NEVADA
FORTRAN

ELLIS COMPUTING™
SOFTWARE TECHNOLOGY
COPYRIGHT

Copyright (C), 1979, 1980, 1981, 1982, 1983 by Ian D. Kettleborough. All rights reserved worldwide. No part of the publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated into any human or computer language in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the express written permission of Ian D. Kettleborough.

TRADEMARKS

NEVADA FORTRAN (tm), NEVADA COBOL (tm), NEVADA PILOT (tm), NEVADA EDIT (tm) and Ellis Computing(tm) are trademarks of Ellis Computing, Inc. CP/M is a registered trademark of Digital Research, Inc.

DISCLAIMER

All Ellis Computing computer programs are distributed on an "AS IS" basis without warranty.

Ellis Computing, Inc. makes no warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event will Ellis Computing, Inc. be liable for consequential damages even if Ellis Computing, Inc. has been advised of the possibility of such damages.

Printed in the U.S.A.
# Table of Contents

1. **GETTING STARTED** .................................................. 2
   1.1. Configuring the NEVADA FORTRAN system ...................... 4
       1.1.1. Generating the error file FORT.ERR ..................... 4
       1.1.2. Configuring the FORTRAN system ...................... 4

2. **COMPILING AND EXECUTING A PROGRAM** .......................... 5
   2.1. CREATING A PROGRAM ........................................... 5
   2.2. RUNNING THE COMPILER ......................................... 5
   2.3. COMPILE OPTIONS ............................................... 6
   2.4. EXECUTING A PROGRAM ......................................... 8
   2.5. CREATING A COM FILE ......................................... 9

3. **THE FORTRAN LANGUAGE** ........................................... 10
   3.1. The FORTRAN Character Set .................................. 10
   3.2. FORTRAN Program Structure .................................. 11
   3.3. FORTRAN Statements ......................................... 13
   3.4. Multi-statements .............................................. 14
   3.5. FORTRAN Program Preparation ................................. 15
   3.6. The COPY Statement ........................................... 16
   3.7. The Options Statement ........................................ 17

4. **NUMBER SYSTEM** ................................................... 20
   4.1. Internal Format of Numbers .................................. 20
   4.2. Number Ranges .................................................. 20
   4.3. Constants ....................................................... 21
       4.3.1. Numerical Constants .................................... 21
       4.3.2. String Constants ....................................... 22
       4.3.3. Logical constants ...................................... 22
   4.4. Variable Names ................................................ 23
   4.5. Type Specification ............................................ 24
       4.5.1. INTEGER .................................................. 25
       4.5.2. LOGICAL .................................................. 26
       4.5.3. REAL ..................................................... 27
       4.5.4. DOUBLE PRECISION ....................................... 28
   4.6. Data Statement ................................................ 29
   4.7. Common Blocks ................................................. 30
   4.8. Implicit Statement ............................................ 32

5. **Expressions** ...................................................... 33
   5.1. Hierarchy of Operators ...................................... 33
   5.2. Expression Evaluation ....................................... 34
   5.3. Integer Operations ........................................... 35
   5.4. Real Operations ............................................... 36
   5.5. Logical Operations .......................................... 37
   5.6. Mixed Expressions ............................................ 39

6. **Control Statements** .............................................. 40
   6.1. Unconditional GO TO Statement ................................ 41
   6.2. Computed GO TO Statement ................................... 42
   6.3. Assigned GO TO ................................................ 43
   6.4. ASSIGN ........................................................ 44
   6.5. Arithmetic IF Statement ..................................... 45
   6.6. Logical IF Statement ......................................... 46
   6.7. IF-THEN-ELSE .................................................. 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8</td>
<td>DO-LOOPS</td>
<td>49</td>
</tr>
<tr>
<td>6.9</td>
<td>CONTINUE Statement</td>
<td>50</td>
</tr>
<tr>
<td>6.10</td>
<td>ERROR TRAPPING</td>
<td>51</td>
</tr>
<tr>
<td>6.11</td>
<td>CONTROL/C CONTROL</td>
<td>53</td>
</tr>
<tr>
<td>6.12</td>
<td>TRACING</td>
<td>54</td>
</tr>
<tr>
<td>6.13</td>
<td>DUMP statement</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>Program Termination Statements</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>PAUSE Statement</td>
<td>56</td>
</tr>
<tr>
<td>7.2</td>
<td>STOP Statement</td>
<td>57</td>
</tr>
<tr>
<td>7.3</td>
<td>END Statement</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>Array Specification</td>
<td>59</td>
</tr>
<tr>
<td>8.1</td>
<td>Dimension Statement</td>
<td>60</td>
</tr>
<tr>
<td>8.2</td>
<td>Subscripts</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>Subprograms</td>
<td>63</td>
</tr>
<tr>
<td>9.1</td>
<td>SUBROUTINE Statement</td>
<td>64</td>
</tr>
<tr>
<td>9.2</td>
<td>FUNCTION Statement</td>
<td>65</td>
</tr>
<tr>
<td>9.3</td>
<td>CALL Statement</td>
<td>66</td>
</tr>
<tr>
<td>9.4</td>
<td>RETURN Statement</td>
<td>67</td>
</tr>
<tr>
<td>9.5</td>
<td>Multiple return</td>
<td>68</td>
</tr>
<tr>
<td>9.6</td>
<td>BLOCK DATA SUBPROGRAM</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>Input/Output</td>
<td>70</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction to FORTRAN I/O</td>
<td>70</td>
</tr>
<tr>
<td>10.1.1</td>
<td>General Information</td>
<td>70</td>
</tr>
<tr>
<td>10.1.2</td>
<td>I/O List Specification</td>
<td>72</td>
</tr>
<tr>
<td>10.2</td>
<td>READ Statement</td>
<td>74</td>
</tr>
<tr>
<td>10.3</td>
<td>WRITE Statement</td>
<td>76</td>
</tr>
<tr>
<td>10.4</td>
<td>MEMORY TO MEMORY I/O statements</td>
<td>77</td>
</tr>
<tr>
<td>10.4.1</td>
<td>DECODE statement</td>
<td>78</td>
</tr>
<tr>
<td>10.4.2</td>
<td>ENCODE statement</td>
<td>79</td>
</tr>
<tr>
<td>10.5</td>
<td>Format Statement and Format Specifications</td>
<td>80</td>
</tr>
<tr>
<td>10.5.1</td>
<td>X-Type (wX)</td>
<td>81</td>
</tr>
<tr>
<td>10.5.2</td>
<td>I-Type (Iw)</td>
<td>81</td>
</tr>
<tr>
<td>10.5.3</td>
<td>A-Type (Aw)</td>
<td>81</td>
</tr>
<tr>
<td>10.5.4</td>
<td>/-TYPE (/)</td>
<td>82</td>
</tr>
<tr>
<td>10.5.5</td>
<td>Z-Type</td>
<td>82</td>
</tr>
<tr>
<td>10.5.6</td>
<td>L-Type (Lw)</td>
<td>83</td>
</tr>
<tr>
<td>10.5.7</td>
<td>T-Type (Tw)</td>
<td>83</td>
</tr>
<tr>
<td>10.5.8</td>
<td>K-Type (Kw)</td>
<td>84</td>
</tr>
<tr>
<td>10.5.9</td>
<td>F-Type (Fw.d)</td>
<td>85</td>
</tr>
<tr>
<td>10.5.10</td>
<td>E-Type (Ew.d)</td>
<td>86</td>
</tr>
<tr>
<td>10.5.11</td>
<td>D-Type (Dw.d)</td>
<td>86</td>
</tr>
<tr>
<td>10.5.12</td>
<td>G-Type (Gw.d)</td>
<td>87</td>
</tr>
<tr>
<td>10.5.13</td>
<td>Repeating field specifications</td>
<td>88</td>
</tr>
<tr>
<td>10.5.14</td>
<td>String Output</td>
<td>89</td>
</tr>
<tr>
<td>10.6</td>
<td>Free Format I/O</td>
<td>90</td>
</tr>
<tr>
<td>10.6.1</td>
<td>INPUT</td>
<td>90</td>
</tr>
<tr>
<td>10.6.2</td>
<td>OUTPUT</td>
<td>90</td>
</tr>
<tr>
<td>10.7</td>
<td>BINARY I/O</td>
<td>92</td>
</tr>
<tr>
<td>10.8</td>
<td>REWIND Statement</td>
<td>93</td>
</tr>
<tr>
<td>10.9</td>
<td>BACKSPACE Statement</td>
<td>94</td>
</tr>
<tr>
<td>10.10</td>
<td>ENDFILE Statement</td>
<td>95</td>
</tr>
<tr>
<td>10.11</td>
<td>GENERAL COMMENTS ON FORTRAN I/O UNDER CP/M</td>
<td>96</td>
</tr>
</tbody>
</table>
10.12. SPECIAL CHARACTERS DURING CONSOLE I/O .......... 97

11. General Purpose SUBROUTINE/FUNCTION Library .......... 98
   11.1. OPEN ........................................... 100
   11.2. LOPEN .......................................... 100
   11.3. CLOSE ........................................... 104
   11.4. DELETE ........................................... 104
   11.5. SEEK ............................................ 105
   11.6. RENAME .......................................... 105
   11.7. CHAIN ............................................ 106
   11.8. LOAD .............................................. 106
   11.9. EXIT .............................................. 107
   11.10. MOVE ............................................. 107
   11.11. DELAY ........................................... 108
   11.12. CIN .............................................. 108
   11.13. CTEST ........................................... 109
   11.14. OUT ............................................... 110
   11.15. SETIO ............................................ 110
   11.16. RESET ............................................ 111
   11.17. POKE .............................................. 111
   11.18. BIT ............................................... 112
   11.19. PUT ............................................... 112
   11.20. CHAR ............................................. 113
   11.21. INP ............................................... 113
   11.22. CALL .............................................. 113
   11.23. CBTOF ............................................ 114
   11.24. PEEK .............................................. 114
   11.25. COMP ............................................. 115

Appendices

A. Statement Summary ........................................ 116
B. Summary of System Function ............................. 120
C. Summary of System Subroutines ........................... 121
D. RUNTIME ERRORS ...................................... 123
E. COMPILe TIME ERRORS .................................... 128
F. ASSEMBLY LANGUAGE INTERFACE ........................... 132
G. GENERAL COMMENTS ...................................... 133
H. USE OF THE NORTH STAR FLOATING POINT BOARD ......... 135
This is an 8080/8085/Z80 version of FORTRAN IV. It is a powerful subset implementation of this widely used language. The compiler works from disk (also using the assembler) to produce 8080/8085/Z80 machine code that executes at maximum CPU speed.

A source program is entered as FORTRAN IV program statements. These statements must follow the conventions outlined in this document or errors may result. The compiler acts upon the source statements to produce assembly code. At this stage any mistakes are flagged with error messages. If an error should occur, the source may be corrected at this time and recompiled. After the program has been compiled without any errors, the final step (normally transparent to the user) is to assemble the intermediate code into 8080 object code. The object module is then ready for execution under CP/M (or compatible operating system).

This manual is intended as a guide to using this version of FORTRAN and is not intended to be a complete instruction manual on the use of FORTRAN. This manual assumes that you already know how to program in FORTRAN and have read the CP/M operating system manuals. If you do not know FORTRAN there are many books that explain the syntax and semantics of the FORTRAN language. This manual explains the subset that is implemented in NEVADA FORTRAN.

This manual describes versions 3.0 and later of the NEVADA FORTRAN compiler.

I would like to thank Colegate V. Spinks for his help and work in developing the initial release of the compiler, runtime and manual for this FORTRAN implementation.
1. GETTING STARTED

HARDWARE REQUIRED

1. 8080/8085/Z80 processor
2. A minimum of 48K of RAM for the compiler
3. At least one disk drive

SOFTWARE REQUIRED

1. CP/M Operating System
2. Any text editor

FILES ON THE DISTRIBUTION DISK

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORT.COM</td>
<td>is the FORTRAN compiler</td>
</tr>
<tr>
<td>FRUN.COM</td>
<td>is the runtime execution package</td>
</tr>
<tr>
<td>CONFIG.COM</td>
<td>is a program to generate the error file and setup compiler and runtime defaults.</td>
</tr>
<tr>
<td>ERRORS</td>
<td>is the error text file used by CONFIG.</td>
</tr>
<tr>
<td>READ.ME</td>
<td>maybe on the disk and contains information and updates to this manual.</td>
</tr>
</tbody>
</table>

If room permits, there maybe some sample programs on the disk.

(Also see the NEVADA Assembler Manual for other files not listed here)

GETTING STARTED

If the master disk is not write protected, do it now!!

The very first thing that you should do is to make at least one backup copy of your NEVADA FORTRAN diskette. The original diskette should be kept in a safe place in case it is ever needed in the future.

1. NEVADA FORTRAN is distributed on a DATA DISK without the CP/M operating system. This disk will not "boot up".

2. On computer systems with the ability to read several disk formats, such as the KayPro computer, the master diskette must be in disk drive B.

3. Do not try to copy the master diskette with a COPY program! On most systems it won't work. You must use the CP/M PIP command to copy the files.
4. You must first, prepare a CP/M diskette for use as your NEVADA FORTRAN operations diskette. Usually you will use your system's FORMAT program (sometimes the COPY program will format the diskette) to prepare this diskette.

5. You should now use PIP to make copies of the original NEVADA FORTRAN diskette. To use PIP use the command:

   PIP A:=B:*.*[V]

   (this specifies that the NEVADA FORTRAN distribution disk is in drive B and the disk to copy onto is in drive A.)

   On 5 1/4" diskettes you may have to remove (erase) other programs to make room for all the NEVADA FORTRAN programs, before the next step. Make sure the default drive has at least 8K of available space. If it does not, you will get a BDOS write error - CP/M's way of letting you know the

   **Disk is full.**

   In some cases you will have to temporarily remove some of the FORTRAN example programs to make the space available.

**NOTE:** The NEVADA assembler, ASSM.COM, must be on the same disk which contains the FORTRAN compiler.
1.1. Configuring the NEVADA FORTRAN system

1.1.1. Generating the error file FORT.ERR

The program CONFIG reads the text file ERRORS which contains the compiler error messages. These messages may be changed with the restriction that they can only be one line, the first 2 characters are the error number followed by a blank followed by the text of the error message. To generate the error file, just enter CONFIG at the CP/M prompt and reply Y to the question about generating the error file. You will then be asked to specify which drive contains the file ERRORS and which drive the file FORT.ERR is to be written to. Reply with a valid drive letter (A, B, ...) for your system. You must generate the error file as it is not supplied on the disk. This only needs to be done once or whenever any of the error text is changed.

1.1.2. Configuring the FORTRAN system

The CONFIG program also allows you to set certain default values in both the compiler and the runtime package. Just enter carriage return (or ENTER) to leave the default as it is, or enter the new default value. You will be asked to enter the drive that contains the compiler (FORT.COM). The default sizes of the parameters that can be changed with an OPTIONS statement can be modified along with the character used to delimit hexadecimal constants in strings. You will also be able to specify if your system console can handle lowercase letters. Enter Y if it can or N if it cannot.

Next, certain parameters in the runtime package can be set. You must first specify which drive contains the runtime package (FRUN.COM). The method that the runtime package performs CP/M console I/O can be specified as either CP/M function 1&2, CP/M function 6 (for CP/M rev 2.0 only) or direct BIOS calls. Specifying CP/M functions 1&2 allows you to use control-P to send a copy of your FORTRAN output to the printer. You will also be able to specify if your system console can handle lowercase letters. Enter Y if it can or N if it cannot.

Since NEVADA FORTRAN supports the North Star floating point board, you must specify the address of the board if you have one and want FORTRAN to use it.

After creating FORT.ERR you can erase the files CONFIG.COM and ERRORS if you need the disk space.
2. COMPILING AND EXECUTING A PROGRAM

2.1. CREATING A PROGRAM

A program can be created with any of the numerous available text editors. The name of the FORTRAN source program should have the filename extension of .FOR, such as PROG.FOR. Refer to section 3.2 for a detailed description of the format of each source statement. Programs are created with a text editor and are later, in a separate step, compiled.

2.2. RUNNING THE COMPILIER

The general format of the command to compile a FORTRAN program is:

FORT U:PGM.LAO $OPTIONS

where:

FORT is FORT.COM, the FORTRAN compiler
PGM is the FORTRAN source program to compile and has the extension .FOR.
U: is the drive where PGM.FOR is located (if not present, the default drive is used).
L is the drive for the listing as follows:

A-P uses that drive for the listing
X listing to CP/M console
Y listing to CP/M LST: device
Z do not generate a listing

The listing will have the same filename as the source file but with the extension .LST.

A is the drive for the intermediate assembly file as follows:

A-P uses that drive
Z do not generate an assembly file.

The assembly file will have the same filename as the source file but with the extension .ASM. This file is normally deleted by the FORTRAN compiler.

O is the drive for the final object program as follows:

A-P uses that drive
Z don't generate an object file
The object program will have the same filename as the source file but with the extension .OBJ.

Notes:

If Z if specified in either the assembly or object drive position, no object program will be generated.

If the three drive specifiers are not specified, then the default drive will be used.

The files FORT.ERR and ASSM.COM must be present on the default drive when the compiler is run.

If the O is not specified as Z, then the assembly file will be automatically assembled and the intermediate .ASM file will be deleted. If Z is specified, then the file will not be assembled and the intermediate .ASM will remain on the disk.

2.3. COMPILIE OPTIONS

Options that effect the compilation of the FORTRAN program can be specified on the command line by preceding the option string with dollar sign ($). For example:

FORT U:PGM.LAO $NP2

The following options can be specified:

N
No assembly file will be produced (and no object file also).

P
The listing file (if specified) will be paginated (66 lines to a page). Each new FORTRAN routine will start on a new page.

1
Source statements will be blank padded to 64 characters.

2
Source statements will be blank padded to 72 characters.
NOTE: Normally source statements are not blank padded. This may cause a problem where blanks are wanted inside a literal string and the string is started on one statement and continued over one or more continuation statements. Without the pad option, the trailing blanks may be lost (of course you could break the continued literal into several, making sure that there is a quote after any blanks at the end of a statement). For example:

```
WRITE (1,10)
10 FORMAT ('THIS IS
*A TEST')
```

Produces: **THIS IS A TEST**
without blank padding and

```
WRITE (1,10)
10 FORMAT ('THIS IS
*A TEST')
```

with blank padding.

If the last form is desired then this could be written as:

```
WRITE (1,10)
10 FORMAT ('THIS IS
*A TEST')
```

to produce the same results as with the blank padding specified.

**H**

This option is used in conjunction with the P option to suppress the heading in the listing.

**C=XXXX**

This option specifies the maximum number of COMMON blocks that may be defined in the program to be compiled. The default is 15.

**B=XXXX**

This option specifies the size of the input statement buffer. The default is 530 characters and the buffer must be large enough to contain a complete statement (first record plus all continuations).

**M=XXXX**

This option specifies the memory address at which blank COMMON will end. In other words, blank COMMON will be allocated downward in memory from the specified address. The address specified must be in hexadecimal.
This option is useful in forcing blank COMMON to be allocated at the same address in memory for passing data between routines that CHAIN to each other.

Examples

FORT MYPROG $C=20

Compiles MYPROG.FOR from the default drive, generating MYPROG.ASM, MYPROG.LST and MYPROG.OBJ on the default drive and allowing for the definition of up to 20 COMMON blocks.

FORT B:READ.XCD $P2

Compiles READ.FOR from drive B, generating READ.ASM on drive C and the listing to the console. The listing will be paginated and source statements will be padded to 72 characters.

FORT TEST.YZZ $P

Compiles TEST.FOR from the default drive, no .ASM or .OBJ file will be produced but a paginated listing will go to the CP/M list (LST:) device.

FORT UPDATE.XBZ $PH

Compiles UPDATE.FOR from the default drive, generating UPDATE.ASM on drive B, a paginated listing minus the heading line to the console and no .OBJ file.

2.4. EXECUTING A PROGRAM

Once the object file has been produced, the program can be executed by simply typing:

FRUN u:filename

Where u: is optional and if not present, the default drive is used. The FORTRAN runtime package, FRUN occupies memory from 100H to 3FFFH. It will load the program to be executed starting at 4000H. The program is then executed and continues until either it terminates normally or a runtime error occurs.

Example

To compile and run the program GRAPH (listing in the sample programs section) the commands would be:

FORT GRAPH (listing, object to default disk)
FRUN GRAPH (will execute the program)
2.5. CREATING A COM FILE

A CP/M .COM file can be created that contains a copy of the runtime package and the program to be executed. This has the advantage that just the filename need be entered to execute the program. Each program generated in this way will be at least 16K in length, that being the size of the FORTRAN runtime package. To create a COM file just add .C to the end of the FRUN command. The command to turn GRAPH.OBJ into GRAPH.COM would be:

FRUN GRAPH.C

Then to execute it, all that is needed is the command:

GRAPH
3. THE FORTRAN LANGUAGE

3.1. The FORTRAN Character Set

The FORTRAN character set is composed of the following characters:

The letters:


The numbers:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The special characters:

blank
= equal sign (for replacement operations)
+ plus sign
- minus sign
* asterisk
/ slash
( left parenthesis
) right parenthesis
, comma
. decimal point
$ dollar sign
# number sign
& ampersand
\ backslash

NOTE: Lowercase letter will be converted to uppercase except when lowercase appears in string literals.
The following is a list of the meanings of the special characters used in this version of FORTRAN:

$  Preceding a constant with a dollar sign indicates that it is a hexadecimal constant.

#  Preceding a constant with a number sign indicates that it is a hexadecimal constant that is to be stored internally in binary format.

&  The & has two functions:
   1) if used in a FORMAT statement contains an ampersand, the character following the ampersand is interpreted as a control character (unless it is also an &).
   2) used to indicate that a statement label is being passed to a SUBROUTINE for use in a multiple RETURN statement.

\  A constant enclosed in backslashes in a character string is assumed to be the hexadecimal code for an ASCII character.

3.2. FORTRAN Program Structure

A FORTRAN program is comprised of statements. Every statement must be of the following format.

1) The first 5 characters of the statement may contain a statement label if the statement is to be branched to.

2) The sixth character is used to indicate a continuation of the previous statement. Continuation is indicated by placing any character except a BLANK or ZERO in column 6 of the continuation statement.

3) Column 7 to the end of the line is used for the body of the statement. This is any one of the following statements which will be described later. All statements are terminated by a CR (Carriage Return) or semicolon (not enclosed in a character string) in the case of multiple statements per line. A statement may be of any length, but only the first 72 characters are retained during compilation. Statements will be processed until the carriage return is encountered. The character positions between the carriage return and character position 72 will NOT be padded with BLANKS as some FORTRAN systems will do, unless the 1 or 2 option is specified This means that if a character string is started on a line, and must be continued, the continuation logically starts immediately after the last character of the previous line.
4) Column 73 through 80 are used for identification purposes and are ignored.

5) A comment line is indicated by place a C in column 1. A comment line has no effect on the program and is ignored. It is only for documentation purposes.

Example

|-- Column 1 |

V

WRITE (1,2)
2 FORMAT ('THIS IS AN
* EXAMPLE CHARACTER STRING')

will output: THIS IS AN EXAMPLE CHARACTER STRING

Uppercase and lowercase letters can be intermixed in a FORTRAN statement. Lowercase letters are retained ONLY when they appear between QUOTES or in the H FORMAT specification in a FORMAT statement. Otherwise they will be converted to uppercase internally. Thus the variables QUANTITY and quantity and QuAnTiTy represent the same variable.

There are four types of statements in FORTRAN:

1) Declaration
2) Assignment
3) Control
4) Input/Output.

These statement types are described in the following sections of this manual.
3.3. FORTRAN Statements

A statement may contain a statement label. A statement label is placed in columns 1 through 5 of the statement.

All labels on statements must be integers ranging between 1 and 99999. Leading zeros will be ignored.

Statement labels are not required to be in any sequence, and they will not be put in order.

