    18H             ; VALUE FOR 30 DEG IN RADS
        DEFB    04H             ; (.5235)
        DEFB    06H
        DEFB    80H             ; EXPONENT
TANGL   DEFS    4               ; WILL HOLD TANGENT OF 30 DEG.
            .
            .
            .
```

Function Derivation

The LEVEL II system supports sixteen arithmetic functions. Seven of those may be called math functions. They are the sine, cosine, arctangent, tangent, square root, exponential (base e) and natural log. Three of these functions are computed from the identities:

$$\cos\theta = \sin\theta + \frac{\pi}{2}$$

$$\tan = \theta \frac{\sin\theta}{\cos\theta}$$

$$\sqrt{x} = e^{\frac{\ln x}{2}}$$

An implied math function exists which computes powers using the identity:

$$x^y = e^{y \ln x}$$

Embeded in LEVEL II are routines for the sine, exponential, natural log and arctangent. The other math functions derive their values using the aforementioned identities.

SINE
The sine routine is based on five terms of the approximation:

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!}$$

Where θ is in radians. The actual approximation used is:

$$\sin\beta(2\pi) = 2\pi\beta - \frac{(2\pi)^3}{3!}\beta^3 + \frac{(2\pi)^5}{5!}\beta^5 - \frac{(2\pi)^7}{7!}\beta^7 + \frac{(2\pi)^9}{9!}\beta^9$$

Where β is a ratio which when multiplied by 2π gives the angle in radians. If x is the angle in degrees, then β is also used to determine the sign of the result according to the following rules:

$$\beta = \beta \frac{X}{360^\circ} \quad \text{if } 0^\circ \leq X \leq 90^\circ$$

$$\beta = \beta \frac{180^\circ}{360^\circ} - \frac{X}{360^\circ} \quad \text{if } 90^\circ < X \leq 180^\circ$$

$$\beta = \beta \frac{180^\circ}{360^\circ} - \frac{X}{360^\circ} \quad \text{if } 180^\circ < X \leq 270^\circ$$

$$\beta = \frac{X}{360^\circ} - \frac{360^\circ}{360^\circ} \quad \text{if } 270^\circ < X \leq 360^\circ$$

The coefficients used with the sine series are correct to four decimal places, the maximum error for sine x is $\leq .000003$, thus all values for sine x would be correct to five places.

EXPONENTIATION
The exponentiation routine computes e^x for all values of x where:

$$-88 \leq x \leq 88$$

The approximation used for this function is derived from the following:

Since $e^x = 2^{x \log_2 e}$

Consider $2^{[\![x \log_2 e]\!]+1}$ where $[\![$ represents the greatest integer function.

Now $e^x = e^{-t}\left[2^{[\![x \log_2 e]\!]+1}\right]$

if $t = -x + [\![x \log_2 e]\!] \ln 2 + \ln 2$

Since $x = \ln e^x = \ln e^{-t}\left[2^{[\![x \log_2 e]\!]+1}\right]$

$$x = -t + \left\{[\![x \log_2 e]\!] + 1\right\} \ln 2$$

Now $t = -x + \left\{[\![x \log_2 e]\!] + 1\right\} \ln 2$

and so $0 < t < \ln 2$

and because

$$e^{-t} = 1 - t + \frac{t^2}{2!} - \frac{t^3}{3!} + \frac{t^4}{4!} - \frac{t^5}{5!} + \frac{t^6}{6!} - \frac{t^7}{7!}$$

The following series is used to approximate e^{-t}.

$$e^{-t} = 1 - t + .5t^2 - .166t^3 + .0416t^4 - .0083t^5 + .00132987t^6 - .0001413t^7$$

Then e^x is found by multiplying the approximate value of e^{-t} by $2^{[\![x \log_2 e]\!]+1}$ giving a result that is usually correct to at least five significant digits or five decimal places whichever is larger.

ARCTANGENT
The arctangent routine uses the approximation:

$$\arctan X = -\frac{X^1}{1} + \frac{X^3}{3} - \frac{X^5}{5} + \frac{X^9}{9} - \frac{X^{11}}{11} + \frac{X^{13}}{13} - \frac{X^{15}}{15} + \frac{X^{17}}{17}$$

If $x < 0$, the series is computed using the absolute value of x and the sign of the result is inverted. If $x > 1$ the series is computed using the value $1/x$ and the result is returned as $\pi/2 - \arctan 1/x$. For values where $0 < x < 1$, the series is computed using the orignal value of x. The coefficients used in the computer series are different from those in the approximating series starting with the seventh term, and the accuracy on the fifth and sixth coefficients is marginal as well. The actual series used is:

$$\arctan X = X - .33331X^3 + .199936X^5 - .142089X^7 + .106563X^9 - .0752896X^{11} + .0429096X^{13} - .01616157X^{15} + .00286623X^{17}$$

The maximum error using this approximation is .026.

NATURAL LOG
The natural log routine is based on three terms from the series:

$$\ln x = 2\left[\left(\frac{x-1}{x+1}\right) + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 \cdots\right]$$

This series is convergent for values of $x < 1$ so x must be redefined as:

$$x = X 2^n$$

Where n is an integer scaling factor and

$$1/2 \leq X < 1$$

Through algebra, not shown here, the x term can be replaced by $\frac{X}{\ln 2}$ giving:

$$\ln x = \frac{2}{\ln 2}\left[\left(\frac{X-1}{\frac{\ln 2}{2}(X+1)}\right) + \frac{1}{3}\left(\frac{X-1}{\frac{\ln 2}{2}(X+1)}\right)^3 + \frac{1}{5}\left(\frac{X-1}{\frac{\ln 2}{2}(X+1)}\right)^5 + \cdots\right]$$

Since $\ln x = \ln X\, 2^n = \ln X + n \ln 2$ and since

$$\frac{\ln \frac{X}{\ln 2}}{\ln 2}$$

from the series it follows that

$$\ln x = \left(\frac{\ln \frac{x}{\ln 2}}{\ln 2} - .5 + n\right)\ln 2$$

In this function $\ln 2$ has been approximated as .707092 and

$$\frac{\ln(\ln 2)}{\ln 2} \text{ as } -.5$$

If x is reasonable where $0 < x$ then $\ln x$ should be accurate to four significant digits. If x is extremely close to zero or very large, this will not be the case.

System Functions are ROM entry points that can be entered at This means that on a disk based system, for example, an assembly language program which CALLS these entry points could be executed immediately after IPL before executing the BASIC utility program first. These entry points are different from the BASIC Functions because they do not require the Communica-- tions Region (CR) to be initialized in order to operate correctly. A Level II system without disks always has an initialized CR because of its IPL processing.

Some of the routines mentioned here do use the Communications Region , but none of them require any particular locations to be initialized. The System Error routine however, which may be called in the event of an error detected by these routines, will assume some words contain meaningfull data, and will return control to the BASIC Interpreter Input Phase.

RST 08 — Compare Symbol

Compares the symbol in the input string pointed to by HL register to the value in the location following the RST 08 call. If there is a match, control is returned to address of the RST 08 instruction 2 with the next symbol in the A-register and HL incremented by one. If the two characters do not match, a syntax error message is given and control returns to the Input Phase.

```
;
; TEST THE STRING POINTED TO BY HL TO SEE IF IT
; CONTAINS THE STRING 'A=B=C'.
;
      RST   08        ; TEST FOR A
      DEFB  41H       ; HEX VALUE FOR A
      RST   08        ; FOUND A, NOW TEST FOR =
      DEFB  3DH       ; HEX VALUE FOR =
      RST   08        ; FOUND =, NOW TEST FOR B
      DEFB  42B       ; HEX VALUE FOR B
      RST   08        ; FOUND B, TEST FOR =
      DEFB   3DH      ; HEX VALUE FOR =
      RST   08        ; FOUND =, TEST FOR C
      DEFB  43H       ; HEX VALUE FOR C
       .              ; FOUND STRING A=B=C
       .
       .
```

RST 10 — Examine Next Symbol

Loads the next character from the string pointed to by the HL register set into the A-register and clears the CARRY flag if it is alphabetic, or sets it if is alphanumeric. Blanks and control codes **09** and **0B** are ignored causing the following character to be loaded and tested. The HL register will be incremented before loading any character therfore on the first call the HL register should contain the string address minus one. The string must be terminated by a byte of zeros.

```
;
; THE CURRENT STRING POINTED TO BY HL IS ASSUMED
; TO BE PART OF AN ASSIGNMENT STATEMENT CONTAINING
; AN OPTIONAL SIGN FOLLOWED BY A CONSTANT OR A
; VARIABLE NAME. MAKE THE NECESSARY TESTS TO DETERMINE
; IF A CONSTANT OR A VARIABLE IS USED.
;
       RST    08       ; TEST FOR '='
       DEFB   3DH      ; HEX VAUE FOR =
NEXT   RST    10H      ; GET SYMBOL FOLLOWING =
       JR     NC,VAR   ; NC IF VARIABLE NAME
       CALL   1E5AH    ; GET VALUE OF CONSTANT
       JR     SKIP     ; JOIN COMMON CODE
VAR    CP     2BH      ; NOT NUMERIC, TEST FOR +,-,
                       ; OR ALPHA
       JR     Z,NEXT   ; SKIP + SIGNS
       CP     2DH      ; NOT A +, TEST FOR A -
       JR     Z,NEXT   ; SKIP - SIGNS
       CALL   260DH    ; ASSUME IT'S A GOOD ALPHA AND
                       ; SEARCH FOR A VARIABLE NAME
         .             ; (SEE SECTION 2.6 FOR A
         .             ; DESCRIPTION OF 260D)
         .
SKIP     .
         .
         .
```

RST 18 — Compare DE:HL

Numerically compares DE and HL. Will not work for signed integers (except positive ones). Uses the A-register only. The result of the comparison is returned in the status register as:

```
CARRY SET   - HL < DE
NO CARRY    - HL > DE
NZ          - UNEQUAL
Z           - EQUAL

;
; THIS EXAMPLE TESTS THE MAGNITUDE OF THE VALUE
; FOLLOWING THE = IN THE STRING POINTED TO BY HL
; TO MAKE SURE IT FALLS BETWEEN 100 AND 500
;
      RST    08       ; TEST FOR =
      DC     3DH      ;  HEX VALUE FOR =
      RST    10H      ; FOUND =, TEST NEXT CHAR
      JR     NC,ERR   ; NC IF NOT NUMERIC
      CALL   1E5AH    ; GET BINARY VALUE
      LD     HL,500   ; UPPER LIMIT VALUE
      RST    18H      ; COMPARE VALUE TO UPPER LIMIT
      JR     C,ERR    ; CARRY IF VALUE > 500
      LD     HL,100   ; LOWER LIMIT VALUE
      RST    18H      ; COMPARE VALUE TO LOWER LIMIT
      JR     NC,ERR   ; NO CARRY IF VALUE < 100
       .
       .
       .
```

RST 20 — Test Data Mode

Returns a combination of STATUS flags and unique numeric values in the A-register according to the data mode flag (40AF). This CALL is usually made to determine the type of the current value in WRA1. It should be used with caution, however since the mode flag and WRA1 can get out of phase particularly if some of the CALLS described here are used to load WRA1.

TYPE	STATUS	A-REGISTER
02 (INTEGER)	NZ/C/M/E	-1
03 (STRING)	Z/C/P/E	0
04 (SINGLE PREC.)	NZ/C/P/O	1
08 (DOUBLE PREC.)	NZ/NC/P/E	5

```
;
; TEST DATA TYPE AFTER INTEGER ADDITION TO
; DETERMINE IF OVERFLOW OCCURED (RESULT WOULD
; BE CONVERTED TO SINGLE PRECISION
;
      LD     A,2          ; TYPE CODE FOR INTEGER
      LD     (40AFH),02;  SET TYPE TO INTEGER
      LD     BC,(VAL1)    ; FIRST QUANTITY
      LD     HL,(VAL2)    ; SECOND QUANTITY
      CALL   0B2DH        ; DO INTEGER ADDITION
      RST    20H          ; TEST FOR OVERFLOW
      JP     M,OK         ; RESULT IS INTEGER
        .                 ; RESULT IS  NOT INTEGER
        .                 ; TEST FOR OTHER TYPES
OK    LD     (SUM),HL     ; SAVE INTEGER RESULT
        .
        .
        .
VAL1  DEFW   125          ; 16 BIT INTEGER VALUE
VAL2  DEFW   4235         ; 16 BIT INTEGER VALUE
SUM   DEFW   0            ; HOLDS 16 BIT VALUE
```

RST 28 — DOS Function CALL

Passes request code in A-register to DOS for processing. Returns for non-disk system. For disk systems, the A-register must contain a legitimate DOS function code. If the code is positive, the CALL is ignored and control returns to the caller. Note that the DOS routine discards the return address stored on the stack by the RST instruction. After processing control will be returned to the previous address on the stack. The calling sequence is:

```
;
; LOAD AND EXECUTE DEBUG
;
      LD       A,87H     ; DOS CODE FOR LOADING DEBUG
      CALL     DOS
        .                 RETURN HERE
        .
DOS   RST      28H       MAKE DOS CALL (WILL RET TO CALLER)
        .
        .
```

RST 30 — Load DEBUG

This CALL loads the DEBUG program and transfers control to it. When DEBUG processing is complete, control is returned to the orginal caller. For non-disk systems control is returned immediately.

```
; IF ILLOGICAL CONDITION ARISES LOAD AND EXECUTE DEBUG.
        .             ; TEST FOR LEGITIMATE CONDITIONS
        .
        .
      JR     Z,OK     ; JMP IF CONDITIONS ARE CORRECT
      RST    30H      ; ELSE LOAD AND EXECUTE DEBUG
 OK     .             ; CONTINUE
        .
        .
```

RST 38 — Interrupt Entry Point

This is the system entry point for all interrupts. It contains a jump to section of code in the Communications Region designed to field interrupts. That section of code consists of a DI (disables further interrupts) followed by a RET (returns to the point of interrupt) for non-disk systems, or a jump to an interrupt processor in SYS0 if it is a DOS system. For DOS systems the interrupt handler consists of a task scheduler, where the exact cause of the interrupt is determined (usually a clock interrupt) and the next task from the task control block is executed. After task completion, control returns to the point of interrupt.

```
;
; INTERCEPT ALL CLOCK INTERRUPTS AND TEST THE WIDGET
; ON PORT AB. IF THE READY LINE (BIT  8) IS TRUE
; (HIGH OR A 1) TURN ON THE COFFEE POT ON PORT DE.
; THEN JUMP TO THE NORMAL DOS INTERRUPT HANDLER
;
      ORG      4012H       ; REPLACE THE JUMP
      JP       HERE        ; TO THE DOS INTERRUPT
                           ; PROCESSOR WITH A JUMP
                           ; TO OUR  WN.
      ORG      0F000H      ; OUR INTERRUPT HANDLER
HERE  DI                   ; DISABLE FURTHER
                           INTERRUPTS
      PUSH     AF          ; WE'LL NEED AF REGS
      IN       A,(0ABH)    ; GET WIDGET STATUS
      OR       A           ; SET STATUS FOR BIT 8
      JP       M,TOCP      ; WIDGET ON IF MINUS
      POP      AF          ; WIDGET OFF, RST REGS
      JP       4518H       ; GO TO DOS INTERRUPT
                           ; HANDLER
TOCP  LD       A,21H       ; CODE TO TURN ON COFFEE
      POT
      OUT      (0DEH),A    ; SEND COMMAND TO POT
      POP      AF          ; THEN RST REGS
      JP       4518H       ; AND GO TO DOS INTERRUPT
                           ; HANDLER
        .
        .
        .
```

CALL 09B4 — Move SP Value In BC/DC Into WRA1

Moves the single precision value in BC/DE into WRA1. HL is destroyed BC/DE is left intact. Note - the mode flag is not updated !

```
        .
        .
      LD     BC,(PART1)   ; GET FIRST ARGUMENT
      LD     DE,(PART2)   ; REMAINDER OF ARGUMENT
                          ; NOTE - WE HAVE ASSUMED THAT
                          ; WRA1 CURRENTLY CONTAINS A
                          ; SINGLE PRECISION VALUE !!!
      CALL   09B4H        ; MOVE PART1 TO WRA1
      LD     BC,(PART3)   ; GET VALUE TO BE ADDED
      LD     DE,(PART4)   ; REST OF VALUE
      CALL   0716H        ; MOVE RESULT (SUM) TO WRA1
        .
        .
PART2 DEFW   0000H        ; LSB OF SP 1.5
PART1 DEFW   8140H        ; EXPONENT AND MSB OF SP 1.5
PART4 DEFW   0000H        ; LSB OF SP XX
PART3 DEFW   0000H        ; EXPONENT/MSB OF SP XX
        .
        .
        .
```

CALL 09B1 — Moves A SP Value Pointed To By HL To WRA1

Loads a single precision value pointed to by HL into BC/DE and then moves it to WRA1. Destroys HL/BC/DE.

```
        .
        .
      LD     HL,VAL       ; GET ADDR OF VALUE TO MOVE
      CALL   09B1H        ; MOVE VALUE TO WRA1
        .
        .
        .
VAL   DEFW   8140H        ; SINGLE PREC 1.5
      DEFW   0000H        ; REMAINDER OF 1.5
        .
        .
        .
```

CALL 09C2 — Load A SP Value Into BC/DE

Loads a single precision value pointed to by HL into BC/DE. Uses all registers.

```
;
; COMPUTE THE PRODUCT OF TWO SP NUMBERS AND MOVE THE
; PRODUCT TO BC/DE.
;
        LD    HL,VAL1       ; ADDR OF VALUE 1
        CALL  09B1H         ; MOVE IT TO WRA1
        LD    HL,VAL2       ; ADDR OF VALUE 2
        CALL  09C2H         ; LOAD IT INTO BC/DE
        LD    BC,(4121H)    ; LOAD EXPONENT/MSB
        LD    DE,(4123H)    ; LOAD LSB
              .
              .
              .
VAL1    DEFW  XXXX
        DEFW  XXXX
VAL2    DEFW  XXXX
        DEFW  XXXX
              .
              .
              .
```

CALL 09BF — Loads A SP Value From WRA1 Into BC/DE

Loads a single precision value from WRA1 into BC/DE. Note, the mode flag is not tested by the move routine. It is up to the caller to insure that WRA1 actually contains a single precision value.

```
              .
              .
        LD    HL,VAL1       ; ADDR OF VALUE TO MOVE TO WRA1
        CALL  09B1H         ; MOVE VAL1 TO WRA1
        LD    HL,VAL2       ; ADDR OF VALUE TO BE ADDED
        CALL  09C2H         ; LOAD VALUE TO BE ADDED TO BC/DE

        CALL  0716H         ; DO SINGLE PRECISION ADD
        CALL  09BFH         ; LOAD RESULT INTO BC/DE
        LD    (SUM1),DE     ; SAVE LSB
        LD    (SUM2),BC     ; SAVE EXPONENT/MSB
              .
              .
              .
SUM1    DEFW  0             ; HOLDS LSB OF SINGLE PRECSION
SUM2    DEFW  0             ; HOLDS EXPONENT/MSB
VAL1    DEFW  0000H         ; LSB OF S.P 2.0
        DEFW  8200H         ; EXPONENT/MSB OF S.P 2.0
VAL2    DEFW  00000;        ; LSB OF S.P. 5.0
        DEFW  8320H         ; EXPONENT/MSB OF S.P. 5.0
              .
              .
              .
```

CALL 09A4 — Move WRA1 To Stack

Moves the single precision value in WRA1 to the stack.It is stored in LSB/MSB/Exponent order. All registers are ~~left intact~~. Note, the mode flag is not tested by the move routine, it is simply assumed that WRA1 contains a single precision value.

```
;
; ADD TWO SINGLE PRECISION VALUES TOGETHER AND SAVE
; THE SUM ON THE STACK. CALL A SUBROUTINE WHICH
; WILL LOAD THE VALUE FROM THE STACK, PERFORM IT'S OWN
; OPERATION AND RETURN.
;
        LD    HL,VAL1       ; ADDR OF VALUE TO MOVE TO WRA1
        CALL  09B1H         ; MOVE VAL1 TO WRA1
        LD    HL,VAL2       ; ADDR OF VALUE TO BE ADDED
        CALL  09C2H         ; LOAD VALUE TO BE ADDED ITO BC/DE
        CALL  0716H         ; DO SINGLE PRECISION ADD
        CALL  09A4H         ; SAVE SUM ON STACK
        CALL  NSUB          ; CALL NEXT SUBROUTINE
              .
              .             ; RETURN WITH NEW VALUE IN
              .             ; IN WRA1.
              .
NSUB    POP   HL            ; GET RETURN ADDR
        LD    (RET),HL      ; MOVE IT TO A SAFE PLACE
        LD    HL,VAL3       ; ADDR OF QUANTITY TO ADD
        CALL  09B1H         ; MOVE VAL3 TO WRA1
        POP   BC            ; GET EXPONENT/MSB
        POP   DE            ; GET LSB
        CALL  0716H         ; ADD TO VALUE PASSED
        LD    HL,(RET)      ; GET RETURN ADDR
        JP    (HL)          ; AND RET TO CALLER
VAL1    DEFW  0000H         ; LSB OF S.P 2.0
        DEFW  8200H         ; EXPONENT/MSB OF S.P 2.0
VAL2    DEFW  00000;        ; LSB OF S.P. 5.0
        DEFW  8320H         ; EXPONENT/MSB OF S.P. 5.0
VAL3    DEFW  0AA6CH        ; LSB OF S.P. -.333333
        DEFW  7FAAH         ; EXPONENT/MSB OF S.P. -.33333
              .
              .
```

CALL 09D7 — General Purpose Move

Moves contents of B-register bytes from the address in DE to the address given in HL. Uses all registers except C.