In any program, a statement label can be used only once as a label.

A statement may contain no more than 530 characters excluding blanks (other than those between single quotes), unless the B= option is specified.

During compilation, blanks are ignored, except between single quotes and in H FORMAT specification.

Comments are indicated by placing a C in column 1; the remaining part of the statement may be in any format and is ignored by the compiler.
3.4. Multi-statements

Statements may be compacted more than one logical statement per line. Statements are separated from each other with a semicolon and a colon is used to separate the label, if any.

Example

```
A=1
3 CONTINUE
A=A+1
TYPE A
GO TO 3
END
```

could be written as:

```
A=1;3:CONTINUE;A=A+1;TYPE A;GOTO 3;END
```
3.5. FORTRAN Program Preparation

A FORTRAN source program is prepared using one of the available CP/M text editors. The FORTRAN file must be in the following format:

Position 1...5....0....5....0....5....0....5....0....5

OPTIONS
  :
  :
  :
FORTRAN program
  :
  END
OPTIONS
  SUBROUTINE X
  :
FORTRAN routine
  END

All FORTRAN routines are **required** to be compiled at one time.
3.6. The COPY Statement

A FORTRAN program can contain COPY statements. The COPY statement contains the word COPY followed by at least one blank, followed by the FILENAME to be inserted at that point. COPY files may contain complete programs or just sections of programs. Copied files may not themselves contain COPY statements.

Example

```
DIMENSION A(1)
COPY ALLDEFS
READ (1,10) I

A=1
T=5
CALL ADDIT
STOP
END
COPY B:ADDIT
```
3.7. The Options Statement

This is an optional statement of each program and/or subprogram which is to be compiled. If present, the OPTIONS statement must appear as the first statement in main program and prior to the SUBROUTINE or FUNCTION statement in each subprogram. The options statement allows the specification of various parameters to be used by the compiler during compilation of a particular routine. The options that are specified on an options statement are only in effect for that routine and revert back to the default unless an OPTIONS statement appears on subsequent routines. Options available are as follows:

\[ S=n \]

\( n \rightarrow \) Is a decimal number indicating the number of allowable symbols. The default is 50. Each entry requires 8 bytes. (\( n \) may be greater than 255). This default can be changed using the CONFIG program.

\[ L=n \]

\( n \rightarrow \) Is a decimal number indicating the number of allowable labels. The default is 50. Each entry requires 6 bytes. (\( n \) may be greater than 255). This default can be changed using the CONFIG program.

\[ T=n \]

\( n \rightarrow \) Is a decimal number indicating the maximum number of temporary variables that are available during EXPRESSION evaluation. Default table size is 15; each variable requires 1 byte. This default can be changed using the CONFIG program.

\[ D=n \]

\( n \rightarrow \) Is a decimal number which indicates the maximum allowable nesting of DO loops. Default is 5, each entry requires 4 bytes. This default can be changed using the CONFIG program.
A=n

n -> Is a decimal number which indicates the maximum number of arrays. Default is a maximum of 15; each entry requires 4 bytes. This default can be changed using the CONFIG program.

O=n

n -> Is a decimal number which indicates the maximum number of operators ever pushed on the internal stack while doing a prefix translation of input expression. Note functions and array subscripting require a double entry. Default is 40; each entry is 2 bytes long. This default can be changed using the CONFIG program.

P=n

n -> Is a decimal number which indicates the maximum number of variables and/or constants ever pushed on the internal stack in evaluation. Default is 40; each entry is 2 bytes long. This default can be changed using the CONFIG program.

I=n

n -> Is a decimal number specifying the depth that IF-THEN-ELSE's may be nested. The default nesting is 5. This default can be changed using the CONFIG program.

E

Instructs the compiler to list, as comments, a reference table equating user symbols, constants, and labels to internally generated ones.

G

Instructs the compiler to list all compile errors as error numbers, instead of explicit error statements. See Appendix 11.6.0 for a list of error numbers and their meanings.
X

Instructs the compiler to generate code which will give explicit runtime errors. In this mode each statement has an extra 5 bytes of overhead to keep track of the statement number of the statement currently being executed.

N

Check for FORTRAN errors only. Do not output an assembly code file.

B

The FORTRAN source statement is included in the assembly file as a comment.

Q

This option must be used whenever the program expects to trap runtime errors. It causes code to be generated for handling user trapping of runtime errors.

n is less than or equal to 255 unless otherwise stated.

Example

$OPTIONS X,G,S=200,L=100

Options used will be:

- EXPLICIT runtime errors will be generated
- EXPLICIT compile errors are not generated
- the SYMBOL table has room for 200 symbols, and
- the LABEL table has room for 100 statement labels.
4. NUMBER SYSTEM

4.1. Internal Format of Numbers

Numbers are stored internally as a 6 byte BCD number containing 8 digits, a one byte exponent, and a sign byte. This allows for the number to range from $1.00000000 \times 10^{-127}$ to $9.99999999 \times 10^{127}$. The sign byte contains the sign of the number; 0 indicating a positive number and 1 indicating a negative number. The exponent is stored in excess 128. A one for the sign of the BCD number indicates a negative number. The number ZERO is stored as an exponent of zero; the rest of the number is ignored.

All numbers in FORTRAN are stored in the following format:

```
+-------+-------+-------+-------+-------+-------+
| 199 | 99 | 199 | 99 | 199 | 0 | S | F | F |
+-------+-------+-------+-------+-------+-------+
```

: BCD Number : Sgn : Exp :

4.2. Number Ranges

Integer variables and constants can have any value from $-99999999$ to $+99999999$. Real variables and constants can take any value between $-0.99999999 \times 10^{-127}$ and $0.99999999 \times 10^{126}$. Integer variables and constants are stored internally in the same format.
4.3. Constants

A constant is a quantity that has a fixed value. A numerical constant is an integer or real number; a string constant is a sequence of characters enclosed in single quotes. A logical constant has a value of .TRUE. or .FALSE.

4.3.1. Numerical Constants

Numerical constants can be either integer or real as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1, 3099, -70</td>
</tr>
<tr>
<td>Real</td>
<td>1.34, -5.98, 1.4E10</td>
</tr>
</tbody>
</table>

A hexadecimal constant can be specified by preceding the number with a dollar sign. A hexadecimal constant is converted internally into an integer and stored that way. The maximum value for a hexadecimal constant is $FFFF.

Example

$8050
I=$1000
z=-$CC00

Another way to specify a hexadecimal constant is to precede the constant with a # sign. This way of representing a hexadecimal number differs in that the number is NOT converted to integer format and is stored in binary in the first two bytes of the constant. The number is stored high byte followed by low byte.

Example

#0D00
i=#127F

$805F is stored internally as: 32 86 30 00 00 00 85
#805F is stored internally as: 5F 80 00 00 00 00 00
4.3.2. String Constants

A string constant is specified by enclosing a sequence of characters in single quotes. A single quote within a character string must be represented by TWO quotes in a row (with no space between these two quotes). By specifying a hexadecimal number within backslashes, any character (even unprintable ones) can be generated.

Example

'This is a string constant'
'This string constant'"contains a single quote'
'Good\21\' is equivalent to 'Good!'
'\7F\' is equivalent to a rubout

Warning: Never include \0\ as part of a string constant as that character is used internally to indicate the end of a string.

NOTE The character used to delimit a hexadecimal number (default is \) can be changed using the CONFIG program.

4.3.3. Logical constants

The two logical constants are .TRUE. and .FALSE.. Numerically, .FALSE. has the value 0 (zero) and .TRUE. has the value 1 however any non-zero value will be considered as .TRUE. Logical operations always return a value of 0 or 1. These logical constants can be assigned to any variable, but is usually used as part of a logical expression.

Example

I=.TRUE.
I=(J .and. .TRUE)
4.4. Variable Names

A variable is a symbolic name given to a quantity which may change depending upon the operation of a program. A variable consists of from 1 to 6 alphanumeric characters, the first of which must be a letter.

An INTEGER variable is a variable that starts with I, J, K, L, M or N by default or explicitly typed INTEGER through the use of an INTEGER or IMPLICIT statement.

A REAL variable is a variable that starts with other than I, J, K, L, M or N by default or explicitly typed REAL through the use of a REAL or IMPLICIT statement.

A DOUBLE PRECISION variable must be explicitly typed DOUBLE PRECISION with a DOUBLE PRECISION or IMPLICIT statement.

A LOGICAL variable must be explicitly typed LOGICAL with a LOGICAL or IMPLICIT statement.

There are four types of variables supported: INTEGER, REAL, DOUBLE PRECISION and LOGICAL.

Example

I10, ALPHA, BETA, I, MAXIM, MINII
IX=34
ALPHA=56.34
ZLOG=.TRUE.
4.5. Type Specification

There are three type specification statements that can be used to override the default types of variables. Remember that variables that begin with the letters I, J, K, L, M, N (unless changed by an IMPLICIT statement) will be of type INTEGER. All others will be of type REAL. The type specification statement overrides the default type of a variable.

Note an array can also be specified in a type statement.

Example

INTEGER A,ZOT,ZAP(10)
REAL INT
LOGICAL LOG1,LOG2
4.5.1. INTEGER

The general format of the INTEGER statement is:

```
INTEGER v1,v2
```

The INTEGER statement is used to explicitly override the default type of the variable. Should a variable occur in the declaration string, the type is automatically set to integer. This works for both subscripted and nonsubscripted variables. A variable can appear only ONCE in a type specification statement.

Example

```
INTEGER MODE,K453,NUMBER(40),MAXNUM
INTEGER ZAPIT
```
The general format of the LOGICAL statement is:

```
LOGICAL vl,v2
```

The LOGICAL statement is used to override the default specification and type a variable as Logical. A logical variable's value is interpreted as:

- .TRUE. if the variable has a non-zero value.
- .FALSE. if the variable has a zero value.

Example

```
LOGICAL FTIME, LTIME
LOGICAL FLAG
```
4.5.3. REAL

The general format of the REAL statement is:

    REAL vl,v2

The REAL statement is used to explicitly override the default type of the variable. Should a variable occur in the declaration string, the type is automatically set to real. This works for both subscripted and nonsubscripted variables. A variable can appear only ONCE in a type specification statement.

Example

    REAL ALPHA,BETA(56),INIT,FIRST,ZAPIT,HI
4.5.4. DOUBLE PRECISION

The general format of the DOUBLE PRECISION statement is:

```
DOUBLE PRECISION v1,v2
```

The DOUBLE PRECISION statement is used to explicitly override the default type of the variable. Should a variable occur in the declaration string, the type is automatically set to real. This works for both subscripted and nonsubscripted variables. A variable can appear only ONCE in a type specification statement.

Example

```
DOUBLE PRECISION ALPHA,BETA(56),INIT,FIRST,ZAPIT,HI
DOUBLE PRECISION VALUE1,VALUE2
```

WARNING: Even though the DOUBLE PRECISION statement is supported, double precision arithmetic is NOT. All DOUBLE PRECISION variables will be treated as if they where REAL. A warning will be issued each time a DOUBLE PRECISION statement is encountered.
4.6. Data Statement

The DATA statement is used to initialize variables or arrays to a numeric value or character string. The general format is:

```
DATA list/nl,n2.../,list1/nl,n2/
```

where list is a list of variables (or array elements) to be initialized and nl, n2.. are numbers or strings (constants) that the corresponding item of list will be initialized to. An exception to this is the array name. If only the name of the array (no subscripts) appears in list, the whole array will be initialized. It is expected that enough constants will be listed to completely fill the array. If not enough constants are supplied to fill the entire array, then portions of the array will be undefined. Subprogram arguments may not appear in list. When a DATA statement is encountered during compilation, it is stored in memory and ALL DATA statements are processed when the END statement for the particular routine is encountered. If there are more DATA statements than can be stored in the available memory, a fatal compile error will result and compilation will terminate. Since DATA statements are processed when the END statement is encountered; errors in a DATA statement will be printed after the END statement. These errors will include the four digit FORTRAN assigned line number and the variable in the DATA statement being processed when the error occurred.

Example

```
DIMENSION B(3),C(3)
DATA A/l/,B/1,2,3/,C/3*0/
DATA LIST/'THIS IS A CHARACTER STRING'/
```

The above statement will assign the value 1 to A and the values 1 to B(1), 2 to B(2) and 3 to B(3). The asterisk is used to indicate a repeat count; thus the array C will be set to zeroes. An error will result if a variable in a DATA statement is not used elsewhere in a program.

NOTE: The other form of the DATA statement:

```
DATA A,B,C/1,2,3/
```

is not supported by NEVADA FORTRAN and must be rewritten as:

```
DATA A/1/,B/2/,C/3/
```
4.7. Common Blocks

The COMMON block declaration sets aside memory (variable space) to be shared between routines (SUBROUTINES, FUNCTIONS and the main program). Common blocks are associated with a name which is used by each declaring routine to point to a specific COMMON block.

The general form of a COMMON statement is:

```
COMMON /namel/ list1 /name2/ list2
```

where namel and name2 are the COMMON block names associated with the corresponding list1 and list2.

Example

```
DIMENSION X(100)
COMMON /ZZZ/ FIRST,LAST,X
CALL ADDEM
.
END

SUBROUTINE ADDEM
REAL NUMBER
COMMON /ZZZ/ F,L,NUMBER(100)
.
END
```

An array declaration may be included in a COMMON statement as shown in the subroutine above. The use of common blocks allow data to be passed to and from a subprogram, but without passing it as arguments (in a heavily called routine, this method can save execution time). If an array is to be included in a common declaration, it must either be declared previously or declared in the COMMON statement.
If the name is omitted or the name is null (i.e. //) then it is called blank COMMON.

Example

COMMON A,B,C,D
COMMON // A,B,C,D are equivalent statements

Blank COMMON differs from named COMMON in the following ways:

1) variables in blank common are allocated their actual memory addresses at runtime and therefore cannot be initialized with a DATA statement.

2) blank common is allocated at runtime directly below the BDOS (at the top of the TPA) in CP/M or at a user specified address. To override the default placing of the blank common block in memory, use the M= compiler option when the program is compiled. If the size of blank common blocks is the same, then blank common can be used to pass data between routines that CHAIN as the blank common variables will occupy the same place in memory.

NOTE: The name of a named COMMON block must not be the same as a SUBROUTINE or FUNCTION name.
4.8. Implicit Statement

The IMPLICIT statement is used to change the default INTEGER, REAL, DOUBLE PRECISION and LOGICAL typing.

The general format of the IMPLICIT statement is:

```
IMPLICIT type (range), type(range)
```

where:

Type is one of INTEGER, REAL, LOGICAL, DOUBLE PRECISION. Range is either a single letter or a range of letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a hyphen or a sequence of single letters separated by commas.

Example

```
IMPLICIT INTEGER (Z), REAL (A,B,C,D,E,G), INTEGER (M-S)
IMPLICIT REAL (I,J)
IMPLICIT REAL (A-Z)
```

An IMPLICIT statement specifies the type of all variables, arrays and functions that begin with any letter that appears in the specification. Type specification by an IMPLICIT statement may be overridden for any particular variable, array or function name by the appearance of that name in a type statement.

The IMPLICIT statement must appear before all other statements in a particular routine: that is immediately after the SUBROUTINE or FUNCTION statement or before the first statement of the main program.

WARNING: Even though the DOUBLE PRECISION specification is supported, double precision arithmetic is NOT. All DOUBLE PRECISION variables will be treated as if they where REAL. A warning will be issued each time a DOUBLE PRECISION statement is encountered.
5. Expressions

An expression is a combination of variables, functions and constants, joined together with one or more operators.

### Arithmetic Operators

- ** or ^: Exponentiation
- *: Multiplication
- /: Division
- +: Addition
- -: Subtraction

### Comparison Operators

- .EQ.: Equal to
- .NE.: Not equal to
- .GT.: Greater than
- .LT.: Less than
- .GE.: Greater than or equal
- .LE.: Less than or equal to

### Logical Operators

- .NOT.: Logical Negation
- .AND.: Logical and
- .OR.: Logical or
- .XOR.: Logical exclusive or

The .NOT. and unary minus (-) operators preceded an operand. All other operators appear between two operands.

#### 5.1. Hierarchy of Operators

The following is the table of operator hierarchy and the correct FORTRAN symbolic representation to be used in coding:

<table>
<thead>
<tr>
<th>Highest</th>
<th>System and User Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>** OR ^ (up arrow)</td>
<td></td>
</tr>
<tr>
<td>* and /</td>
<td></td>
</tr>
<tr>
<td>+ and - (including unary -)</td>
<td></td>
</tr>
<tr>
<td>.LT., .LE., .NE., .EQ., .GE., .GT.</td>
<td></td>
</tr>
<tr>
<td>.NOT.</td>
<td></td>
</tr>
<tr>
<td>.AND.</td>
<td></td>
</tr>
<tr>
<td>.OR. and .XOR.</td>
<td></td>
</tr>
<tr>
<td>Replacement (=)</td>
<td></td>
</tr>
</tbody>
</table>

| Lowest | |
|--------| |
5.2. Expression Evaluation

FORTRAN expressions are evaluated as follows:

1. Parenthesised expressions are always evaluated first, with the inner most set being evaluated first.

2. Within parentheses (or whenever there are none) the order of expression evaluation is:
   a. FUNCTION references
   b. Exponentiation
   c. Multiplication and division
   d. Addition and subtraction

3. Operators of the same precedence are evaluated from left to right during expression evaluation.

Example

\[ A + 1 + Z \times 5 \]

will be evaluated as:

\[ ((A + (1 + (Z \times 5))) \]

\[ VAL \times Z + (T + 4) / 6 \times X^{\text{**}Y} \]

will be evaluated as:

\[ (((VAL \times Z) + ((T + 4) / 6) \times (X^{\text{**}Y}))) \]

NOTE: Operators of equal precedence are executed from left to right.
5.3. Integer Operations

A fundamental difference between INTEGER and REAL arithmetic operation, is the manner in which rounding occurs. If you were to divide 3.0 by 2.0 using floating point arithmetic, the answer would be 1.5. However, if the same operation were to be performed using integer arithmetic, 3/2 would equal 1.

Note in using integer arithmetic, the fractional part of the number is truncated. Another example is in the multiplication of two real numbers. 2.9 times 4.8 would equal 13.92. However in integer mode, the result is be 13. Also, no more than 8 digits of accuracy are maintained. Should more than 8 digits be generated by an integer operation, a runtime error of INT RANG will result.

Example

\[
\begin{align*}
6/3 &= 2 \\
\text{but } 7/3 &= 2 \text{ (NOTE: no fraction is retained) and } 7/9 &= 0 \\
99999999+5 &= \text{? integer overflow}
\end{align*}
\]
5.4. Real Operations

Unlike integers, Real operations and their results have a precision of eight significant digits plus an exponent (base 10) between -127 and +127.

Example

\[
\begin{align*}
12/6.0 &= 2.0 \\
15.0/2 &= 7.5 \\
1./2. &= 0.5
\end{align*}
\]
5.5. Logical Operations

Logical operations are unlike INTEGER and REAL operations in that they always return a value of zero (0) or one (1). All the logical operations will return a one for a TRUE condition, however any NON-ZERO value will be interpreted as TRUE. If the logical operation is a logically true statement, the result is a one, if the statement is false, a zero is returned.

Example

\[
\begin{align*}
A &= 1 \text{ .GT. } 2 \quad \text{(false)} \quad \text{A would evaluate to 0} \\
A &= 1 \text{ .EQ. } 1 \quad \text{(true)} \quad \text{A would evaluate to 1} \\
A &= 1 \text{ .LT. } 2 \quad \text{(true)} \quad \text{A would evaluate to 1}
\end{align*}
\]

The relational operator abbreviations in the previous table represent the following operations:

- \text{.LT.}  \quad \text{Less Than}
- \text{.LE.}  \quad \text{Less Than or Equal}
- \text{.NE.}  \quad \text{Not Equal}
- \text{.EQ.}  \quad \text{Equal}
- \text{.GE.}  \quad \text{Greater Than or Equal}
- \text{.GT.}  \quad \text{Greater Than}

- \text{.AND.} \quad \text{True only if both operands are true.}
- \text{.OR.} \quad \text{True if either operand is true.}
- \text{.XOR.} \quad \text{True if operands are different.}

Example

\[
\begin{align*}
\text{IF (A .EQ. B) GO TO 500} \\
\text{IF (A .EQ. B .OR. K .EQ. D) STOP}
\end{align*}
\]

Logical variables can also be used in assignment statements:

\[
\begin{align*}
A &= \text{A .AND. B} \\
I &= (A \text{ .OR. B}) \text{.XOR.} ((T \text{ .EQ. 35.4}) \text{.OR.} (T \text{ .EQ. 39}))
\end{align*}
\]
The following logical operators are also available, as listed in the following truth charts.

<table>
<thead>
<tr>
<th>.AND.</th>
<th>.OR.</th>
<th>.XOR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B  R</td>
<td>A  B  R</td>
<td>A  B  R</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0  0  0</td>
<td>0  0  0</td>
</tr>
<tr>
<td>0  1  0</td>
<td>0  1  1</td>
<td>0  1  1</td>
</tr>
<tr>
<td>1  1  0</td>
<td>1  1  1</td>
<td>1  1  1</td>
</tr>
<tr>
<td>1  1  1</td>
<td>1  1  1</td>
<td>1  1  1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.NOT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  R</td>
</tr>
<tr>
<td>0  1</td>
</tr>
<tr>
<td>1  0</td>
</tr>
</tbody>
</table>

06-21-83 NEVADA FORTRAN PAGE 38
5.6. Mixed Expressions

The standard FORTRAN rules for mixed mode expressions are:

- integer <op> integer gives an integer result
- real <op> integer gives a real result (with the integer being converted to real before the operation is performed).
- real <op> real gives a real result
- integer <op> real gives a real result, with the integer being converted to REAL before the operation is performed.
- integer = real will cause truncation of any fractional part of real and an error if the truncated result is outside the range of integers.
- real = integer will cause integer to be converted to real.

In general, in a mixed expression, integers are converted to real before the operation take place, giving a real result (unless both operands are integer).

<op> represents one of the operators: + - / *
6. Control Statements

There are several different statements that control the execution flow of a FORTRAN program:

These are:

1. GO TO statements
   a. Unconditional GO TO
   b. Computed GO TO
   c. Assigned GO TO

2. IF statements
   a. Arithmetic IF
   b. Logical IF
   c. IF-THEN-ELSE

3. DO

4. CONTINUE

5. PAUSE

6. STOP

7. CALL

8. RETURN
   a. explicit RETURN
   b. multiple RETURN
6.1. Unconditional GO TO Statement

The general format of the unconditional GO TO is:

\[
\text{GO TO n}
\]

where \( n \) is a label on an executable statement.

The unconditional GO TO Statement performs a transfer of control to the statement number specified as the object of the branch. If the statement number does not exist, an undefined label error will occur; this error is detected during compilation.

Note: Labels on FORMAT statements in most FORTRAN systems may not receive transfer of control. This is not true in this implementation of FORTRAN. FORMAT statements act the same as a CONTINUE statement which will be discussed later.

Example

\[
\begin{align*}
\text{GO TO 10} \\
\text{GO TO 400} \\
10 & \text{ CONTINUE} \\
400 & \text{ FORMAT (1X)}
\end{align*}
\]
6.2. Computed GO TO Statement

The general format of the COMPUTED GO TO is:

```
GO TO (n1,n2,...nm),i
```

The computed GO TO statement works in a manner similar to the GO TO statement. However, one of the distinct advantages is that under program control, you may direct which statement is the next to be executed, based on the value of i. The computed GO TO works as follows:

<table>
<thead>
<tr>
<th>Computed GO TO Statement</th>
<th>Present Value of Variable</th>
<th>Next Executed Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO TO (1,5,98,167,4),K2</td>
<td>K2=5</td>
<td>4</td>
</tr>
<tr>
<td>GO TO (44,28),J</td>
<td>J=1</td>
<td>44</td>
</tr>
<tr>
<td>GO TO (51,6,7,1,46),M</td>
<td>M=4</td>
<td>1</td>
</tr>
<tr>
<td>GO TO (1,1,1,1,2,2),LOOT</td>
<td>LOOT=3</td>
<td>1</td>
</tr>
</tbody>
</table>

If the value of i exceeds the number of statement labels in the computed GOTO, a runtime error COM GOTO will be generated. If the value of i is less than 1, a runtime error will also be generated.
6.3. Assigned GO TO

The general format of the assigned GO TO is:

\[ \text{GO TO } v,(n_1,n_2,...) \]

where \( v \) is the variable used in an ASSIGN statement and \( n_1, n_2 \) are statement labels.

Example

\[ \text{GO TO LABL,}(100,400,500) \]
\[ \text{GO TO K,}(1,2,3,4,5) \]
6.4. ASSIGN

The general format of the ASSIGN statement is:

`ASSIGN n TO V`

where n is the statement label to be `ASSIGNed` to v. The ASSIGN statement assigns a statement label to be used in conjunction with the `ASSIGNED GO TO` statement.

Example

```
ASSIGN 20 TO LABEL

IF (KNT .GT. 10) ASSIGN 10 TO LABEL

GO TO LABEL,(10,20)
```
6.5. Arithmetic IF Statement

The Arithmetic IF allows the programmer to evaluate an expression which may be any combination of integer, real, or logical operators, and based upon its relationship to zero, transfers control to one of three specified statements.

The general form of the arithmetic IF is:

\[ \text{IF } (e) \text{ n1,n2,n3} \]

where \( e \) is an arithmetic expression which when evaluated is used to determine the next statement to be executed.

If \( e \) is:

<table>
<thead>
<tr>
<th>next statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;0 )</td>
</tr>
<tr>
<td>( =0 )</td>
</tr>
<tr>
<td>( &gt;0 )</td>
</tr>
</tbody>
</table>

Example

\[ \begin{align*}
\text{IF } (A) & 1,2,3 \\
\text{IF } (BETA*SIN(BETA/\text{DEGREE})) & 100,150,432 \\
\text{IF } (A-1) & 1,1,99 \\
\text{IF } (.\text{NOT. } \text{FLAG}) & 1,5,7
\end{align*} \]
6.6. Logical IF Statement

The general format of the Logical IF is:

\[
\text{IF (e) s}
\]

The logical IF statement operates as follows:

1. The expression e is evaluated, and a logical result is derived, .TRUE. or .FALSE. (numerically 1 or 0, respectively).

2. Depending on the value which is derived, one of the following two conditions occurs:
   - If e is evaluated as .TRUE. then the statement s is executed, and once the IF has completed, transfer is then passed to the next consecutive statement.
   - If e is evaluated as .FALSE. the statement s is NOT executed and control is then passed to the next sequential executable statement.

The statement s can be any statement other than an END, another Logical IF or a DO.

Example

\[
\begin{align*}
\text{IF (DEGREE .EQ. 100) WRITE (1,*)) RADIAN} \\
\text{IF ((A .EQ. 12).OR.(LOOP .LE. 500)) RETURN} \\
\text{IF (SIN(30)/WHERE-.00005 .LT. .00004) STOP} \\
\text{IF (A .NE. B) GO TO 500} \\
\text{IF (A .EQ. 1) GO TO (1,2,3), J} \\
\text{IF (VALUE .EQ. 6) IF (J)99,33,67} \\
\text{IF (I .GE. 500) J=I+20/8} \\
\text{IF (FLAG) A=2*A+5}
\end{align*}
\]
6.7. IF-THEN-ELSE

The general format of the IF-THEN-ELSE statement is:

```plaintext
IF (e) THEN
    statement 1
    statement 2
    ...
ELSIF
    statement 3
    statement 4
    ...
ENDIF
```

The IF-THEN-ELSE is an extension of the logical IF with two additions:

1) there can be more than 1 statement to execute if the IF is true
2) there is the provision of specifying one or more statements to be executed if the IF is false.

The ENDIF is required to indicate the end of the complete IF-THEN-ELSE statement.

To indicate an IF-THEN-ELSE the s part of the logical IF is replaced with the THEN statement. All statements between the THEN and the matching ELSE or ENDIF will be executed if the specified condition is true. All statements between the ELSE and ENDIF will be executed if the specified condition is false. The ELSE is optional and if the condition is false, all statements between the THEN and ENDIF will be skipped.

If no ELSE condition is to be specified, then the THEN can be terminated with an ENDIF. For example:

```plaintext
If (e) THEN
    statement 1
    statement 2
    ENDIF
```

The statements to be executed can be any statement including another IF-THEN-ELSE.
Note: THEN, ELSE and ENDIF are individual statements terminated by either a carriage return or semicolon.

Example

\[
\begin{align*}
\text{IF (I .EQ. 0) THEN} \\
& L=K+1 \\
& K=I \\
& \text{ELSE} \\
& K=0 \\
& \text{ENDIF}
\end{align*}
\]

\[
\begin{align*}
\text{IF (J .LT. 7) THEN} \\
& LL=L+1 \\
& \text{ELSE} \\
& \text{IF (A .EQ. B)THEN} \\
& Q=0 \\
& D=N \\
& \text{ELSE} \\
& \text{TYPE 'ERROR'} \\
& \text{STOP} \\
& \text{ENDIF} \\
& \text{ENDIF}
\end{align*}
\]
6.8. DO-LOOPS

The general format for a DO loop is:

```
DO n i=m1,m2,m3
```

The DO loop is the basic loop structure in FORTRAN. It works in a manner similar to the FOR-NEXT loop in BASIC. The DO Loop works as follows:

1) i is set to the value of ml.

2) After each pass through the loop (which ends with the statement labelled n), the step value, m3 is added to i. If the m3 term (step value), is omitted, then the step value is assumed to be one. Unlike other versions of FORTRAN the i and m terms do not have to be INTEGER values and the step may be negative. This allows fractional increments of the DO loop index, i. The ability of a negative increment, m3, allows loop to step in a downward direction. If the step value is positive, the loop continues until the value of i is greater than that of m2. If the step value is negative, the loop continues until the value of i is less than that of ml. n in the DO loop specifies the range of the DO loop. This is the statement number of the last statement of the DO loop.

Irrespective of the relation of the initial and ending values, the DO will always be executed once.

Note that 2 or more DO loops may end on the same statement.

DO loops may not terminate on GO TO, STOP, IF-THEN-ELSE, END or RETURN statements. A common way to terminate a DO is with a CONTINUE statement.

Example

```
DO 800 I=1,100
DO 1 J=I,END,.005
DO 99 A=START,END,AINCR

DO 10 I=1,5
DO 20 J=3,99
.
.
20 CONTINUE
.
10 CONTINUE
```
6.9. CONTINUE Statement

The format of the CONTINUE statement is:

```
CONTINUE
```

The CONTINUE statement is an executable FORTRAN statement. It generates no code and is generally used as the terminal statement of a DO loop.

Example

```
DO 100 I=1,50
  :
  :
100 CONTINUE
```

The CONTINUE statement simply serves to mark the range of the DO. It is also used for transfer of control i.e. you can GO TO it.
6.10. ERROR TRAPPING

Normally when an error occurs during the execution of a FORTRAN program a runtime error message will be generated. However using the ERRSET and ERRCLR statements it is possible to control and trap runtime errors.

The general format of these statements is:

```
ERRSET n,v
ERRCLR
```

where n is the label of the statement to go to, if a runtime error occurs. And v is the variable to contain the error code of the runtime error that occurred.

The ERRSET statement causes control to be transferred to the statement labeled n when a runtime error occurs. No runtime error message will be printed if the error is trapped with an ERRSET statement. The ERRSET statement can only be used if the Q option was specified on the OPTIONS statement for the routine in which the error occurred. If an ERRSET or ERRCLR statement is encountered and the Q option was not specified, a compilation error will be generated.

The value placed in the variable v corresponds to the runtime error that occurred as follows:

```
1  Integer overflow
2  Convert error
3  Argument count error
4  Computed GOTO index out of range
5  Overflow
6  Division by zero
7  Square root of negative number
8  Log of negative number
9  Call stack push error
10 Call stack pop error
11 CHAIN/LOAD error
12 Illegal FORTRAN logical unit number
13 Unit already open
14 Disk full
15 Unit not open
16 Binary I/O to system console
17 Line too long on READ or WRITE
18 FORMAT error
19 Input/Output error in READ or WRITE
20 Invalid character on input
21 Invalid input/output list (impossible)
22 Assigned GOTO error
23 CONTROL/C abort
24 Illegal character in input
```
If more than one ERRSET statement is executed in a routine, then the last one executed is the one in effect. If a runtime error should be trapped with an ERRSET statement, the ERRSET statement is automatically cleared after control has transferred to the statement n.

The ERRCLR statement clears the effect of the ERRSET statement that was last executed in the routine in which the ERRCLR is executed.

Example

OPTIONS Q

   ERRSET 10, CODE
   
   ERRCLR
   
   STOP

10 TYPE 'ERROR ,ERROR CODE = ', CODE

END
6.11. CONTROL/C CONTROL

At the beginning of each READ or WRITE statement the state of the CONTROL/C abort flag is tested. If the CONTROL/C abort flag is set, then the console is tested to see if CONTROL/C has been hit. If CONTROL/C has been hit, then one of two actions will occur:

1) if there is an ERRSET in effect, the error branch will be taken with a CONTROL/C error.
2) otherwise a runtime error of CONTROL/C will be generated.

The user has control of the CONTROL/C flag through the CTRL ENABLE and CTRL DISABLE statements. CTRL ENABLE sets the CONTROL/C flag and allows a CONTROL/C from the console to abort the program. CTRL DISABLE resets the flag and causes the CONTROL/C to be ignored.

Example

```
DO 1 I=1,100
   IF (I .EQ. 51)CTRL DISABLE
   1 TYPE I
END
```

The above program will only abort if CONTROL/C is entered while the first 50 numbers are being typed.

When program execution starts, the CONTROL/C flag will be set which allows CONTROL/C to abort the program.

Note: The CONTROL/C error, if enabled, can be trapped with an ERRSET statement. However, the CIN function will return a control-C to the caller, regardless of the setting of the CONTROL/C flag.
6.12. TRACING

There are two statements that are used to trace a program:

```
TRACE ON
TRACE OFF
```

When program execution begins, tracing is initially off and must be explicitly turned on. Once tracing is on, it remains on until the program terminates or a TRACE OFF statement is executed. The effect of the both trace statements is global over the whole program and tracing does not have to be turned on in each subroutine. The trace function will output the line number of the FORTRAN statement before execution only if the X option was specified on the options statement for this routine. Otherwise the program will be traced only up to the entrance to the subroutine. It should be noted that the line number for any entrance to a subroutine (either SUBROUTINE or FUNCTION) will always be output as 1111 regardless of the state of the X option.

Example

```fortran
IF (FLAG .EQ. 0) TRACE ON
TRACE OFF
DO 1 I=1,100
   IF (I .EQ. 50) TRACE ON
1   TYPE I
```
6.13. DUMP statement

The general format of the DUMP statement is:

```
DUMP /ident/ output list
```

where `ident` is up to a 10 character identifier for this DUMP statement and `output list` is a standard WRITE output list that may contain variables, constants, character strings, array elements, array names and implied DO loops.

The DUMP statement is used to display information when a runtime error occurs that is not trapped by an ERRSET statement.

More than one DUMP statement may be executed in a routine and the last one executed is the one that will be output on a runtime error. Each subprogram may contain its own DUMP statement, but only the last DUMP statement executed in a particular routine will be saved and output if a runtime error occurs.

Example

```
DUMP /AFTER-DIV/ 'Index after divide is ',K
  .
  .
  .
A=K/I
  .
  .
END
```

will cause the dump statement to output if I is zero.
7. Program Termination Statements

7.1. PAUSE Statement

The general format of the PAUSE statement is:

```
PAUSE 'any char string'
```

This statement causes the program to wait for any input from the system console. To continue execution, press any key on the system keyboard. If the character string option is specified, the string will be displayed on the system console. The string is enclosed in single quotes ('). To output a quote, two quotes in a row must be entered; e.g. ("') outputs as ('). The quotes surrounding the text are not displayed.

Example

```
PAUSE
PAUSE 'DATA OUT OF SEQUENCE, IGNORED'
PAUSE 'THIS IS A SINGLE QUOTE (" )'

ISUM=0
DO 10 I=1,10
   ISUM=ISUM+I
   IF (ISUM .EQ. 5) PAUSE 'SUM = 5'
10 CONTINUE
STOP
END
```
7.2. STOP Statement

The general format of the STOP statement is:

```
STOP 'any char string'
STOP n
```

When a STOP statement is executed, termination of the executing program will occur. If the character string is specified it will be printed on the system console when the statement is executed. After the character string is output, the program terminates and returns to CP/M. The string is enclosed in single quotes. To output a quote, two quotes in a row must be entered. The quotes surrounding the text will not be output.

In the second form, n is a 1 to 5 digit integer number that will display. n is optional.

To terminate a program without the STOP being typed on the console, use the EXIT subroutine.

Example

```
STOP 'PROGRAM COMPLETE'
STOP 1267
STOP
STOP 'ERROR OCCURRED, CHECK OUTPUT'

IF (ERROR .NE. 0)STOP 'ERROR'
IF (FLAG .AND. STOPIT)STOP 'ALL DONE'
```
7.3. END Statement

The format of the END statement is:

```
END
```

This is a required statement for every FORTRAN routine. It is used by the compiler to indicate the end of one logical routine. If an END statement is executed, then the message STOP END IN - XXXXX will be output to the system console, with XXXXX being replaced by the name of the FORTRAN routine in which the END statement was executed and the program will terminate.
8. Array Specification

An array is a collection of values that are referenced by the same name and the particular element is specified by a subscript. Subscripts can be real or integer expressions or constants and will be truncated to an integer value after the expression is evaluated.

Every array that is to be used must be dimensioned.

An array may have from one to seven dimensions.

Example

If GRADE has 3 elements then:

- GRADE(1) refers to the first element
- GRADE(2) refers to the second element
- GRADE(3) refers to the third element

NOTE: Subscripted variables cannot be used as subscripts, thus GRADE(A(I)) is invalid, where both GRADE and A are arrays.
8.1. Dimension Statement

The general format for a DIMENSION statement is:

```
DIMENSION v(n1,n2,..,nm),..
```

Where v is the array name and N1, N2,.. are the size of each of the dimensions of the array v.

The DIMENSION statement is used to define an array. The rules for using the DIMENSION statement are as follows:

1) Every subscripted variable must appear in a DIMENSION statement whether explicit (in a dimension statement) or implied (in a REAL, DOUBLE PRECISION, INTEGER, LOGICAL or COMMON statement) prior to the use of the first executable statement.

2) A DIMENSION specification must contain the maximum dimensions for the array being defined.

3) The dimensions specified in the statement must be numeric in the main routine. However, in subprograms the subscripts may be integer variables. Hence the following statement is valid only in a SUBROUTINE or FUNCTION:

```
DIMENSION A(I,J)
```

In the case where the dimensions of an array are specified as variables, the value of the variable at runtime will be used in computing the position within the array to be accessed.

4) All arrays passed to subprograms must be DIMENSIONED in the subroutine as well as in the main program. If the arguments in the subroutine differ from those in the main program, then only those sections of the array specified by the DIMENSION statement in the subroutine will be accessible in the subroutine.

5) The number of dimensions specified for a particular array cannot exceed 7.

6) No single array can exceed 32,767 bytes in size (5461 elements).
Note: The following is a method that can be used to get around the size limit of arrays. Allocate the large array in a named common block as 2 or more sequential arrays. Use just the first array and subscript out of it as necessary. The way that common blocks are allocated will assure you that the arrays are allocated sequentially in memory. For example, if you want an array of 7000 elements, it can be defined as:

```
COMMON /DUMMY/ TABLE(4000),TABLE1(3000)
```

Then you would just use the array TABLE. To access the 4945th element, just use TABLE(4945) (which is actually the 945th element of TABLE1).

Example

```
DIMENSION GUN(S,E) (this statement is valid only in a subprogram as it uses variable dimensions).

DIMENSION A(2,2),B(10)
DIMENSION ZIT(10)
REAL APPLE(10)
LOGICAL FUNCT(100)
DOUBLE PRECISION ARR(10),B(8),A(SIZE)

DIMENSION A(3,2,3),C(10),ZOT(10,10)
INTEGER SWITCH(15)

"A" would require 3*2*3*(6) = 108 bytes
"C" would require 10*(6) = 60 bytes
"ZOT" would require 10*10*(6) = 600 bytes
"SWITCH" would require 15*(6) = 90 bytes
```

In calculating the memory used by an array, multiply each of the dimensions times each other, then times 6. The result will be the number of bytes used by the array for storage.

WARNING: No subscript range checking is performed at runtime.
8.2. Subscripts

Subscripts are used to specify an entry into an array (i.e. the value specified in the subscript is the element of the array referenced). Subscripts may be integers, real (fractions are truncated), logical expressions or any other valid expression. Expressions are evaluated as explained in the EXPRESSION section (5.2.0).

Example

ZIT(8)
A(1+2)
ORANGES(I+5-(K*10)/2)
APPLE(5)
9. Subprograms

Subprograms provide a means to define often needed sections of code that can be considered as a unit. FORTRAN provides the means to execute these subprograms whenever they are referenced.

There are 3 types of subprograms supported in this version of FORTRAN:

1) SUBROUTINE subprograms
2) FUNCTION subprograms
3) Built in library functions

The major differences between FUNCTIONS and SUBROUTINES are listed below.

1) FUNCTIONS are used in expressions, while SUBROUTINES must be CALLED.

2) FUNCTIONS require at least one parameter; SUBROUTINES do not require any.

3) The name on the FUNCTION statement must be the object of a replacement statement somewhere in the FUNCTION; this is not the case for a SUBROUTINE.

WARNING: If a constant is passed as an argument in either a CALL or FUNCTION reference, and the corresponding parameter in the SUBROUTINE or FUNCTION is modified, then the value of the constant that was passed will be changed, and remain that of the new value.

NOTE: All SUBROUTINES and FUNCTIONS must be compiled at the same time.
9.1. SUBROUTINE Statement

The general format of the SUBROUTINE statement is:

```
SUBROUTINE name(list)
```

The SUBROUTINE statement is used to identify the beginning of a logical routine. This statement is required at the beginning of every SUBROUTINE. The list that is to receive the values being passed to the subroutine is optional if no parameters are to be passed.

Example

```
SUBROUTINE ADDIT (RESULT,X,Y)
RESULT=X+Y
RETURN
END
```
9.2. FUNCTION Statement

The general format of the FUNCTION statement is:

```
FUNCTION name(list)
```

A FUNCTION Statement is used to define a logical routine as a FUNCTION. The type of result of a FUNCTION can be specified by preceding the FUNCTION with REAL, DOUBLE PRECISION, INTEGER, or LOGICAL; or the name of the FUNCTION may appear in a type statement within the FUNCTION.

Example

```
FUNCTION SWAP(A)
SWAP=A
RETURN
END

INTEGER FUNCTION SWAP (A)
SWAP=IFIX(A)
RETURN
END

FUNCTION SWAP (A)
INTEGER SWAP
SWAP=A/2
RETURN
END

DOUBLE PRECISION VALUE(WHAT)
VALUE=WHAT/4*7+5
RETURN
END
```
9.3. CALL Statement

The general format of the CALL statement is:

CALL name(list)

The CALL statement is used to transfer control to a SUBROUTINE. List specifies the parameters to be passed to the SUBROUTINE and may be omitted if no parameters are to be passed.

The number of parameters in a CALL and SUBROUTINE statement referring to the same subprogram must be the same, otherwise a runtime error will result.

Example

CALL XSWAP (NUM1, NUM2, TOTAL)
CALL XSWAP
9.4. RETURN Statement

There are 2 types of RETURN statements:

1) normal RETURN

2) multiple RETURN

9.4.1 Normal return

RETURN

The RETURN Statement is used to terminate execution of a subprogram whether it is a FUNCTION or a SUBROUTINE. Return is transferred to the next statement following the CALL statement, or in the case of a FUNCTION, return is transferred back to the point where it was called with the value of the FUNCTION returned. A RETURN statement is not valid in the MAIN routine and will cause an error during compilation if encountered in the MAIN routine.

Example

SUBROUTINE ZERO(I,J)
I=0
J=0
RETURN
END

FUNCTION ZZ(VAL)
COMMON /A/ A,B,C
ZZ=A+B/C-VAL
RETURN
END
9.5. Multiple return

The general format of the multiple RETURN statement is:

```
RETURN I
```

This variation of the RETURN statement is used to transfer back from a SUBROUTINE to a point other than the statement that immediately follows the CALL. The I in the RETURN is the name of a variable in the argument list of the subroutine and must have been passed as a label in the CALL. The CALL statement that invokes a routine that contains a multiple return must pass the label as one of the parameters. The statement label is indicated in the argument list by preceding the label with an ampersand (&).

Example

```
CALL X(&1,Y,2,&2)
  .
  SUBROUTINE X(I,A,IC,J)
  .
C THE FOLLOWING RETURN WILL TRANSFER TO THE STATEMENT
C LABELLED '1' IN THE CALLING PROGRAM.
C
  RETURN I
  .
C THE FOLLOWING RETURN WILL TRANSFER TO THE STATEMENT
C LABELLED '2' IN THE CALLING PROGRAM.
C
  RETURN J
  END
```

NOTE: Multiple RETURNS are only valid for SUBROUTINES.
9.6. BLOCK DATA SUBPROGRAM

The BLOCK DATA subprogram is used to initialize variables in named COMMON. The BLOCK DATA subprogram must contain no executable statements. It may contain only declaration statements for specifying variable types, array dimensions, COMMON blocks and DATA statements.

Example

```
BLOCK DATA
INTEGER FIRST, LAST
COMMON /ONE/ NAMES(100) /TWO/ FIRST, LAST
DATA FIRST /1/, LAST/10/
DATA NAMES /1, 2, 0, 4, 5, 6, 7, 8, 9, 10, 90*99999/
END
```

NOTE: The variable in named COMMON can be initialized in any routine. The BLOCK DATA subprogram appears only for compatibility with other FORTRAN systems.
10. Input/Output

10.1. Introduction to FORTRAN I/O

10.1.1. General Information

Input and Output (I/O) under FORTRAN may take one of the following forms:

1) Standard Formatted I/O
2) Free Format I/O
3) Binary I/O

In formatted I/O, input and output is defined in terms of fields which are right justified on the decimal point, with zero suppression. In a FORMAT statement, no more than three levels of nested parentheses are allowed (outer set and two nested inner sets).

Free Format I/O is used as in BASIC. All the values are entered using commas (,) or carriage returns to delimit the numbers.

Binary I/O is a third option that allows passing of large files between FORTRAN programs, with the minimal amount of wasted disk space. Each variable written in binary format uses six bytes of disk space.

FORTRAN logical units 0 and 1 are dedicated to console input and output and cannot be either opened or closed. An attempt to open or close 0 or 1 will result in a runtime error. Logical unit 0 is used for console input and logical unit 1 is used for console output. Binary I/O cannot be specified for logical units 0 or 1 and doing so will result in a runtime error.

There are two special I/O statements:

```
TYPE
ACCEPT
```

Both of these are followed by a standard I/O list. TYPE is equivalent to WRITE (1,*) and ACCEPT to READ (0,*). This is just a convenient method of doing console I/O.

Example

```
TYPE I,J,(A(I),I=1,10)
ACCEPT 'INPUT THE MAX COUNT',COUNT
```
A RUNTIME FORMAT can be specified for any formatted I/O statement by substituting an ARRAY name for the FORMAT number. At runtime, the array is assumed to contain a valid FORTRAN FORMAT (complete with its outer set of parentheses). This allows a FORMAT statement to be input at runtime and then to be used in either READ or WRITE statements within the program. Thus a particular FORMAT can be changed at runtime instead of having to recompile the program. The FORMAT should be input using an A6 format specification as imbedded blanks (added if using less than an A6) will cause a runtime error.