```
;
; BLANK FILL A DCB THEN MOVE A NAME INTO IT
;
        LD    A,20H         ; HEX VALUE FOR BLANK
        LD    B,32          ; NO. OF BYTES TO BLANK
        LD    DE,IDCB       ; DE = ADDR OF DCB
LOOP    LD    (DE),A        ; STORE A BLANK INTO DCB
        INC   DE            ; BUMP STORE ADDR
        DJNZ  LOOP          ; LOOP TILL DCB BLANKED
        LD    DE,NAME       ; NOW, MOVE FILE NAME TO IDCB
        LD    HL,IDCB       ; DE = NAME ADDR, HL = DCB ADDR
        LD    B,LNG         ; NO. OF CHARS IN NAME TO MOVE
        CALL  09D7H         ; MOVE NAME TO DCB
              .
              .
              .
IDCB    DEFS  32            ; EMPTY DCB
LNG     EQU   ENDX-$        ; LET ASSEMBLER COMPUTE LNG OF
                            ; FILE NAME
NAME    DEFM  'FILE1/TXT'   ; NAME TO BE MOVED TO DCB
ENDX    EQU   $             ; SIGNAL END OF NAME
              .
              .
```

CALL 0982 — Variable Move Routine

Moves the number of bytes specified in the type flag (40AF) from the address in DE to the address in HL, uses registers A, DE, HL.

```
;
; LOCATE THE ADDRESS OF A DOUBLE PRECISION VARIABLE
; THEN MOVE IT TO A LOCAL STORAGE AREA.
;
        LD    HL,NAME1      ; NAME OF VARIABLE TO LOCATE
        CALL  260DH         ; GET ADDR OF STRING X
        RST   20H           ; MAKE SURE IT'S DBL PREC.
        JR    NC,OK         ; JMP IF DBL PREC.
        JP    ERR           ; ELSE ERROR
OK      LD    HL,LOCAL      ; HL = LOCAL ADDR
                            ; DE = VARIABLE ADDR
        CALL  0982H         ; MOVE VALUE FROM VLT TO LOCAL
        AREA.
              .
              .
              .
ERR           .
              .
NAME1   DEFM  'X'           ; NAME OF VARIABLE TO LOCATE
        DEFB  0             ; MUST TERM WITH A ZERO
LOCAL   DEFS  8             ; ENOUGH ROOM FOR DBL PREC. VALUE
              .
              .
              .
```

CALL 29C8 — String Move

On entry HL points to the string control block for the string to be moved, and DE contains the destination address. All registers are used. The string length and address are not moved. String control blocks have the format:

```
DEFB    X           STRING LENGTH
DEFW    ADDR        STRING ADDRESS
```

```
;
; LOCATE THE ADDRESS OF A STRING VARIABLE CALLED F$.
; MOVE THE STRING F$ TO A LOCAL STORAGE AREA CALLED
  DCB.
;
        LD   HL,NAME    ; NAME OF VARIABLE TO LOCATE
        CALL 28DBH      ; FIND ADDR OF $STRING F$
        RST  20H        ; MAKE SURE IT'S A STRING
        JR   Z,OK       ; JMP IF STRING
        JP   ERR        ; ELSE ERROR
OK      LD   A,(DE)     ; GET LENGTH OF STRING
        CP   33         ; WHICH MUST BE < 33
        JP   P,ERR      ; ERR, STRING LNG > 32
        PUSH DE         ; SHORTCUT FOR MOVING DE TO HL

        POP  HL         ; ADDR OF STRING TO HL
        LD   DE,LOCAL   ; DE = LOCAL ADDR
        CALL 29C8H      ; MOVE STRING VARIABLE TO
                        ; LOCAL AREA
          .
ERR       .
          .
NAME    DEFM 'F$'       ; NAME OF VARIABLE TO FIND
        DEFB 0          ; REQUIRED TO TERM NAME
LOCAL   DEFS 32         ; LOCAL STORAGE AREA
          .
          .
          .
```

Basic Functions

Basic Functions differ from System Functions because they deal mainly with tables in the Communications Region (CR). Because of this, these entry points assume that the CR has been initialized and properly maintained. This means that the BASIC Interpreter must have been entered prior to calling any of these routines, and the BASIC utility in RAM must be intact. The assembly program making the CALL must be running as a subroutine called by a BASIC program.
For a complete description of the tables and storage areas in the Communication Region see chapter 4.

CALL 1B2C — Search For Line Number

Searches the Program Statement Table (PST) for a BASIC statement with the line number specified in the DE register pair. All registers are used. The exit conditions are:

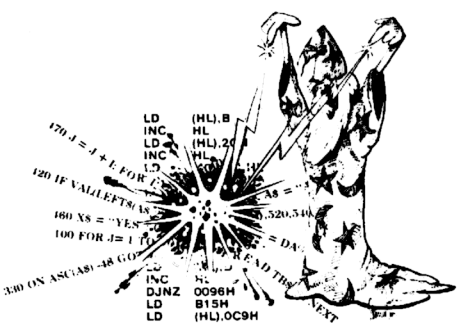
STATUS	CONDITION	REGISTERS
C/Z	LINE FOUND. BC = STARTING ADDRESS OF LINE IN PST.	
NC/Z	LINE DOES NOT EXIST. LINE NUMBER TOO LARGE	HL = ADDRESS OF FOLLOWING LINE IN PST. HL/BC = ADDRESS OF NEXT AVAILABLE LOCATION IN
NC/NZ	LINE DOES NOT EXIST. BC = ADDRESS OF FIRST LINE NUMBER GREATER THAN THE ONE SPECIFIED.	HL = ADDRESS OF FOLLOWING LINE.

```
;
; LOCATE THE ADDRESS OF BASIC STATEMENT NUMBER 750
; IN THE PST. IF THE LINE DOES NOT EXIST RETURN A
; STATUS OF -1 IF IT IS LARGER THAN ANY CURRENT LINE
; NUMBER, OR A -2 IF IT THERE ARE LINES GREATER THAN
; 750. IF THE LINE IS FOUND RETURN A STATUS OF ZERO.
;
          .
        LD   DE,750     ; LINE NUMBER TO SEARCH FOR
        CALL 1B2CH      ; SEEK LINE IN PST
        JR   NC,NO      ; NC SET IF LINE NOT THERE
        LD   HL,3       ; INCREMENT TO STEP OVER
        ADD  HL,BC      ; POINTER TO NEXT LINE/LINE NO.

                        ; RST BELOW WILL INCREMENT
                        ; BEFORE  LOADING
        RST  10H        ; FETCH FIRST CHAR OF OF
                        ; STATEMENT.
          .
        LD   A,0        ; SIGNAL LINE FOUND
        RET             ; RETURN TO CALLER
NO      JR   NC,M2      ; JMP IF LINE NO. TOO BIG
        LD   A,0FFH     ; SIGNAL LINE NOT THERE
        RET             ; RETURN TO CALLER
M2      LD   A,0FEH     ; SIGNAL LINE NOT THERE
                        ; TOO BIG
        RET             ; RETURN TO CALLER
```

CALL 260D — Find Address Of Variable

This entry point searches the Variable List Table (VLT) for a variable name which matches the name in the string pointed to by HL. If the variable exists, its address is returned in DE. If it is not defined, then it is created with an initial value of zero and its address is returned in DE. Dimensioned and non-dimensioned variables may be located, and suffixs for data mode may be included in the name string. A byte of machine zeros must terminate the name string. All registers are used.

```
;
; LOCATE THE ADDRESS OF THE VARIABLE A3
;
        LD   HL,STRNG   ; NAME OF VARIABLE TO LOCATE
        CALL 260DH      ; FIND IT'S ADDRESS IN VLT
        LD   (ADDR),DE  ; SAVE FOR FUTURE REFERENCE
          .
          .
STRNG   DEFM 'A3'       ; VARIABLE NAME IS  A3
        DEFB 0
STRNG   DEFM 'A(25)'    ; VARIABLE NAME IS A(25)
        DEFB 0
STRNG   DEFM 'AI'       ; VARIABLE NAME IS AI
        DEFB 0
```



CALL 1EB1 GOSUB

Can be used to execute the equivalent of a GOSUB statement from an assembly program. It allows a BASIC subroutine to be called from an assembly subroutine. After the BASIC subroutine executes, control returns to the next statement in the assembly program. All registers are used. On entry, the HL must contain an ASCII string with the starting line number of the subroutine.

```
;
; SIMULATE A GOSUB STATEMENT FROM AN ASSEMBLY LANGUAGE PROGRAM
;
    LD      HL,STRNG    ; ADDRESS OF BASIC LINE NUMBER TO GOSUB TO
    CALL    1EB1H       ; EQUIVALENT OF A GOSUB 1020
      .
      .                 ; WILL RETURN HERE WHEN BASIC PROGRAM
      .                 ; EXECUTES A RETURN
      .
STRNG DEFM    '1020'    ; LINE NO. OF BASIC SUBROUTINE
    DEFB      0
```

CALL 1DF7 TRON

Turns TRON feature on. Causes line numbers for each BASIC statement executed to be displayed. Uses A-register.

```
;
; TURN TRACE ON THEN EXECUTE A BASIC SUBROUTINE
;
      CALL  1DF7H     ; TURN TRACE ON
      LD    HL,LN     ; LINE NO. TO GOSUB
      CALL  1EB1H     ; DO A GOSUB 1500
        .
        .
        .
LN    DEFM  '1500'    ; LINE NO. OF BASIC SUBROUTINE
      DEFB  0
```

CALL 1DF8 TROFF

Disables tracing feature. Uses A register.

```
;
; ENABLE TRACE. EXECUTE BASIC SUBROUTINE. UPON
; RETURN DISABLE TRACING.
;
      CALL 1DF7H    ; TURN TRACE ON
      LD   HL,LN    ; LINE NO. OF BASIC SUBROUTINE
      CALL 1EB1H    ; DO A GOSUB 2000
      CALL 1DF8H    ; TURN OFF TRACING
      RET           ; RETURN TO CALLER
LN    DEFM '2000'   ; LINE NO. OF BASIC SUBROUTINE
      DEFB 0
```

JP 1EDF RETURN

Returns control to the BASIC statement following the last GOSUB call. An assembly program called by a BASIC subroutine may wish to return directly to the orginal caller without returning through the subroutine entry point. This exit can be used for that return. The return address on the stack for the call to the assembly program must be cleared before returning via 1EDF.

```
300 GOSUB 1500          CALL BASIC SUBROUTINE
310 GOSUB 1510          RETURN HERE FROM SUBROUTINE CALL
320   .
      .
      .
1500 Z=USR1(0)          CALL ASSEMBLY SUBROUTINE & RETURN

1510 Z=USR2(0)          CALL ANOTHER SUBROUTINE & RETURN

1530  .
      .
      .
;
; ENTRY POINT FOR USR1 SUBROUTINE
;
      .                 ; DO WHATEVER PROCESSING IS
                          REQUIRED
      .
      .
    POP   AF            ; CLEAR RETURN  ADDR TO 1510
                        ; FROM STACK
    JP    1EDFH         ; RETURN DIRECTLY TO 310
;
; ENTRY POINT FOR USR2 SUBROUTINE
;
      .                 ; PERFORM NECESSARY PROCESSING
                        ; FOR USR2 CALL
      POP   AF          ; CLEAR RETURN ADDR TO 1520
      JP    1EDFH       ; RETURN DIRECTLY TO 320
```

CALL 28A7 Write Message

Displays message pointed to by HL on current system output device (usually video). The string to be displayed must be terminated by a byte of machine zeros or a carriage return code 0D. If terminated with a carriage return, control is returned to the caller after taking the DOS exit at 41D0 (JP 5B99). This subroutine uses the literal string pool table and the string area. It should not be called if the communications region and the string area are not properly maintained.

```
;
; WRITE THE MESSAGE IN MLIST TO THE CURRENT SYSTEM
; OUTPUT DEVICE.
;
      .
      .
      LD   HL,MLIST     ; HL = ADDR OF MESSAGE
      CALL 28A7H        ; SEND TO SYSTEM OUTPUT DEVICE
      .
      .
      .
MLIST DEFM  ' THIS IS A TEST'
      DEFB  0DH         ; THIS TERMINATOR REQUIRED
      .
      .
      .
```

CALL 27C9 Return Amount Of Free Memory

Computes the amount of memory remaianing between the end of the variable list and the end of the stack. The result is returned as a single precision number in WRA1 (4121-4124).

```
;
; TAKE ALL AVAILABLE MEMORY BETWEEN THE STACK AND
; THE END OF THE VLT AND DIVIDE IT INTO REGIONS FOR
; USE IN A TOURNAMENT SORT
      .
      .
```

```
DI              ; MUST GO INHIBITED BECAUSE
                ; THERE WILL BE NO STACK SPACE

                ; FOR INTERRUPT PROCESSING
CALL 27C9H      ; GET  AMT OF FREE SPACE
CALL 0A7FH      ; CONVERT IT TO INTEGER
LD   DE,(4121H) ; GET IT INTO DE
LD   HL,500     ; MAKE SURE IT'S AT
RST  18H        ; LEAST 500  BYTES
JR   C,ERR      ; ERR - INSUFFICIENT SPACE
LD   HL,(40D1H) ; START OF AREA
LD   (EVLT),HL  ; SAVE FOR RESTORATION
LD   HL,0       ; SO WE CAN LOAD CSP
ADD  HL,SP      ; END OF AREA
LD   (ECSP),HL  ; SAVE FOR RESTORATION
      .
      .
      .
```

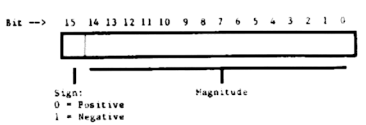
CALL 2B75 Print Message

Writes string pointed to by HL to the current output device. String must be terminated by a byte of zeros. This call is different from 28A7 because it does not use the literal string pool area, but it does use the same display routine and it takes the same DOS Exit at 41C1. Uses all registers. This routine can be called without loading the BASIC utility, if a C9 (RET) is stored in 41C1.

```
;
; WRITE MESSAGE TO CURRENT OUTPUT DEVICE
;
        .
        .
      LD   HL,MLIST     ; ADDRESS OF MESSAGE
      CALL 2B75H        ; SEND MSG TO SYSTEM DEVICE
        .
        .
MLIST DEFM 'THIS IS A TEST'
      DEFB 0            ; REQUIRED TERMINATOR
        .
        .
        .
```

Internal Number Representation

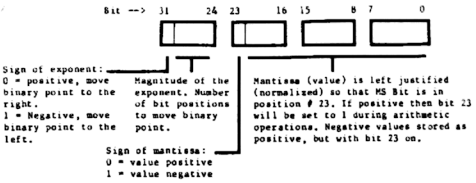
BASIC represents integers as signed 16 bit quantites. Bit 15 contains the sign bit while bits 0-14 hold the magnitude. The largest possible positive value that can be represented is 32767 (dec.) or 7FFF (hex). The smallest possible negative value that can be represented is -32768 (dec.) or 8000 (hex).



positive values 0000 - 7FFF (hex.) : 0 to 32767 (dec.)
Negative values FFFF - 8000 (hex.) : -1 to -32768 (dec.)

Note - negative values are represented as the one's complement of the positive equivalent.

BASIC supports two forms of floating point numbers. One type is single precision and the other is double precision. Both types have a signed seven bit exponent. Single precision numbers have a signed 24 bit mantissa while double precision values have a signed 56 bit mantissa. Both types have the following format



The only difference bettween single and double precision is in the number of bits in the mantissa. The maximum number of significant bits representable in a positive single precision value is 2 ** 24 -1 or 8 388 607 decimal or 7F FF FF hex. Double precision numbers have an extended mantissa so positive values up to 2 ** 56- 1, or 3.578 X 10 ** 16 can be represented accurately.

These numbers 8 388 607 and 3.578 X 10 ** 16 are not the largest numbers that can be represented in a single or double precision number, but they are the largest that can be represented without some loss of accuracy. This is due to the fact that the exponent for either type of number ranges between 2 ** -128 and 2 ** 127. This means that theoretically the binary point can be extended 127 places to the right for positive values and 128 to the left for negative values even though there are only 24 or 56 bits of significance in the mantissa. Depending of the type of data being used (the number of significant digits) this may be all right. For example Planck's constant which is 6.625 X 10 ** -34 J-SEC could be represented as a single precision value without any loss of accuracy because it has only four significant digits. However if we were totalling a money value of the same magnitude it would have to be a double precision value because all digits would be significant

Chapter 3

Cassette & Disk

This chapter contains an introductory description of physical I/O operations for the cassette and disk. The sample programs are for purposes of illustration only and are not recommended for adaptation to general applications. There may be special situations, however when a simple READ/WRITE function is needed and for limited applications they will serve the purpose.

Cassette I/O

Cassette I/O is unusual from several aspects. First, each byte is tranmitted on a bit-by-bit basis under software control. This is radically different from all other forms of I/O where an entire byte is transfered at one time. For most I/O operations, referencing memory or executing an IN or OUT instruction, is all that is required to transfer an entrie byte between the CPU and an external device. However, If the device is a cassette, each bit (of a byte to be transferred) must be transfered individually by the software.

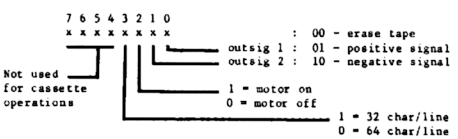
The second unusual aspect is the procedure used for transmitting these bits. Exact timing must be adhered to and the program must use different code depending on whether a binary zero or one is to be written. Each bit recorded consists of a clock pulse (CP) followed by a fixed amount of erased tape followed by either another CP if a binary one is represented, or a streach of erased tape if a binary zero is being represented. A binary one and zero would appear as:

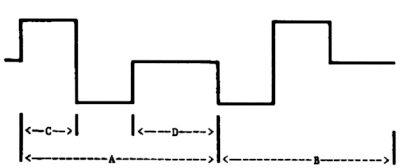
The distance between points A, B, C, and D is measured in units of time. Because time can be measured in machine cycles the value given for distances will be in machine cycles where one instruction (any instruction regardless of how long it is) equals one cycle and one cycle equals one microsecond. This is crude but workable. The sum of A B is supposed to be 2 milliseconds for Level II.

Using the crudity described above and counting instructions used in the Level II software gives the following values.

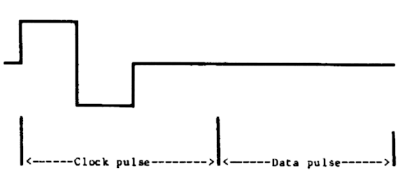
A B 1.4 millisec per half bit 2.8 millisec per bit.
C .20 millisec * 2 per CP .40 millisec
D 1.0 millisec

Before discussing programming for cassette I/O in any detail we should review the fundementals. Drive selection is accomplished by storing either a **01** (drive 1) or **02** (drive 2) in **37E4**. Motor start and loading or clearing the data latch is achieved by sending a command value to the cassette controller on port **FF**. The command value is shown below.





Binary one



Binary zero

Be careful to preserve the current video character size when sending commands to the cassette. The system maintains a copy of the last command sent to the video controller in 403D. Bit 3 of that word should be merged with any commans issued to the cassette.

A write operation of one bit (called a bit cell) can be divided into two steps. First a clock pulse (CP) is written to signal the start of a bit. It is followed by a strip of erased tape which is considered part of the CP. Next, another CP is written if the bit is a one, or more blank tape is written if the bit is a zero.

Read operations begin by searching for the clock pulse and skipping to the data pulse area. The data pulse area is then read returning a zero if blank tape was encountered or a one if non-blank tape was found. Below are examples of code that could be used for cassette operations. The code used by Level II can be found around the area **01D9 - 02A8** in the Level II listing.

Assembler Object Code Format

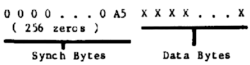
DOS loads disk object files with a utility program called LOAD. They can also be loaded under DOS by entering the name of a file that has an extension of CMD. The format of a disk object file is shown below. It is more complex than a cassette file because it has control codes embedded in the object code. The loader reads the file into a buffer before moving the object code to its designated address. The control codes are used to indicated to the loader where the code is to be loaded, how many bytes are to be loaded, and where execution is to begin.

```
Control Code:    01   (data to be loaded follows)
Count       :    XX   (count of bytes to load, 0 = 256)
Load Address:    XX   (load address in LSB/MSB order)
                 XX
Load Data   :    XX
                 XX
                 .

Control Code:    02   (beginning execution address follows)
                 XX   (this byte is to be discarded)
Address     :    XX   (execution address in
                 XX   (LSB/MSB order)


Control Code:  03 - 05  (following data is to be skipped)
Count       :    XX   (count of bytes to skip)
Skip Data   :    XX   (this data is to be skipped)
                 XX
                 .
```

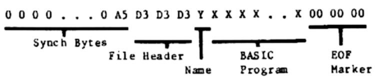
Cassette Recording Format

The recording format used by Level II is as follows:

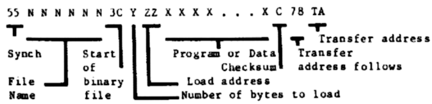
1: BASIC Data Files



2: BASIC Programs



3: Absolute Assembler Programs




```
SELECT UNIT AND TURN ON MOTOR
        LD    A,01       ; CODE FOR UNIT 1
        LD    (37E4H),A ; SELECT UNIT 1
        LD    A,04       ; COMMAND VALUE: TURN ON MOTOR
        OUT   (OFFH),A   ; START MOTOR, CLEAR DATA LATCH

WRITE BYTE CONTAINED IN THE A REGISTER
        PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL           ; SAVE CALLERS REGISTERS
        LD    L,8          ; NUMBER OF BITS TO WRITE
        LD    H,A          ; H = DATA BYTE
LOOP    CALL  CP           ; WRITE CLOCK PULSE FIRST
        LD    A,H          ; GET DATA BYTE
        RLCA               ; HIGH ORDER BIT TO CARRY
        LD    H,A          ; SAVE REPOSITIONED BYTE
        JR    NC,WR        ; BIT WAS ZERO. WRITE BLANK TAPE
        CALL  CP           ; BIT WAS ONE. WRITE A ONE DATA PULSE
TEST    DEC   L            ; ALL BITS FROM DATA BYTES WRITTEN ?
        JR    NZ,LOOP      ; NO!  JUMP TO LOOP
        POP   HL           ; YES! RESTORE CALLERS REGISTERS
        POP   DE
        POP   BC
        POP   AF
        RET                ; RETURN TO CALLER
WR      LD    B,135        ; DELAY FOR 135 CYCLES (988 USEC) WHILE
WR1     DJNZ  WR1          ; BLANK TAPE IS BEING WRITTEN
        JR    TEST         ; GO TEST FOR MORE BITS TO WRITE
CP      LD    A,05         ; COMMAND VALUE MOTOR ONE, OUTSIG 1
        OUT   (OFFH),A     ; START OF CLOCK PULSE
        LD    B,57         ; DELAY FOR 57 (417 USEC) CYCLES
CP1     DJNZ  CP1          ; GIVES PART OF CP
        LD    A,06         ; COMMAND VALUE: MOTOR ON, OUTSIG 2
        OUT   (OFFH),A     ; 2ND PART OF CLOCK PULSE
        LD    B,57         ; DELAY FOR 57 CYCLES (417 USEC)
CP2     DJNZ  CP2          ; GIVES PART OF CP
        LD    A,4          ; COMMAND VALUE: MOTOR ON, NO OUTSIG
        OUT   (OFFH),A     ; START ERASING TAPE
        LD    B,136        ; DELAY FOR 136 CYCLES (995 USEC)
CP3     DJNZ  CP3          ; GIVES TAIL OF CLOCK PULSE
        RET                ; RETURN TO CALLER

READ NEXT BYTE FROM CASSETTE INTO A REGISTER
        XOR   A            ; CLEAR DESGINATION REGISTER
        PUSH  BC
        PUSH  DE
        PUSH  HL           ; SAVE CALLERS REGISTERS
LOOP    LD    B,8          ; NUMBER OF BITSS TO READ
        CALL  RB           ; READ NEXT BIT. ASSEMBLE INTO
                           ; BYTE BUILT THUS FAR.
        POP   HL
        DJNZ  LOOP         ; LOOP UNTIL 8 BITS USED
        POP   DE
        POP   BC           ; RESTORE CALLERS REGISTERS
        RET                ; RETURN TO CALLER


RB      PUSH  BC
        PUSH  AF
RB1     IN    (OFFH),A     ; READ DATA LATCH
        RLA                ; TEST FOR BLANK/NON-BLANK TAPE
        JR    NC,RB1       ; BLANK, SCAN TILL NON-BLANK
                           ; IT WILL BE ASSUMED TO BE START
                           ; OF A CLOCK PULSE.
        LD    B,57         ; DELAY FOR 57 CYCLES WHILE
RB2     DJNZ  RB2          ; SKIPPING OVER FIRST PART OF CP
        LD    A,04         ; COMMAND VALUE: MOTOR ON, CLEAR
        OUT   (OFFH),A     ; DATA LATCHES
        LD    B,193        ; DELAY FOR 193 CYCLES WHILE
RB3     DJNZ  RB3          ; PASSING OVER END OF CP
        IN    A,(OFFH)     ; WE SHOULD BE POSITIONED INTO
                           ; THE DATA PULSE AREA. READ
                           ; THE DATA PULSE.
        LD    B,A          ; SAVE DATA PULSE
        POP   AF           ; ACCUMULATED BYTE THUS FAR
        RL    B            ; DATA PULSE TO CARRY WILL BE A
                           ; ZERO IF BLANK TAPE, 1 IF NON-BLANK
        RLA                ; COMBINE NEW DATA PULSE (1 BIT)
        PUSH  AF           ; WITH REST OF BYTE AND SAVE
        LD    A,4          ; COMMAND VALUE: MOTOR ON, CLEAR OUTSIG
        OUT   (OFFH),A     ; CLEAR DATA LATCHES
        LD    B,240        ; DELAY LONG ENOUGH TO SKIP TO
RB4     DJNZ  RB4          ; END OF DATA PULSE
        POP   BC
        POP   AF           ; A = DATA BYTE
        RET

TURN OFF MOTOR
        LD    A,00         ; COMMAND VALUE: MOTOR OFF
        OUT   (OFFH),A     ; TURN MOTOR OFF
        RET
```

Disk I/O

The disk operations discussed in this section are elementary inasmuch as there is no consideration given to disk space management or other functions normally associated with disk I/O. What is presented are the fundamental steps necessary to position, read, and write any area of the disk without going through DOS. It will be assumed that the reader is familiar with the I/O facility provided by DOS and is aware of the pitfalls of writing a diskette without going through DOS.

Disks which normally come with a Model I system are single sided, 35 track 5 1/4' mini-drives. It is possible to substitute other drives with a higher track capacity such as 40, 77, or 80 tracks, but then a modified version of DOS must be used. Dual sided mini-drives are becoming available and eventually they should replace the single sided drives. Dual density drives are another type of mini-drive that are available, but like the dual sided drives they require a modified version of DOS.

The type of programming used in this example is called programmed I/O. It is called that because the program must constantly monitor the controller status in order to determine if it is ready to send or recieve the next data byte. Thus each byte is transfered individually under program control. An alternative to programmed I/O is DMA or Direct Memory Access. Using this method the controller is told the number of bytes to transfer and the starting transfer address and it controls the tranfer of data leaving the CPU free to perform other tasks. On the Model I systems there is no DMA facility so programmed I/O must be used.

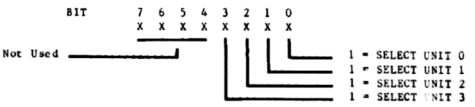
This example will assume that a DOS formatted diskette is being used. New diskettes are magnetically erased. Before they can be used they must be formatted. That is each sector and track must be uniquely identified by recording its track and sector number in front of the data area of each sector. There is some variability in the coded information which preceeds each sector so it is not always possible to read any mini-diskette unless it orginated on the same type of machine.

Like most of the I/O devices on the Model I the disk is memory mapped. There are five memory locations dedicated to the disk. They are:

37E1	Unit Select Register
37EC	Command/Status Register
37ED	Track Update Register
37EE	Sector Register
37EF	Data Register

All disk commands except for unit selection are sent to 37EC. If the command being issued will require additional information such as a track or sector number, then that data should be stored in the appropriate register before the command is issued. You may have noticed that the command and status register have the same address. Because of that, a request for status (load 37EC) cannot occur for 50 microseconds following the issuing a command (store 37EC).

Unit selection is accomplished by storing a unit mask value into location 37E1. That mask has the format:

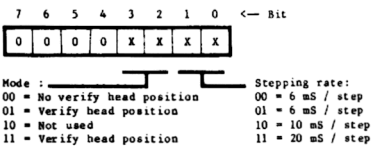


More than one unit can be selected at a time. For example a mask of 3 would select units 0 and 1. When any unit is selected the motor on all units are automatically turned on. This function is performed automatically by the expansion interface.

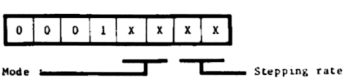
Controller Commands

The Model I uses a Western Digital FD1771B-01 floppy disk controller chip. It supports twelve 8-bit commands. They are:

Restore: Positions the head to track 0



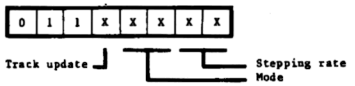
Seek: Positions the head to the track specified in the data register (37E).



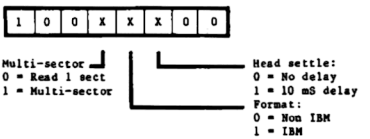
Step: Moves the head one step in the same direction as last head motion.

Step Head In: Moves the head in towards the innermost track one position.

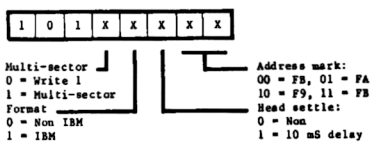
Step Head Out: Moves the head out towards the outermost track one position.



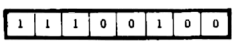
Read Data: Transmits the next byte of data from the sector specified by the value in the sector register.



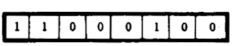
Write Data: Sends the byte of data in the data register to the next position in the sector specified by the value in the sector register.



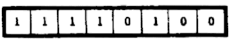
Read Track: Reads an entire track beginning with the index mark.



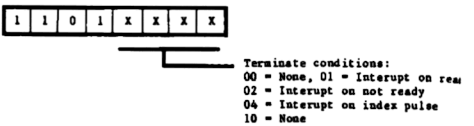
Read Address: Reads the address field from the next sector to pass under the head.



Write Track: Writes a full track starting at the index mark and continuing until the next index mark is encountered.



Force Interrupt: Terminates the current operation and / or generates an interrupt if one of the following four conditions is true:



Read Status: The status of the Floppy Controller is returned whenever location 37EC is read. The status word has the following format:

Disk Programming Details

Disk programming can be broken down into several easily managed steps. They are:

1. Select the unit and wait for ready.
2. Position the head over the desired track.
3. Isue the Read/Write command for the required sector
4. Transfer a sectors worth of data, on a byte at a time basis.

Each transfer must be preceeded by a test to see if the controller either has the next data byte, or is ready to accept the next data byte.

This program demonstrates a single sector read from track 25 (decimal), sector 3.

```
        ORG   7000H
        LD    BC,256        ; BYTE COUNT
        PUSH  BC            ; B= =1 C = 0
        LD    HL,BUFF       ; BUFFER ADDRESS
        LD    A,1           ; UNIT SELECT MASK (DRIVE 0)
        LD    (37E1H),A     ; SELECT DRIVE 0, START MOTOR
        LD    D,25          ; TRACK NUMBER
        LD    E,3           ; SECTOR NUMBER
        LD    (37EEH),DE    ; SPECIFY TRACK AND SECTOR
                            ; TRACK NO. TO DATA REGISTER
                            ; (37EFH)
                            ; SECTOR NO. TO SECTOR REGISTER.
        LD    A,1BH         ; SEEK OP CODE. NO VERIFY
                            ; (FOR VERIFY 17H)
        LD    (37ECH),A     ; SEEK REQ. TO COMMAND REGISTER.
        LD    B,6           ; GIVE CONTROLLER A CHANCE
                            ; TO DIGEST
DELAY   DJNZ  DELAY         ; COMMAND BEFORE ASKING STATUS
WAIT    LD    A,(37ECH)     ; GET STATUS OF SEEK OP
        BIT   0,A           ; TEST IF CONTROLLER BUSY
        JR    NZ,WAIT       ; IF YES, THEN SEEK NOT DONE
        LD    A,88H         ; SEEK FINISHED. LOAD READ
                            COMMAND
        LD    (37ECH),A     ; AND SEND TO CONTROLLER
        LD    B,6           ; GIVE CONTROLLER A CHANCE TO
DELAY1  DJNZ  DELAY1        ; DIGEST COMMAND BEFORE
                            REQUESTING
                            ; A STATUS
WAIT1   LD    A,(37ECH)     ; NOW, ASK FOR STATUS
        BIT   1,A           ; IS THERE A DATA BYTE PRESENT ?
        JR    Z,WAIT1       ; NO, WAIT TILL ONE COMES IN
        LD    A,(37EFH)     ; YES, LOAD DATA BYTE
        LD    (HL),A        ; STORE IN BUFFER
        INC   HL            ; BUMP TO NEXT BUFF ADDR
        DEC   BC            ; TEST FOR 256 BYTES TRANSFERED
        LD    A,B           ; COMBINE B AND C
        OR    C             ; TO TEST BOTH REGISTERS
        JR    NZ,WAIT       ; GO GET NEXT BYTE
              .
              .
              .
```

DOS Exits

DOS Exits were discussed in general terms in chapter 1. They are used as a means of passing control between Level II BASIC and Disk BASIC. The Exit itself is a CALL instruction in the ROM portion of the system to a fixed address in the Communications Region. Contained at that CALL'd address will be either a RETURN instruction or a JUMP to another address in Disk BASIC. On a Level II system without disks these CALL'd locations are set to RETURNS during IPL processing. On disk based systems they are not initialized until the BASIC command is executed. At that time JUMPS to specific addresses within Disk BASIC are stored at the CALL locations.

The term DOS Exit really has two different meanings. DOS Exits are calls from ROM BASIC to Disk BASIC while in the Input Phase, while executing a system level command, or while executing a verb action routine. These exits allow extensions to be made to the routines in ROM. The exits are not stragetically located so that an entire ROM routine could be usurped, but they are conveniently placed for intercepting the majority of the ROM routine processing. Another type of DOS Exit is the Disk BASIC Exit. These exits are radically different from the other ones, they are only entered on demand when a Disk BASIC token is encountered during the Execution Phase. All of the processing associated with these tokens is contained in the Disk BASIC program. There is no code in ROM for executing these tokens.

The following descriptions are for DOS Exits as opposed to Disk BASIC Exits. The calling sequence for each of the DOS Exits vary. Before writing a program to replace any of these Exits study the code around the CALL, paying particular attention to register usage. What happens at the exits is not discussed here. If it is important, disassemble the Disk BASIC utility program and examine the code at the BASIC address assigned to the exit. An example of how both types of Exits can be intercepted can be found in chapter 6.

All these addresses are for NEWDOS 2.1, TRSDOS addresses will differ.

Level II ADDRESS	DESCRIPTION	DOS Exits ADDRESS	BASIC ADDRESS
19EC	Call to load DISK BASIC error processing. Error number must be in E-register.	41A6	
27FE	Start of USR processing	41A9	5679
1A1C	BASIC start up, Just before BASIC's 'READY' message.	41AC	5FFC
0368.....	At start of keyboard input.	41AF	598E
1AA1	Input scanner after tokenizing current statement.	41B2	6033
1A8C	Input scanner after udating program statement table.	41B5	5BD7
1AF2	Input scanner after reinitial- izing BASIC.	41B8	5B8C
1B8C/1DB0	Initializing BASIC for new routine. During END processing.	41BB	60A1
2174	During initializing of system output device.	41BE	577C
032C	During writing to system output device.	41C1	59CD
0358	When scanning keyboard. Called from INKEY$, at end of execution of each BASIC statement.	41C4	59CD
1EA6	At start of RUN NNN processing.	41C7	5F78
206F	At beginning of PRINT processing.	41CA	5A15
20C6	During PRINT # or PRINT item processing.	41CD	5B9A
2103	When skipping to next line on video during a BASIC output operation.	41D0	5B99
2108/2141	At start of PRINT on cassette and during PRINT TAB processing.	41D3	5B65
219E	At beginning of INPUT processing.	41D6	5784
222D	During READ processing when a variable has been read.	41DC	5E63
2278/2278	At end of READ processing	41DF	579C
2B44/2B44	From LIST processing.		
02B2.....	During SYSTEM command operation.	41E2	5B51

Disk BASIC Exits

These exits are made from Level II during the Execution Phase whenever a token in the range of BC - FA is encountered. Tokens with those values are assigned to statements which are executed entirely by Disk BASIC. When a token in the given range is found control is passed indirectly through the Verb Action Routine List (see chapter 4) to the appropriate Disk BASIC Exit in the Communications Region. Control is returned to Level II at the end of the verb routine's processing.

TOKEN	VERB	CR ADDRESS	DISK BASIC ADDRESS
E6	CVI	4152	5E46
BE	FN	4155	558E
E7	CVS	4158	5E49
B0	DEF	415B	5655
E8	CVD	415E	5E4C
E9	EOF	4161	61EB
EA	LOC	4164	6231
EB	LOF	4167	6242
EC	MKI$	416A	5E2D
ED	MKS$	416D	5E30
EE	MKD$	4170	5E33
85	CMD	4173	56C4
C7	TIME$	4176	5714
A2	OPEN	4179	6349
A3	FIELD	417C	60AB
A4	GET	417F	627C
A5	PUT	4182	627B
A6	CLOSE	4185	606F
A7	LOAD	4188	5F7B
A8	MERGE	418B	600B
A9	NAME	418E	6346
AA	KILL	4191	63C0
NONE	&	4194	5887
AB	LSET	4197	60E6
AC	RSET	419A	60E5
C5	INSTR	419D	582F
AD	SAVE	41A0	6044
9C	LINE	41A3	5756
C1	USR	41A9	5679

Disk Tables

The most frequently used disks on the Model I series are 5 1/4' single sided single density mini-floppy drives. A variety of other units are available and could be used, however some hardware and software modifications would be necessary. Examples of other units would be: 5 1/4' dual headed and dual density drives; 8' single and dual headed plus single and dual density units; and various hard disks with capacities up to 20 Mbytes.

The terms single and dual headed refer to the number of read/write heads in a unit.Most micro-computer systems use single headed drives but dual headed drives are now becoming more commonplace. A dual headed drive has twice the capacity of a single headed unit because two disk surfaces can be accessed rather than one.

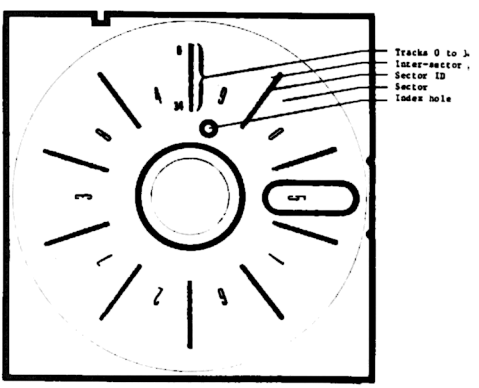
Dual density describes the recording method used. In single density mode each bit cell consists of a clock pulse followed by a data pulse while in dual density recording clock pulses may be omitted if the data pulse is repetitious. Using this method more sectors can be written on a track than in single density format. The recording method used is dictated by the controller and the software, but with dual density drives clock pulses may be omitted and the timing is more critical, hence not all drives can be used for dual density.

Eight inch drives are essentially the same as 5 1/4' drives except they usually only come in one track size (77 tracks). As with the smaller units they come in both single and dual density. Since their radius is larger they have more sectors per track. Track capacities for 8' drives are typically: 26 - 128 byte sectors / track; 15 - 256 byte sectors / track; 8 - 512 byte sectors / track; 4 - 1024 byte sectors / track.

Track capacities for 5 1/4' single density are: 20 - 128 byte sectors / track; 10 - 256 byte sectors / track; 5 - 512 byte sectors / track; and 2 - 1024 byte sectors / track. Dual density 5 1/4' drives have capacities of: 32 - 128 byte sectors / track; 18 - 256 byte sectors / track; 08 - 512 byte sectors / track; and 4 - 1024 byte sectors / track.

Hard disks are too varied to classify. Basically a hard disk has more capacity, faster access time, higher transfer rates, but the disk itself may not be removeable. Without a removeable disk file backup can be a serious problem, a second hard disk is an expensive solution.

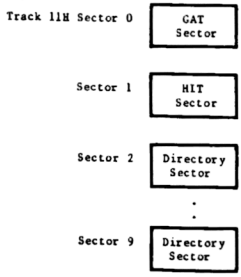
Shown below is a diagram of a 5 1/4' 35 track diskette.



Each diskette has 35, 40, 77, or 80 tracks depending on the drive used. Each track has 10 sectors of 256 bytes. Sector sizes can vary from 2 to 1024 bytes per sector. But the software must be modified to handle anything other than 256, because that is the size assumed by DOS. The Model I uses a semi IBM compatible sector format. It is not 100% compatible because track and sector numbers on IBM diskettes are numbered from 1 not 0 as in TRSDOS.

DOS uses a file directory to keep a record of file names and their assigned tracks and sectors . The directory occupies all 10 sectors of track number 11. It is composed of three parts: a disk map showing available sectors (track 11, sector 1): a file name in use index that allows the

directory to be searched from an advanced starting point (called the Hash Index Table track 11, sector 2); and the directory sectors themselves (track 11 sector 3 thru track 11 sector 10).



As well as the directory track there is one other special area on a diskette. Track 0 sector 0 contains a system loader used during the disk IPL sequence to load DOS. The loader is read into RAM locations 4200 - 4300 by the ROM IPL code which then passes control to it so that the DOS can be loaded.

Disk Track Format

Before any diskette can be used it must be initialized rmatted either the FORMAT or COPY (BACKUP if using TRSDOS) utility programs. Formatting initializes the diskette which is orginally magnetically erased. The formatting operation writes the sector addresses for every addressable sector plus synch bytes which will be used by the controller to aid it locating specific addresses. In addition the formatting operation specifies the sector size, the number of sectors per track, and the physical order of the sectors .

Mini-floppys are usually formatted with 128, 256, 512, or 1024 byte sectors although other sizes may be formatted. DOS uses the following track format:

Position	Number of Bytes	Contents
Index	14	FF
	6	00
	1	FE (Address marker)
	1	Track Number
	1	Head Number
One Sector.	1	Sector Number
	1	Sector Length Code : 00 = 128 bytes 01 = 256 bytes 02 = 512 bytes 03 = 1024 bytes
Ten per track.		
Sector order is 0,5,1,6, 2,7,3,8, 4,9.	2	CRC
	11	FF : Sector 0 only, 12
	1	A0 : bytes of FF all others
	1	FA (Data Field Mark)
	256	Data
	2	CRC
	12	FF : Except the last (9)
	6	00 : which is followed by
	FE	130 bytes of FF

GAT Sector (Track 11 Sector 1)

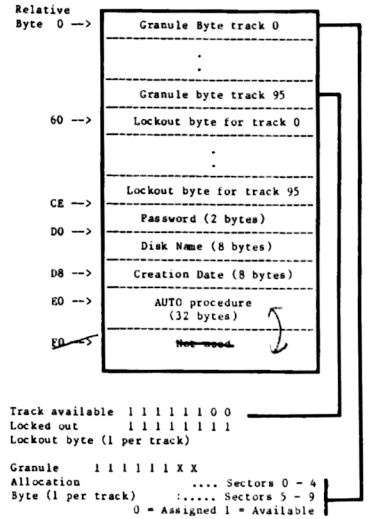
Previously we mentioned the file directory system used by DOS. It is based in part on the ability to dynamically assign disk space on an as-needed basis. Conversly, it must be possible to reuse space which has been released and is no longer needed. The basic vehicle used for keeping track of assigned and available disk space is the Granuale Allocation Table (GAT). Obviously, GAT data must be stored outside the machine if a permanent record is to be maintained. The GAT sector is used for this storage.

With the disk descripton there was a definition for a track and sector. These terms will now be re-defined into the DOS term granule. A granule is 5 sectors or half of a track. It is the minimum unit of disk space that is allocated or de-allocated. Granules are numbered from 0 to N, where N is a function of the number of tracks on a diskette. A record of all granules assigned is maintained in the GAT sector. Recalling the disk dimensions mentioned earlier we can compute the number of granules on a diskette as:

```
Granule = (Number of tracks * 10) / 5
```

Using a 35 track drive with the default DOS disk values of 10 sectors per track and 5 sectors per granule this gives 70 granules per diskette.

The GAT sector is divided into three parts. The first part is the actual GAT table where a record of GAT's assigned is maintained. Part two contains a track lock out table, and part three system initialization information.



Hash Index Table (Track 11 Sector 2)

The Hash Index is a method used to rapidly locate a file without searching all of the directory sectors until it is found. Each file has a unique value computed from its name. This value is called the Hash Code. A special sector in the directory contains the Hash Codes for all

active files on a diskette. When a file is created, its Hash Code is stored in the hash sector in a position that corresponds to the directory for that file. Note, the hash position does not give the file position, just its directory sector position. When a file is KILL'd it code is removed from the hash sector.

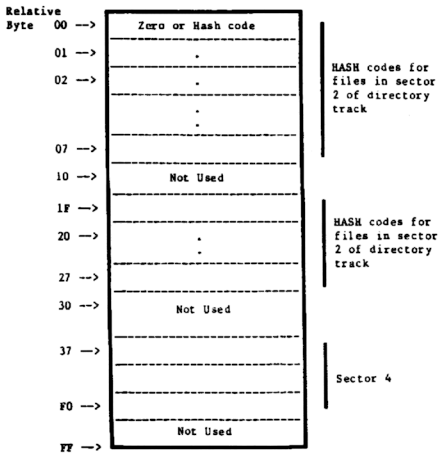
Files are located by first computing their hash value, the Hash Index Sector is then searched for this value. If it is not found then the file does not exist. If the code is found then its position in the Hash Index Sector is used to compute the address for the directory sector containing the file name entry.

Hash code values range from **01** to **FF**. They are computed from an 11 character file name that has been left justified, blank filled. Any file name extension is the last three characters of the name. The code used for computing a hash value is shown below:

```
        LD      B,11        NO. OF CHARS TO HASH
        LD      C,0         ZERO HASH REGISTER
LOOP    LD      A,(DE)      GET ONE CHAR OF NAME
        INC     DE          BUMP TO NEXT CHAR
        XOR     C           HASH REG. XOR. NEXT CHAR
        RLCA                2*(HR. XOR. NC)
        LD      C,A         NEW HR
        DJNZ    LOOP        HASH ALL CHARS
        LD      A,C         GET HASH VALUE
        OR      A           DON'T ALLOW ZERO
        JMP     DONE        EXIT, HASH IN A
        INC     A           FORCE HASH TO 1
DONE      .                 EXIT, HASH IN A
          .
```

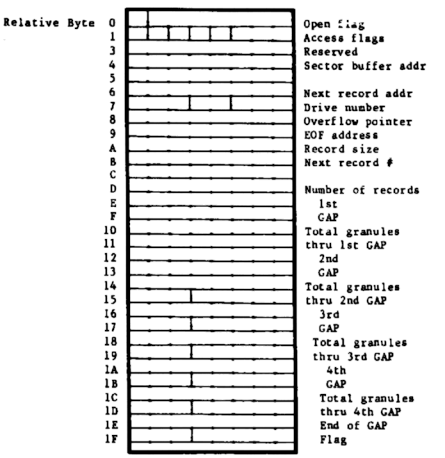
Space for codes in the Hash Sector is assigned sequentially beginning at an arbitrary point. If the hash sector is full a DOS error code of **1A** is given otherwise the sector is scanned in a circular manner until the first available (zero) entry is found.

Not all words in the Hash Sector are used. Addresses in the range **10 - 1F, 30 - 3F, 50 - 5F** are excluded. Only those addresses ending in the digits **00 - 07, 20 - 27** etc are assigned. This speeds the computation of the directory sector number from the hash code value address. The Hash Sector is shown below.



Disk DCB

Each disk file has associated with it a 32 byte DCB which is defined in the user's memory space. When the file is opened the DCB must contain the file name, a name extension if any, and an optional drive specification. As part of the OPEN processing the DCB is initialized for READ and WRITE operations by copying portions of the directory entry into the DCB. After initialization the DCB appears as shown.



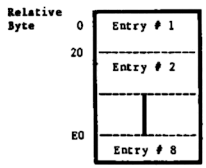
where

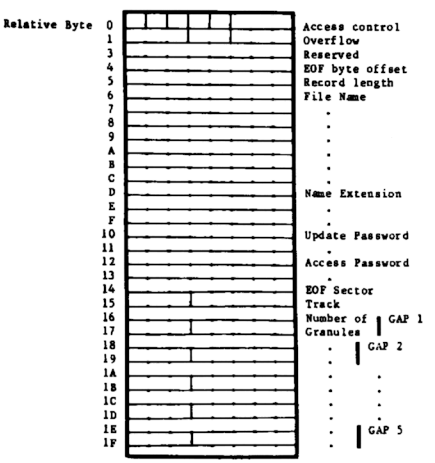
```
BYTE  0   bits 0-6 : reserved
          bit 7   : 0 = file not opened
                    1 = file opened

BYTE  1   bits 0-2 : access permission flag.
          bit  3   : reserved
          bit  4   : 0 = sector buffer available
                     1 = flush sector buffer befor using
          bit  5   : 0 = look for record in current buffer
                     1 = unconditionally read next sector
          bit  6   : reserved
          bit  7   : 0 = sector I/O
                     1 = logical record I/O
BYTE  2       reserved
BYTE  3 - 4   sector buffer address in LSB/MSB order
BYTE  5       pointer to next record in buffer
BYTE  6       drive number
BYTE  7       bits 0-3  sector number - 2 of  overflow entry
              bits 3-4  reserved
              bits 5-7  offset/16 to primary entry in directory
BYTE  8         pointer to end of file in last sector
BYTE  9         record size
BYTE  10 - 11 next record number in LSB/MSB format
BYTE  12 - 13 number of records in file
BYTE  14 - 15 first GAP
BYTE  16 - 17 total granules assigned thru first
BYTE  18 - 19 second GAP
BYTE  20 - 21 total granules assigned thru second GAP
BYTE  22 - 23 third GAP
BYTE  24 - 25 total granules assigned thru third GAP
BYTE  26 - 27 fourth GAP
BYTE  28 - 29 total granules assigned thru fourth GAP
BYTE  30 - 31 end of GAP string flag (FFFF)
```

Directory Sector (Track 11 Sector 3 - Track 11 Sector 9)

Directory sectors contain file descriptions used when accessing a disk file. These descriptions contain among other things the file name, passwords, and a list of the disk addresses occuppied by the file. The directory sectors are divided into eight fixed-length partitions of thirty two bytes each. Each partition contains one file description. Empty partitions are indicated by a flag in the first byte of the partition.

Space in the directory is assigned when a file is initially created using a DOS OPEN or INIT call. There is no particular order in the way space is assigned because the directory sector number used is determined by a hash code derived from the file name. Partition space in the sector is assigned in sequential order.





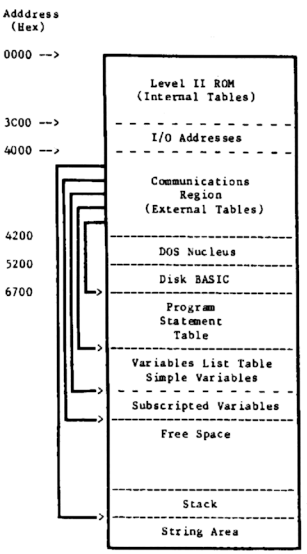

```
BYTE 0    bits 0-2 = file access control flags
          000 - unrestricted access
          001 - KILL/RENAME/WRITE/READ/EXECUTE access
          010 - RENAME/WRITE/READ/EXECUTE access
          011 - reserved
          100 - WRITE/READ/EXECUTE access
          101 - READ/EXECUTE access
          110 - EXECUTE access only
          111 - restricted file no access

          bit 3  = 0, file is displayable. 1, file is invisible.
          bit 4  =  0, this entry is available. 1, entry is
          bit 5  =  reserved
          bit 6  =  0, user file. 1, SYSTEM file.
          bit 7  =  0, primary entry. 1, overflow entry.

BYTE  1   used for overflow entries only.
          Bits 0 - 3 byte offset/10 in primary sector to the entry
          for this file
          Bits 4 - 7  sector number - 2 of primary entry.
BYTE  2   Reserved
BYTE  3   Bits 0 - 7  byte offset to end of file in last sector.
BYTE  4   Bits 0 - 7  record length.

BYTES  5 - 12   File name in ASCII, left justified, blank filled.
BYTES  13 - 15  File name extension in ASCII left justified, blank filled
BYTES  16 - 17  Update password (encoded).
BYTES  18 - 19  Access password (encoded).
BYTES  20 - 21  Last sector number in file. LSB/MSB order.
BYTES  22 - 31  Five two-byte entries called Granule Assignment
                Pairs (GAPs). Each GAP consists of a starting track numbe
                (byte 1) and a count of the number of  consecutively
                assigned granules (byte 2). A string of these GAP's in
                proper order define the disk addresses assigned
                to a file. The end of a GAP string will be signaled by
                a FF in bytes 1 and 2 if there are no more than five
                GAP assigned, for an FE followed by the disk address of
                another directory sector containing the remainder of
                the GAP's. The directory entry containing the overflow
                GAP's is called an overflow entry and contains only the
                continuation of the GAP string. There is no limit to the
                number of overflow entries that may be assigned.
                GAP bytes are formatted as shown below

1st Byte: Bits 0 - 7 contain one of the following:
  a) If the contents of 1st byte is less than FE it is assumed
     to be a track number.
  b) An FF if there are no more GAP's. This is the end of a GAP string
  c) An FE if there are more GAP entries in an overflow sector.
     The next byte contains the overflow sector address.

2nd Byte: The intrepretation of this byte depends on the contents of
  the preceeding byte. If = FF, then this byte is not contains an FF.
  If preceeding byte = FE, then:
  holds in bits  0 - 3 the sector number - 2 of overflow sector.
  bits  4 - 7 the byte offset/10 in the overflow sector to the
  entry with the remainder of the GAPs'.
  If preceeding byte < FE, then this byte has in bits 0 - 3 the number of
  consecutive granules minus 1. This value varies from 0 up to 12.
  Bit 4 = a flag indicating whether the first or second granule in
  the starting track has been assigned. If bit 4 = 0, then the
  first granule was assigned. if bit 4 = 1, then the second granule
  starts with sector.
  5) was assigned.
```

Following is an example of a GAP string:

```
byte 22: 23       file starts on track 23
byte 23: 06       there are 7 granules assigned
                  TRK (23) S(0-9), TRK (24) S(0-9)
                  TRK (25) S(0-0), TRK (26) S(0-4)

-------------------------------------------------------

byte 24: 15       file continues on track 15
byte 25: 23       for 4 granules
                  TRK (15) S(5-9), TRK (16) S(0-9)
                  TRK (17) S(0-4)

-------------------------------------------------------

byte 26: FF       end of GAP string
byte 27: FF       end of GAP string

-------------------------------------------------------
```

Chapter 4

Addresses & Tables



Level II Internal Tables

Internal tables are those lists and tables that are resident in the Level II system. Since they are ROM resident their contents and address are fixed. They are used by BASIC for syntax analysis, during expression evaluation, for data conversions, and while executing such statements as FOR and IF.

Reserved Word List (1650 - 1821)

This table contains all of the word reserved for use by the BASIC interpreter. Each entry contains a reserved word with bit 8 turned on. During the Input Phase the incoming line is scanned for words in this list. Any occurance of one is replaced by a token representing it. The token is computed as 80 plus the index into the table where the word was found. A list of those words and their token values follows:

Word	Token	Word	Token	Word	Token
END	80	FOR	81	RESET	82
SET	83	CLS	84	*CMD	85
RANDOM	86	NEXT	87	DATA	88
INPUT	89	DIM	8A	READ	8B
LET	8C	GOTO	8D	RUN	8E
IF	8F	RESTORE	90	GOSUB	91
RETURN	92	REM	93	STOP	94
ELSE	95	TRON	96	TROFF	97
DEFSTR	98	DEFINT	99	DEFSNG	9A
DEFDBL	9B	*LINE	9C	EDIT	9D
ERROR	9E	RESUM	9F	OUT	A0
ON	A1	*OPEN	A2	*FIELD	A3
*GET	A4	*PUT	A5	*CLOSE	A6
*LOAD	A7	*MERGE	A8	*NAME	A9
*KILL	AA	*LSET	AB	*RSET	AC
*SAVE	AD	SYSTEM	AE	LPRINT	AF
*DEF	B0	POKE	B1	PRINT	B2
CONT	B3	LIST	B4	LLIST	B5
DELETE	B6	AUTO	B7	CLEAR	B8
CLOAD	B9	CSAVE	BA	NEW	B8
TAB(	BC	TO	BD	*FN	BE
USING	BF	VARPTR	C0	USR	C1
ERL	C2	ERR	C3	STRING$	C4
INSTR	C5	POINT	C6	*TIME$	C7
MEM	C8	INKEY$	C9	THEN	CA
NOT	CB	STEP	CC	+	CD
-	CE	*	CF	/	D0
UP ARROW	D1	AND	D2	OR	D3
>	D4	=	D5	<	D6
SGN	D7	INT	D8	ABS	D9
FRE	DA	INP	DB	POS	DC
SQR	DD	RND	DE	LOG	DF
EXP	E0	COS	E1	SIN	E2
TAN	E3	ATN	E4	PEEK	E5
*CVI	E6	*CVS	E7	*CVD	E8
*EOF	E9	*LOC	EA	*LOF	EB
*MKI$	EC	*MKS$	ED	CINT	EF
CSNG	F0	CDBL	F1	FIX	F2
LEN	F3	STR$	F4	VAL	F5
ASC	F6	CHR$	F7	LEFT$	F8
RIGHT$	F9	*MID$	FA	'	FB

* Disk BASIC tokens

Precedence Operator Values (189A - 18A0)

This table contains numeric values used to determine the order of arithmetic operations when evaluating an expression. As the expression is scanned each operator/ operand pair plus the precedence value for the previous operand is stored on the stack. When an operator of higher precedence than the preceeding one is found the current operation is performed giving an intermediate value that is carried forward on the stack. The values shown for relational operations are computed rather than being derived from a table look-up.

Operator	Function	Precedence Value
UP ARROW	(Exponent)	7F
*	(Multiplication)	7C
/	(Division)	7C
+	(Addition)	79
-	(Subtraction)	79
ANY	(Relational)	64
AND	(Logical)	50
OR	(Logical)	46
<=	(Relational)	06
<>	(Relational)	05
>=	(Relational)	03
<	(Relational)	04
=	(Relational)	02
>	(Relational)	01

Arithmetic Routines (18AB - 18C8)

There are really three tables back-to-back here. They are used during expression evaluation to compute intermediate values when a higher precedence operator is found.

Arithmetic Routine Addresses

	Integer	Single Precision	Double Precision	String
Addition	0BD2	0716	0C77	298F
Subtraction	0BC7	0713	0C70	NONE
Multiplication	0BF2	0847	0DA1	NONE
Division	2490	08A2	0DE5	NONE
Comparison	0A39	0A0C	0A78	NONE

Data Conversion Routines (18A1 - 18AA)

These routines convert the value in WRA1 from one mode to another. They are called by the expression evaluator when an intermediate computation has been made, and the result needs to be make compatible with the rest of the expression.

Conversion Routine Addresses

Destination Mode	Address
String	0AF4
Integer	0A7F
Single Precision	0A31
Double Precision	0ADB

Verb Action Addresses

Verb Action Routines (1822 - 1899)

There are two Verb Action Address Lists. The first one is used by the execution driver when beginning execution of a new statement. It contains address of of verb routines for the tokens **80 - BB**. The first token of the statement is used as an index in the range of **0 - 60** into the table at **1822 - 1899** to find the address of the verb routine to be executed. If the statement does not begin with a token control goes to assignment statement processing. The second table contains the addresses of verb routines which can only occur on the right side of an equals sign. If during the expression evaluation stage a token in the range of **D7 - FA** is encountred it is used as an index into the table at **1608 - 164F**, where the address of the verb routine to be executed is found. There is no address list for the tokens **BC - D6** because they are associated with and follow other tokens that expect and process them.

Table Address 1822 - 1899)

Token	Verb	Address	Token	Verb	Address
80	END	1DAE	81	FOR	1CA1
82	RESET	0138	83	SET	0135
84	CLS	01C9	85	CMD	4135
86	RANDOM	01D3	87	NEXT	2286
88	DATA	1F05	89	INPUT	219A
8A	DIM	2608	8B	READ	21EF
8C	LET	1F21	8D	GOTO	1EC2
8E	RUN	1EA3	8F	IF	2039
90	RESTORE	1D91	91	GOSUB	1EB1
92	RETURN	1EDE	93	REM	1F07
94	STOP	1DA9	95	ELSE	1F07
96	TRON	1DF7	97	TROFF	1DF8
98	DEFSTR	1E00	99	DEFINT	1E03
9A	DEFSNG	1E06	9B	DEFDBL	1E09
9C	LINE	41A3	9D	EDIT	2E60
9E	ERROR	1FF4	9F	RESUME	1FAF
A0	OUT	2AFB	A1	ON	1FC6
A2	OPEN	4179	A3	FIELD	417C
A4	GET	417F	A5	PUT	4182
A6	CLOSE	4185	A7	LOAD	4188
A8	MERGE	418B	A9	NAME	418E
AA	KILL	4191	AB	LSET	4197
AC	RSET	419A	AD	SAVE	41A0
AE	SYSTEM	02B2	AF	LPRINT	2067
B0	DEF	41B5	B1	POKE	2CB1
B2	PRINT	206F	B3	CONT	1DE4
B4	LIST	2B2E	B5	LLIST	2B29
B6	DELETE	2BC6	B7	AUTO	2008
B8	CLEAR	1E7A	B9	CLOAD	2C1F
BA	CSAVE	2BF5	BB	NEW	1B49

(Table Address 1608 - 164F)

TOKEN	VERB	Address	TOKEN	VERB	Address
D7	SGN	098A	D8	INT	0B37
D9	ABS	0977	DA	FRE	27D4
DB	INP	2AEF	DC	POS	27A5
DD	SQR	13E7	DE	RND	14C9
DF	LOG	0809	E0	EXP	1439
E1	COS	1541	E2	SIN	1547
E3	TAN	15A8	E4	ATN	15BD
E5	PEEK	2CAA	E6	CVI	4152
E7	CVS	4158	E8	CVD	415E
E9	EOF	4161	EA	LOC	4164
EB	LOF	4167	EC	MKI$	416A
ED	MKS$	416D	EE	MKD$	4170
EF	CINT	0A7F	F0	CSNG	0AB1
F1	CDBL	0DAB	F2	FIX	0B26
F3	LEN	2A03	F4	STR$	2B36
F5	VAL	2AC5	F6	ASC	2A0F
F7	CHR$	2A1F	F8	LEFT$	2A61
F9	RIGHT$	2A91	FA	MID$	2A9A

Error Code Table (18C9 - 18F6)

Error codes printed under Level II are intrepreted by using the error number as in index into a table of two letter error abbreviations. The format of the error code table is as follows:

Error Number	Code	Cause	Originating Address
0	NF	NEXT WITHOUT FOR	22C2
2	SN	SYNTAX ERROR (NUMEROUS CAUSES)	DA,2C7,EEF 1C9E,1D32,1E0E 1E66,2022,235B 2615,2AE9,2DE2
4	RG	RETURN WITHOUT GOSUB	1EEC
6	OD	OUT OF DATA (READ)	2214,22A2
8	FC	NUMEROUS	1E4C
A	OV	NUMERIC OVERFLOW	7B2
C	OM	OUT OF MEMORY	197C
E	UL	MISSING LINE NUMBER	1EDB
10	BS	INDEX TOO LARGE	273F
12	DD	DOUBLY DEFINED SYMBOL	2735
14	/0	DIVISION BY 0	8A5,DE9,1401
16	ID	INPUT USE INCORRECT	2833
18	TM	VAIRABLE NOT A STRING	AF8
1A	OS	OUT OF STRING SPACE	28DD
1C	LS	STRING TOO LONG	29A5
1E	ST	LITERAL STRING POOL TABLE FULL	28A3
20	CN	CONTINUE NOT ALLOWED	1DEB
22	NR	RESUME NOT ALLOWED	198C
24	UE	INVALID ERROR CODE	2005
26	UE	INVALID ERROR CODE	2005
28	MO	OPERAND MISSING	24A2
2A	FD	DATA ERROR ON CASSETTE	218C
2C	L3	DISK BASIC STATEMENT ATTEMPTED UNDER LEVEL II	12DF

Level II External Tables

External tables used by Level II are those which are kept in RAM. They are kept there because their contents and size, as well as their address, may change. A pointer to each of the External tables is maintained in the Communications Region.

Mode Table (4101 - 411A)

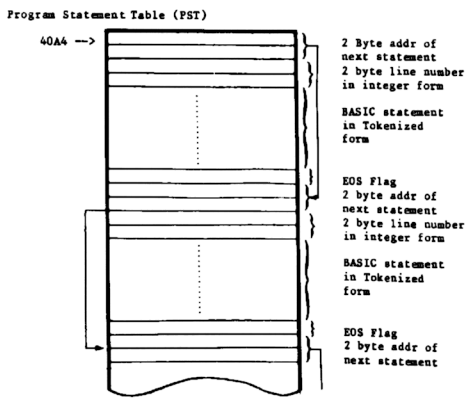
This table is used by the BASIC intrepreter to determine the data type mode (integer, string, single or double precision) for each variable. Although it never moves its contents may change when a DEF declaration is encountered, and therefore it must be in RAM. It is the only RAM table with a fixed address and consequently there is no pointer to it in the Communications Region. The table is 26 decimal words long and is indexed by using the first character of a variable name as an index. Each entry in the table contains a code indicating the variable type e.g. 02 - integer, 03 - string, 04 - single precision, 08 - double precision.

The mode table is initialized during the IPL sequence to 04 for all variables. It appears as:

Address	Letter	Type	Address	Letter	Type
4101	A	04	4102	B	04
4103	C	04	4104	D	04
4105	E	04	4106	F	04
4107	G	04	4108	H	04
4109	I	04	410A	J	04
410B	K	04	410C	L	04
410D	M	04	410E	N	04
410F	O	04	4110	P	04
4111	Q	04	4112	R	04
4113	S	04	4114	T	04
4115	U	04	4116	V	04
4117	W	04	4118	X	04
4119	Y	04	411A	Z	04

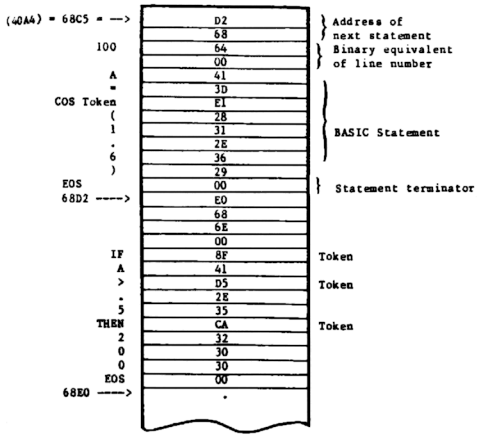
Program Statement Table (PST)

The Program Statement Table contains BASIC statements entered as a program. Since it is RAM resident and its orgin may change from system to system there is a pointer to it in the Communications Region at address **40A4**. As each line is entered it is tokenized and stored in the PST. Statements are stored in ascending order by line number regardless of the order in which they are entered. Each entry begins with a two byte pointer to the next line followed by a two byte integer equivalent of the line number then the text of the BASIC statement. The body of the statement is terminated with a single byte of zeros called the End Of Statement or EOS flag. The ending address of the PST is contained in **40F9**. It is terminated by two bytes of zeros.



Shown below are two statements and their representation in the PST:

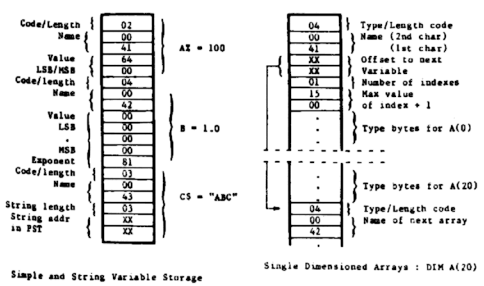
```
100  A = COS ( 1.6 )
110  IF A>.5 THEN 500
```



Variable List Table (VLT)

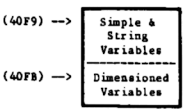
This table contains all variables assigned to a BASIC program. Internally the table is divided into two sections. Section one contains entries for all non-subscripted and string variables while section two contains the values for all subscripted variables. Like the PST the VLT is RAM resident and it has two pointers in the Communications Region. Location **40F9** contains the address of the first section, and **40FB** contains the address of section two. The starting address of the VLT is considered as the end of the PST.

Regardless of which section a variable is defined in, the first three bytes of each entry have the same format. Byte one has a type code (2,3,4,or 8), which doubles as the length of the entry. Bytes two and three contain the variable name in last/first character order. Following this is the value itself in LSB/MSB order, or if it as a string variable a pointer to the string in the String Area.



Section two contains all dimensioned arrays. These entries have the same three byte header followed by a another header which defines the extents of the array. The array is stored after the second header in column-major order.

Variables are assigned space in the VLT as they are encountered (in a DIM statement or in any part of an assignment statement). There is no alphabetical ordering. Because space is assigned on demand it is possible for previously defined variables to be moved down. For example, if A, B, and C(5) were defined followed by D, C(5) would be moved down because section one would be increased for D. This would force section two to be moved.



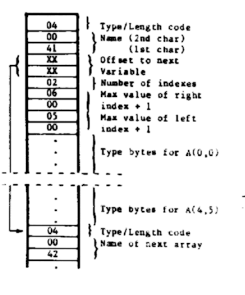
Arrays are stored in column-major order. In that order the left most index varies the fastest. For example the array A(2,3) would be stored in memory as:

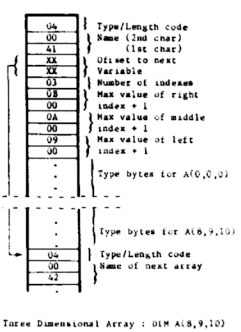
A(0,0)
A(1,0)
A(2,0)
.
.
.
A(0,3)
A(1,3)
A(2,3)

An index for any element can be computed using the formula:

```
INDEX = (((LRI*0)+URI)*LMI)+UMI)*LLI)+ULI

where

          LRI = limit of right index
          LMI = limit of middle index
          LLI = limit of left index

          URI = user's current right index
          UMI = user's current middle index
          ULI = user's current left index
```

The code used to compute these indexes may be found at address **2595** to **27C8**.

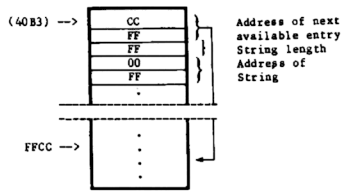




Literal String Pool (40D2)

This table is used by BASIC to keep track of intermediate strings which result from operations such as string addition or some print operations. The table has eleven three byte entries which are assigned sequentially. The start of the table has a two byte pointer to the next available entry. It is initialized during IPL to point to the head of the list.

Each entry contains the length and the address of a string which is usually (although not necessarily) in the PST. Entries are assigned in a top down fashion and released in a bottom up manner. A pointer to the next available entry is kept in 40B3. If the table overflows an ST error is given.



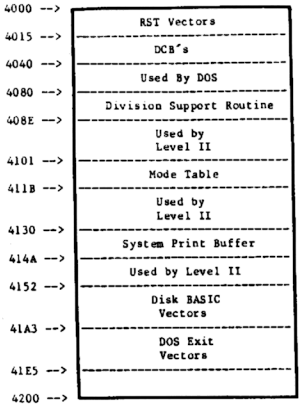
Literal String Pool

Communications Region (4000 - 4200)

The Communications Region has been defined as RAM locations 4000 to 4200. These addresses give the definition an air of precision that is not warranted. In reality only a portion of the area is used in the sense given to the term Communications Region. Those boundaries were chosen because they represent the end of ROM and the approximate starting address of DOS in RAM. In a Level II system without disk there would be no DOS and the RAM tables such as the PST, VLT, etc. would begin at a much lower address. But they would still be above 4200 so it is safe to think of that region as reserved.

The Communications Region has many uses other than those mentioned so far. The following diagram shows the major areas discussed up to this point. Following it is a description of all bytes in the Communications Region and their known use.

Communications Region



Address	Level II Contents	DOS Contents	Description
4000	JP 1C96	....	RST 8 VECTOR
4003	JP 1D78	....	RST 10 VECTOR
4006	JP 1C90	....	RST 18 VECTOR
4009	JP 25D9	....	RST 20 VECTOR
400C	RET	JP 4BA2	RST 28 DOS REQUEST PROCESSING
400F	RET	JP 44B4	LOAD DEBUG (LD A,XX/RST 28)
4012	DI/RET	CALL 4518	RST 38 INTERRUPT SERVICE CALL
4015	.....		KEYBOARD DCB (8 BYTES)
401D	.....		VIDEO DCB (8 BYTES)
4025	.....		PRINTER DCB (8 BYTES)
402D	JP 5000	JP 4400	MAKE SYS1 (10) DOS REQUEST
4030	RST 0	LD A,A3	DOS REQUEST CODE FOR SYS1
4032	LD A,0	RST 28	WRITE 'DOS READY' MSG
4033	RET	JP 44BB	CALL DEVICE DRIVER ALA DOS
4036	.....		KEYBOARD WORK AREA USED BY SYS0 AND KEYBOARD DRIVER
403D	.....		DISPLAY CONTROL WORD (U/L CASE)
403E	.....		USED BY DOS
403F	.....		USED BY DOS
4040	.....		SYSTEM BST'S
4041	.....		SECONDS
4042	.....		MINUTES
4043	.....		HOURS
4044	.....		YEAR
4045	.....		DAY
4046	.....		MONTH
4047	.....		LOAD ADDRESS FOR SYSTEM UTILITIES 2 BYTES, INITIALIZED TO 5200 BY SYS0/SYS
4049	.....		MEMORY SIZE. COMPUTED BY SYS0/SYS
404A	.....		RESERVED
404B	.....		CURRENT INTERRUPT STATUS WORD
404C	.....		INTERRUPT SUBROUTINE MASK
404D	.....		RESERVED (INTERRUPT BIT 0)
404F	.....		RESERVED (INTERRUPT BIT 1)
4051	.....		COMMUNICATIONS INTERRUPT SUBROUTINE
4053	.....		RESERVED (INTERRUPT BIT 3)
4055	.....		RESERVED (INTERRUPT BIT 4)
4057	.....		RESERVED (INTERRUPT BIT 5)
4059		45F7	ADDR OF DISK INTERRUPT ROUTINE
405B		4560	ADDR OF CLOCK INTERRUPT ROUTINE
405D	.....		STACK DURING IPL
407D	.....		START OF STACK DURING ROM IPL
407E	.....		RESERVED
407F	.....		RESERVED
4080	.....		SUBTRACTION ROUTINE USED BY DIVISION CODE. CODE IS MOVED FROM '18F7' - '1904' DURING NON-DISK IPL OR BY BASIC UTILITY FOR DISK SYSTEMS.

408E	.....	CONTAINS ADDRESS OF USER SUBROUTINE.
4090	.....	RANDOM NUMBER SEED
4093	.....	IN A,00
4096	.....	OUT A,00
4099	.....	HOLDS LAST CHAR TYPED AFTER BREAK
409A	.....	FLAG (SIGNALS RESUME ENTERED)
409B	.....	NO. OF CHARS. IN CURRENT PRINT LINE
409D	.....	OUTPUT DEVICE CODE (1-PRINTER 0-VIDEO, MINUS 1-CASSETTE)
409D	.....	SIZE OF DISPLAY LINE (VIDEO)
409E	.....	SIZE OF PRINT LINE
409F	.....	RESERVED
40A0	.....	ADDR OF STRING AREA BOUNDARY
40A2	.....	CURRENT LINE NUMBER
40A4	.....	ADDR OF PST
40A6	.....	CURSOR POSITION
40A7	.....	ADDR OF KEYBOARD BUFFR.
40A9	.....	0 IF CASSETTE INPUT, ELSE NON-ZERO
40AA	.....	RANDOM NUMBER SEED
40AB	.....	VALUE FROM REFRESH REGISTER
40AC	.....	LAST RANDOM NUMBER (2 BYTES)
40AE	.....	FLAG: 0 - LOCATE NAMED VARIABLE -1 - CREATE ENTRY FOR NAMED VARIABLE
40AF	.....	TYPE FLAG FOR VALUE IN WRA1. 2 - INTEGER 3 - STRING 4 - SINGLE PRECISION 8 DOUBLE PRECISION
40B0	.....	HOLDS INTERMEDIATE VALUE DURING EXPRESSION EVALUATION
40B1	.....	MEMORY SIZE
40B2	.....	RESERVED
40B3	.....	ADDR OF NEXT AVAILABLE LOC. IN LSPT.
40B5	.....	LSPT (LITERAL STRING POOL TABLE)
40D2	.....	END OF LSPT
40D3	.....	THE NEXT 3 BYTES ARE USED TO HOLD THE LENGTH AND ADDR OF A STRING WHEN IT IS MOVED TO THE STRING AREA.
40D6	.....	POINTER TO NEXT AVAILABLE LOC. IN STRING AREA.
40D8	.....	1: INDEX OF LAST BYTE EXECUTED IN CURRENT STATEMENT. 2: EDIT FLAG DURING PRINT USING
40DA	.....	LINE NO. OF LAST DATA STATEMENT READ
40DC	.....	FOR FLAG (1 = FOR IN PRGORESS 0 = NO FOR IN PROGRESS)
40DD	.....	0 DURING INPUT PHASE, ZERO OTHERWISE
40DE	.....	READ FLAG: 0 = READ STATEMENT ACTIVE 1 = INPUT STATEMENT ACTIVE ALSO USED IN PRINT USING TO HOLD SEPERATOR BETWEEN STRING AND VARIABLE
40DF	.....	HOLDS EXECUTION ADDR FOR PGM LOADED WITH DOS REQUEST
40E1	.....	AUTO INCREMENT FLAG 0 = NO AUTO MODE NON-ZERO HOLDS NEXT LINE NUMBER
40E2	.....	CURRENT LINE NUMBER IN BINARY (DURING INPUT PHASE)
40E4	.....	AUTO LINE INCREMENT
40E6	.....	DURING INPUT: ADDR OF CODE STRING FOR CURRENT STATEMENT. DURING EXECUTION: LINE NO. FOR CURRENT STATEMENT
40E8	.....	DURING EXECUTION: HOLDS STACK POINTER VALUE WHEN STATEMENT EXECUTION BEGINS
40EA	.....	LINE NO. IN WHICH ERROR OCCURED
40EC	.....	LINE NO. IN WHICH ERROR OCCURED.
40ED	.....	LAST BYTE EXECUTED IN CURRENT STATEMENT
40EF	.....	ADDR OF POSITION IN ERROR LINE
40F0	.....	ON ERROR ADDRESS
40F2	.....	FLAG. FF DURING ON ERROR PROCESSING CLEARED BY RESUME ROUTINE
40F3	.....	ADDR OF DECIMAL POINT IN FBUF
40F5	.....	LAST LINE NUMBER EXECUTED SAVED BY STOP/END
40F7	.....	ADDR OF LAST BYTE EXECUTED DURING ERROR
40F9	.....	ADDR OF SIMPLE VARIABLES
40FB	.....	ADDR OF DIMENSIONED VARIABLES
40FD	.....	STARTING ADDRESS OF FREE SPACE LIST (FSL)
40FF	.....	POINTS TO BYTE FOLLOWING LAST CHAR READ DURING READ STMNT EXECUTION
4101	.....	VARIABLE DECLARATION LIST. THERE ARE 26 ENTRIES (1 FOR EACH LETTER OF THE ALPHABET) EACH ENTRY CONTAINS A CODE INDICATING DEFAULT MODE FOR VARIABLES STARTING WITH THAT LETTER.
411A	.....	END OF DECLARATION LIST
411B	.....	TRACE FLAG (0 = NO TRACE, NON-ZERO = TRACE)
411C	.....	TEMP STORAGE USED BY NUMERIC ROUTINES WHEN UNPACKING A FLOATING POINT NUMBER. USUALLY IT HOLDS THE LAST BYTE SHIFTED OUT OF THE LSB POSITION
411D	.....	WRA1 - LSB OF DBL PREC. VALUE
411E	.....	WRA1 - DBL PREC. VALUE
411F	.....	WRA1 - DBL PREC VALUE
4120	.....	WRA1 - DBL PREC VALUE
4121	.....	WRA1 - LSB OF INTEGER SINGLE PREC
4122	.....	WRA1
4123	.....	WRA1 - MSB FOR SINGLE PREC
4124	.....	WRA1 - EXPONENT FOR SINGLE PREC
4125	.....	SIGN OF RESULT DURING MATH & ARITHMETIC OPERATIONS
4126	.....	BIT BUCKET USED DURING DP ADDITION
4127	.....	WRA2 - LSB
4128	.....	WRA2
4129	.....	WRA2
412A	.....	WRA2
422B	.....	WRA2
412C	.....	WRA2
412D	.....	WRA2 - MSB
412E	.....	WRA2 - EXPONENT
412F	.....	NOT USED
4130	.....	START OF INTERNAL PRINT BUFFER USED DURING PRINT PROCESSING
4149	.....	LAST BYTE OF PRINT BUFFER
414A	.....	TEMP. STORAGE USED BY DBL PRECISION DIVISION ROUTINE. HOLDS DIVISOR
4151	.....	END OF TEMP AREA

LOCATIONS 4152 THRU 41E2 CONTAIN DOS EXITS AND DISK BASIC EXITS. ON NON-DI
SYSTEMS THESE LOCATIONS ARE INITIALIZED TO RETURNS (RET'S) WHILE ON DISK
BASED SYSTEMS THEY WILL BE INITIALIZED AS SHOWN.

4152	...RET. .JP	5E46.....	DISK BASIC EXIT (CVI)
4155	...RET. .JP	558E.....	DISK BASIC EXIT (FN)
4158	...RET. .JP	5E49.....	DISK BASIC EXIT (CVS)
415B	...RET. .JP	5655.....	DISK BASIC EXIT (DEF)
415E	...RET. .JP	5E4C.....	DISK BASIC EXIT (CVD)
4161	...RET. .JP	61EB.....	DISK BASIC EXIT (EOF)
4164	...RET. .JP	6231.....	DISK BASIC EXIT (LOC)
4167	...RET. .JP	6242.....	DISK BASIC EXIT (LOF)
416A	...RET. .JP	5E20.....	DISK BASIC EXIT (MKI$)
416D	...RET. .JP	5E30.....	DISK BASIC EXIT (MKS$)
4170	...RET. .JP	5E33.....	DISK BASIC EXIT (MKD$)
4173	...RET. .JP	56C4.....	DISK BASIC EXIT (CMD)
4176	...RET. .JP	5714.....	DISK BASIC EXIT (TIME$)
4179	...RET. .JP	6349.....	DISK BASIC EXIT (OPEN)
417C	...RET. .JP	60AB.....	DISK BASIC EXIT (FIELD)
417F	...RET. .JP	627C.....	DISK BASIC EXIT (GET)
4182	...RET. .JP	627B.....	DISK BASIC EXIT (PUT)
4185	...RET. .JP	606F.....	DISK BASIC EXIT (CLOSE)
4188	...RET. .JP	5F78.....	DISK BASIC EXIT (LOAD)
418B	...RET. .JP	600B.....	DISK BASIC EXIT (MERGE)
418E	...RET. .JP	6346.....	DISK BASIC EXIT (NAME)
4191	...RET. .JP	63C0.....	DISK BASIC EXIT (KILL)
4194	...RET. .JP	58B7.....	DISK BASIC EXIT (&)
4197	...RET. .JP	60E6.....	DISK BASIC EXIT (LSET)
419A	...RET. .JP	60E5.....	DISK BASIC EXIT (RSET)
419D	...RET. .JP	582F.....	DISK BASIC EXIT (INSTR)
41A0	...RET. .JP	6044.....	DISK BASIC EXIT (SAVE)
41A3	...RET. .JP	5756.....	DISK BASIC EXIT (LINE)
41A6	...RET. .JP	5679.....	DISK BASIC EXIT (USR)

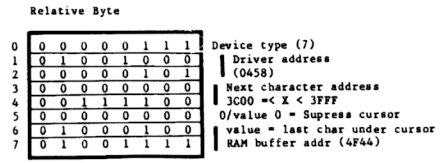
THE FOLLOWING ADDRESSES ARE THE DOS EXIT ADDRESSES.

41A9	...RET. .JP	XXXX.....	DOS EXIT FROM
41AC	...RET. .JP	5FFC.....	DOS EXIT FROM 1A1C
41AF	...RET. .JP	598E.....	DOS EXIT FROM 0368
41B2	...RET. .JP	6033.....	DOS EXIT FROM ROM address 1AA1
41B5	...RET. .JP	5BD7.....	DOS EXIT FROM ROM address 1AEC
41B8	...RET. .JP	58BC.....	DOS EXIT FROM ROM address 1AF2
41BB	...RET. .JP	60A1.....	DOS EXIT FROM ROM address 1B8C
41BE	...RET. .JP	577C.....	DOS EXIT FROM ROM address 2174
41C1	...RET. .JP	59CD.....	DOS EXIT FROM ROM address 032C
41C4	...RET. .JP	XXXX.....	DOS EXIT FROM ROM address 0358
41C7	...RET. .JP	5F78.....	DOS EXIT FROM ROM address 1EA6
41CA	...RET. .JP	5A15.....	DOS EXIT FROM ROM address 206F
41CD	...RET. .JP	5B9A.....	DOS EXIT FROM ROM address 2103
41D0	...RET. .JP	5B99.....	DOS EXIT FROM ROM address 2103
41D3	...RET. .JP	5B65.....	DOS EXIT FROM ROM address 2108
41D6	...RET. .JP	5784.....	DOS EXIT FROM ROM address 219E
41DC	...RET. .JP	5E63.....	DOS EXIT FROM ROM address 222D
41DF	...RET. .JP	579C.....	DOS EXIT FROM ROM address 2278
41D2	...RET. .JP	5B51.....	DOS EXIT FROM ROM address 02B2

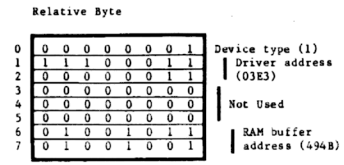
DCB Descriptions

The keyboard, video, and printer DCB'S (Device Control Blocks) are defined in ROM at locations@@EQ2 06E7 - 06FF. They are moved to the address show in the Communications Region during the IPL sequence.

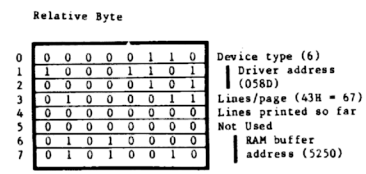
Video DCB (Address 401D)



Keyboard DCB (Address 4015)



Printer DCB (Address 4025)



Interrupt Vectors

Interrupts are a means of allowing an external event to interrupt the CPU and redirect it to execute some specific portion of code. The signal that causes this to happen is called an interrupt and the code executed in response to that interrupt is called a service routine. After the service routine executes it returns control of the CPU to the point where the interrupt occurred and normal processing continues.

In order for interrupts to occur the system must be primed to accecpt them. When the system is primed it is ENABLED which is shorthand for the instruction used to enable the interrupt system (EI-Enable Interrupts). A system that is not enabled is DISABLED and again that is shorthand for the disable instruction (DI-Disable Interrupts). Besides priming the system for interrupts there must be some outside event to stimulate the interrupt. On Level II systems that could be a clock or a disk. Actually both of them generate interrupts - the clock gives one every 25 milliseconds, and the disk on demand for certain operations.

When running a Level II system without disks the interrupts are disabled. It is only when DOS is loaded that interrupts are enabled and service routines to support those interrupts are loaded. Interrupts are disabled at the start of the IPL sequence that is common to Level II and DOS. For Level II they will remain off, but on a DOS system they will be enabled at the end of the initilization in SYSO/SYS.

When an interrupt occurs two things happen. First a bit indicating the exact cause of the interrupt is set in byte 37E0. Second an RST 56H instruction is executed. As a result of the RST (which is like a CALL) the address of the next instruction to be executed is saved on the stack (PUSH'd) and control is passed to location 0038. Stored at 00 38 is a JP 4012. During the IPL sequence 4012 was initialized to:

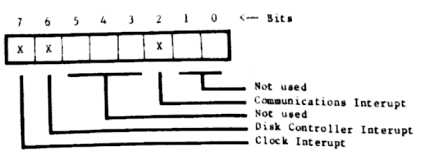
```
4012   DI    Disable further interupts
4013   RET   Return to point of interupt
```

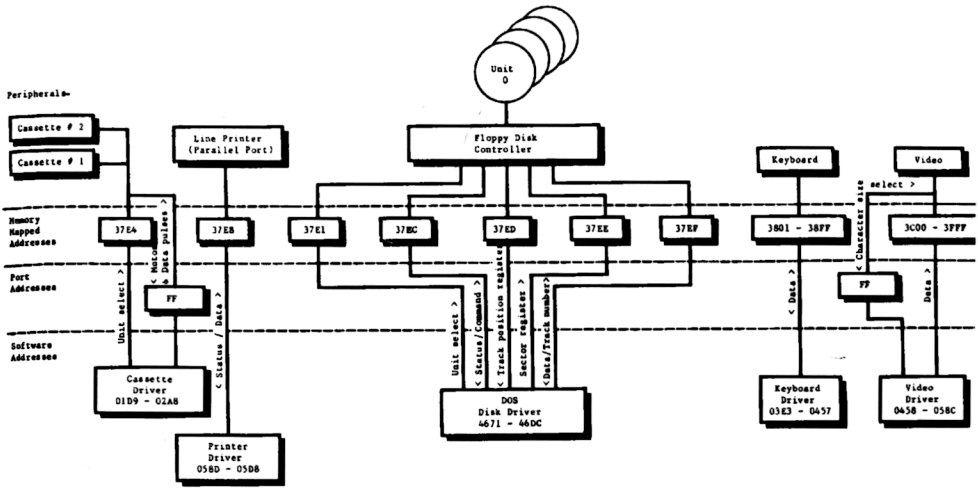
for non-disk systems or:

```
4012   CALL 4518   Service Interupt
```

for disk systems

The service routine at 4518 examines the contents of 37E0 and executes a subroutine for each bit that is turned on and for which DOS has a subroutine. The format of the interupt status word at 37E0 is:





Memory Mapped I/O

DOS maintains an interrupt service mask at 404C that it uses to decide if there is a subroutine to be executed for each of the interrupt status. As released 404C contains a C0 which indicates subroutines for clock and disk interrupts.

The service routine at 4518 combines the status byte and the mask byte by AND'ing them together. The result is used as a bit index into a table of subroutine addresses stored at 404D - 405C. Each entry is a two byte address of an interrupt subroutine. Bit 0 of the index corresponds to the addess at 404D/404E, bit 1 404F/4050, etc.

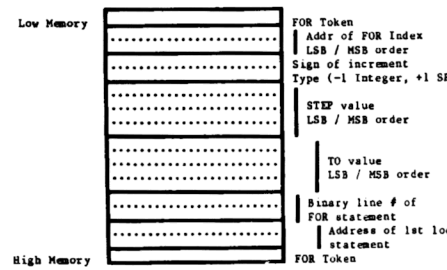
The service routine runs disabled. It scans the interrupt status from left to right jumping to a subroutine whenever a bit is found on. All registers are saved before subroutine entry and a return address in the service routine is PUSH'd onto the stack so a RET instruction can be used to exit the subroutine. When all bits in the status have been tested control returns to the point of interrupt with interrupts enabled.

Stack Frame Configurations

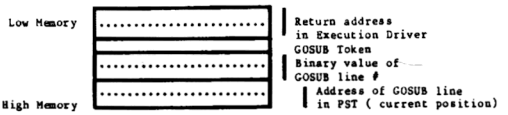
Level II usually uses the Communications Region for temporary storage. There are special cases, however where that is not possible because a routine may call itself (called recursion) and each call would destroy the values saved by the previous call. In those cases the stack is used to save some of the variables. Of course an indexed table could be used, but in these cases the stack serves the purpose.

FOR Statement Stack Frame

All variable addresses associated with a FOR loop are carried on the stack until the loop completes. When a NEXT statement is processed, it searches the stack looking for a FOR frame with the same index address as the current one. The routine that searches the stack is at location 1936. Its only parameter is the address of the current index which is passed in the DE register set. The stack is searched backwards from its current position to the beginning of the stack. If a FOR frame with a matching index address is not found an NF error is generated. The stack frame searched for is given below.



GOSUB Stack Configuration



Expression Evaluation

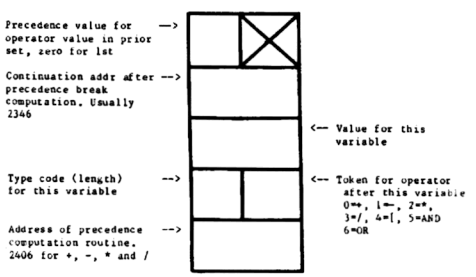
Expression evaluation involves scanning an expression and breaking it intó seperate operations which can be executed in their proper order according to the hierarchary of operators. This means a statement must be scanned and the operations with the highest heiracherial value (called precedence value) must be performed first. Any new terms which result from those operations must be carried forward and combined with the rest of the expression.

The method used for evaluation is an operator precedence parse. An expression is scanned from left to right. Scanning stops as soon as an operator token or EOS is found. The variable to the left of the operator (called the current variable), and the operator (any arithmetic token for - * / or exp) are called a 'set', and are either:

a) pushed onto the stack as a set or,

b) if a precedence break is detected the operation between the previous set pushed onto the stack and the current variable is performed. The result of that operation then becomes the current variable and the previous set is removed from the stack. After the computation another attempt is made to push the new current variable and operator onto the stack as a set.
This step is repeated until the new set is pushed or there are no more sets on the stack with which to combine the current value. In that case the expression has been evaluated.

The variable/operator sets that are pushed on the stack have the following format:



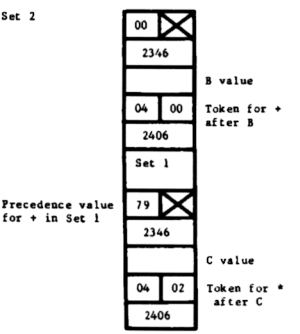
The test for precedence break is simple. If the operator (the token where the scan stopped) has the same or a lower precedence value as the precedence value for the last set pushed on the stack then a break has occured, and an intermediate computation is required. The computation is performed automatically by POPing the last set. When this occurs control is transfered to a routine (usually at 2406) which will perform the operation specified in the set between that value (the one from the set on the stack), and the current variable. The result then becomes the current variable. When the computation is finished control returns to a point where the precedence break test is repeated. This time the set which caused the last break is not there, so the test will be between the same operator as before and the operator in the previous set. If there is no previous set then the current variable and operator are pushed as the next set. Note, an EOS or a non-arithmetic token are treated as precedence breaks.

Assuming no break occurs the curent variable and operator are pushed on the stack as the next set, and the scan of the expession continues from the point where it left off. Let's take an example. Assume we have the expression,

A equals B plus C * D / E 5

Scanning begins with the first character to the right of the equals sign and will stop at the first token (plus). B plus would be pushed as the first set because: a) there was no prior set so there could not have been a precedence break, and b) the scan stopped on an arithmetic token (plus).

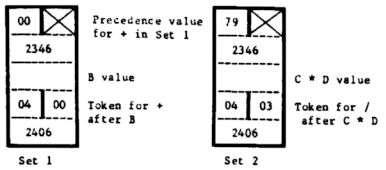
The next scan would stop at the *. Again the variable/operator pair of C* would be pushed this time as set 2 although for slightly different reasons than before. The * precedence value is higher than the plus precedence value already pushed so there is no break. At this time the stack contains,



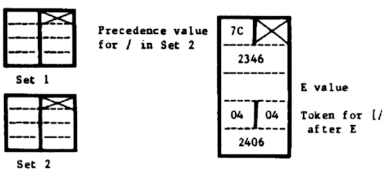
Scan three would stop on the / following D. This time there would be a precedence break because * and / have the same values. Consequently set 2 would be POP'd from the stack and control passes to the precedence break routine at 2406 (other routines may be used depending on the operation to be peformed - ckeck the listing for details). Here the operation between set 2 (C*) and the current value (D) would be performed. This would result in a new current value that will be called M. M equals C * D

After the multiplication control goes back to 2346 (continuation after break processing) where the rules from above are used. This time the current value is pushed as set 2 because it has a higher precedence value (/) than that

in set 1 (plus). Now the stack contains



After pushing set 2 the scan continues, stopping at the operator. It has a higher precedence value than the (/) in set 2 so a third set is added to the stack giving:



The next scan is made and an EOS is found following the 5 (which is now the current value). As mentioned earlier an EOS or non-arithmetic token is an automatic precedence break, so set 3 is POP'd from the stack and E 5 is computed and becomes the current value. Control passes to 2346 where the rules for pushing the next set are applied and set 2 get's POP'd because the current operator is an EOS. Set 2 (M/) and th current value are operated on giving a current value of

M / E 5 or
C * D / E 5

Again control goes to 2346 which forces set 1 to be POP'd because the current operator is an EOS. When the set is POP'd control goes to the computation routine where the current value and set 1 are operated on. This yields a current value of

B plus C * D / E 5

Now control goes to 2346 and this time the stack is empty causing control to be returned to the caller. The expression has been evaluated and its value is left in WRA1.

DOS Request Codes

DOS request codes provide a mechanism for executing system level commands from within a program. The way they work is to cause the DOS overlay module SYSX/SYS associated with the request to be loaded into 4200 - 5200 and executed. When the request has been satisfied control is returned to the caller as though a subroutine call had been made.

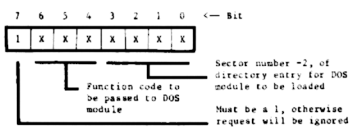
DOS functions may be executed by loading a DOS request code into the a register and executing a RST 28 instruction. Because of the way DOS processes these request codes the push on the stack that resulted from the RST instruction is lost, and control will be returned to the next address found on the stack - rather than to the address following the RST instruction. For example,

```
LD   A,VAL     LOAD DOS  FUNTION CODE
RST   28       EXECUTE DOS FUNCTION
  .                THIS  IS  WHERE  WE  WANT  TO  RETURN
                   TO
  .            BUT WILL NOT BECAUSE OF THE WAY
  .            THE STACK IS MANAGED BY DOS
```

This will not work because the return address (stored on the stack by the RST 28) has been lost during processing. Instead the following sequence should be used:

```
             LD      A,VAL     LOAD REQUEST CODE
             CALL    DOS       PUT RETURN ADDR ON STACK
                     .
                     .
                     .
DOS          RST     28        EXECUTE DOS FUNCTION
                               ALL REGISTERS ARE PRESERVED
                               WE WILL AUTOMATICALLY RET TO
                               CALLER OF DOS
```

The request code value loaded into the A-register must contain the sector number minus 2 of the directory sectory for the overlay to be loaded and a code specifying the exact operation to be performed. The format of the request code is:



As it is presently implemented the file pointed to by the first entry in the specified directory sector will be loaded. There is no way for example, to load the file associated with the 3rd or 4th entry. A list of the system overlay modules and their functions follows. These descriptions are incomplete. See the individual modules for a complete description.

MODULE	DIRECTORY SECTOR MINUS 2	REQUEST CODE	SUB-FUNCTIONS
SYS1	1	93	10 - write 'DOS READY'
		AC	20 - write 'DOS READY'
		BC	30 - scan input string
		C3	40 - move input string to DCB
		D3	50 - scan and move input string
		E3	60 - append extension to DCB
		F3	70 - reserved
SYS2	2	94	10 - OPEN file processing
		A4	20 - INIT file processing
		B4	30 - create directory overflow entry
		C4	40 -
		D4	50 - reserved
		E4	60 -
		F4	70 -
SYS3	3	95	10 - CLOSE file processing
		A5	20 - KILL file processing
		B5	30 -
		C5	40 - reserved
		D5	50 -
		E5	60 - load SYS3/SYS
		F5	70 - format diskette
SYS4			
SYS5			

Chapter 5

A BASIC SORT Verb

Contained in this chapter is a sample assembly program that demonstrates the use of the ROM calls and tables described in the previous chapters. In this example DOS Exits and Disk BASIC Exits are used to add a SORT verb to BASIC.

In this case a SORT verb will be added so that the statement

100 SORT I$, O$, K1$

be used to read and sort a file specified by the string I$. O$ and K1$ are strings which specify the output file name and the sort key descriptors. The procedure for doing this is simple. First we must modify the Input Phase to recognize the word SORT and replace it with a token. This can be accomplished by using one of the DOS Exits.

A DOS Exit is taken during the Input Phase immediately after the scan for reserved words. We will intercept this exit to make a furthur test for the word SORT and replace it with a token. Processing will then continue as before. Before using any DOS Exit study the surrounding code to determine exact register usage. In this case it is important to note that the length of the incoming line is in the BC register when the exit is taken. If the subroutine compresses the line (by replacing the word SORT with a token) then its length will have changed and the new length must replace the orginal contents of BC.

A second modification must be made to the Execution Driver, or somewhere in its chain, to recognize the new token value and branch to the SORT action routine. This presents a slight problem because there are no DOS Exits in the execution driver before calling the verb routine, and since the driver code and its tables are in ROM they cannot be changed. In short there is no easy way to incorporate new tokens into the Execution Phase.

The solution is to borrow a Disk BASIC token and piggy-back another token behind it. Then any calls to the verb routine associated with the borrowed token must be intercepted and a test make for the piggy-backed token. If one is found control goes to the SORT verb routine otherwise it passes to the assigned verb routine. In this example the tokenq FA will be borrowed and another FA will be tacked behind it giving a token FAFA.

This example is incomplete because the LIST function has not been modified to recognize the sort token. If a LIST command is issued the verb MIDMID will be given for the SORT verb. There is one more detail that needs attention before discussing the verb routine. Using the memory layout figure in Chapter 1 we can see that there is no obvious place to load an assembly language program without interfering somehow with one of BASIC's areas. Depending on where we loaded our verb routine it could overlay the String Area, or the Stack, or maybe even reach as low as the PST or VLT. Of course we might get lucky and find an area in the middle of the Free Space List that never gets used but thats too risky.

BASIC has a facility for setting the upper limit of the memory space it will use. By using this feature we can reserve a region in high memory where our verb routine can be loaded without disturbing any of BASIC's tables. Now for the details of verb routine.

Because a sort can be a lengthy piece of code only the details that pertain to DOS Exits, Disk BASIC, and some of the ROM calls from Chapter 2 will be illustrated. The verb routine has two sections. The first section will be called once to modify the DOS and Disk BASIC exit addresses (also called vectors) in the Communications Region to point to locations within the verb routine. The vector addresses must be modified after BASIC has been entered on a DOS system because they are initialized by the BASIC command. The second section has two parts.

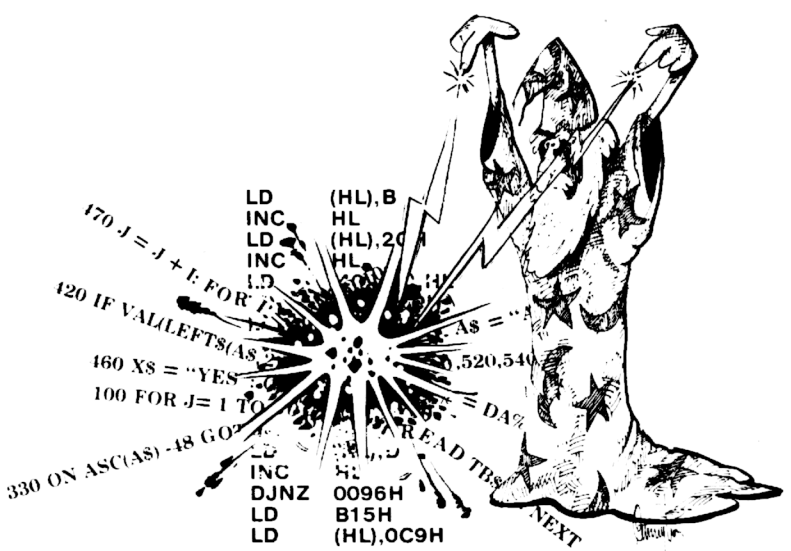
Part one is the DOS Exit code called from the Input Scanner. Part two is the verb action routine for the SORT verb. It is entered when a FA token is encountered during the Execution Phase.

The system being used will be assumed to have 48K of RAM, at least 1 disk, and NEWDOS 2.1. The verb routine will occupy locations E000 - FFFF. The entry point for initializing the vectors will be at qt22E000. All buffers used will be assigned dynamically in the stack portion of the Free Space List. The verb routine will be loaded before exiting DOS and entering Level II BASIC. Although it could be loaded from the BASIC program by using the CMD'LOAD.....' feature of NEWDOS.