Example

DIMENSION FORM(10)
READ (0,10) 'ENTER DATA FORMAT ',FORM
10 FORMAT (10A6)
.
.
READ (4,FORM) A,B,C
.
.
WRITE (5,FORM) R1,R2,R3
10.1.2. I/O List Specification

The I/O List is used to specify which variables are to be READ or WRITTEN in a particular I/O statement. The list has the same form for both READ and WRITE statements. The list can be composed of one or more of the following:

1) simple (non-subscripted) variable
2) array element
3) array name
4) implied DO loop
5) literal
6) constant (WRITE only)

The above types are combined to form the I/O list specification. Items 1-4 are self explanatory, however item 4, the implied DO loop is explained further below:

The implied DO loop is used mainly to output sections of one or more arrays and functions in the same way as does a regular DO loop. An example of an implied DO loop is:

```
WRITE (1,*) (F(I),I=1,3,1)
```

It should be noted that the outer parentheses and the comma preceding the DO index are always necessary when using an implied DO loop. Nested loops can be used. Each loop must be enclosed in parentheses. An example follows:

```
WRITE (1,*) (J,(F(I,J),I=1,4),J=1,30,2)
```

The inner DO (I) is performed for each iteration of the outer DO (J). Note that other than array elements can be included within the range of an implied DO. Implied DO's can be nested to any depth, each within its own set of parentheses.
LITERALS (character strings enclosed in quotes) can be used in any WRITE statement and in READ statements that reference the system console. The literals can be used as prompts for input or identification on output.

Example

```
WRITE (1,*) 'A= ',A
TYPE 'The answer is ',ANS
WRITE (5,3) 'X= ',X

READ (0,*), 'A= ',A, ' B= ',B
ACCEPT 'Enter quantity ',QUANT
```

NOTE: An attempt to use a literal in a READ statement that doesn't reference the console will result in an INPUT ERR runtime error.
10.2. READ Statement

READ(unit[,format]{,END=end}{,ERR=error}) I/O list

The READ statement is required in order for the user to do input through the FORTRAN system. If a unit number of 0 is used there is no need to open this file as it is assumed to be system console input. Note: do not use 1 as the logical unit as it is reserved for the system console output. Any other unit number must first have been opened by the user through the OPEN or LOPEN subroutine. The FORMAT entry may take one of the following forms:

1) The FORMAT number is the label on the FORMAT statement which is to be used.

2) An asterisk (*) in the FORMAT entry indicates that input is to be free format. The exact format of the output depends on the value of the number being output and is determined at runtime.

3) If the FORMAT entry is left blank (or not specified), binary input is assumed.

4) The name of an array that contains the FORMAT to be used.

END= is the label to which transfer of control is to be made should an end of file condition be encountered. ERR= is the label to which control will be transferred, should an error other than end of file occur during input, such as a bad sector. The ERR= does not handle input format errors (such as decimal point in an integer field). Use ERRSET to handle these input errors. I/O list is the string of variables which accept the data to be read.

Example

READ (0,2) A  
read from the system console the variable 'A' under FORMAT number 2

READ (0,*) A  
read from the system console the variable 'A' in free format.

READ (4) A  
read from logical file 4, the variable A in binary.
READ (4,,END=10) A
read from logical file 4, the variable A, in binary, and if end-of-file is encountered, go to statement label 10

READ (4,,END=10,ERR=100) A
read from logical file 4, the variable A in free format, should end-of-file be encountered, go to statement label 10. If an error occurs, go to statement label 100.

READ (4,,ERR=100) A
read from logical file 4, the variable A in binary format, and if an error occurs, go to statement label 100.

NOTE: The END= and ERR= parameters are optional and can appear in any order.
10.3. WRITE Statement

WRITE (unit[,format]{,END=end}{,ERR=error}) I/O list

The WRITE statement is used to output to either disk files or the console. It performs a function that is the opposite of the READ statement. The I/O list is specified exactly the same as for the READ statement with the exception that a string can always be used in the I/O list. However the END= serves no function and will never be used by the WRITE statement.

Example

WRITE (1,2) I,J,PAY,WITHOLD
WRITE (1) (I, I=1,10)
WRITE (10,*) THIS
WRITE (6,12,END=99,ERR=66) LOOP,COUNT
10.4. MEMORY TO MEMORY I/O statements

The ENCODE and DECODE statements allow I/O to be performed to or from a specified memory location. This allows data in memory to be read (using DECODE) with perhaps a different format code depending on the data itself. The ENCODE statement is similar to a WRITE statement in that data is formatted according to the specified format type, but instead of being output to a file it will be placed in memory at the specified location for further processing.
10.4.1. DECODE statement

The general form of the DECODE statement is:

```
DECODE (variable, length, format) I/O list
```

The DECODE statement is similar to a READ statement in that it causes data to be converted from external ASCII format to internal FORTRAN type. Variable is either an unsubscripted variable name or an array name. Length is the number of bytes to process for this READ starting at variable. If multiple records are required by the I/O list, successive records of length will be retrieved from memory. Input records will be blank padded on the right end as necessary as in a READ statement. FORMAT is either an asterisk for free formatting or the number of a FORMAT statement.

Example

```
DIMENSION A(15)
READ (1,10) A
10 FORMAT (15A6)
DECODE (A,80,11) KNT1,KNT2,CNT3
11 FORMAT (I10,I3,F10.5)
```
10.4.2. ENCODE statement

The general form of the ENCODE statement is:

```
ENCODE (variable, length, format) I/O list
```

The ENCODE statement is similar to a WRITE statement in that it is used for a memory to memory formatted WRITE. Variable is either an unsubscripted variable name or an array name. Length is the number of bytes (or characters) that the output record is to contain. If the number of characters generated by the ENCODE statement is less than length, then the record will be blank padded to length. If the number of characters in the generated record is greater than length, then the record will be truncated after the length character. If multiple output records are generated, successive records of length character will be placed in memory starting at variable. FORMAT is either an asterisk for free formatting or the number of a FORMAT statement.

Example

```
DIMENSION A(15)
ENCODE (A, 80, *) (I, I=1, 5)
```
10.5. Format Statement and Format Specifications

The general form of the FORMAT statement is:

\[ n \text{ FORMAT} (s1, s2, \ldots, sn) \]

The FORMAT statement is used in FORTRAN to do formatted input and output. Through the use of this statement the programmer has the ability to select the fields in which to read, or specify the columns on which to write. It is the use of this statement which gives FORTRAN its I/O power. On FORMATTED input, blanks are treated as if they were zeros except when reading in A format. A constant enclosed in backslashes (eg \A\) can be used to enter a binary constant from a string within a FORMAT statement.

If a number cannot be written in the specified field width, then the entire field will be filled with asterisks (*) to indicate the error condition. Note: some FORTRANS will print a negative number, even when there is not room enough to place the negative sign in the field, by omitting the negative sign. In this case, NEVADA FORTRAN will fill the field with asterisks. Asterisk filling of a field that is not large enough to output a number applies on all output specifications.

A ZERO will always be printed as 0.0 under a F, E or D field specification. If a field is printed as 0.000... this indicates that the digits have been truncated because the d portion of the field specification was not large enough.

All floating numbers output using the F, E or D (and G with a floating point number) specifications will be rounded to the appropriate number of digits specified by the d portion of the field specifier.
10.5.1. X-Type (wX)

The X-Type specification is used to space over any number of columns with a maximum of 255 character positions. 
\( w \) may have any value from 1 to 255.

On output the columns spaced over will be set to blanks. On input \( w \) characters of the input record will be skipped.

Example

10 FORMAT (10X, I10, 3X, I5)
99 FORMAT (IX, 'THIS IS A LITERAL', 5X, '$$$$')

10.5.2. I-Type (Iw)

The I-Type specification is used as a method of performing I/O with integer numbers. On input, the number must be right justified in the specified field with leading zeros or blanks. On output the leading zeros are replaced by blanks, and the number is right justified in the field.

Example

10 FORMAT (10I10)

10.5.3. A-Type (Aw)

The A-Type specification is used to perform the input of alphanumeric data in ASCII character form. Up to 6 ASCII characters may be stored per variable name. However, this is entirely under program control. For example the user may choose to store only one character per variable in a dimensioned array, in order to do character manipulation. Characters are stored in the variable left justified and zero filled. On output these padding zeros will be printed as blanks. It is not advisable to perform any arithmetic operations on a variable that contains character data as unpredictable results may occur. A format code of A6 is the maximum field width for both input and output.

Example

10 FORMAT (A10, I10, A6)
10.5.4. /-Type (/)

The /-Type specification is used to cause I/O to skip to the next record. During input this causes a new input record to be read, even though the previous one was not fully used. On output the slash will cause the current line to be written out to the associated file.

Example

```
54 FORMAT (I10/)
WRITE (1,100) 1,20,45
100 FORMAT (I3/2I3)
```

will generate

```
1
20 45
```

10.5.5. Z-Type

The Z-Type specification is used only for output, to indicate to the system that a carriage return/line feed is not to be written at the end of the record. The Z specification is ignored on input.

Example

```
WRITE (1,10)
10 FORMAT ('INPUT X ',Z)
READ (0,*) X
```
10.5.6. L-Type (Lw)

The L-Type specification is used with LOGICAL variables, where \( w \) is the width of the field. On output, the letter T or F is printed (for .TRUE. or .FALSE. respectively). The T or F will be right justified in the field. On input, the field is scanned from left to right until a T or F is found. The T or F can be located anywhere in the field and all characters that follow the T or F in the remainder of the field are ignored. If the first character found is not a T or F an error will be generated. If the input field is completely blank, then a .FALSE. value will be used.

**Example**

```fortran
LOGICAL WHICH
WRITE (1,11) WHICH
11 FORMAT (8L10)
```

10.5.7. T-Type (Tw)

The T-Type code can be used on both input and output. It is used to move to a explicit column within the input or output buffer. \( w \) specifies an absolute column number that the next character is to be read from (on input) or to be placed upon (on output). The first column number is 1. On input the T format code can be used to re-read a particular set of columns in different format codes in the same read statement. Tabbing beyond the end of the input record causes the input record to be blank padded. On output, the output cursor can be moved back (to the left) over text already inserted into the output buffer, thus causing text already there to be over written with new data. Tabbing beyond the maximum character inserted into the output buffer will cause blanks to be inserted into the output buffer to the indicated column. The maximum value of \( w \) is 255.

**Example**

```fortran
WRITE (1,56) I,LOT
56 FORMAT (I10,T50,I4)

J=1234
WRITE (1,34) J
34 FORMAT ('$$$$$$$$$$',T5,I4)
```

will produce:

```
$$$$$1234$
```
10.5.8. K-Type (Kw)

The K-Type format code is used to transmit data in hexadecimal format. Each byte of internal memory occupies 2 hexadecimal characters. If \( w \) is less than 12 characters (6 bytes/variable, 2 hex characters/byte), the hexadecimal characters will be either input or output starting from the low order memory address (beginning of the variable).

**Example**

```fortran
WRITE (1,99) 1
99 FORMAT (K12)
```

will output the line:

```
10000000081
```
10.5.9. F-Type (Fw.d)

The F-Type specification is one of several specifications for performing I/O with floating point numbers. The digit portion of the decimal number works the same as in the I-Type format. The fractional part of the number is always printed, including trailing zeros. During input, the decimal point is assumed to be at the indicated position, unless explicitly overridden in the input field. The number ZERO will always print as 0.0 (with the decimal point aligned where specified) regardless of the field width or decimal digits specified. Remember to consider the decimal point and negative sign of the number when specifying the width of the output field.

Example

<table>
<thead>
<tr>
<th>Output</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4.1</td>
<td>32.2</td>
</tr>
<tr>
<td>F7.5</td>
<td>0.00001</td>
</tr>
<tr>
<td>F3.0</td>
<td>7.</td>
</tr>
<tr>
<td>F7.2</td>
<td>bbb4.50</td>
</tr>
</tbody>
</table>

NOTE: b is used to indicate a blank position.

During input the F field specifier reads \( w \) characters. If there is not a decimal point in the field read, a decimal is inserted \( d \) digits from the right. A decimal point in the input field overrides the field specification.
10.5.10. E-Type (Ew.d)

The E-Type specification is another method of performing I/O with floating point (real) numbers. It is through this specification that the programmer may perform I/O using an exponential format. That is a mantissa followed by an exponent of ten. Again as with the F type, the decimal point is assumed to be at the indicated position if not overridden in the input field. The exponent part of the input number can be omitted, in which case it is treated as if it were an F type specification. The number will be printed as d digits followed by the letter E, exponent sign, and a three digits exponent. The d part cannot be zero for output.

<table>
<thead>
<tr>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9.2</td>
<td>0.00E+000</td>
</tr>
<tr>
<td>E9.2</td>
<td>0.12E+004</td>
</tr>
<tr>
<td>E10.0</td>
<td>invalid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>E10.0</td>
</tr>
<tr>
<td>E9.2</td>
</tr>
<tr>
<td>E9.2</td>
</tr>
</tbody>
</table>

10.5.11. D-Type (Dw.d)

The D-Type format is treated exactly the same way as the E format code, except on output a D is inserted into the number instead of an E. On input they are treated exactly the same.
10.5.12. G-Type (Gw.d)

The G-Type can be used on either input or output and for both integer and real values where \( w \) and \( d \) have the same meaning as in the E, D and F type formats. The G format is treated as follows:

**Output**

If the output element is of type integer, then the format code used will be \( Iw \).

If the output element is of type real, the actual format code used depends on the value of the number being output:

\( Ew.d \) will be used if the number is outside the range of \( 0.1 \leq \text{number} < 10^{**d} \), or

\[
\begin{align*}
F(w-5).d,5X & \quad \text{if} \quad 0.1 \leq \text{number} < 1 \\
F(w-5).(d-1),5X & \quad \text{if} \quad 1 \leq \text{number} < 10 \\
& \quad \vdots \\
F(w-5).1,5X & \quad \text{if} \quad 10^{**(d-2)} \leq \text{number} < 10^{**(d-1)} \\
F(w-5).0,5X & \quad \text{if} \quad 10^{**(d-1)} \leq \text{number} < 10^{**d}
\end{align*}
\]

In general in this range:

\( F(w-5).(d-(\text{exponent of number})),5X \)

**Input**

If the input element is of type integer: \( Iw \)

If the input element is of type real: \( Ew.d \)

**Example**

\[
\begin{align*}
A &= 5.67 \\
\text{WRITE} &\ (1,34) \ A \\
34 &\ \text{FORMAT (G10.5)} \\
\text{READ} &\ (0,9) \ A \\
9 &\ \text{FORMAT (G9.3)}
\end{align*}
\]
10.5.13. Repeating field specifications

A field specification can be repeated in a FORMAT statement by preceding it with the number of times that it should be repeated. Thus 4I10 is the same as I10, I10, I10, I10. The following FORMATS are equivalent:

10 FORMAT (3I4,2F10.4)
10 FORMAT (I4,I4,I4,F10.4,F10.4)

A single field specification or a group of field specifications can be enclosed in parentheses and preceded by a group count. In this case, the entire group is repeated the specified number of times. The following FORMATS are equivalent:

19 FORMAT (I4,2(I3,F4.1))
19 FORMAT (I4,I3,F4.1,I3,F4.1)

The FORMATS:

10 FORMAT (I5,2(I3,F5.1))
10 FORMAT (I5,I3,F5.1,I3,F5.1)

execute exactly the same for output, but differ for input. In a FORMAT without group counts, control goes to the beginning of the FORMAT statement for reading or writing of additional values. In a FORMAT with group counts, additional values are read according to the last complete group.

Example

READ (2,10) KNT, (Z(I), I=1,KNT)
10 FORMAT (I5/(F10.5))

The I5 specification will be used once and the array values will be read using the F10.5 specification.

Group counts can be nested to a maximum depth of two. Thus:

10 FORMAT (2(I5,3(I10))) is ok, while
10 FORMAT (2(I5,3(I10,2(I1)))) is not legal.
10.5.14. String Output

Character strings are written using a FORMATTED write. The string to be written is enclosed in SINGLE QUOTES (') and may not contain a backslash (\). To output a single quote within the string, two single quotes in a row must be entered. The string format type is only valid on output and if used with a READ will result in a runtime error being produced.

A character string can also be specified using the H (or Hollerith) field specification. This is an awkward method of specifying a character string as the number of characters in the string must be specified in front of the H. The H type should be avoided as it can lead to problems.

The hexadecimal code for any character (except 0) can be inserted in a string by enclosing it in backslashes (\). The backslash character can be changed using the CONFIG program.

Placing an ampersand (&) in front of a character in a string causes the character to be treated as a control character. To output an ampersand, two ampersands in a row must be used.

Example

```fortran
WRITE (1,46)
46 FORMAT ('THIS IS A TEXT STRING')
65 FORMAT (21HTHIS IS A TEXT STRING)
48 FORMAT ('This is an exclamation point\21\')
```

generates: This is an exclamation point!

```fortran
99 FORMAT ('This is a control L: &L')
```

generates: This is a control L: (followed by a control/L

```fortran
11 FORMAT ('This is an ampersand: &\')
```

generates: This is an ampersand: &
10.6. Free Format I/O

10.6.1. INPUT

FREE format input is similar to BASIC. Blanks in this mode of input are ignored completely. Numbers are entered in any format (F, D, I or E) and can be intermixed as desired. Numbers must be separated from each other by a comma or a carriage return. A comma may appear after the last number on an input line and is ignored if present. If the I/O list specifies more variables than there are in an input record, succeeding records will be read until the list is satisfied. Blank input records and blanks imbedded in numbers are ignored in this mode. The last number in any input record does not have to be followed by a comma.

10.6.2. OUTPUT

With FREE format output the exact output format used depends on the type of the variable or constant being output. An integer will result in an I type format being used, and a real will use a G-type. (The actual format used in this case depends on the value being output).

Example

ACCEPT I
ACCEPT 'PLEASE ENTER ID NUMBER', ID,'HOW MUCH', AMOUNT
READ (Ø,*) A, B, C

TYPE 'THE RESULTING VALUE IS', VALUE
WRITE (Ø,*) 'THE RESULTING VALUE IS', VALUE
TYPE '1+1=', 2
Free format I/O can also be used to any file, not just the console. The file must first be opened using either the OPEN or LOPEN routine. Then specifying an asterisk as the FORMAT number will perform free format I/O to the specified file.

Example

CALL OPEN (2,'INFILE')
CALL OPEN (4,'B:FILE')
CALL OPEN (3,'LST:')
READ (2,*)(A(I),I=1,10)
WRITE (3,*)(A(I),I=10,1,-1)
10.7. BINARY I/O

BINARY I/O provides a quick and efficient means of transferring information to and from a file. The variables are READ or WRITTEN in BINARY format. That is, six bytes for each item in the I/O list. WRITE causes the item in the I/O list to be written exactly as it is stored in memory without any additional conversion. READ does the opposite, reading six bytes directly into the I/O list item. No conversion or check is made on the data being read.

Example

```
WRITE (1) (I, I=1,100)
WRITE (1,,ERR=66) ARRAY

READ (1,,END=99) VALUE
READ (1) THIS,IS,IT
```

NOTE: the binary READ and WRITE transfers 6 bytes from the file specified directly to the variable in the I/O list. No check on the validity of the data is performed and the user should be sure that the variable contains valid numerical data before any arithmetic operations are done on the variable. An end-of-file is indicated by either the physical end of the file or a six byte field of all FF (hex). This is the value that ENDFILE will place at the end of a file that has had binary writes performed on it.
10.8. REWIND Statement

The general format of the REWIND statement is:

```
REWIND unit
```

The REWIND Statement is used to position the file pointer associated with unit to the beginning of the file. Essentially this statement closes and then re-opens the file at the beginning.

Example

```
REWIND 3
REWIND INFIL
REWIND OUTF
```
10.9. BACKSPACE Statement

The general format of the BACKSPACE statement is:

BACKSPACE unit[,error]

The BACKSPACE statement is currently not implemented and will produce a message to that effect if encountered at runtime.
10.10. ENDFILE Statement

The general format of the ENDFILE statement is:

```
ENDFILE unit
```

The ENDFILE statement is used to force an end-of-file on unit. Any data that existed beyond the point in the file where the ENDFILE was executed will be lost.

Note: The ENDFILE file statement will also CLOSE the specified file. Essentially the ENDFILE is equivalent to just closing the file; both do the same thing.

Example

```
ENDFILE 4
ENDFILE FILE
```
10.11. GENERAL COMMENTS ON FORTRAN I/O UNDER CP/M

The OPEN or LOPEN subroutine is used to associate a file with a FORTRAN logical unit. Eight files are available, numbered 0 through 7 with 0 being permanently open and associated with input from the CP/M console, logical file 1 also is permanently open and is associated with output to the CP/M console. Logical files 0 and 1 cannot be opened or closed. Additionally any logical unit associated with the CP/M console (through the use of the filename CON:) cannot have binary I/O done to it, cannot be rewound (using REWIND), endfiled (using ENDFILE) or seeked (using the SEEK routine).

A file that is going to be written on should be deleted, using the DELETE subroutine, before the file is opened. The OPEN routine does not delete a file as it does not know what type of I/O will be performed on it.

The CLOSE routine will not place any end-of-file indicator in a file that was written to; the ENDFILE statement must be used to write an end-of-file indicator to a file. The ENDFILE statement will write the normal CP/M end-of-file indicator (control-Z) if the file specified in the ENDFILE has been written to and no binary I/O was done to the file. If binary I/O has been done to the file, then an end-of-file of 6 bytes of FF (hex) will be written instead. If a file is written and then read without being ENDFILEd, it is possible to encounter unwritten data of unknown characters that may cause an error during the READ (illegal character, end-of-file, etc). All files that are written to should be ENDFILEd.

When SEEKing within a file, remember that it is a BYTE position that is specified in the call to SEEK. Each record written to a file will contain a carriage return and line feed appended to the end of it. Remember that the carriage return and line feed MUST be included in the count of characters that make up a record. If SEEKing on a record, it is up the user to insure that each record written contains the same number of characters. If the records do not contain the same number, SEEKing can become a very complicated task.
10.12. SPECIAL CHARACTERS DURING CONSOLE I/O

Entering a control-X during input from the CP/M console will cancel the current line and echo an exclamation point (!) followed by a carriage return and line feed.

End-of-file from the CP/M console is indicated by a control-Z being entered as the first character of an input line during console I/O.

Entering a DELETE (7F hex) or control-H will erase the last character entered.
11. General Purpose SUBROUTINE/FUNCTION Library

The following list of subroutines are available for the user of FORTRAN.

SUBROUTINE Name

OPEN
LOPEN
CLOSE
DELETE
SEEK
RENAME
SETUNT
MOVE
CHAIN
LOAD
EXIT
DELAY
CIN
CTEST
OUT
SETIO
RESET
POKE
BIT
PUT

FUNCTION Name

CHAR
INP
CALL
CBTOF
PEEK
COMP

For details as to the parameters required, see the following descriptions of the individual routines.
If the error is present in the CALL statement and a CP/M error should occur, return will be to the statement following the call and error will contain the appropriate error code as listed below. If error is present and the routine completes successfully, then a zero will be returned for error. However if error is not specified and the routine encounters an error, the program will terminate with a runtime error.

The following is a list of possible errors that may returned through the optional error parameter.

\[
\begin{align*}
0 &= \text{OK} \\
1 &= \text{specified file not found} \\
2 &= \text{disk is full} \\
3 &= \text{end of file encountered} \\
4 &= \text{new filename for RENAME already exists} \\
5 &= \text{seek error} \\
6 &= \text{seek error (but file is closed)} \\
7 &= \text{format error in CHAIN or LOAD file}
\end{align*}
\]
11.1. OPEN

CALL OPEN(unit,'file'{,error})

The OPEN routine is used to open a CP/M file the user may wish to access. unit and file are required entries. If the CP/M file does not exist and error is not specified, then the file will be created. However, if error is specified and the file does not exist, the appropriate CP/M error code will be returned and the file will not be opened.