```
1. IPL
2. LOAD,SORT      :(load verb into    E000 - FFFF  1)
3. BASIC,57344    :(protect verb area)


100 DEF USR1(0) = &HE000 : initialization entry point
110 A = USR1(0)          : initialize vectors

RUN                      : initialize the sort

100 I$="SORTIN/PAY:1"    : (sort input file)
110 O$="SORTOUT/PAY:1"   : (sort output file)
120 K$="A,A,100-120"     : (sort key: ascending order ASCII
                         : key, sort field is 100 - 120)
130 SORT I$, O$,K$       : (sort file)

RUN
```

```
00100          ORG     0E000H
00110 ; INITIAL ENTRY POINT TO INITIALIZE DOS EXIT AND
00120 ; DISK BASIC ADDRESSES.
00130          LD      HL,(41B3H)      ; ORGINAL DOS EXIT VALUE
00140          LD      (ADR1+1),HL     ; IS STILL USED AFTER OUR
00150 ;                                ; PROCESSING
00160          LD      HL,(41DAH)      ; ORGINAL DISK BASIC ADDR FOR
00170 ;                                ; MID$ TOKEN (FA)
00180          LD      (ADR2+1),HL     ; SAVE IN CASE FA TOKEN FOUND
00190          LD      HL,NDX
00200          LD      (41B3H),HL
00210          LD      HL,NDB
00220 ;                                ; OUR ADDR
00230          LD      (41DAH),HL
00240 ;                                ; FA TOKEN W/OUR ADDR
00250          RET                     ; RET TO EXECUTION DRIVER
00260 ;*   GET ADDRESS OF VARIABLE
00270 ;* THIS SECTION OF CODE IS ENTERED AS A DOS EXIT DURING THE
00280 ;* INPUT PHASE. IT WILL TEST FOR A 'SORT' COMMAND AND REPLACE
00290 ;* IT WITH A 'FAFA' TOKEN. THE ORGINAL DOS EXIT ADDR HAS BEEN
00300 ;* SAVED AND WILL BE TAKEN AT ADR1.
00310 ;*
00320 NDX      CALL    SAV             ; SAV ALL REGISTERS
00330          LD      IX,SORT-1       ; TEST STRING
00340          LD      B,3             ; NO. OF CHARS TO MATCH
00350 NDX1     INC     HL              ; START OF LOOP
00360          INC     IX              ; BUMP TO NEXT TEST CHAR
00370          LD      A,(IX+0)        ; GET A TEST CHAR
00380          CP      (HL)            ; COMPARE W/INPUT STRING
00390          JR      NZ,OUT          ; STOP WHEN FIRST MIS-MATCH
00400          DJNZ    NDX1            ; ALL 4 CHARS MUST MATCH
00410 ;*
00420 ;* WE HAVE A MATCH. NOW REPLACE THE WORD 'SORT' WITH A TOKEN
00430 ;* 'FAFA' AND COMPRESS THE STRING
00440 ;*
00450          INC     HL              ; FIRST CHAR AFTER 'SORT'
00460          PUSH    HL              ; SAVE FOR COMPRESSION CODE
00470          LD      BC,-3           ; BACKSPACE INPUT STRING
00480          ADD     HL,BC           ; START OF WORD 'SORT'
00490          LD      (HL),0FAH       ; TOKEN REPLACES 'S'
00500          INC     HL              ; NEXT LOC IN INPUT STRING
00510          LD      (HL),0FAH       ; TOKEN REPLACES 'O'
00520          INC     HL              ; NEXT LOC IN INPUT STRING
00530          POP     DE              ; STRING ADDR AFTER SORT
00540          EX      DE,HL           ; SO WE CAN USE RST 10
00550 ;*                               ; TO FETCH NEXT CHAR
00560 ;* NOW COMPRESS THE INPUT STRING
00570 ;*
00580          LD      BC,3            ; SET COUNT OF CHARS IN
00590 ;*                               ; EQUAL TO NO SKIPPED OVER
00600 NDX2     RST     10H             ; GET NEXT CHAR, DISCARD
00610 ;*                               ; BLANKS
00620          LD      (DE),A          ; MOVE IT DOWN
00630          INC     DE              ; BUMP SOURCE ADDR
00640          INC     C               ; COUNT 1 CHAR IN LINE
00650          OR      A               ; TEST FOR END OF STRING
00660          JR      NZ,NDX2         ; NOT END, LOOP
00670          LD      (DE),A          ; EACH LINE MUST END WITH
00680 ;*                               ; 3 BYTES OF ZEROS
00690          INC     DE              ; BUMP TO LAST BYTE
00700          LD      (DE),A          ; STORE 3 RD ZERO
00710          INC     C               ; THEN SET BC = LENGTH OF
00720          INC     C               ; LINE + 1
00730          INC     C               ; SO BASIC CAN MOVE IT
00740          LD      (TEMP),BC       ; SAVE NEW LINE LENGTH
00750          CALL    RES             ; RESTORE REGISTERS
00760          LD      BC,(TEMP)       ; NEW LINE LENGTH TO BC
00770          JR      ADR1            ; EXIT
00780 OUT      CALL    RES             ; RESTORE REGISTERS
00790 ADR1     JP      0               ; CONTINUE ON TO ORGINAL
00800 ;*                               ; DOS EXIT
00810 ;* DISK BASIC EXIT FOR FA TOKEN. TEST FOR SORT TOKEN FAFA
00820 ;*
00830 NDB      CALL    SAV             ; SAVE ALL REGISTERS
00840          INC     HL              ; SKIP TO CHAR AFTER TOKEN
00850          LD      A,(HL)          ; TEST FOR SECOND 'FA'
00860          CP      0FAH            ; IS FOLLOWING CHAR A FA
00870          JR      Z,NDB1          ; Z IF SORT TOKEN
00880          CALL    RES             ; RESTORE REGISTERS
00890 ADR2     JP      0               ; CONTINUE WITH MID$ PROCESSI!
00900 ;*
00910 ;* WE HAVE A SORT TOKEN
00920 ;*
00930 NDB1     INC     HL              ; SKIP OVER REST OF TOKEN
00940          CALL    GADR            ; GET ADDR OF 1ST PARAM
00950          LD      (PARM1),DE      ; SAV ADDR OF INPUT FILE NAME
00960          RST     08              ; LOOK FOR COMMA
00970          DEFM    ','             ; SYMBOL TO LOOK FOR
00980          CALL    GADR            ; GET ADDR OF 2ND PARAM
00990          LD      (PARM2),DE      ; SAV ADDR OF OUTPUT FILE NAM
01000          RST     08              ; LOOK FOR COMMA
01010          DEFM    ','             ; SYMBOL TO LOOK FOR
01020          CALL    GADR            ; GET ADDR OF SORT KEYS
01030          LD      (PARM3),DE      ; SAV ADDR OF SORT KEY
01040          LD      (TEMP),HL       ; SAVE ENDING POSITION
01050 ;*                               ; IN CURRENT STATEMENT
01060 ;*
01070 ;* NOW, BLANK FILL I/O DCBS
01080 ;*
01090          LD      IX,DCBL         ; LIST OF DCB ADDRS
01100          LD      C,2             ; NO OF DCBS TO BLAINK
01110          LD      A,20H           ; ASCII BLANK
01120 L1       LD      L,(IX+0)        ; LSB OF DCB ADDR
01130          LD      H,(IX+1)        ; MSB OF DCB ADDR
01140          LD      B,32            ; NO OF BYTES TO BLANK
01150 L2       LD      (HL),A          ; BLANK LOOP
01160          INC     HL
01170          DJNZ    L2              ; LOOP TILL BLANKED
01180          INC     IX              ; BUMP TO NXT DCB ADDR
01190          INC     IX              ; BUMP AGAIN
01200          DEC     C               ; ALL DCBS BLNKD
01210          JR      NZ,L1           ; NO
01220 ;*
01230 ; YES, MOVE FILE NAMES TO DCB AREAS
01240 ;*
01250          LD      HL,(PARM1)      ; ADDR OF INPUT FILE NAME STR
01260          LD      DE,DCBI         ; INPUT DCB
01270          CALL    29C8H           ; MOVE NAME TO DCB
01280          LD      HL,(PARM2)      ; ADDR OF OUTPUT FILE NAME
01290          LD      DE,DCBO         ; OUTPUT DCB
01300          CALL    29C8H           ; MOVE NAME TO DCB
01310          LD      HL,(PARM3)      ; GET ADDR OF KEY STRING
01320          INC     HL              ; SKIP OVER BYTE COUNT
01330          LD      C,(HL)          ; GET LSB OF STRNG ADDR
01340          INC     HL              ; BUMP TO REST OF ADDR
01350          LD      H,(HL)          ; GET MSB OF STRNG ADDR
01360          LD      L,C             ; NOW HL = STRNG ADDR
01370          CALL    1E3DH           ; MUST BE ALPHA
01380          JR      NC,YA1          ; OK
01390          JP      ERROR           ; INCORRECT SORT ORDER
01400 YA1      LD      (ORDER),A       ; SAVE SORT ORDER (A/D)
01410          INC     HL              ; SKIP TO TERMINAL CHAR
01420          RST     08              ; TEST FOR COMMA
01430          DEFM    ','
01440          CALL    1E3DH           ; MUST BE ALPHA
01450          JR      NC,YA5          ; OK
```

```
01460           JP      ERROR
01470 YA5       LD      (TYPE),A        ; SAVE TYPE (A/B)
01480           INC     HL              ; SKIP TO TEMINAL CHAR
01490           RST     8               ; TEST FOR COMMA
01500           DEFM    ','
01510           CALL    0E6CH           ; GET RECORD SIZE
01520           LD      DE,(4121H)      ; GET SIZE FROM WRA1
01530           LD      (SIZE),DE       ; SAVE IT
01540           RST     20H             ; MUST BE AN INTEGER
01550           JP      M,YA10          ; MINUS IF INTEGER
01560           JP      ERROR
01570 YA10      RST     08              ; LOOK FOR COMMA
01580           DEFM    ','
01590           CALL    0E6CH           ; GET STARTING POSITION
01600           LD      DE,(4121H)      ; GET POS FROM WRA1
01610           LD      (START),DE      ; SAVE IT
01620           RST     08              ; LOOK FOR -
01630           DEFM    '-'             ; CHAR TO TEST FOR
01640           CALL    0E6CH           ; GET ENDING POS OF KEY
01650           LD      DE,(4121H)      ; GET VALUE FROM WRA1
01660           LD      (END),DE        ; SAVE ENDING POS
01670           LD      HL,(TEMP)       ; RESTORE CURRENT LINE ADDR
01680 ;*                                ; TO EOS ON RETURN
01690           CALL    RES             ; RESTORE REGISTERS
01700           LD      HL,(TEMP)       ; RESTORE EOS ADDR
01710           RET                     ; RETURN TO BASIC
01720 ;*
01730 ;*
01740 ;*
01750 SORT      DEFM    'S'             ; S OF SORT
01760           DEFB    0D3H            ; TOKEN FOR OR OF SORT
01770           DEFM    'T'             ; T OF SORT
01780 ;*
01790 ;*  SAVE ALL REGISTERS
01800 ;*
01810 SAV       EX      DE,HL
01820           EX      (SP),HL         ; SAV DE/RTN ADDR TO HL
01830           PUSH    BC
01840           PUSH    AF
01850           PUSH    IX
01860           PUSH    DE              ; SAV ORGINAL HL
01870           EX      DE,HL           ; RESTORE HL RET ADDR TO DE
01880           PUSH    DE              ; RET ADDR TO STK
01890           RET                     ; RET TO CALLER
01900 ;*
01910 ;*  RESTORE ALL REGISTERS
01920 ;*
01930 RES       POP     HL              ; RTN ADDR TO HL
01940           POP     DE              ; REAL HL
01950           POP     IX
01960           POP     AF
01970           POP     BC
01980           EX      (SP),HL         ; RTN ADDR TO STK
01990           EX      DE,HL
02000           RET
02010           JP      (HL)            ; RTN TO CALLER
02020 ;*
02030 ;*  GET THE ADDRESS OF THE NEXT VARIABLE INTO DE
02040 ;*
02050 GADR      LD      A,(HL)          ; GET NEXT CHAR FROM INPU
02060 ;*                                ; STRNG, TST FOR LITERAL
02070           CP      22H             ; IS IT A QUOTE -START OF
02080 ;*                                ; A LITERAL-
02090           JR      NZ,GADR2        ; NO, GO FIND ADDR OF VAR
02100           CALL    2866H           ; YES, GO BUILD A LSPT EN
02110           JR      GADR5           ; THEN JOIN COMMON CODE
02120 GADR2     CALL    2540H           ; GET ADDR OF NEXT VARIAB
02130 GADR5     RST     20H             ; IS IT A STRING
02140           LD      DE,(4121H)      ; ADDR OF NEXT VAR
02150           RET     Z               ; RET IF STRING VAR
02160           POP     HL              ; CLEAR STACK
02170           POP     HL              ; CLEAR STACK
02180           LD      A,2             ; ERROR CODE FOR SYNTAX E
02190           JP      1997H           ; GO TO ERROR ROUTINE
02200 ;*
02210 ;*  ERROR EXIT
02220 ;*
02230 ERROR     CALL    RES             ; RESTORE REGISTERS
02240           POP     HL              ; CLEAR STACK
02250           LD      A,2             ; SYNTAX ERROR CODE
02260           JP      1997H           ; PRINT ERROR MESSAGE
02270 ;*
02280 ;*
02290 ;*
02300 DCB1      DEFW    DCB1
02310           DEFW    DCB0
02320 PARM1     DEFW    0               ; INPUT FILE NAME STRING
02330 PARM2     DEFW    0               ; OUTPUT FILE NAME STRING
02340 PARM3     DEFW    0               ; KEY STRING ADDR
02350 TYPE      DEFB    0               ; RECORD TYPE (A/B/C)
02360 ORDER     DEFB    0               ; SORT ORDER (A/D)
02370 SIZE      DEFW    0               ; RECORD SIZE
02380 START     DEFW    0               ; STARTING POSITION OF KE
02390 END       DEFW    0               ; ENDING POSITION OF KEY
02400 TEMP      DEFW    0               ; HOLDS EOS ADDR
02410 DCB1      DEFS    32              ; INPUT DCB
02420 DCB0      DEFS    32              ; OUTPUT DCB
02430           END
```



Chapter 6

BASIC Overlay Routine

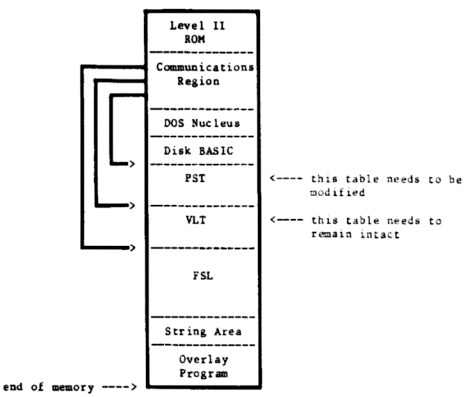
This example shows how the tables in the Communications Region can be manipulated so that a BASIC program can load and execute overlays. The overlay program will add statements to an executing BASIC program while preserving all the current variables. The calling sequence to be used is:

```
100  DEF USR1=&HE000   : Address of of overlay program
              .        : Main body of application program
              .
              .
300  F$="FILE1/BAS"    : File containing overlay
310  Z=USR1(500)       : Replace lines 500 thru the end
                       : of the program with the
                       : statement from FILE1/BAS.)
320 GOSUB 500          : Execute the overlay
              .
              .
              .
500 REM START OF OVERLAY AREA
              .
              .
```

The operating assumptions for this example will be the same as those in chapter 5. Note, overlay files containing the ASCII file must have been saved in the A mode.

The program itself will be considerably different, however. For instance, there will be no use of DOS Exits. This means that the CR will not need modifica-tion so there will be no need for an initial entry point. One parameter will be passed in the calling sequence while the other one will have an agreed name so that it can be located in the VLT.

When a BASIC program is executing there are three major tables that it uses. First is the PST where the BASIC statements to be executed have been stored. Second is the VLT where the variables assigned to the program are stored, and the third table is the FSL which represents available memory. All of these tables occur in the order mentioned. The problem we need to overcome in order to support overlays is to find a way to change the first table while maintaining the contents of the second one. A diagram of memory showing the tables follows.



Fortunately this can be accomplished quite easily. By moving the VLT to the high end of FSL we can seperate it from the PST. Then the overlay statements can be read from disk and added to the PST. Obviously the PST would either grow or shrink during this step unless the overlay being loaded was exactly the same size as the one before it. After the overlay statements have been added the VLT is moved back so it is adjacent to the PST. Then the pointers to the tables moved are updated and control is returned to the BASIC Execution Driver

The overlay loader used in this example assumes that the file containing the overlay statements is in ASCII format. This means that each incoming line must be tokenized before being moved to the PST. To speed up processing the loader could be modified to accept tokenized files.

There is no limit to the number of overlays that can be loaded. The program will exit with an error if a line number less than the starting number is detected. The loader does not test for a higher level overlay destroy-ing a lower one, this would be disasterous - as the return path would be destroyed.

A sample program to load three seperate overlays is given as an example.

```
100 A = 1.2345
110 B = 1
120 IF B = 1 THEN F$ = "FILE1"
130 IF B = 2 THEN F$ = "FILE2"
140 IF B = 3 THEN F$ = "FILE3"
150 Z = USR1(500)
160 GOSUB 500
170 B = B + 1
180 IF B > 3 THEN 110
190 GOTO 120


500 PRINT"OVERLAY #1 ENTERED"
510 PRINT A
520 C = 25
530 D = 30
540 E = C+D+A                          Contents
550 PRINT"C = ";                       of File 1
560 PRINT"D = ";D
570 PRINT"E = ";E
580 RETURN



500 PRINT"OVERLAY #2 ENTERED"
510 PRINT A
520 C = C + 1
530 D = D + 1
540 E = E + 1                          Contents
550 REM                                of File 2
560 REM
570 REM
580 REM
590 PRINT"C, D, E ";C,D,E
600 RETURN


500 PRINT"OVERLAY #3 ENTERED"
510 A = A + 1                          Contents
520 PRINT"A = ";A                      of File 3
530 RETURN
```

```
00100          ORG     0F000H
00110 OPEN     EQU     4424H           ; DOS ADDRESS
00120 READ     EQU     4436H           ; DOS ADDRESS
00130 ERN      EQU     12              ; DISK DCB ADDRESS
00140 NRN      EQU     10              ; DISK DCB ADDRESS
00150 EOF      EQU     8               ; DISK DCB ADDRESS
00160 ;*
00170 ;*   ENTRY POINT FOR OVERLAY LOADING OF BASIC PROGRAMS
00180 ;*
00190          PUSH    AF              ; SAVE ALL REGISTERS
00200          PUSH    BC
00210          PUSH    DE
00220          PUSH    HL
00230          LD      HL,-1           ; INITIALIZE SECTOR COUNT
00240          LD      (RCOUNT),HL     ; TO MINUS 1
00250          LD      HL,00           ; SO WE CAN LOAD CSP
00260          ADD     HL,SP           ; LOAD CSP
00270          LD      (CSP),HL        ; SAVE FOR RESTORATION
00280          LD      DE,(4121H)      ; LINE NO TO START OVERLAY
00290          LD      (LINE),DE       ; SAVE FOR FUTURE REF
00300          LD      A,(40AFH)       ; FUNCTION VALUE TYPE
00310          LD      (TYPE),A        ; MUST BE RESTORED AT END
00320 ;*
00330 ;*   BLANK FILL DCB BEFORE MOVING NAME INTO IT
00340 ;*
00350          LD      B,32            ; NO. OF BYTES TO BLANK
00360          LD      HL,DCB          ; DCB ADDR
00370          LD      A,20H           ; ASCII BLANK
00380 BFL      LD      (HL),A          ; MOVE ONE BLANK
00390          INC     HL              ; BUMP TO NEXT WORD
00400          DJNZ    BFL             ; LOOP TILL DCB FILLED
00410 ;*
00420 ;*  GET OVERLAY FILE NAME FROM VARIABLE F$
00430 ;*  MOVE IT INTO THE BLANKED DCB
00440 ;*
00450          LD      HL,LFN          ; STRING FOR COMMON VAR NAME
00460          CALL    2540H           ; GET ADDR OF F$
00470          RST     20H             ; MAKE SURE IT'S A STRING
00480          JR      Z,OK            ; ZERO IF STRING
00490          JP      ERR             ; WRONG TYPE OF VARIABLE
00500 OK       LD      HL,(4121H)      ; GET ADDR OF F$ INTO HL
00510          LD      DE,DCB          ; DCB ADDR
00520          CALL    29C8H           ; MOVE F$ NAME TO DCB
00530 ;*
00540 ;*   INITIALIZE ALL LOCAL VARIABLES
00550 ;*
00560          LD      A,0             ; SET PASS FLAG TO ZERO
00570          LD      (PF),A          ; PASS FLAG
00580          LD      (FI),A          ; SECTOR BUFFER INDEX
00590 ;*
00600 ;*  LOCATE ADDR OF VARIABLE ASSIGND TO FUNCTION CALL. IT
00610 ;*  MUST BE RECOMPUTED AFTER THE OVERLAY HAS BEEN LOADED
00620 ;*  BECAUSE THE VLT WILL HAVE BEEN MOVED. NEXT, ALLOCATE
00630 ;*  SPACE IN THE FSL FOR THE SECTOR BUFFER USED FOR
00640 ;*  READING THE OVERLAY FILE.
00650 ;*
00660          LD      HL,00           ; SO WE CAN LOAD CSP
00670          ADD     HL,SP           ; HL = CSP
00680          PUSH    HL              ; SAVE CSP
00690          LD      BC,20           ; AMT TO BACKSPACE CSP
00700          ADD     HL,BC           ; GIVES CSP - 20  OR ADDR
00710 ;*                               ; OF FUNCTION VARIABLE
00720          LD      (VARADR),HL     ; SAVE STK ADDR OF VAR
00730          POP     HL              ; RESTORE CSP TO HL
00740          LD      BC,-256         ; AMT OF SPACE TO ALLOCTAT
00750 ;*                               ; IN FSL FOR SECTOR BUFFR
00760          ADD     HL,BC           ; COMPUTE NEW CSP
00770          LD      (BADDR),HL      ; START OF SECTOR BUFFER
00780          LD      SP,HL           ; IS ALSO NEW CSP
00790          PUSH    HL
00800          LD      DE,(40F9H)      ; CURRENT END OF PST
00810          LD      (CEPST),DE      ; SAVE FOR COMPUTATIONS
00820          LD      HL,(40FBH)      ; START OF ARRAYS
00830          XOR     A               ; CLEAR CARRY
00840          SBC     HL,DE           ; COMPUTE OFFSET FROM STAR
00850 ;*                               ; OF VLT TO START OF ARRAY
00860          LD      (LSVLT),HL      ; SAVE OFFSET
00870 ;*
00880 ;*
00890 ;*
00900          LD      DE,(LINE)       ; FIND ADDR OF LINE WHERE
00910          CALL    1B2CH           ; OVERLAY STARTS IN PST
00920          LD      (40F9H),BC      ; MAKE IT TEMP END OF PST
00930 ;*
00940 ;*  COMPUTE LENGTH OF VLT
00950 ;*
00960          LD      DE,(CEPST)      ; ORGINAL END OF PST
00970          LD      HL,(40FDH)      ; START OF FSL
00980          XOR     A               ; CLEAR CARRY
00990          SBC     HL,DE           ; GIVES LNG -1 OF VLT
01000          INC     HL              ; CORRECT FOR -1
01010          LD      (LVLT),HL       ; SAVE LENGHT OF VLT
01020          POP     HL              ; RESTORE CSP TO HL
01030          LD      BC,-50          ; ASSUMED STK LENG NEEDED
01040          ADD     HL,BC           ; GIVE END OF TEMP VLT
01050          LD      BC,(LVLT)       ; NOW, SUBTRACT LENGTH OF
01060          XOR     A               ; VLT FROM END TO GET STAR
01070          SBC     HL,BC           ; ADDRESS
01080          LD      (SNVLT),HL      ; SAVE END OF TEMP VLT
01090          PUSH    HL              ; SO WE CAN
01100          POP     DE              ; LOAD IT INTO DE
01110          LD      HL,(CEPST)      ; START OF OLD PST
01120          LD      BC,(LVLT)       ; SIZE OF VLT
01130          LDIR                    ; MOVE VLT TO TEMP LOC.
01140 ;*
01150 ;*  BEGIN OVERLAY LOADING
01160 ;*
01170          LD      DE,DCB          ; DCB FOR OVERLAY FILE
01180          LD      HL,(BADDR)      ; SECTOR BUFF ADDR
01190          LD      BC,0            ; SPECIFY SECTOR I/O
01200          CALL    OPEN            ; OPEN OVERLAY FILE
01210 LOOP     CALL    GNL             ; GET NEXT LINE FROM FILE
01220          JR      Z,OUT           ; ZERO IF NO MORE LINES
01230 ;*                               ; IN OVERLAY FILE
01240          CALL    ATOB            ; ADD LINE TO PST
01250          JR      LOOP            ; LOOP TILL FILE EXHAUSTED
01260 ;*
01270 ;*  OVERLAY STATEMENTS HAVE BEEN ADDED. RESET POINTERS
01280 ;*  TO VLT AFTER MOVING IT DOWN (ADJACENT TO PST).
01290 ;*
01300 OUT      LD      HL,(SNVLT)      ; START OF TEMP VLT
01310          LD      DE,(40F9H)      ; CURRENT END OF PST
01320          INC     DE              ; LEAVE TWO BYTES
01330          INC     DE              ; OF ZEROS AT END OF PST
01340          LD      (40F9H),DE      ; SAVE START ADDR OF NEW VL
01350          LD      BC,(LVLT)       ; LENGTH OF VLT
01360          LDIR                    ; MOVE VLT TO END OF PST
01370          INC     DE              ; GIVES ADDR OF FLS
01380          PUSH    DE              ; SAVE FSL ADDR
01390          LD      HL,(40F9H)      ; START OF VLT
01400          LD      BC,(LSVLT)      ; PLUS LNG OF SIMP VAR
01410          ADD     HL,BC           ; GIVES ADDR OF ARRAYS PTI
01420          LD      (40FBH),HL      ; SAVE NEW ARRAYS POINTER
01430          POP     HL              ; HL = NEW FSL ADDR
01440          LD      (40FDH),HL      ; UPDATE FSL
01450 ;*
01460 ;*  COMPUTE DISTANCE VLT HAS MOVED AND UPDATE THE ADDR OF
01470 ;*  THE FUNCTION VARIABLE BEING CARRIED ON THE STACK.
01480 ;*
```