There are 2 special filenames that are recognized by the OPEN routine:

CON: used to specify either CP/M console input or output
LST: used to specify CP/M list device

Example

CALL OPEN (3,'CON:')
WRITE (3,*) 'A= ',A

will output the text to the system console. Files opened with the name CON: can also use a literal in an input statement such as:

CALL OPEN (4,'CON:')
READ (4,*), QUANT

Output can be directed to the CP/M LST device by opening the file LST: as in:

CALL OPEN (2,'LST:')
WRITE (2,*), (I, I=1,123)
To open a disk file, just the filename needs to be specified such as:

```
CALL OPEN (4,'C:FILE.BAS')
CALL OPEN (2,'DISKFILE')
CALL OPEN (3,'B:INPUT')
READ (3,*) VALUE
WRITE (2,22) VALUE
```

```
22 FORMAT (F10.4)
```

```
WRITE (4,55)
```

```
55 FORMAT ('THIS LINE WILL BE WRITTEN TO THE FILE')
```

To open a file and check if the file exists, the optional error parameter must be specified such as:

```
CALL OPEN (3,'INPUT',IERROR)
IF (IERROR .NE. 0) THEN
   TYPE 'CANNOT OPEN INPUT FILE'
   STOP 'RUN ABORTED'
ENDIF
```

NOTE: The filename (whether a character string or array name) is defined as terminating when:

1) 13 characters are encountered.
2) a NULL is encountered.
11.2. LOPEN

CALL LOPEN(unit,'file'[,error])

This subroutine is functionally the same as OPEN in that it associates a FORTRAN unit with a CP/M file except that the first character of all output records will be processed as the printer's carriage control. This is usually used for a listing device such as a printer. The first character of the record will not be output to the file but processed as follows:

<table>
<thead>
<tr>
<th>first character</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>overprint the last record</td>
</tr>
<tr>
<td>blank (space)</td>
<td>single skip</td>
</tr>
<tr>
<td>Ø</td>
<td>double skip</td>
</tr>
<tr>
<td>-</td>
<td>triple skip</td>
</tr>
<tr>
<td>l</td>
<td>page eject</td>
</tr>
</tbody>
</table>

If none of the above characters is present, then single-line spacing will be assumed. Overprinting is implemented by only generating a carriage return at the end of the line (not followed by a line feed). A page eject generates a form feed character (ØCH).

The output device that finally prints the output from this file must respond in the following manner:

ØDH (carriage return) -- return to beginning of this line
ØAH (line feed) -- space 1 line, do not return to beginning of line

ØCH (form feed) -- space to the top of the next page

A carriage return must cause the line to be printed on a line oriented device.
Example

CALL LOPEN (2,'LST:')
C
C PAGE EJECT TO TOP OF NEW PAGE
C
WRITE (2,1)
1 FORMAT ('THIS SHOULD BE ON THE TOP OF A NEW PAGE')
C
WRITE (2,2)
2 FORMAT ('ONE BLANK LINE ABOVE THIS ONE'/
* ' + ', 'THIS LINE WILL OVERPRINT THE ONE ABOVE'/
* ' - ', 5X, 'THIS LINE WILL HAVE 2 LINES ABOVE IT')
STOP
END
11.3. CLOSE

CALL CLOSE(unit)

The CLOSE routine is used as a method of closing FORTRAN files which were previously opened through the OPEN or LOPEN routine. Once the file has been closed, the file number is then available for reuse.

Example

CALL CLOSE(3)
CALL CLOSE(FILE)

11.4. DELETE

CALL DELETE ('file'{, error})

The DELETE routine is used by the FORTRAN user to remove a file from the CP/M system. Note that once a file is deleted it cannot be recovered. No error is generated if the file does not exist and the error is not present.

Example

CALL DELETE('OUTFILE')
CALL DELETE ('OUTFILE', ERROR)
CALL DELETE (FILE)
11.5. SEEK

The SEEK routines allow random positioning within a file. The file associated with unit will be positioned to position which specifies a displacement in bytes from the beginning of the file. If error is specified, there are two possible values that may be returned on a seek error. A 5 indicates a seek to a part of the file that doesn't exist, and a 6 indicates a seek to an extent of the file that does not exist. The difference between the two is that if error code 6 is return, the file associated with unit is closed. The file will have to be re-opened before it can be used again.

Example

CALL SEEK (FILE, IPOS*10+4)
CALL SEEK (3,100,ERROR)

11.6. RENAME

The RENAME routine will rename old file to new file. A runtime error occurs if old file does not exist and error is not specified or new file already exists.

Example

CALL RENAME ('OLD','NEW')

DIMENSION OFILE(2), NFILE(2)
READ (0,1) OFILE, NFILE
1 FORMAT (2A6/2A6)
CALL RENAME (OFILE, NFILE, ERROR)
11.7. CHAIN

CALL CHAIN('program name',error)

The CHAIN routine is used to load in another program overwriting the existing one in memory. This is NOT an overlay, the program that issues the CALL CHAIN will be overwritten by the new program. If program name specified does not exist; and error, was not specified, a CHAIN FL runtime error will be produced. If the format of the program name file is incorrect, program execution will be terminated. The new program to be loaded is assumed to have the .OBJ extension. The CHAIN routine will NOT close any files that may be open. Thus the new routine will be able to use the same files as the routine that issued the CHAIN without having to reopen them.

Example

CALL CHAIN ('GRAPH')
CALL CHAIN ('NXTPGM',ERROR)
CALL CHAIN (NEXT)

11.8. LOAD

CALL LOAD('file to load',load-type,error)

The LOAD routine is used to load either a standard CP/M .HEX file or a NEVADA ASSEMBLER .OBJ file. If load-type is zero, then the type of the file to be loaded will be .HEX, if load-type is non-zero, then the type will be .OBJ. This routine can be used to load assembly language routines into memory that can then be accessed through the CALL function. No check is made during the loading process to see whether the object code being read into memory overlays the program or runtime package.

It is left up to the user to insure that it does not occur. Normally the runtime package occupies memory from 100H to 4000H. If file to load does not exist and error is not specified a CHAIN FL runtime error will be produced. If the format of the program name file is incorrect, program execution will be terminated.
Example
CALL LOAD ('ASMFILE',0)
CALL LOAD ('ASMOBJ',1,ERROR)

11.9. EXIT

CALL EXIT

The EXIT routine will terminate execution of the FORTRAN program in the same manner as the STOP statement, except that EXIT does not output STOP to the system console.

Example
CALL EXIT

11.10. MOVE

CALL MOVE(count,from,displacement,to,displacement)

The MOVE routine allows direct access to memory for both reads and writes. The count specifies the number of bytes to be moved. The arguments from and to specify either a memory address to be used or a character string to be moved. Which interpretation of, from, and to is based on the respective displacement. If the displacement is negative, then the associated from or to specifies an address to be used in memory access. If the displacement is positive then the from or to that is associated with it is a string.

Example
CALL MOVE(2,A,-1,CC00,-1)
This MOVES 2 bytes from the address specified by A to address CC00 (HEX).
CALL MOVE(6,'STRING',0,CC00,-1)
   This MOVES 6 bytes of the string 'STRING' to address CC00 (HEX).

CALL MOVE(1024,CC00,-1,A,0)
   This MOVES 1024 bytes from address CC00 (HEX) to the address of A.

NOTE: The DOLLAR ($) sign indicates a hexadecimal constant. This hexadecimal constant is converted to floating point notation internally.

11.11. DELAY

CALL DELAY(wait time)

The DELAY routine enables the user to implement a time DELAY of \( \frac{1}{100} \) of a second to 655.36 seconds. Wait time must be in range of 0 to 65535 with 0 being the maximum delay time, 1 being the shortest and 65535 being \( \frac{1}{100} \) less than 0. This time is based on a 2 MHZ 8080 processor.

Example

   CALL DELAY(10)
   CALL DELAY(HOWMUCH)
   CALL DELAY(WAIT)

11.12. CIN

CALL CIN(variable)

The CIN routine enables the user to obtain a single character from the system console. The character is returned as the left most byte of variable. The left most bit of value read will be zeroed. The other 5 bytes of variable remain unchanged.
Example

C WAIT FOR A CARRIAGE RETURN (ODH) FROM THE CONSOLE
C BEFORE CONTINUING.
80 CALL CIN(CHAR)
   IF (COMP(CHAR,#0D00,1) .NE. 0) GO TO 80

In the above example, #D000 must be specified like this as the # operator stores the number as OD 00 00 00 00 00 in memory. This forces the hex value of a carriage return (0D) to be placed in the left most byte for the COMP routine.

11.13. CTEST

CALL CTEST(status)

The CTEST routine is used to test the status of the system console. A zero is returned in status if there is no character ready to input on the system console. A one is returned if there is a character.

Example

C WAIT IN A LOOP UNTIL A CHARACTER IS HIT ON THE C SYSTEM CONSOLE, THEN CHECK THE CHARACTER FOR A C LINE FEED (0AH) BEFORE CONTINUING.
ARAND=.3478
10 ARAND=RAND(ARAND)
   CALL CTEST(STATUS)
   IF (STATUS .EQ. 0) GO TO 10
C
C CHARACTER HIT, READ IT
C
   CALL CIN(CHAR)
   IF (COMP(CHAR,#0A00,1) .NE. 0) GO TO 10
11.14. OUT

CALL OUT (port, value)

This routine allows access to the 8080/8085/Z80 output ports. Value will be converted to an 8 bit number and output to port.

Example

CALL OUT(10, 1)
CALL OUT (PORT, 10)
CALL OUT (CONTROL, BITVL)

11.15. SETIO

CALL SETIO(new I/O)

This routine allows changing how the runtime package performs console I/O. The default method is setup using the CONFIG program, however it can be changed as follows:

new I/O = 0 to use direct BIOS I/O
new I/O = 2 to use CP/M function 1&2
new I/O <> 0 or 2 to use CP/M function 6
  (should be used with CP/M 2.X only).

The use of CP/M functions 1&2 permits the use of the control-p ability of CP/M to echo all the console output to the LST device. Use of other options will bypass this ability.

Example

CALL SETIO (2)
CALL SETIO (10)
11.16. RESET

CALL RESET

The RESET routine is used to inform CP/M that a diskette has been changed at runtime. This routine must be called if a diskette is changed and you wish to write on the new diskette. If the RESET routine is not called, then a BDOS: R/O will occur and the program will abort if a write is attempted on the changed diskette. This routine will prompt for a change in diskette and wait for the change to occur. Also, all open files on the diskette to be changed should be closed (using the CLOSE routine) before RESET is called. The programmer is responsible for closing the file. They can be done as follows:

Example

```
..  
CALL CLOSE (4)
CALL CLOSE (5)
C
THE "RESET" ROUTINE WILL PROMPT FOR THE CHANGE
C
CALL RESET
....
```

11.17. POKE

CALL POKE(memory location, value)

This routine allows changing of memory locations. Value will be converted to an 8 bit quantity and stored at the location specified by memory location.

Example

```
CALL POKE (0,34)
CALL POKE (1,PEEK(1)+1)
```

The last example will increment the contents of memory location 0001.
11.18. BIT

CALL BIT(variable,bit displacement, 'S', 'R', 'F', 'T', value)

The BIT subroutine allows the setting (S), resetting (R), flipping (F), or testing (T) of individual bits.

The bit at bit displacement from the start of variable will be set if S is specified, reset if R is specified, flipped (1 will become 0 and 0 will become 1) if F is specified; and, finally, the value of the selected bit will be returned in value if T is specified. Value must be present only for T. Displacement is specified starting with the leftmost bit.

Example

CALL BIT (ZAPIT, 0, 'S')
CALL BIT (ZAPIT, 0, 'T', VALUE)

11.19. PUT

CALL PUT(value)

The PUT routine is used to output a character to the console without the FORTRAN system interpreting it. Using this routine it is possible to do such things as control the position of the cursor. Value must be a number or variable and cannot be a string.

Example

CALL PUT(27)
CALL PUT(61) will clear the screen on an ADM 3A
CALL PUT (CHAR('A', 0)) will output an A
11.20. CHAR

A=CHAR(variable,displacement)

The CHAR routine is used to return the numerical value of an ASCII character located at variable+displacement where displacement is a byte displacement from the beginning of variable. For example:

Example

A='ABCDEF'
B=CHAR(A,0) returns 61
B=CHAR(A,1) returns 62
B=CHAR(A,5) returns 66

11.21. INP

A=INP(port)

This routine allows access to the 8080/8085/Z80 input ports. This is a function whose value is the current setting of the input port specified by port. No wait is done using the INP routine, it will return the current value that the input port contains.

Example

I=INP(10)
10 IF (INP(CONSOLE) .NE. 0)GO TO 10
    VALID=INP(CLOCK) .AND. 48

11.22. CALL

A=CALL(address,argument)

The CALL function causes execution of assembly language routines that have been loaded into memory (usually by the LOAD subroutine). Address is the memory location to be CALLeD. Argument will be converted to a 16 bit binary number and then passed to the called routine in both the BC and DE register pairs. The assembly routine places the value to be returned in register pair HL. The return address is placed on the 8080 stack and the called routine can just issue a standard RET instruction to return to the
FORTRAN program.

Example

CALL LOAD ('ASMFILE',1)
A=CALL ($DE00,VALUE)

11.23. CBTOF

A=CBTOF(from,displacement[,8-bit])

The CBTOF function is used to convert either a 16 bit or 8 bit binary number to its equivalent floating point value. The number to be converted is located at from+displacement if displacement is positive. If displacement is negative, then from contains the address to be used. The number is assumed to be 16 bit value (store in standard 8080 format) unless 8-bit is present, in which case it will be assumed to be an 8 bit value. The binary number is considered to be unsigned.

Example

BIOS=CBTOF($0006,1)-3

gets the base address of the CP/M bios jump table by reading the 16 bit address at location $0001 and subtracting 3 from it.

11.24. PEEK

A=PEEK(memory location)

The PEEK routine is used to read an 8 bit value from a memory location. The byte at the address specified by memory location will be returned as the value of the function.

Example

BIOS=(PEEK(1)+PEEK(2)*256)-3

This is equivalent to the example of using CBTOF.
11.25. COMP

A=COMP(string1,string2,length)

The COMP routine is used to compare character strings in the following manner:

A=COMP('string1','string2',length).

The strings will be compared on a byte basis for a byte count of length. The routine returns the following:

-1 if string1 < string2
0 if string1 = string2
+1 if string1 > string2
A. Statement Summary

variable = expression
Assigns the value of the expression to the variable.

ACCEPT input list
Reads values from the system console and assigns them to the variables in the input list.

ASSIGN n TO V
Assigns a statement label to a variable to be used in an assigned go to.

BACKSPACE unit
Positions the specified unit to the beginning of the previous record.

BLOCK DATA
Begin a BLOCK DATA subprogram for initializing variables in COMMON.

CALL name(argument list)
Call the subroutine passing the argument list.

COMMON /label1/list1 /label2/list2
Declares the variables and array that are to be placed in COMMON with the various routines.

CONTINUE
Causes no action to take place, usually used as the object of a GOTO or DO loop.

COPY filename
The specified filename is inserted into the source at the point of the COPY statement.

CTRL DISABLE
Disables program termination by control/c from the console.

CTRL ENABLE
Enables program termination by control/C from the console. Control C being enabled is the default.

DATA /var1/const1, const2/var2/c1,c2,.../
Initializes the specified variable, array element or arrays to the specified constants.

DIMENSION v(n1,n2,...),v2(n1,n2,...)
Sets aside space for arrays v and v2.
DO n i=ni,n2,n3
   Executes statements from DO to statement n, using i as index, increasing or decreasing from n1 to n2 by steps of n3.

DOUBLE PRECISION v1,v2,..
   Declares v1, v2, etc, to be double precision variables.

DUMP /id/  output list
   When a runtime error occurs, displayed id and items in output list.

END
   This statement must be the last statement of every routine.

ENDFILE unit
   Write an end of file at the current position of unit.

ERRCLR
   Clears the effect of the ERRSET statement.

ERRSET n,v
   When a runtime error occurs, control goes to the statement labeled n with variable v containing the error code.

FORMAT (field specifications)
   Used to specify input and output record formats.

FUNCTION name(argument list)
   Begins the definition of a function subprogram.

GO TO n
   Transfer control to the statement labelled n.

GO TO v,(ni,n2,...),v
   The COMPUTED GOTO transfers control to ni if v=1, n2 if v=2, etc.

GO TO v,(ni,n2,..)
   The ASSIGNED GOTO transfers control to statement ni, n2,.. depending on the value of v. V must have appeared in an ASSIGN statement.

IF (e)ni,n2,n3
   The arithmetic IF transfers control to ni if e<0, n2 if e=0 or n3 if e>0.

IF (e)statement
   The logical IF executes statement if the value of expression e is true (non-zero).
IF (e) THEN statement1 ELSE statement2 ENDIF
The IF-THEN-ELSE executes blocks of statements-
statement1 if e is true, or blocks of statements-
statement2 if e is false.

IMPLICIT type(letter list)
Changes the default type of variables that start
with the letters in the letter list.

INTEGER v1, v2, ...
Declares v1, v2, etc, to be integer variables.

LOGICAL v1, v2, ...
Declares v1, v2, etc, to be logical variables.

PAUSE 'character string'
Suspends program execution until any key is hit,
displaying PAUSE and character string.

READ (unit, format[, ERR=]{, END=}) input list
Reads values from unit according to format and
assigns them to the variables in input list.

REAL v1, v2, ...
Declares v1, v2, etc, to be real variables.

RETURN
Returns control from a subprogram to the
statement following either the call or the
function reference.

RETURN i
The multiple return statement returns control from
a subprogram to statement i in the calling
routine.

REWIND unit
The file associated with unit is closed, then
reopened at the beginning of the same file.

STOP 'character string'
Terminates program execution and displays
character string on the system console.

STOP n
Terminates program execution and displays n on the
system console.

SUBROUTINE name(argument list)
Begins the definition of a subroutine subprogram.

TRACE OFF
Turns statement tracing off.
TRACE ON
   Turns statement tracing on.

TYPE output list
   Displays the value of the variables in output list on the system console.

WRITE (unit,format[,ERR=]) output list
   Writes the values of the variable in output list to unit according to format.
## B. Summary of System Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Arg</th>
<th>Result</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>Sine(x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>COS</td>
<td>Cosine(x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>TAN</td>
<td>Tangent(x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>ATAN</td>
<td>Arctangent(x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>ATAN2</td>
<td>Arctangent(y/x)</td>
<td>2</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>ALOG</td>
<td>Log base e (x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>ALOG10</td>
<td>Log base 10 (x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>MOD</td>
<td>Remainder (x/y)</td>
<td>2</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>AMOD</td>
<td>Remainder (x/y)</td>
<td>2</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>SQRT</td>
<td>Square Root (x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Make real (x)</td>
<td>1</td>
<td>real</td>
<td>integer</td>
</tr>
<tr>
<td>IFIX</td>
<td>Truncate (x)</td>
<td>1</td>
<td>integer</td>
<td>real</td>
</tr>
<tr>
<td>ABS</td>
<td>Absolute (x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>IABS</td>
<td>Absolute (x)</td>
<td>1</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>RAND</td>
<td>Random Number (x)</td>
<td>1</td>
<td>real</td>
<td>real 0 &lt; R &lt; 1.0</td>
</tr>
<tr>
<td>EXP</td>
<td>e**(x)</td>
<td>1</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>COMP</td>
<td>Compare strings</td>
<td>3</td>
<td>either</td>
<td>real</td>
</tr>
<tr>
<td>CALL</td>
<td>CALL assembly pgm</td>
<td>2</td>
<td>either</td>
<td>real</td>
</tr>
<tr>
<td>AMAXØ</td>
<td>Maximum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>AMAX1</td>
<td>Maximum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>MAXØ</td>
<td>Maximum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>MAX1</td>
<td>Maximum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>AMINØ</td>
<td>Minimum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>AMINØ1</td>
<td>Minimum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>MINØ</td>
<td>Minimum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>MIN1</td>
<td>Minimum</td>
<td>&lt;255</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>BIT</td>
<td>Bit handling</td>
<td>3/4</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>SIGN</td>
<td>Transfer of sign</td>
<td>2</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>ISIGN</td>
<td>Transfer of sign</td>
<td>2</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>DIM</td>
<td>Positive difference</td>
<td>2</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>IDIM</td>
<td>Positive difference</td>
<td>2</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>CHAR</td>
<td>character value</td>
<td>1</td>
<td>either</td>
<td>character</td>
</tr>
<tr>
<td>CALL</td>
<td>execute asm pgm</td>
<td>2</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>INP</td>
<td>input from a port</td>
<td>1</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>CBTOF</td>
<td>convert to real</td>
<td>2/3</td>
<td>real</td>
<td>both</td>
</tr>
<tr>
<td>COMP</td>
<td>compare strings</td>
<td>3</td>
<td>either</td>
<td>either</td>
</tr>
<tr>
<td>PEEK</td>
<td>examine mem loc.</td>
<td>1</td>
<td>either</td>
<td>either</td>
</tr>
</tbody>
</table>

Most of the above functions are ANSI standard except for RAND. This function behaves as if it were returning an entry from a table of random numbers. The argument of RAND determines which entry of this table will be returned:

**Rand Arg.**

<table>
<thead>
<tr>
<th>Rand Arg.</th>
<th>Value returned for RAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>The next entry in the table</td>
</tr>
<tr>
<td>-1</td>
<td>The first entry in the table. Also the pointer for the next entry (arg=Ø) is reset to the second entry in the table.</td>
</tr>
<tr>
<td>n</td>
<td>Returns the table entry following n.</td>
</tr>
</tbody>
</table>
C. Summary of System Subroutines

CALL BIT (variable, disp, code)
Set, resets or flips bit \(0+\text{disp}\) of variable according the code.

CALL CBTOF(loc1, displ, loc2{, flag})
Converts a binary number to its floating point equivalent.

CALL CHAIN('program name'[, error])
Loads another program and executes it.

CALL CIN(var)
Reads a single character from the system console.

CALL CLOSE(unit)
Close the file associated with unit.

CALL CTEST(status)
Determines if a character has been entered on the system console.

CALL DELAY(time)
Delays execution the specified time in one hundreds of a second.

CALL DELETE('filename'[, error])
Delete the specified filename from the disk.

CALL EXIT
Terminates program execution.

CALL MOVE(n, loc1, displ, loc2, disp2)
Moves n bytes from loc1 to loc2.

CALL OPEN(unit, 'filename'[, error])
Opens the specified filename and associates it with unit.

CALL LOAD('filename', load-type[, error])
This routine is used to load a file of type .HEX or .OBJ into memory depending on the value of load-type. It is usually used to load assembly language routines into memory. No check is made to see if the code that is loaded into memory would overwrite the program or CP/M.

CALL LOPEN(unit, 'filename'[, error])
Opens the specified filename and associates it with unit. This file is also treated as a printer file with the first character of each output record controlling paper movement.
CALL POKE(LOCATION, VALUE)
The POKE routine is used to change a memory location. Value will be stored in memory at location.

CALL OUT(port, value)
value is converted to an 8 bit number and output to port port.

CALL PUT(CHARACTER)
The PUT routine is used to output a character to the system console directly. It is commonly used to do such things as clear the screen or position the cursor.

CALL RENAME('old name', 'new name'[, error])
Renames old name to be new name.

CALL RESET
The RESET routine allows for the changing of a diskette at runtime and then being able to write on the changed diskette. Without using this function, an attempt to write on a diskette that has been changed will result in a BDOS read only error.

CALL SEEK (unit, position)
Positions the file associated with unit to the byte position specified by position.

CALL SETIO (new I/O)
Allows changing the way that console I/O is performed during program execution.
D. RUNTIME ERRORS

During execution of a program, there are numerous conditions that can occur which cause program termination. When one of these conditions is encountered, a RUNTIME ERROR message will be generated to the system console file. The message has the format:

Runtime error: XXXXXXXX, called from loc. YYYYY
Pgm was executing line LLLL in routine NNNN

where: XXXXXXXX is the ERROR, YYYYY is the memory location of the CALL to the runtime package in which the error occurred.

The second line of the error message will be generated as a traceback of CALL statements that have been executed. The LLLL is the FORTRAN generated line number (shown on the listing of the source from the compiler) of the statement which caused the error, and NNNN if the name of the routine in which that line number corresponds. The line number will be output as ???? if the X option was not specified on the $OPTIONS statement for a given routine. If multiple 'PGM WAS ...' lines are printed, the first one specifies the line in which the error actually occurred.
Runtime Errors

INT RANG

INTEGER OVERFLOW: a result greater than 8 digits has been generated in an expression.

CONVERT

16 BIT CONVERSION ERROR: in converting a number from integer to internal 16 bit binary, an overflow has occurred. This can occur on all statements associated with I/O (unit number), subscript evaluation and anywhere that a number has to be converted from floating to 16 BIT binary. Also subscripting outside of the DIMENSIONED space for an array can cause this error.

ARG CNT

ARGUMENT COUNT ERROR: a subprogram call had too many or too few arguments. In other words, the number of arguments in the CALL or function reference is not the same as the number of parameters specified for this SUBROUTINE or FUNCTION.

COM GOTO

COMPUTED GO TO INDEX OUT OF RANGE: the variable specified in a computed GOTO is either zero or greater than the number of statement labels that where specified.

OVERFLOW

FLOATING POINT OVERFLOW: the result of a floating point operation has resulted in a number whose value is too large to be stored.

DIV ZERO

DIVIDE BY ZERO: an attempt has been made to divide by zero.
SQRT NEG

SQRT OF NEGATIVE NUMBER: argument of the square root function is negative.

LOG NEG

LOG OF NEGATIVE NUMBER: argument of the log either (ALOG or ALOG10) function is negative.

CALL PSH

CALL STACK PUSH ERROR: this error is caused by a recursive subprogram CALLS of depth greater than 36. Only in very special cases should a subprogram CALL itself or one of those that has CALLED it.

CALL POP

CALL STACK POP ERROR: this error should never occur (This means that a RETURN has been executed that does not have a corresponding CALL or FUNCTION reference. Usually caused by user assembly language programs).

CHAIN FL

CHAIN FILE ERROR: the filename specified in a call to the CHAIN or LOAD routine was not found on the disk.

ILL UNIT

ILLEGAL UNIT NUMBER (<2 OR >7): the unit number in a READ, WRITE, OPEN, LOPEN, REWIND, CLOSE, SEEK is either < 2 or greater than 7.

UNIT OPN

UNIT ALREADY OPEN: this is generated by the OPEN or LOPEN routine when an attempt to open a file on an already open FORTRAN logical unit. This error will also occur if units 0 or 1 is specified in the OPEN or LOPEN call.

DSK FULL

DISK FULL: either the disk is full or the directory is full.
UNIT CLO

UNIT CLOSED: the unit number passed to the CLOSE routine specifies a unit number that has not been OPENed.

CON BIN

BINARY I/O TO CONSOLE: binary I/O is not supported to the system console.

LINE LEN

LINE LENGTH ERROR: an attempt has been made to READ or WRITE a record whose length exceeds 250 characters. This count also includes a carriage return at the end of the line.

FORMAT

FORMAT ERROR: an unrecognized or invalid FORMAT specification has been encountered in a FORMATTED READ or WRITE. The most likely error is an unrecognized format specification or blanks in a variable format.

I/O ERR

I/O ERROR: an error occurred during a READ or WRITE operation and the ERROR label was not specified in the statement. It will also be generated during a READ if END OF FILE is encountered and an EOF label was not specified.

ILL CHAR

ILLEGAL CHARACTER: an illegal character has been encountered during a READ.

I/O LIST

INVALID I/O LIST: this error indicates an error in the I/O list specification of a formatted WRITE or READ. This error will not normally occur.
ASN GOTO

ASSIGNED GO TO ERROR: the value of the variable specified in an ASSIGNED GO TO does not match that of one of the statement labels listed.

CONTROL/C

CONTROL/C error: CONTROL/C was hit and the CONTROL/C was not trapped.

INPT ERR

INPUT ERROR: during a READ an invalid character has been encountered for the number being processed. This will be generated for such things as: two decimal points in a number, an E in an F type field, decimal point in an I type field, etc.

FILE OPR

FILE OPERATION ERROR: an error has occurred while trying to do some file operation, such as renaming when the new file already exists.

SEEK ERR

SEEK ERROR: an error has occurred while positioning a file to the specified position and no error variable was specified in the CALL.
E. Compile Time Errors

The following is a list of errors that may occur during the compilation of a FORTRAN program. If the G option is not selected a two digit error number will be printed instead. This number can be found at the beginning of each line.

00 *FATAL* compiler error
01 Syntax error, 2 operators in a row
02 unexpected continuation (column 6 not blank or 0)
03 input buffer overflow (increase B= compiler option)
04 invalid character for FORTRAN statement
05 unmatched parenthesis
06 statement label > 99999
07 invalid character encountered in statement label
08 invalid HEX digit encountered in constant
09 expected constant or variable not found
0A 8 bit overflow in constant
0B unidentifiable statement
0C statement not implemented
0D quote missing
0E SUBROUTINE/FUNCTION/BLOCK DATA not first statement in routine
0F columns 1-5 of continuation statement are not blank
10 cannot initialize BLANK COMMON
11 RETURN is not valid in main program
12 syntax error on unit specification
13 missing comma after ) in COMPUTED GO TO
14 missing variable in COMPUTED GO TO
15 invalid variable in ASSIGNED/COMPUTED GO TO
16 invalid LITERAL, no beginning quote
17 number of subscripts exceeds maximum of 7
18 invalid SUBROUTINE or FUNCTION name
19 subscript not POSITIVE INTEGER CONSTANT
1A FUNCTION requires at least one argument
1B syntax error
1C invalid argument in SUBROUTINE/FUNCTION call
1D first character of variable not alphabetic
1E ASSIGNED/COMPUTED GOTO variable not integer
1F label has already defined
20 specification of array must be integer
21 invalid variable name
22 invalid DIMENSION specification
23 dimension specification is invalid
24 variable has already appeared in type statement
25 invalid subroutine name in CALL
26 SUBPROGRAM argument cannot be initialized
27 improperly nested DO loops
28 unit not integer constant or variable
29 Array size exceeds 32K
2A invalid use of unary operator
2B variable DIMENSION not valid in MAIN program
2C variable dimensioned array must be argument
2D DO/END/LOGICAL IF cannot follow LOGICAL IF
2E undefined label
2F unreferenced label
30 FUNCTION or ARRAY missing left parenthesis
31 invalid argument of FUNCTION or ARRAY
32 DIMENSION specification must precede first executable statement
33 unexpected character in expression
34 unrecognized logical opcode
35 argument count for FUNCTION or ARRAY wrong
36 *COMPILER ERROR* popped off bottom of operand stack
37 expecting end of statement, not found
38 statement too complex; increase P and/or O table
39 invalid delimiter in ARITHMETIC IF
3A invalid statement number in IF
3B HEX constant > FFFF (HEX)
3C replacement not allowed within IF
3D multiple assignment statement not implemented
3E subscripted-subscripts not allowed
3F subscript stack overflow; increase P= or O=
40 missing left ( in READ/WRITE
41 invalid unit specified
42 invalid FORMAT, END= or ERR= label
43 invalid element in I/O list
44 built-in function invalid in I/O list
45 cannot subscript a constant
46 variable not dimensioned
47 invalid subscript
48 missing comma
49 index in IMPLIED DO must be a variable
4A invalid starting value for IMPLIED DO
4B invalid ending value of IMPLIED DO
4C invalid increment of IMPLIED DO
4D illegal use of built-in function
4E variable cannot be dimensioned in this context
4F invalid or multiple END= or ERR=
50 invalid constant
51 exponent overflow in constant
52 invalid exponent
53 character after . invalid
54 integer overflow
55 integer underflow (too small)
56 missing = in DO
57 string constant not allowed
58 invalid variable in DATA list
59 DATA symbol not used in program, line
5A invalid constant in DATA list
5B error in DATA list specification
5C FUNCTION invalid in DATA list
5D no filename specified on COPY
5E runtime format not array name
5F DUMP label invalid or more than 10 characters
60 more than 1 IMPLICIT is not allowed
61 IMPLICIT not first statement in MAIN, 2nd statement in SUBPROGRAM
62 data type not REAL, INTEGER or LOGICAL
63 illegal IMPLICIT specification
64 improper character sequence in IMPLICIT
65 variable already DIMENSIONED
66 Q option must be specified for ERRSET/ERRCLR
67 Hex constant of zero (0) invalid in I/O stmt
68 Argument cannot also be in COMMON
69 Illegal COMMON block name
6A Variable already in COMMON
6B Array specification must precede COMMON
6C Executable statement invalid in BLOCK DATA
6D Hex constant of 27H (') invalid in FORMAT
6E Invalid number following STOP or PAUSE
6F invalid TRACE statement (operand not ON/OFF)
70 invalid IOSTAT= variable
71 missing , in ENCODE/DECODE
72 invalid label in ASSIGNED GOTO
73 invalid variable in ASSIGNED GOTO
74 label not allowed on this statement
75 multiple RETURN not valid in FUNCTION
76 UNUSED
77 no matching IF-THEN for ELSE or ENDF
78 invalid ELSE or ENDF
79 missing ENDF
7A initialization of non-COMMON variable
7B "DOUBLE PRECISION" not supported, treated as "REAL"
7C UNUSED
7D UNUSED
7E UNUSED
7F UNUSED
80 *FATAL* no program to compile
81 *FATAL* missing $OPTIONS statement
82 *FATAL* missing = in $OPTIONS statement
83 *FATAL* invalid digit in number in $OPTIONS
84 *FATAL* value exceeds 255 in $OPTIONS
85 *FATAL* COMMON table overflow, increase C=
86 *FATAL* unknown option (letter before =)
87 *FATAL* missing END statement
88 *FATAL* LABEL TABLE overflow, increase L=
89 *FATAL* SYMBOL TABLE overflow, increase S=
8A *FATAL* ARRAY STACK overflow, increase A=
8B *FATAL* DO LOOP STACK overflow, increase D=
8C *FATAL* stack overflow (compiler error)
8D *FATAL* stack overflow (compiler error)
8E *FATAL* internal tables exceed user memory
8F *FATAL* MEMORY ERROR
90 *FATAL* OPEN error on COPY file
91 *FATAL* too many routines to compile (> 62)
92 *FATAL* no more room to store DATA statements
93 *FATAL* IF-THEN stack overflow, increase I=
94 *FATAL* Nested "COPY" statements not permitted
95 *FATAL* Disk write error (disk probably full)
96 *FATAL* Cannot close file (disk probably full)
97 *FATAL* Input file not found
98 *FATAL* Invalid drive specifier
99 *FATAL* No filename found on COPY statement
9A *FATAL* File specified on COPY not found
F. ASSEMBLY LANGUAGE INTERFACE

ASSEMBLY statements can be directly inserted into a FORTRAN program by preceding the statement with an ASTERISK (*). The line that contains that asterisk will be directly output to the assembly file without further processing (the asterisk is deleted first). Because of the nature of the FORTRAN compiler (it actually reads one statement ahead of where it is processing) it is ALWAYS a good idea to put a CONTINUE statement immediately preceding the first assembly statement in each separated group of assembly statements. The CONTINUE will cause the assembly statements to be inserted at the expected place. FORTRAN maintains nothing in the registers between statements, but does use the 8080 stack for saving RETURN addresses for user called FUNCTIONS and SUBROUTINES.

Example

    CONTINUE
    * MVI A,'A'
    * STA STRING
G. GENERAL COMMENTS

1. In the description of the individual routines, anywhere that a character string is specified, a variable or array name can be used. The variable or array can be set to the desired character string.

2. A variable can be set to a character string using an assignment statement such as:

   A='STRING'

No more than 6 characters will be retained for any variable and if less than 6 will be zero filled in the low order bytes of the variable.

3. If a variable or array name is used to reference a CP/M file (such as in the OPEN routine) the filename itself within the variable (or array) is terminated after:

   1) the first 15 characters,

   2) a NULL is encountered.

4. Hexadecimal constants can be used anywhere that a constant or variable is permitted. A hexadecimal constant is specified by preceding it by a dollar sign ($). Examples are:

   A=$E060
   A=-$CC00

Hexadecimal constants are limited to a maximum value of FFFF. An error is generated if a hexadecimal constant exceeds this limit. Internally a hexadecimal constant is treated as any other INTEGER constant would be.
5. A hexadecimal constant that is preceded by a # instead of a $ will be stored internally in binary format in the first two bytes of the variable. Numbers of this form should not be used in any expression as they are not stored in the normal floating point format. The number is stored in standard 8080 format (HIGH byte followed by LOW byte).

6. A backslash (\) can be used in a literal to specify an 8 bit binary constant to be inserted at that point. The constant is enclosed in backslashes and is assumed to be a hexadecimal constant. The backslash can be changed using the CONFIG program supplied.

Example

A='THIS \32\ IS AN EXAMPLE'
CALL OUTIT(3,1,'\7F\\FF\',2,32)

10 FORMAT ('IT IS ALLOWED \1\ HERE \FF\ ALSO')

NOTE The backslash is the default character and can be changed using the CONFIG program.

7. 8 Fortran files may be open at any one time (file numbers 0-7). Remember that files 0 and 1 are permanently open.
H. USE OF THE NORTH STAR FLOATING POINT BOARD

A feature of NEVADA FORTRAN is that it will directly use the North Star floating point board if one is installed in the computer upon which the program is executed. Use the CONFIG program to specify the 2 memory addresses that the floating point board uses. If you should specify that you have a floating point board, at runtime the runtime package checks to see if the floating point board is actually in the system and if not, then uses the software floating point routines instead of the hardware. One disadvantage of using the floating point board is that the range of real number decreases to $10^{-63}$ to $10^{+63}$. Any number generated outside this range will produce an overflow error message.
I. COMPARISON OF NEVADA FORTRAN AND ANSI FORTRAN

NEVADA FORTRAN includes the following extensions to version X3.9-1966 of ANSI Standard FORTRAN:

1. Free-format input and output.
2. IMPLICIT statement for setting default variable types
3. Options end-of-file and error branches in READ and WRITE statements.
4. COPY statement to insert source files into a FORTRAN program.
5. Direct inline assembly language.
6. Access to file system for such functions as creating, deleting and renaming files.
7. Random access on a byte level to files.
8. Access to absolute memory locations.
9. Program controlled time delay.
10. A pseudo random number generator function.
11. Program control of runtime error trapping.
12. Ability to chain a series of programs.
13. Ability to load object code into memory.
14. CALL function to execute previously loaded code.
15. Program tracing.
16. IF-THEN-ELSE statement.
17. Enabling and disabling console abort of program.
18. ENCODE and DECODE memory to memory I/O.
19. Multiple returns from subroutines.

NEVADA FORTRAN does not include the following features of ANSI standard FORTRAN:

1. Double precision, double precision functions.
   (Double precision is treated as single precision).
2. Complex numbers, complex statements and functions.
3. EQUIVALENCE statement.
4. Extended DATA statement of the form:
   
   DATA A,B,C/1,2/3/

5. The P format specifications.
7. The following are reserved names and cannot be used for functions, subroutines, or COMMON block names:

   A, B, C, D, E, H, L, M, SP, PSW

8. EXTERNAL statement.
9. Subscripted subscripts.
10. Certain of the numerical library functions such as the hyperbolic functions and others.
J. SAMPLE PROGRAMS

On the following pages you will find listings of the sample programs that may have been included on your diskette.

C
C "CHAIN.FOR"
C
C THIS ROUTINE DEMONSTRATED THE 'CHAIN' FUNCTION, ALL IT DOES IS REQUEST THE NAME OF THE PROGRAM TO CHAIN TO AND THEN CHAIN.
C
DIMENSION IF(3)
TYPE 'FILE?'
C
GET THE FILENAME TO CHAIN TO
READ (0,1) IF
FORMAT (3A6)
C
CHAIN TO IT
CALL CHAIN (IF,IER)
C
ONLY GETS HERE IF AN ERROR HAPPENS
TYPE 'ERROR FROM CHAIN = ',IER
CALL EXIT
END
OPTIONS X
C
C "DUMP.FOR"
C
C THIS PROGRAM DEMONSTRATED THE USE OF THE DUMP STATEMENT.
C
C CALL 'X' FOR TRACEBACK PRINTOUT (JUST FOR SHOW)
C
        CALL X
        END

OPTIONS X
SUBROUTINE X

C
C DEFINE THE DUMP STATEMENT TO BE USED IN CASE OF AN
C ERROR, WITH DUMP ID OF 'ROUTINE-X'
C
            DUMP /ROUTINE-X/ I,J,K
            I=1
            J=2
            K=I+J

C
C CREATE AN ERROR TO CAUSE DUMP STATEMENT TO BE ACTIVE
C
            Z=1/0
            END
OPTIONS X
C
C "GRAPH.FOR"
C
C GRAPH SINE FUNCTION FROM -PI TO PI IN INCREMENT OF .12
C
        DIMENSION LINE(70)
        INTEGER WHERE
C
C OPEN UNIT 6 TO WRITE TO CONSOLE
C
        CALL OPEN (6, 'CON: ')
C
C WRITE TITLE
C
        WRITE (6,2)
2        FORMAT (28X,'GRAPH OF SIN')
        TYPE
        TYPE

C SET PI AND -PI
C
        PI=3.1415926
        MPI=-PI
C
C MAIN LOOP
C
        DO 100 ANGLE=MPI,PI,.12
C
C FIGURE OUT WHICH ELEMENT IN ARRAY SHOULD BE SET TO *,
C SIN RETURNS -1 TO 1 WHICH IS CONVERTED TO -35 TO 35
C AND THEN OFFSET SO FINAL RANGE IS 1 TO 70
C
        WHERE=SIN(ANGLE)*35+35
C
C FIGURE OUT HOW MUCH TO BLANK IN THE OUTPUT ARRAY
C
        IBLANK=MAX0(35,WHERE)
C
C AND BLANK IT
C
        DO 15 I=1,IBLANK
15        LINE(I)=''
C
C HMM... WHICH SIDE OF ZERO ARE WE ON?
C
        IF (WHERE .GT. 35) THEN
C
C RIGHT SIDE
C
        DO 20 I=36,WHERE
        LINE(I)='*'
20        ELSE
C LEFT SIDE

DO 30 I=WHERE,35
   LINE(I)='*'
   ENDF

C SET "ZERO"
C
   LINE(35)='+'
C
C AND THE SIN VALUE
C
   LINE(WHERE)='*'
C
C IF THIS VALUE IS < 35, SET SO WE OUTPUT TO ZERO LINE
C
   IF (WHERE .LE. 35)WHERE=35
C
C AND FINALLY OUTPUT THE LINE
C
   WRITE (6,21) (LINE(I),I=1,WHERE)
21   FORMAT (70A1)
100 CONTINUE
   CALL EXIT
END
"LOAD.FOR"

THIS ROUTINE DEMONSTRATED THE USE OF THE 'LOAD' ROUTINE TO LOAD AN ASSEMBLY LANGUAGE FILE INTO MEMORY AND THEN CALL IT FORM FORTRAN

INTEGER A

FIND OUT WHICH ONE TO LOAD

TYPE 'Enter 0 to "LOAD" LD.HEX'
TYPE 'Enter 1 to "LOAD" LD.OBJ'
ACCEPT 'Which one: ',LTYPE

IF (LTYPE .EQ. 0) THEN
    TYPE '"LOAD"ing "LD.HEX"'
ELSE
    TYPE '"LOAD"ing "LD.OBJ"
ENDIF

MUST LOAD "LD.HEX" OR "LD.OBJ" INTO MEMORY BEFORE WE CAN CALL IT

CALL LOAD ('LD',LTYPE,IER)

TYPE 'ERROR FOR LOAD=',IER

CHECK THE RETURNED ERROR CODE FROM LOAD

IF (IER .NE. 0)STOP 'LOAD ERROR'

"CALL" THE ROUTINE

A=CALL ($8000,1)

RESULT SHOULD BE 2

TYPE 'THE RESULT OF THE ASSEMBLY ROUTINE IS: ',A

CALL EXIT
END

"LD.ASM"

THIS ROUTINE IS USED BY "LOAD.FOR", ALL IT DOES IS TO DOUBLE THE NUMBER SENT TO IT NUMBER IS PASSED IN DE FROM FORTRAN PROGRAM AND RESULT IS PASSED BACK IN HL TO FORTRAN PROGRAM

ORG 8000H
PUSH D ;NUMBER FROM FORTRAN PROGRAM
POP H ;GET IT TO HL
DAD H ;HL*2
RET ;RETURN IT
OPTIONS X,Q
C
C "RAND.FOR"
C
C THIS PROGRAM GENERATES A SEQUENCE OF RANDOM NUMBERS, 
C DIVIDES THEM INTO 10 INTERVALS AND COUNTS HOW MANY 
C RANDOM NUMBER FALL INTO EACH INTERVAL. FINALLY IT 
C PRINTS OUT THE COUNTS OF EACH INTERVAL.
C
DIMENSION NUM(10)
DATA NUM/10*0/
INTEGER T,A,D,FLAG,TIME(6),DATE(6),START,END
99 DO 50 I=1,10
50 NUM(I)=0
ACCEPT 'How many? ',K
DO 1 I=1,K
L=RAND(0)*10+1
1 NUM(L)=NUM(L)+1
TYPE NUM
GO TO 99
END
OPTIONS X,Q
C
C "SEEK.FOR"
C
C THIS PROGRAM DEMONSTRATES RANDOM ACCESS I/O
C
C IT FIRST WRITES A FILE OF NUMBERS, THEN REQUESTS
C A RECORD, READ NUMBER THAT THE RECORD CONTAINS,
C ADDS 1 TO THE NUMBER READ AND WRITES IT BACK INTO
C THE SAME RECORD
C
ERRSET 500,I
CALL DELETE('TEST')
C
OPEN THE TEST FILE
C
CALL OPEN (2,'TEST')
C
READ HOW MANY RECORDS TO CREATE
C
ACCEPT 'HOW MANY RECORDS? ',K
C
WRITE THE FILE
C
DO 1 I=0,K
1 WRITE (2,2) I
2 FORMAT (I5)
TYPE 'FILE WRITTEN'
TYPE
REWIND 2
GO TO 10
CALL OPEN(2,'TEST')
C
REQUEST RECORD TO DISPLAY
C
10 ACCEPT 'WHICH RECORD? ',K
C
POSITION THE FILE (EACH RECORD IS 7 CHARACTERS,
C 5 FOR NUMBER, 1 FOR CARRIAGE RETURN AND 1 FOR LINE FEED
C
CALL SEEK (2,7*K,IER)
C
CHECK THE ERROR CODE
C
IF (IER .NE. 0) THEN
TYPE 'SEEK ERROR, CODE= ',IER
CALL CLOSE (2)
CALL DELETE ('TEST')
STOP
ENDIF
C READ THE CURRENT VALUE
C
READ (2, 2) I
TYPE 'CURRENT VALUE OF RECORD ',K,' IS ',I
C
C POSITION BACK TO THE SAME RECORD
C
CALL SEEK(2, 7*K)
I=I+1
C
C WRITE THE UPDATED VALUE
C
WRITE (2, 2) I
GO TO 10
C
C TRAP ERROR
C
500 TYPE '*** ERROR TRAPPED ***'
TYPE 'ERROR CODE = ',I
CALL CLOSE (2)
CALL DELETE ('TEST')
STOP 'ERROR EXIT'
END
C
C "SORT.FOR"
C
C THIS ROUTINE IS A DEMONSTRATION OF A SHELL SORT
C
INTEGER T, A, D, FLAG, TIME(6), DATE(6), START, END
DIMENSION A(2000)
TYPE 'Shell sort'
TYPE