```
01490           LD      DE,(CEPST)      ; ORGIANL START OF VLT
01500           LD      HL,(40F9H)      ; CURRENT START OF VLT
01510           RST     18H             ; COMPARE TWE ADDRESSES
01520           JR      NC,UP           ; NEW VLT WAS MOVED UP
01530           PUSH    HL              ; REVERSE OPERANDS
01540           PUSH    DE
01550           XOR     A               ; CLEAR CARRY
01560           POP     HL              ; RERSTORE OPERANDS
01570           POP     DE
01580           JR      UP1             ; GO COMPUTE DISTANCE
01590 UP        XOR     A               ; CLEAR CARRY FOR SUB
01600 UP1       SBC     HL,DE           ; COMPUTE AMT VLT HAS MOVED
01610           PUSH    HL              ; SAVE DISTANCE
01620           LD      HL,(VARADR)     ; THEN ADDR IT TO ADDR
01630           LD      C,(HL)          ; CARRIED ON STK
01640           INC     HL              ; BUMP TO MSB OF ADDR
01650           LD      B,(HL)          ; BC = ADDR OF VAR THAT WAS
01660 ;*                                ; CARRIED ON STK
01670           POP     HL              ; GET DISPLACEMENT
01680           ADD     HL,BC           ; GET NEW ADDR (BECAUSE VLT
01690 ;*                                ; HAS BEEN MOVED
01700           PUSH    HL              ; SO WE CAN LOAD IT INTO
01710           POP     DE              ; LOAD NEW ADDR INTO DE
01720           LD      HL,(VARADR)     ; REFETCH STK ADDR
01730           LD      (HL),E          ; LSB OF FUNCTION VAR ADDR
01740           INC     HL              ; NEXT BYTE ADDR ON STK
01750           LD      (HL),D          ; MSB OF FUNCTION VAR ADDR
01760 ;*
01770 ;*  RESET TYPE TO IT'S ORGINAL VALUE
01780 ;*
01790           LD      A,(TYPE)        ; GET MODE FLAG WHEN ENTERED
01800           LD      (40AFH),A       ; RESTORE MODE TO ORGIANL
01810           LD      HL,(CSP)        ; RESET CSP
01820           LD      SP,HL           ; TO IT'S ORGINAL VALUE
01830           POP     HL              ; RESTORE REGISTERS
01840           POP     DE
01850           POP     BC
01860           POP     AF
01870           RET                     ; RETURN TO BASIC
01880 ;*
01890 ;*  GNL -  GETS NEXT LINE OF BASIC PROGRAM FROM A FILE
01900 ;*         MOVES IT TO BASIC LINE BUFFER AREA AND THEN
01910 ;*         TOKENIZES IT.
01920 ;*         FILE IS ASSUMED TO BE IN ASCII FORMAT. LINES ARE
01930 ;*         TERMINATED BY A CARRIAGE RET. (0D).
01940 ;*
01950 GNL       LD      A,(PF)          ; GET PASS FLAG
01960           OR      A               ; IS IT TIME TO READ SECTOR
01970           JR      NZ,GNL5         ; NO IF NON-ZERO
01980 GNL3      LD      A,0             ; RESET SECTOR BUFF INDEX
01990           LD      (FI),A          ; TO ZERO
02000           LD      HL,(RCOUNT)     ; PREPARE TO TEST FOR
02010           INC     HL              ; END OF FILE. BUMP COUNT
02020           LD      (RCOUNT),HL     ; OF SECTORS READ
02030           LD      BC,0            ; READ NEXT SECTOR
02040           LD      DE,DCB          ; OVERLAY DCB ADDR
02050           LD      HL,(BADDR)      ; SECTOR BUFF ADDR
02060           CALL    READ            ; READ NEXT SECTOR
02070           LD      A,1             ; RESET PASS FLAG
02080           LD      (PF),A          ; TO DATA IN BUFFER
02090 GNL5      LD      DE,(RCOUNT)     ; NOW TEST FOR END OF FILE
02100           LD      HL,(DCB+ERN)    ; LAST SECTOR NO FROM DCB
02110           XOR     A               ; CLEAR CARRY FOR SUB
02120           SBC     HL,DE           ; HAS LAST SECTOR BEEN READ
02130           JR      NZ,GNL10        ; NON-ZERO IF NOT LAST SECT
02140           LD      A,(DCB+EOF)     ; IN LAST SECTOR. END OF D
02150           LD      B,A             ; DATA REACHED YET?
02160           LD      A,(FI)          ; CURRENT SECTOR INDEX
02170           SUB     B               ; MUST BE LE TO EOD INDEX
02180           JR      C,GNL10         ; CARRY IF NOT END OF DATA
02190           XOR     A               ; SIGNAL END OF FILE
02200           RET                     ; RET TO MAIN PGM
02210 GNL10     LD      HL,(BADDR)      ; SECTOR BUFF ADDR
02220           LD      A,(FI)          ; CURRENT BUFF INDEX
02230           LD      C,A             ; FOR 16 BIT AIRTH
02240           LD      B,0             ; DITTO
02250           ADD     HL,BC           ; CURRENT LINE ADDR IN BUFF
02260           LD      DE,(40A7H)      ; BA   : LINE BUFF ADDR
02270 GNL15     LD      A,(HL)          ; MOVE LINE FROM SECT BUFF
02280           LD      (DE),A          ; TO BASIC LINE BUFF
02290           INC     DE              ; BUMP DEST ADDR
02300           INC     C               ; COUNT 1 CHAR MOVED
02310           JR      C,GNL3          ; JMP IF LINE OVERFLOWS
02320 ;*                                ; SECTOR
02330           INC     HL              ; NO OVERFLOW, BUMP FETCH
02340           SUB     0DH             ; ADDR. TEST FOR END OF LINE
02350           JR      NZ,GNL15        ; LOOP TILL END OF LINE
02360           DEC     DE              ; BKSPC 1 CHAR IN LINE BUFF
02370           LD      (DE),A          ; AND TERM IT WITH A ZERO
02380           LD      A,C             ; SAVE ENDING BUFF INDEX
02390           LD      (FI),A          ; FOR NEXT LINE
02400           OR      A               ; SIGNAL MORE DATA
02410           RET                     ; RET TO CALLER
02420 ;*
02430 ;*  TOKENIZE LINE IN BUFFER. THEN ADD IT TO PST
02440 ;*
02450 ATOB      LD      HL,(40A7H)      ; LINE BUFFER ADDR
02460           CALL    1E5AH           ; GET BINARY LINE NO
02470           PUSH    DE              ; SAVE IT
02480           PUSH    HL              ; SAVE LINE BUFF ADDR
02490           LD      HL,(LINE)       ; BEG OVERLAY LINE NO
02500           RST     18H             ; COMPARED W/CURRENT LINE
02510           JR      Z,ATOB5         ; OK IF EQUAL
02520           JR      NC,ERR          ; ERR IF INCOMMING LESS
02530 ;*                                ; THAN OVERLAY LINE NO
02540 ATOB5     POP     HL              ; RESTORE LINE ADDR
02550           CALL    1BC0H           ; TOKENIZE LINE
02560           LD      HL,(40F9H)      ; CURRENT END OF PST
02570           PUSH    HL              ; SAVE ADDR OF THIS LINE
02580           ADD     HL,BC           ; ADD LNG OF NEW LINE
02590           LD      (40F9H),HL      ; START OF NEXT LINE
02600           PUSH    HL              ; SO WE CAN
02610           POP     DE              ; LOAD IT INTO DE
02620 ;*
02630 ;*  UPDATE POINTER TO NEXT LINE IN NEW LINE BEING ADDED.
02640 ;*  THEN MOVE BINARY LINE NO. FOR THIS LINE TO PST.
02650 ;*
02660           POP     HL              ; ADDR OF THIS LINE IN PST
02670           LD      (HL),E          ; LSB OF ADDR NEXT LINE
02680           INC     HL
02690           LD      (HL),D          ; MSB OF ADDR NEXT LINE
02700           INC     HL              ; START OF BIN LINE NO
02710           POP     DE              ; BINARY LINE NO
02720           LD      (HL),E          ; LSB OF LINE NO
02730           INC     HL
02740           LD      (HL),D          ; MSB OF LINE NO
02750           INC     HL              ; BUMP TO FIRST CHAR IN LINE
02760           EX      DE,HL           ; DE = PST FOR LINE
02770           LD      HL,(40A7H)      ; TOKENIZED LINE ADDR
02780           DEC     HL
02790           DEC     HL
02800 ATOB10    LD      A,(HL)          ; GET A TOKENIZED BYTE
02810           LD      (DE),A          ; MOVE IT TO PST
02820           INC     HL
02830           INC     DE
02840           OR      A               ; TEST OF EOS
02850           JR      NZ,ATOB10       ; JMP IF NOT END OF STAT.
02860           LD      (DE),A          ; OF MACHINE ZEROS
02870           INC     DE
02880           LD      (DE),A
02890           RET                     ; RET TO CALLER
02900 ;*
02910 ;*  ERROR PROCESSNG - RECOVER STACK SPACE
02920 ;*
02930 ERR       POP     AF              ; CLEAR STACK
02940           POP     AF              ; CLEAR STACK
02950           POP     AF              ; CLEAR STACK
02960           LD      HL,0            ; DEALLOCATE SECTOR BUFFER
02970           ADD     HL,SP           ; CSP
02980           LD      BC,256          ; SIZE OF SECTOR BUFF
02990           ADD     HL,BC           ; COMPUTE NEW CSP
03000           LD      SP,HL           ; SETUP NEW CSP
03010 ERR10     POP     AF              ; CLEAR STACK
03020           POP     AF              ; CLEAR STACK
03030           POP     AF              ; CLEAR STACK
03040           POP     AF              ; CLEAR STACK
03050           POP     AF              ; CLEAR STACK
03060           LD      A,2             ; CODE FOR SYNTAX ERROR
03070           JP      1997H           ; GIVE ERR, RTN TO BASIC
03080 ;*
03090 ;*  CONSTANTS AND COUNTERS
03100 ;*
03110 LINE      DEFW    0               ; OVERLAY LINE NO
03120 CSP       DEFW    0               ; HOLDS CSP ON ENTRY
03130 TYPE      DEFB    0               ; ORGINAL DATA TYPE
03140 LFN       DEFM    'FS'            ; COMMON VARIABLE NAME
03150           DEFB    0
03160 DCB       DEFS    32              ; OVERLAY DCB
03170 BADDR     DEFW    0               ; SECTOR BUFF ADDR ON STK
03180 VARADR    DEFW    0               ; VARIABLE ADDR ON STK
03190 CEPST     DEFW    0               ; CURRENT END OF PST
03200 LVLT      DEFW    0               ; LENGTH OF VLT
03210 SNVLT     DEFW    0               ; START ADDR OF NEW VLT
03220 LSVLT     DEFW    0               ; LENGTH OF SIMP VAR VLT
03230 PF        DEFB    0               ; PASS FLAG
03240 FI        DEFB    0               ; SECTOR BUFF INDEX
03250 RCOUNT    DEFW    -1              ; COUNT OF SECTORS READ
03260           END
```

Chapter 7

BASIC Decoded: New ROMs

The comments in chapter 8 are based on the original three chip ROM set, if you have a 2 chip ROM configuration your dissassembly will probably be slightly different.

Differences between the latest 'MEM SIZE ?' ROMs and the old ROMs are given below. Locations with an asterisk next to them have different contents than the next chapter.

When running a Disassembler be careful to check the page sequence where differences occur.

This comment chapter was designed to be used in conjunction with a disassembler that produces 62 lines per page. The Apparat NEWDOS plus Disassembler was used during the books production.

```
0050    0D       DEC --- Enter no shift (0D)                  * ASCII values
0051    0D       DEC --- Enter shift (0D)
0052    1F       RRA --- Clear no shift (1F)
0053    1F       RRA --- Clear shift (1F)
0054    01015B   LD  --- BREAK ns (01) / BREAK shift (01) / up arrow ns (5B)
0057    1B       DEC --- Up arrow shift (1B)
0058    0A       LD  --- Down arrow no shift (0A)
0059   *00       NOP --- Down arrow shift (00)
005A    08       EX  --- Left arrow no shift (08)
005B    1809     JR  --- Left arrow shift (18) / right arrow no shift (09)
005D    19       ADD --- Right arrow shift (19)
005E    2020     JR  --- Space no shift (20) / space shift (20)

00FC   *210E01   LD  --- Address of 'R/S L2 BASIC' message

0105    4D       LD  --- M                                    * MEM SIZE
0106    45       LD  --- E
0107    4D       LD  --- M
0108   *2053     JR  --- Space, S
010A   *49       LD  --- I
010B   *5A       LD  --- Z
010C   *45       LD  --- E
```

```
010D   *00       NOP   --- Message terminator
010E   *52       LD    --- R                                    * R/S L2 BASIC
010F   *2F       CL    --- /
0110   *53       LD    --- S
0111   *204C     JR    --- Space, L
0113   *322042   LD    --- 2, space, B
0116   *41       LD    --- A
0117    53       LD    --- C
0118   *49       LD    --- I
0119   *43        LD   --- C
011A   *0D        DEC  --- Carriage return
011B   *00       NOP   --- Message terminator
011C   *C5       PUSH  --- Save active row address
011D   *010005   LD    --- Delay count value
0120   *CD6000   CALL  --- Delay for 7.33 milliseconds       * Debounce routine
0123   *C1       POP   --- Restore row address
0124   *0A       LD    --- And reload original flags from active row
0125   *A3       AND   --- Then combine current flag lists with original flag bits
0126   *C8       RET   --- Rtn to caller if zero because row was not active on 2nd test
0127   *7A       LD    --- Otherwise we have a legimately active row
0128   *07       RLCA  --- Row index * 2
0129   *07       RLCA  --- Row index * 4
012A   *C3FE03   JP    --- Return to rest of keyboard driver routine

0248   *0660     LD    --- Now, delay for 476/703 microseconds

024F   *0685     LD    --- Then delay for 865/975 microseconds

02E2   *20ED     JR    --- If no match, skip to next program on cassette
02E4   *23       INC   --- We have a character match. Bump to next char of typed in name.

03FB   *C31C01   JP    --- Go to debounce routine. If legimate char rtn to 3FE, else rtn t
caller.

0683   *20F1     JR    --- Loop thru block move routine 128 times

1225    E7       RST   --- Double precision or string
1226   *300B     JR    --- Jmp if double precision

124D   *B7       OR    --- Set status flags

1265   *F24312   JP    --- No change in this comment

2067    3E01     LD    --- A = device code for printer          * LPRINT routine
2069    329C40   LD    --- Set current system device to printer
206C   *C37C20   JP
206F    CDCA41   CALL  --- DOS Exit                             * PRINT routine
2072   *FE23     CP    --- Test for #
2074   *2006     JR    --- Jmp if not PRINT #
2076   *CD8402   CALL  --- Write header on cassette file        * PRINT # routine
2079   *329C40   LD    --- Set current system device to cassette
207C   *2B        DEC  --- Backspace over previous symbol in code string
207D   *D7        RST  --- Re-examine previous char in code string
207E   *CCFE20   CALL  --- If end of string write a Carrige Return
2081   *CA6921   JP    --- If end of string turn off cassette and return
2084   *F620     OR    --- Not end of string.  Convert possible 40 to 60
2086   *FE60     CP    --- Then test for @
2088   *201B     JR    --- Jmp if not PRINT @
208A   *CD012B   CALL  --- Evaluate @ expression, result in DE  * PRINT @ routine
208D   *FE04     CP    --- A = MSB, test for @ value > 1023
```

```
208F   *D24A1E  JP    --- FC error if @ position > 1023
2092   *E5      PUSH  --- Save current code string addr
2093   *21003C  LD    --- HL = starting addr of video buffer
2096   *19      ADD   --- All tab position
2097   *222040  LD    --- And save addr in video DCB as cursor addr
209A   *7B      LD    --- Then get position within line
209B   *E63F    CP    --- And truncate it to 63
209D   *32A640  LD    --- Then save as current position within line
20A0   *E1      POP   --- Restore code string addr (starting addr of item list)
20A1   *CF      RST   --- But make sure a comma follows the tab position
20A2   *2C      INC   --- DC  2C  ','
20A3   *18C7    JR    --- Go get first variable from item list
20A5   *7E      LD    --- Reload next element from code string
20A6   *FEBF    CP    --- Test for USING token
20A8   *CABD2C  JP    --- Jmp if USING token
20AB   *FEBC    CP    --- Test for TAB token
20AD   *CA3721  JP    --- Jmp if TAB token
20B0   *E5      PUSH  --- Save current code string addr
20B1   *FEC2    CP    --- Test for a comma
20B3   *2853    JR    --- Go get next item if a comma
20B5   *FE3B    CP    --- Not comma, test for semi-colon
20B7   *285E    JR    --- Go get next item if semi-colon
20B9    CD3723  CALL  --- Evaluate next item to be printed
20BC   *E3      EX    --- Save current code string addr HL = addr of current item

20F6   *C37C20  JP    --- And loop till end of statement (EOS)

213A   *E67F    AND   --- Result in A-reg. Do not let it exceed 127

2166   *C38120  JP    --- Process next of PRINT TAB statement

226A   *00      NOP   --- Remove
226B   *00      NOP   --- Erroneous
226C   *00      NOP   --- Test
226D   *00      NOP   --- For
226E   *00      NOP   --- FD error

2C1F   *D6B2          --- Test for CLOAD?                    * CLOAD routine
2C21   *2802          --- Jmp if CLOAD?
2C23   *AF            --- Signal CLOAD
2C24   *012F23        --- 2C25: CPL A=-1 if CLOAD?, 0000 if CLOAD
2C27   *F5            --- 2C26: INC HL position to file name Save CLOAD? / CLOAD flag
2C28   *7E            --- Get next element from code string. Should be file name
2C29   *B7            --- Set status flags
2C2A   *2807          --- Jmp if end of line
2C2C   *CD2723        --- Evaluate expression (get file name)
2C2F   *CD132A        --- Get addr of file name into DE
2C32   *1A            --- Get file name
2C33   *6F            --- And move it to L-reg
2C34   *F1            --- Restore CLOAD? / CLOAD flags
2C35   *B7            --- Set status register according to flags
2C36   *67            --- H=CLOAD?/CLOAD flag, L=file name
2C37   *222141        --- Save flag and file name in WRA1
2C3A   *CC4D1B        --- If CLOAD call NEW routine to initialize system variables
2C3D   *210000        --- This will cause the drive to be selected when
2C40   *CD9302        --- We look for leader and synch byte
2C43                  --- Restore CLOAD? / CLOAD flag, file name

2FFB   *DEC3          --- These instructions
2FFD   *C344B2        --- Are not used by Level II
```