C
C GET HOW MANY NUMBERS TO SORT
C
88 ACCEPT 'How many numbers (2-2000) ', NN
IF (NN .LT. 2.0.R. NN .GT. 2000) STOP

C
C GENERATE ARRAY OF NUMBERS TO SORT
C
DO 10 I = 1, NN
   10 A(I) = (RAND(0)*NN)+1
TYPE 'Starting sort'
D = NN
FLAG = 0

100 D = IFIX((D+1)/2)
C
1000 D = IFIX((D+1)/2)
C
C TYPE OUT INTERMEDIATE STUFF
C
TYPE ' D=', D

110 ND = NN - D
DO 150 N = 1, ND
   IF (A(N) .LE. A(N+D)) GO TO 150
NPD = N + D
T = A(N)
A(N) = A(NPD)
A(NPD) = T
FLAG = 1

150 CONTINUE
IF (FLAG .EQ. 1) THEN
   FLAG = 0
   GO TO 110
ENDIF
IF (D .GT. 1) GO TO 100
C
TYPE 'All done'

C
C TYPE OUT SORTED ARRAY
C
TYPE (A(I), I = 1, NN)
GO TO 88
END
OPTIONS X,Q
C
C "TRACE.FOR"
C
C THIS ROUTINE DEMONSTRATES THE USE OF THE 'TRACE' AND
C 'ERROR' TRAPPING FUNCTIONS
C
C TYPE 'STARTING EXECUTION'
C
C SET ERROR TRAPPING: ON ERROR GO TO STATEMENT 500 WITH
C ERROR CODE IN VARIABLE I
C
ERRSET 500,I
10 CONTINUE
C
C TURN TRACING OFF
C
TRACE OFF
C
C GET AN INPUT # FROM THE USER
C
ACCEPT '#: ',K
C
C IF <0, TERMINATE
C
IF (K .LE. 0)GO TO 99
C
C IF INPUT # > 100, THEN TURN TRACING ON
C
IF (K .GT. 100)TRACE ON
C
C AND OUTPUT THE NUMBERS, TO SEE EFFECT OF THE
C ERROR TRAPPING, HIT CONTROL-C
C
DO 20 I=1,K
20 TYPE I
    GO TO 10
C
99 TYPE 'DONE'
    STOP
C
C ERROR TRAPPING HAPPENS HERE
C
500 TYPE 'ERROR TRAPPED, IER= ',I
END
THIS SAMPLE PROGRAM SHOWS HOW TO HANDLE THE CURSOR AND CLEAR SCREEN FUNCTIONS OF A CRT

    PAUSE 'CLEAR SCREEN'
    CALL SCREEN (1,0,0)

WRITE THE LINE NUMBERS OUT FROM THE BOTTOM LINE TO THE TOP

    DO 10 I=23,0,-1
    CALL SCREEN (2,0,1)
    WRITE (1,2) I
    2 FORMAT ('LINE ',I2,Z)
    CONTINUE

WAIT 5 SECONDS

    CALL DELAY (500)

CLEAR THE SCREEN AGAIN

    CALL SCREEN(1,0,0)
    ACCEPT 'ENTER ANY NUMBER ',A
    I=AMOD(A,300)+1
    DO 99 J=1,I
    99 Z=RAND(0)

AGAIN CLEAR IT

    CALL SCREEN(1,0,0)

NOW PUT RANDOM PLUS SIGNS ALL OVER THE SCREEN

    DO 40 I=1,400
    CALL SCREEN(2,IFIX(RAND(0)*80),IFIX(RAND(0)*23))
    CALL PUT ( CHAR (I+I),0)
    CALL DELAY(500)

OUTPUT A BELL

    CALL SCREEN(5,0,0)

CLEAR IT NOW

    CALL SCREEN(1,0,0)
    CALL EXIT
END
SUBROUTINE SCREEN(FUNCT, X, Y)

C
C THIS IS A SAMPLE SCREEN DRIVER FOR AN ADM-3A TERMINAL
C
INTEGER FUNCT, X, Y
IF (FUNCT .LE. 0. OR. FUNCT .GT. 5) RETURN
C
FUNCTION:
C
1 = CLEAR SCREEN
2 = POSITION CURSOR
3 = SET REVERSE VIDEO
4 = SET NORMAL VIDEO
5 = BELL
C
GO TO (100, 200, 300, 400, 500), FUNCT
C
CLEAR SCREEN
C
100 CALL PUT (12)
RETURN
C
SET CURSOR TO X, Y
C
200 CALL PUT (27)
CALL PUT (102)
CALL PUT (X+32)
CALL PUT (Y+32)
RETURN
C
SET REVERSE VIDEO
C
300 RETURN
C
SET NORMAL VIDEO
C
400 RETURN
C
OUTPUT A BELL
C
500 CALL PUT (7)
RETURN
END
C THIS PROGRAM DEMONSTRATES THE USE OF THE SET FUNCTION OF THE "BIT" ROUTINE. YOU ENTER THE BIT TO BE SET AND CAN SEE EXACTLY WHICH BIT IS CHANGED.

C

1

A=0
ACCEPT 'BIT?',B
IF (B .GT. 47) THEN
    TYPE 'INVALID BIT NUMBER, ONLY 0-47'
    GO TO 1
ENDIF
IF (B .LT. 0) STOP

C SET THE BIT

CALL BIT (A,B,'S')

C OUTPUT THE WORD IN HEX FORMAT

WRITE (1,2) A
2 FORMAT (K12)
GOTO 1
END
K. SAMPLE PROGRAM COMPILATIONS AND EXECUTION

On the following pages you will find examples of the compilation and execution of several of the sample programs listed above. The following notes refer to the next few pages:

1) input is underlined.

2) CP/M output is printed in bold.

3) FORTRAN output (either compiler or execution) is neither underlined or bold.

4) notes are in { }. 
NEVADA FORTRAN 3.0 (MOD 0)

***** NEVADA Fortran 3.0 (Mod 0) ** Compiling File: GRAPH.FOR *****

0001 OPTIONS X
C GRAPH SINE FUNCTION FROM -PI TO PI IN
C INCREMENT OF .12

0002 DIMENSION LINE(70)
0003 INTEGER WHERE
C OPEN UNIT 6 TO WRITE TO CONSOLE
C
0004 CALL OPEN (6,'CON: ')

C WRITE TITLE
C
0005 WRITE (6,2)
0006 2 FORMAT (28X,'GRAPH OF SIN')

C SET PI AND -PI
C
0009 PI=3.1415926
0010 MPI=-PI

C MAIN LOOP
C
0011 DO 100 ANGLE=MPI,PI,.12
C FIGURE OUT WHICH ELEMENT IN ARRAY SHOULD BE SET TO *,
C SIN RETURNS -1 TO 1 WHICH IS CONVERTED TO -35 TO 35
C AND THEN OFFSET SO FINAL RANGE IS 1 TO 70
C
0012 WHERE=SIN(ANGLE)*35+35
C FIGURE OUT HOW MUCH TO BLANK IN THE OUTPUT ARRAY
C
0013 IBLANK=MAX0(35,WHERE)
C AND BLANK IT
C
0014 DO 15 I=1,IBLANK
0015 15 LINE(I)= ' ' 
C HMM... WHICH SIDE OF ZERO ARE WE ON?
IF (WHERE .GT. 35) THEN

RIGHT SIDE

DO 20 I=36,WHERE
LINE(I)='*'
ELSE

LEFT SIDE

DO 30 I=WHERE,35
LINE(I)='*'
ENDIF

SET "ZERO"
LINE(35)='+'
AND THE SIN VALUE
LINE(WHERE)='*'

IF THIS VALUE IS < 35, SET SO WE OUTPUT TO ZERO LINE

IF (WHERE .LE. 35)WHERE=35

AND FINALLY OUTPUT THE LINE

WRITE (6,21) (LINE(I),I=1,WHERE)
FORMAT (70A1)
CONTINUE
CALL EXIT
END

** Generated Code = 687 (Decimal), 02AF (Hex) Bytes
** Array Area = 420 (Decimal), 01A4 (Hex) Bytes

No Compile errors

NO ASSEMBLY ERRORS. 175 LABELS WERE DEFINED.
B>FRUN GRAPH
{execute the program}

GRAPH OF SIN

***+
************+
*************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
***************+
B>FORT LOAD.X  {compile, listing to console, .ASM and .OBJ to default drive}

NEVADA FORTRAN 3.0 (MOD 0)

***** NEVADA Fortran 3.0 (Mod 0) ** Compiling File: LOAD.FOR
*****

0001 C
0002 C "LOAD.FOR"
0003 C THIS ROUTINE DEMONSTRATED THE USE
0004 C OF THE 'LOAD' ROUTINE
0005 C TO LOAD AN ASSEMBLY LANGUAGE FILE INTO MEMORY AND
0006 C THEN CALL IT FROM FORTRAN
0007 C
0008 INTEGER A
0009 C FIND OUT WHICH ONE TO LOAD
0010 C
0011 TYPE 'Enter 0 to "LOAD" LD.HEX'  
0012 TYPE 'Enter 1 to "LOAD" LD.OBJ'  
0013 ACCEPT 'Which one: ',LTYPE
0014 C
0015 IF (LTYPE .EQ. 0) THEN
0016 TYPE "LOAD"ing "LD.HEX"
0017 ELSE
0018 TYPE "LOAD"ing "LD.OBJ"
0019 ENDIF
0020 C
0021 MUST LOAD "LD.HEX" OR "LD.OBJ" INTO MEMORY
0022 BEFORE WE CAN CALL IT
0023 C
0024 CALL LOAD ('LD',LTYPE,IER)
0025 C
0026 TYPE 'ERROR FOR LOAD=',IER
0027 C
0028 CHECK THE RETURNED ERROR CODE FROM LOAD
0029 C
0030 IF (IER .NE. 0) STOP 'LOAD ERROR'
0031 C
0032 "CALL" THE ROUTINE
0033 C
0034 A=CALL ($8000,1)
0035 C
0036 RESULT SHOULD BE 2
0037 C
0038 TYPE 'THE RESULT OF THE ASSEMBLY ROUTINE ',A
0039 C
0040 CALL EXIT
0041 C
0042 END

** Generated Code = 405 (Decimal), 0195 (Hex) Bytes

NO ASSEMBLY ERRORS. 135 LABELS WERE DEFINED.
B>FRUN LOAD  {execute the program}

Enter 0 to "LOAD" LD.HEX
Enter 1 to "LOAD" LD.OBJ
Which one: 0
"LOAD"ing "LD.HEX"
ERROR FOR LOAD= 0
THE RESULT OF THE ASSEMBLY ROUTINE IS: 2
B> FORT TRACE.ZCC {compile with no listing, .ASM and .OBJ to drive C}

B> FRUN TRACE {execute the program}

STARTING EXECUTION

#: 4
1
2
3
4

#: 140
Pgm is executing line 0009 in routine MAIN
Pgm is executing line 0010 in routine MAIN
1
Pgm is executing line 0010 in routine MAIN
Pgm is executing line 0014 in routine MAIN
ERROR TRAPPED, IER= 23
STOP END IN - MAIN
B> FORT SORT.BBB  {compile with listing, .ASM and .OBJ files to drive B}  

B> FRUN SORT  {execute the program}  

Shell sort  

How many numbers (2-2000) 100  
Starting sort  
D= 50  
D= 25  
D= 13  
D= 7  
D= 4  
D= 2  
D= 1  
All done  

2 6 6 6 7 8  
8 9 9 12 12 14  
14 18 19 21 21 21  
21 22 25 27 28 28  
29 29 30 32 33 33  
34 34 37 38 39 40  
40 41 46 46 48 48  
48 49 50 51 52 54  
58 58 59 60 60 61  
62 62 64 64 65 65  
66 67 68 70 70 71  
73 74 75 76 76 77  
80 80 82 82 83 84  
85 86 88 89 89 91  
91 91 93 93 93 93  
94 96 96 96 97 98  
99 99 100 100  

How many numbers (2-2000) 0  
STOP  
B>
L. SUGGESTED FURTHER READING

SOFTWARE TOOLS
Brian W. Kernigham and P. J. Plauger
Addison-Wesley Publishing Co. 1976.

A GUIDE TO FORTRAN PROGRAMMING
Daniel D. McCracken

FORTRAN IV WITH WATFOR AND WATFIV
Cress, Dirksen, and Graham

FORTRAN IV
Elliot I. Organick and Loren P. Meissner
Addison-Wesley Publishing Co. 1966.

PROGRAMMING PROVERBS FOR FORTRAN PROGRAMMERS
Henry F. Ledgard
Hayden, 1975.

ELLIS C., NEVADA COBOL Application Package Book1, ELLIS COMPUTING, INC., 1980.


ELLIS COMPUTING, NEVADA BASIC, ELLIS COMPUTING, INC., 1983.

Ian D. Kettleborough, FORTRAN SELF-TEACHING COURSE,
ELLIS COMPUTING, INC., 1983.

SOFTWARE LICENSE AGREEMENT

IMPORTANT: All Ellis Computing, Inc. programs are sold only on the condition that the purchaser agrees to the following license.

Ellis Computing, Inc. agrees to grant to the Customer, and the Customer agrees to accept on the following terms and conditions nontransferable and nonexclusive licenses to use the software program(s) (Licensed Programs) herein delivered with this Agreement.

TERM:

This Agreement is effective from the date of receipt of the above-referenced program(s) and shall remain in force until terminated by the Customer upon one month's prior written notice, or by Ellis Computing, Inc. as provided below.

Any License under this agreement may be discontinued by the Customer at any time upon one month's prior written notice. Ellis Computing, Inc. may discontinue any License or terminate this Agreement if the Customer fails to comply with any of terms and conditions of the Agreement.

LICENSE:

Each program License granted under this Agreement authorizes the Customer to use the Licensed Program in any machine readable form on any single computer system (referred to as System). A separate license is require for each System on which the Licensed Program will be used.

This Agreement and any of the Licenses, program or materials to which it applies may not be assigned, sublicensed or otherwise transferred by the Customer without prior written consent from Ellis Computing, Inc. No right to print or copy, in whole or part, the Licensed Programs is granted except as hereinafter expressly provided.

PERMISSION TO COPY OR MODIFY PROGRAMS:

The Customer shall not copy, in whole or part, any Licensed Programs which are provided by Ellis Computing, Inc. in printed form under this Agreement. Additional copies of printed materials may be acquired from Ellis Computing, Inc.

The NEVADA FORTRAN COMPILER Licensed Programs which are provided by Ellis Computing, Inc. in machine readable form may be copied, in whole or in part, in machine readable form in sufficient number for use by the Customer with the designated System, for back-up purposes, or for archive
purposes. The original, and any copied of the Licensed Programs, in whole or in part, which are made by the Customer shall be the property of Ellis Computing, Inc. This does not imply that Ellis Computing, Inc. owns the media on which the Licensed Programs are recorded.

The NEVADA FORTRAN Licensed Program call "FRUN" which is provided by Ellis Computing, Inc. may be distributed to third parties.

The Customer agrees to reproduce and include the copyright notice of Ellis Computing, Inc. and Ian D. Kettleborough on all copies, in whole or in part, in any form, including partial copies of modification, of Licensed program made hereunder.

PROTECTION AND SECURITY:

The Customer agrees not to provide or otherwise make available the NEVADA FORTRAN COMPILER Program including but not limited to program listings, object code, and source code, in any form, to any person other than Customer, Ellis Computing, Inc. employees or Ian D. Kettleborough, except with the Customer's permission for purposes specifically related to the Customer's use of the Licensed Program.

DISCLAIMER OF WARRANTY:

Ellis Computing, Inc. makes no warranties with respect to the Licensed Programs.

LIMITATION OF LIABILITY:

THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTED OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL ELLIS COMPUTING, INC. BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF ELLIS COMPUTING, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

GENERAL:

If any of the provision, or portions thereof, of the Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted. This is the complete and exclusive statement of the Agreement between the parties which supercedes all proposals, oral or written, and all other communications between the parties relating to the subject matter of the Agreement. This Agreement will be governed by the laws of the State of California.
\#
\#, 11, 21

\$
\$, 11

&
&, 11, 68, 89

*
*, 33, 74, 80
***, 33

+
+, 33

-
-, 33

..AND., 33, 37, 38
.COM, 9
.EQ., 33, 37
.FALSE., 21, 22, 46, 83
.GE., 33, 37
.GT., 33, 37
.HEX, 106
.LE., 33, 37
.LT., 33, 37
.NE., 33, 37
.NOT., 33, 38
.OBJ, 106
.OR., 33, 37, 38
.TRUE., 21, 22, 46, 83
.XOR., 33, 37, 38

/
/, 33
/-Type, 82

1
1, 6

2
2, 6

A
A-Type, 81
A=n, 18
ABS, 120
ACCEPT, 70, 116
Addition, 33
ALOG, 120
ALOG10, 120
AMAX0, 120
AMAX1, 120
AMIN0, 120
AMOD, 120
Ampersand, 10, 68, 89
ARG CNT, 124
Arithmetic IF, 40, 45, 117
Arithmetic Operators, 33
Array, 30, 59, 71, 72, 74, 78, 133
ASN GOTO, 127
ASSIGN, 43, 44, 116
Assigned GO TO, 40, 43, 44
ASSIGNED GOTO, 117
ASSM.COM, 6
Asterisk, 10, 74, 80
ATAN, 120
ATAN2, 120

B
B, 19
B=XXXX, 7
Backslash, 10, 11, 22, 89, 134
BACKSPACE, 94, 116
Binary I/O, 70, 92
BIT, 112, 120, 121
Blank COMMON, 8, 31
Blank padding, 7
BLOCK DATA, 69, 116

C
C=XXXX, 7
CALL, 40, 66, 67, 68, 113, 116, 120
CALL POP, 125
CALL PSH, 125
CBTOF, 114, 120, 121
CHAIN, 8, 106, 121
CHAIN FL, 125
CHAR, 113, 120
CIN, 53, 108, 121
CLOSE, 95, 104, 111, 121
Colon, 14
COM GOTO, 124
Comma, 10
Comment, 12, 13
COMMON, 30, 31, 60, 69, 116
COMP, 115, 120
Comparison Operators, 33
Computed GO TO, 40, 42
COMPUTED GOTO, 117
CON BIN, 126
CON:, 100
CONFIG, 4, 110, 134, 135
CONFIG.COM, 2
Constant, 21
Continuation, 11
CONTINUE, 40, 41, 50, 116
CONTRL/C, 127
Control-H, 97
Control-X, 97
Control-Z, 97
CONTROL/C, 53
CONVERT, 124
COPY, 16, 116
COS, 120
CTEST, 109, 121
CTRL DISABLE, 53, 116
CTRL ENABLE, 53, 116

D
D-Type, 86
D=n, 17
DATA, 29, 69, 116
Decimal point, 10
DECODE, 77, 78
DELAY, 108, 121
DELETE, 96, 104, 121
DIM, 120
DIMENSION, 60, 116
Disk is full!, 3
DIV ZERO, 124
Division, 33
DO, 40, 46, 49, 50, 117
Dollar, 10
DOUBLE PRECISION, 23, 28, 32, 60, 65, 117
DSK FULL, 125
DUMP, 55, 117

E
E, 18
E-Type, 86
ELSE, 47
ENCODE, 77, 79
END, 29, 46, 58, 117
END=, 74, 76
ENDFILE, 92, 95, 96, 117
ENDIF, 47
Equal sign, 10
ERR=, 74
ERRCLR, 51, 52, 117
ERRORS, 2, 4
ERRSET, 51, 52, 53, 55, 117
EXIT, 57, 107, 121
EXP, 120
Explicit RETURN, 40
Exponentiation, 30
EXPRESSION, 62
Expressions, 34

F
F-Type, 85
FILE OPR, 127
FLOAT, 120
FORMAT, 12, 41, 70, 71, 74, 78, 79, 80, 88, 117, 126
Formatted I/O, 70
FORT.COM, 2
FORT.ERR, 4, 6
Free format, 74, 78
Free Format I/O, 70
FREE format input, 90
FREE format output, 90
FRUN, 8
FRUN.COM, 2, 4
FUNCTION, 17, 30, 31, 60, 63, 65, 67, 117
FUNCTION statement, 32

G
G, 18
G-Type, 87
GO TO, 50, 117

H
H, 7
Hexadecimal, 11, 21, 22, 84, 89, 108, 133

I
I-Type, 81, 85
I/O ERR, 126
I/O LIST, 126
I=n, 18
IABS, 120
IDIM, 120
IF-THEN-ELSE, 40, 47, 118
IFIX, 120
ILL CHAR, 126
ILL UNIT, 125
IMPLICIT, 24, 32, 118
Implied DO loop, 72
INP, 113, 120
INPT ERR, 127
INT RANG, 124
INTEGER, 23, 25, 32, 35, 37, 49, 60, 65, 118
ISIGN, 120

K
K-Type, 84

L
L-Type, 83
L=, 17
Library functions, 63
LINE LEN, 126
LOAD, 106, 121
LOG NEG, 125
LOGICAL, 23, 26, 32, 60, 65, 118
Logical constants, 22
Logical IF, 40, 46, 117
Logical Operators, 33
Logical unit, 70, 96
LOPEN, 74, 91, 96, 102, 104, 121
Lowercase, 12
LST:, 100

M
M=XXXX, 7
Main, 30, 32
MAX0, 120
MAX1, 120
MIN0, 120
MIN1, 120
Minus sign, 10
Mixed mode expressions, 39
MOD, 120
MOVE, 107, 121
Multiple RETURN, 11, 40, 68, 118
Multiplication, 33

N
N, 6, 19
Normal return, 67
North Star floating point board, 135
NOT, 32
Number sign, 10
Numerical constants, 21

O
O=n, 18
OPEN, 74, 91, 96, 100, 104, 121
Operator hierarchy, 33
OPTIONS, 17
OUT, 110, 122
OVERFLOW, 124

P
P, 6
P=n, 18
Parenthesised expressions, 34
PAUSE, 40, 56, 118
PEEK, 114, 120
Plus sign, 10
POKE, 111, 122
PUT, 112, 122

Q
Q, 19
Q option, 51

R
RAND, 120
READ, 53, 71, 72, 74, 78, 92, 118
REAL, 23, 27, 32, 35, 37, 60, 65, 118
RENAME, 105, 122
RESET, 111, 122
RETURN, 67, 118
REWIND, 93, 96, 118
Runtime error, 35, 42, 51, 53, 55, 66, 89, 105, 106, 123
RUNTIME FORMAT, 71

S
S=, 17
SEEK, 96, 105, 122
SEEK ERR, 127
Semicolon, 14
SETIO, 110, 122
SIGN, 120
SIN, 120
Slash, 10
SQRT, 120
SQRT NEG, 125
Statement label, 11, 13
STOP, 40, 57, 107, 118
String constant, 22
Strings, 89
Subprograms, 63
SUBROUTINE, 11, 17, 30, 31, 32, 60, 63, 64, 66, 67, 68, 118
Subscripts, 59, 62
Subtraction, 33

T
T-Type, 83
T=n, 17
TAN, 120
THEN, 47
TRACE OFF, 54, 118
TRACE ON, 54, 119
TYPE, 70, 119

U
Unary minus, 33
Unconditional GO TO, 40, 41
UNIT CLO, 126
UNIT OPN, 125

W
WRITE, 53, 55, 71, 72, 76, 79, 92, 119

X
X, 19
X-Type, 81

Z
Z-Type, 82
\, 11, 80, 89
^, 33
COPYRIGHT

Copyright, 1982 by Ellis Computing, Inc. All rights reserved worldwide. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system or translated into any human or computer language in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the express written permission of Ellis Computing, Inc.

TRADEMARKS

NEVADA COBOL(tm), NEVADA FORTRAN(tm), NEVADA PILOT(tm), NEVADA EDIT(tm), NEVADA ASSEMBLER(tm) and Ellis Computing(tm) are trademarks of Ellis Computing, Inc. CP/M is a registered trademark of Digital Research, Inc.

DISCLAIMER

All Ellis Computing, Inc. computer programs are distributed on an "AS IS" basis without warranty.

Ellis Computing makes no warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event will Ellis Computing be liable for consequential damages even if Ellis Computing has been advised of the possibility of such damages.

Printed in the U.S.A.
NEVADA ASSEMBLER
AN ASSEMBLER FOR CP/M

TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>4</td>
</tr>
<tr>
<td>2 OPERATING PROCEDURES</td>
<td>5</td>
</tr>
<tr>
<td>HARDWARE REQUIREMENTS</td>
<td>5</td>
</tr>
<tr>
<td>SOFTWARE REQUIREMENTS</td>
<td>5</td>
</tr>
<tr>
<td>FILES ON DISTRIBUTION DISKETTE</td>
<td>5</td>
</tr>
<tr>
<td>FILE TYPE CONVENTIONS</td>
<td>5</td>
</tr>
<tr>
<td>GETTING STARTED</td>
<td>5</td>
</tr>
<tr>
<td>EXECUTING THE ASSEMBLER</td>
<td>6</td>
</tr>
<tr>
<td>STARTUP</td>
<td>8</td>
</tr>
<tr>
<td>EXECUTING THE .OBJ FILE</td>
<td>9</td>
</tr>
<tr>
<td>MEMORY USAGE</td>
<td>9</td>
</tr>
<tr>
<td>TERMINATION</td>
<td>10</td>
</tr>
<tr>
<td>3 STATEMENTS</td>
<td>11</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>LINE NUMBERS</td>
<td>11</td>
</tr>
<tr>
<td>LABEL FIELD</td>
<td>12</td>
</tr>
<tr>
<td>OPERATION FIELD</td>
<td>12</td>
</tr>
<tr>
<td>OPERAND FIELD</td>
<td>13</td>
</tr>
<tr>
<td>Register Names</td>
<td>13</td>
</tr>
<tr>
<td>Labels</td>
<td>13</td>
</tr>
<tr>
<td>Constants</td>
<td>14</td>
</tr>
<tr>
<td>Expressions</td>
<td>15</td>
</tr>
<tr>
<td>High and Low Order byte extraction</td>
<td>15</td>
</tr>
<tr>
<td>COMMENT FIELD</td>
<td>16</td>
</tr>
<tr>
<td>4 PSEUDO-OPERATIONS</td>
<td>17</td>
</tr>
<tr>
<td>5 ERROR CODES AND MESSAGES</td>
<td>23</td>
</tr>
<tr>
<td>6 APPENDICES</td>
<td>25</td>
</tr>
<tr>
<td>8080 OPERATION CODES</td>
<td>25</td>
</tr>
<tr>
<td>TABLE OF ASCII</td>
<td>27</td>
</tr>
<tr>
<td>SAMPLEx ASSEMBLER LISTING</td>
<td>29</td>
</tr>
<tr>
<td>SAMPLE PROGRAM</td>
<td>30</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>40</td>
</tr>
<tr>
<td>SOFTWARE LICENSE</td>
<td>41</td>
</tr>
<tr>
<td>CORRECTIONS AND SUGGESTIONS</td>
<td>43</td>
</tr>
<tr>
<td>INDEX</td>
<td>44</td>
</tr>
</tbody>
</table>
SECTION 1

INTRODUCTION

The assembler translates a symbolic 8080 assembly language program "source code" into the binary instructions "object code" required by the computer to execute the program.

The assembler operates on standard CP/M text files. Each line of a normal text file consists of the characters of that line followed by a carriage return (0DH) and a line feed (0AH).

When the assembler is invoked, it is loaded into memory starting at location 100H. It processes the source code file in two passes. On the first pass, it builds a symbol table containing all of the labels defined in the source program. The symbol table begins at the memory location immediately following the assembler; each entry in the table is 7 bytes long. Certain errors may be detected during the first pass, causing error messages to be output to an error file (usually the console). On the second pass, the object code is generated and usually output to an object code file. In addition, a formatted listing of both source and object code may be output to a listing file, and the symbol table may be output to a file. Any errors detected during this pass cause messages to be output to the error file.

To abort the assembly process at any time, press the Control and C keys.

After the assembly runs to completion and no errors are detected, the resulting object code file (type .OBJ) can be executed by typing RUNA and its name.

EXAMPLE:

   RUNA PROG  loads and executes a file called PROG.
SECTION 2
OPERATING PROCEDURES

HARDWARE REQUIREMENTS

1. 8080/Z80/8085 microprocessor.
   Z80 is a trademark of Zilog.
2. A minimum of 32K RAM.
3. Any disk drive.
4. CRT or Video display and keyboard.

SOFTWARE REQUIREMENTS

Digital Research, Inc.s. CP/M(R) operating system
Version 1.4 or 2.2.
CP/M is a registered trademark of Digital Research, Inc.

FILES ON THE DISTRIBUTION DISKETTE

ASSM.COM is the assembler program.
RUNA.COM is the runtime loader.
(note: These files are on the Nevada FORTRAN diskette.)

FILE TYPE CONVENTIONS

Assembly source code files .ASM
COBOL source code files .CBL
FORTRAN source code files .FOR
Object code run time files .OBJ
Printer listing files .PRN
Symbol table listing files .SYM
Error files .ERR
Work files .WRK

GETTING STARTED

If the master disk is not write protected, do it now!

1. NEVADA ASSEMBLER is distributed on a DATA DISK without
   the CP/M operating system. There is no information on the
   system tracks, so don't try to "boot it up", it won't work!
2. On computer systems with the ability to read several
   disk formats, such as the KayPro computer, the master
   diskette must be used in disk drive B.
3. Do not try to copy the master diskette with a COPY program! On most systems it won't work. You must use the CP/M PIP command to copy the files.

4. First, prepare a CP/M system's diskette for use as your NEVADA ASSEMBLER operations diskette. On 5 1/4 inch diskettes you may have to remove (use the CP/M ERA command) most of the files in order to make room for the ASSEMBLER files. None of the CP/M files are needed for NEVADA ASSEMBLER, but PIP.COM and STAT.COM are useful if you have the space. You may want to do a CP/M STAT command on the distribution disk so you will know how much space you need on your operational diskette. For more information read the CP/M manuals about the STAT command.

5. Then insert the newly created CP/M diskette in disk drive A, and insert the NEVADA ASSEMBLER diskette in drive B and type (ctl-c) to initialize CP/M. Now copy all the files from the ASSEMBLER diskette onto the CP/M diskette:

   PIP A:=B:*.*[VO]

If you get a BDOS WRITE ERROR message from CP/M during the PIP operation it usually means the disk is full and you should erase more files from the operational diskette.

At this point, put the NEVADA ASSEMBLER diskette in a safe place. You will not need it unless something happens to your operations diskette. By the way, back up your operations diskette with a copy each week! If your system malfunctions you can then pat yourself on the back for having a safe back up copy of your work.

Now, boot up the newly created NEVADA ASSEMBLER operations diskette. Notice that CP/M displays the amount of memory for which this version of CP/M has been specialized. The amount of memory available determines the size of the programs that can be assembled. The more memory available the larger the program that can be assembled.

EXECUTING THE ASSEMBLER

The assembler is invoked by a CP/M command with the following formats.

FORMAT-1:

   ASSM file<CR>
DESCRIPTION:

where:

file = [unit:]source-file-name
The name of the source code input file. This parameter must be present; all others are optional.

[] = optional parameters

unit: = disk drive unit letter. If this parameter is not included, the default drive is used.

u = the disk drive unit letter or the letter "X" for output to the console, or the letter "Z" for no output.

u for position one.
This single character code, if present, represents the drive onto which the listing file is to be written. If this argument is absent then the listing will be written on the default drive. Also, if the character is an X the listing will be sent to the console. If the character is a Z then no listing is produced.

u for position two.
The second letter of the file type represents the drive for the object (.OBJ) file. If this argument is absent then the file will be written on the default drive. If this character is a Z then no object code file will be produced.

u for position three.
The third letter of the file type represents the drive for the error (.ERR) file. If this argument is absent the console will be used to display the errors. This argument must be followed by a space or carriage return.

u for position one of the second set.
The first letter of the second set of arguments represents the drive for the symbol (.SYM) file. If this argument is absent no symbol table file will be produced.
Various assembler options may be controlled by following the $ with one or more of the following option specifiers. The list of options is terminated by a carriage return. For those options that may be preceded by + or -, the + is optional and will be assumed if absent.

+L
The source file has line numbers in column 1-4 of each line.

-L
The source file has no line numbers.

If neither of these is specified, the assembler will examine the first line to determine if the file has line numbers.

#
Instructs the assembler to generate its own line numbers in the listing in place of those in the source file (if any).

P
Instructs the assembler to paginate output to the listing file. The file name of the source code file will be printed on the top left-hand corner of each page. A page number will be printed on the top right-hand corner of each page. If a TITL pseudo-operation occurs in the source code, a one- or two-line title will be centered at the top of each page.

0, 1, 2, or 3
Specifies the spacing on the listing:

0 = no additional spacing
1 = 72 column output
2 = 80 column output (default)
3 = 132 column output

S
Specifies output symbol table in format for SID.

EXAMPLES:

ASSM TST
ASSM TST.AAX A$-L#P0

STARTUP

To assemble your program, type ASSM and the source file name. The first thing that happens is the copyright message is displayed on the screen and the disk drive(s) begin working. When the assembly process is complete, a message
will be displayed and machine control will return to the operating system.

A>ASSM source-file<CR>

EXECUTING THE .OBJ FILE

To execute the program, type RUNA and the file-name. The assembly process creates a file with the extension type of (.OBJ). This object program file will be loaded into memory and executed.

A>RUNA file-name

There are several options that also can be specified with the RUNA command.

RUNA file-name[.ZLC]

Z = zero memory before loading the .OBJ file.

L = load the program but don't execute it. Control returns to CP/M.

C = create a .COM file for later execution. Control returns to CP/M. Remember .COM files always begin execution at location 100H.

Example:

A>RUNA PROG.ZC this will zero memory and create a file named PROG.COM.
A>RUNA PROG.L this will load PROG but not execute it.

NOTE: These object code files (.OBJ), if properly orged, can also be loaded and executed by the NEVADA COBOL and NEVADA FORTRAN run time packages. However, Nevada COBOL and Nevada FORTRAN generated type (.OBJ) files cannot be converted to (.COM) files by RUNA because of runtime package requirements. Please see the Nevada FORTRAN manual for the procedure to generate a FORTRAN (.COM) file.

MEMORY USAGE

The ASSEMBLER program is read into memory starting at location 100H and uses all memory available up to the bottom of CP/M.
The runtime package RUNA loads into memory at location 100H and relocates itself to just below CP/M and then begins loading your program.

**TERMINATION**

The normal termination of the assembly is signaled by the display of the following messages and return to CP/M.

NO ASSEMBLY ERRORS.  
4 LABELS WERE DEFINED.  
A>

The assembly process can be interrupted at any time by pressing the Control and C keys.
INTRODUCTION

An assembly language program (source code) is a series of statements specifying the sequence of machine operations to be performed by the program.

Each statement resides on a single line and may contain up to four fields as well as an optional line number. These fields, label, operation, operand, and comment, are scanned from left to right by the assembler, and are separated by spaces. The assembler can handle lines up to 80 characters in length.

LINE NUMBERS

Line numbers in the range 0000-9999 may appear in columns 1-4. Line numbers need not be ordered and have no meaning to the assembler, except that they appear in listings. Line numbers may also make it easier to locate lines in the source code file when it is being edited. The disk and memory space required for normal text files will be increased by five bytes per line if line numbers are used; this may become significant for large files.

If line numbers are not used, the label field starts in column 1 and the operation field may not start before column 2. If line numbers are used, they must be followed by at least one space, so the label field starts in column 6 and the operand may not start before column 7.

Once the starting column for the label has been established, the same format must be followed throughout the file: either all of the lines or none of the lines can have line numbers. Any other file(s) assembled along with the main file (using COPY pseudo-operation) must conform to the format of the main file.

Example of source statements with line numbers:

Column
1234567
0001 LABEL ORA A  label field must start at column 6.
0002 JNZ NEXT   operation field starts at column 7
0003 ;               (minimum).
0004 LOOP MOV A,B  operation field starts one space after
0005 *              label.
Example of source statements without line numbers:

Column
1234567
LABEL ORA A label field must start at column 1.
   JNZ NEXT operation field starts at column 2 (minimum).
LOOP MOV A,B operation field starts one space after label.

LABEL FIELD

The label field must start in column 1 of the line (column 6 if line numbers are used). A label gives the line a symbolic name that can be referenced by any statement in the program. Labels must start with an alphabetic character (A-Z,a-z), and may consist of any number of characters, though the assembler will ignore all characters beyond the sixth; e.g. the labels BRIDGE, BRIDGE2 and BRIDGET cannot be distinguished by the assembler. A duplicate label error will occur if any two labels in a program begin with the same six letters.

A label may be separated from the operations field by a colon (:) instead of, or in addition to, a blank.

The labels A, B, C, D, E, H, L, M, PSW, and SP are pre-defined by the assembler to serve as symbolic names for the 8080 registers. They must not appear in the label field.

An asterisk (*) or semi-colon (;) in place of a label in column 1 (column 6 if line numbers are used) will designate the entire line as a comment line.

OPERATION FIELD

The operation field contains either 8080 instruction mnemonics or assembler pseudo-operation mnemonics. Appendix 1 summarizes the standard instruction mnemonics recognized by the assembler, and Appendix 4 lists several references to consult if more information on the 8080 machine instructions is needed. Assembler pseudo-operations are directives that control various aspects of the assembly process, such as storage allocation, conditional assembly, file inclusion, and listing control.

An operation mnemonic may not start before column 2 (column 7 if line numbers are used) and must be separated from a label by at least one space (or a colon).
OPERAND FIELD

Most machine instructions and pseudo-operations require one or two operands, either register names, labels, constants, or arithmetic expressions involving labels and constants.

The operands must be separated from the operator by at least one space. If two operands are required, they must be separated by a comma. No spaces may occur within the operand field, since the first space following the operands delimits the comments field.

Register Names

Many 8080 machine instructions require one or two registers or a register pair to be designated in the operand field. The symbolic names for the general-purpose registers are A, B, C, D, E, H and L. SP stands for the stack pointer, while M refers to the memory location whose address is in the HL register pair. The register pairs BC, DE, and HL are designated by the symbolic names B, D, and H, respectively. The A register and condition flags, when operated upon as a register pair, are given the symbolic name PSW.

The values assigned to be register names A, B, C, D, E, H, L, M, PSW and SP are 7, 0, 1, 2, 3, 4, 5, 6, 6, and 6, respectively. These constants, or any label or expression whose value lies in the range 0 to 7, may be used in place of the pre-defined symbolic register names where a register name is required; such a substitution of a value for the pre-defined register name is not recommended, however.

Labels

Any label that is defined elsewhere in the program may be used as an operand. If a label is used where an 8-bit quantity is required (e.g., MVI C, LABEL), its value must lie in the range -256 to 255, or it will be flagged as a value error.

If a label is used as a register name, its value must lie in the range 0 to 7, or be 0, 2, 4, or 6 if it designates a register pair. Otherwise, it will be flagged as a register error.

During each pass, the assembler maintains an instruction location counter that keeps track of the next location at which an instruction may be stored; this is analogous to the program counter used by the processor during program execution to keep track of the location of the next instruction to be fetched.

The special label $ (dollar Sign) stands for the current
value of the assembler's instruction location counter. When $ appears within the operand field of a machine instruction, its value is the address of the first byte of the next instruction.

EXAMPLE:

FIRST EQU $  The label FIRST is set to the address
TABLE DB ENTRY of the entry in a table and LAST
*  points to the location immediately after
*  the end of the table. TABLN is then
*  the length of the table and will remain
LAST EQU $  correct, even if later additions or
TABLN EQU LAST-FIRST deletions are made in the table.

CONSTANTS

Decimal, hexadecimal, octal, binary and ASCII constants may be used as operands.

The base for numeric constants is indicated by a single letter immediately following the number, as follows:

D = decimal
H = hexadecimal
O = octal
Q = octal
B = binary

If the letter is omitted, the number is assumed to be decimal. Q is usually preferred for octal constants, since O is so easily confused with 0 (zero). Numeric constants must begin with a numeric character (0-9) so that they can be distinguished from labels; a hexadecimal constant beginning with A-F must be preceded by a zero.

ASCII constants are one or two characters surrounded by single quotes ('). A single quote within an ASCII constant is represented by two single quotes in a row with no intervening spaces. For example, the expression '''', where the two outer quote marks represent the string itself, i.e., the single quote character. A single character ASCII constant has the numerical value of the corresponding ASCII code. A double character ASCII constant has the 16-bit value whose high-order byte is the ASCII code of the first character and whose low-order byte is the ASCII code of the second character.

If a constant is used where an 8-bit quantity is required (e.g., MVI C,10H), its numeric value must lie in the range -256 to 255 or it will be flagged as a value error.

If a constant is used as a register name, its numeric value
must lie in the range 0 to 7, or be 0, 2, 4, or 6 if it
designates a register pair. Otherwise it will be flagged as
a register error.

Examples:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI A,128</td>
<td>Move 128 decimal to register A.</td>
</tr>
<tr>
<td>MVI C,10D</td>
<td>Move 10 decimal to register C.</td>
</tr>
<tr>
<td>LXI H,2FH</td>
<td>Move 2F hexadecimal to registers HL.</td>
</tr>
<tr>
<td>MVI B,303Q</td>
<td>Move 303 octal to register B.</td>
</tr>
<tr>
<td>MVI A,'Y'</td>
<td>Move the ASCII value for Y to reg A.</td>
</tr>
<tr>
<td>MVI A,101B</td>
<td>Move 101 binary to register A.</td>
</tr>
<tr>
<td>JMP OFFH</td>
<td>Jump to address FF hexadecimal.</td>
</tr>
</tbody>
</table>

**EXPRESSIONS**

Operands may be arithmetic expressions constructed from
labels, constants, and the following operators:

- `+` addition or unary plus
- `-` subtraction or unary minus
- `*` multiplication
- `/` division (remainder discarded)

Values are treated as 16-bit unsigned 2's complement
numbers. Positive or negative overflow is allowed during
expression evaluation, e.g., 32767+1=7FFFH+1=-32768 and
-32768-1=7FFFH=32767. Expressions are evaluated from left
to right; there is no operator precedence.

If an expression is used where an 8-bit quantity is required
(e.g., MVI C,TEMP+10H), it must evaluate to a value in the
range -256 to 255, or it will be flagged as a value error.

Examples:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI</td>
<td>A,255D/10H-5</td>
</tr>
<tr>
<td>LDA</td>
<td>POTTS/256*OFFSET</td>
</tr>
<tr>
<td>LXI</td>
<td>SP,30*2+STACK</td>
</tr>
</tbody>
</table>

**High- and Low-order Byte Extraction**

If an operand is preceded by the symbol `<`, the high-order
byte of the evaluated expression will be used as the value
of the operand. If an operand is preceded by the symbol `>`,
the low-order byte will be used.

Note that the symbols `<` and `>` are not operators that may be
applied to labels or constants within an expression. If
more than one `<` or `>` appears within an expression, the
rightmost will be used to determine whether to use the high-
or low-order byte of the evaluated expression as the value of the operand. That is, the rightmost < or > is treated as if it preceded the entire expression, and the others will be totally ignored.

Examples:

MVI A,>TEST  Loads register A with the least significant 8 bits of the value of the label TEST.
*               *
MVI B,<0CC00H  Loads register B with the most significant byte of the 16-bit value CC00H, i.e., CCH.
*               *
MVI C,<1234H  Loads register C with the value 12H.
MVI C,>1234H  Loads register C with the value 34H.

COMMENT FIELD

The comment field must be separated from the operand field (or operation field for instructions or pseudo-operations that require no operand) by at least one space. Comments are not processed by the assembler, but are solely for the benefit of the programmer. Good comments are essential if a program is to be understood very long after it is written or is to be maintained by someone other than its author.

An entire line will be treated as a comment if it starts with an asterisk (*) or semicolon (;) in column 1 (column 6 if line numbers are used).

Examples:

0001 ; is input ready?
0002 LOOP IN STAT input device status
0003 ANI 1 test status bit
0004 JZ LOOP wait for data
0005 *data is now available

If listing file formatting is specified in the ASM command ($=options contains 1, 2, or 3), the comment field must be preceded by at least two spaces to ensure proper output formatting. Furthermore, instructions and pseudo-operations requiring no operand must be followed by a dummy operand (a period is recommended).

Examples:

MVI A,10 comments
RZ . comments
SECTION 4

PSEUDO-OPERATIONS

Pseudo-operations appear in a source program as instructions to the assembler and do not always generate object code. This section describes the pseudo-operations recognized by the NEVADA assembler.

In the following pseudo-operation formats, <expression> stands for a constant, label, arithmetic expression constructed from constants and labels. Optional elements are enclosed in square brackets [].

Equate

EQU <expression>

This pseudo-operation sets a label name to the 16-bit value that is represented in the operand field. That value holds for the entire assembly and may not be changed by another EQU.

Any label that appears in the operand field of an EQU statement must be defined in a statement earlier in the program.

Examples:

BELL EQU 7 The value of the label BELL is set to 7.
BELL2 EQU BELL*2 The label BELL2 is set to 7*2.

Set Origin

ORG <expression>

This pseudo-operation sets the assembler's instruction location counter to the 16-bit value specified in the operand field. In other words, the object code generated by the statements that follow must be loaded beginning at the specified address in order to execute properly. The label, if present, is given the specified 16-bit value.

Any label that appears in the operand field of an ORG statement must be defined in a statement earlier in the program.

If no origin is specified at the beginning of the source code, the assembler will set the origin to 100H. If no ORG pseudo-operation is used anywhere in the source program,
successive bytes of object code will be stored at successive memory locations.

Examples:

\[
\begin{align*}
&\text{ORG 4000H} & \text{Determines that the object code generated} \\
&\text{START ORG 100H} & \text{by subsequent statements must be loaded} \\
&\text{in locations beginning at 4000H.} \\
&\text{in locations beginning at 100H.}
\end{align*}
\]

**Set Execution Address**  \(\text{XEQ} \ <\text{expression}>\)

This pseudo-operation specifies the entry point address for the program, i.e., the address at which it is to begin execution. If a program contains no \(\text{XEQ}\) pseudo-operation, the object code file will contain a starting address of 100H. If more than one \(\text{XEQ}\) appears in a program, the last will be used.

An example of the difference between ORG and XEQ is that a program whose first 100 bytes are occupied by data will have an ORG address 100 bytes lower in memory than its XEQ address.

Example:

\[
\begin{align*}
&\text{XEQ 100H} & \text{The entry point address for the assembled program is set to 100H.}
\end{align*}
\]

**Define Storage**  \([\langle\text{label}\rangle]\) \(\text{DS} \ <\text{expression}>\)  
\([\langle\text{label}\rangle]\) \(\text{RES} \ <\text{expression}>\)

Either of these pseudo-operations reserves the specified number of successive memory locations starting at the current address within the program. The contents of these locations are not defined and are not initialized at load time.

Any label that appears in the operand field of a DS or RES statement must be defined in a statement earlier in the program.

Examples:

\[
\begin{align*}
&\text{SPEED DS 1} & \text{Reserves one byte.} \\
&\text{DS 400} & \text{Reserves 400 bytes.} \\
&\text{RES 177Q} & \text{Reserves 177 (octal) bytes.}
\end{align*}
\]
Define byte  \[<\text{label}>\] DB <expression>[,<expression>,...]

This pseudo-operation sets a memory location to an 8-bit value. If the operand field contains multiple expressions separated by commas, the expressions will define successive bytes of memory beginning at the current address. Each expression must evaluate to a number that can be represented in 8 bits.

Examples:

DB 1   one byte is defined.
DB OFFH,303Q,100D,11010011B,3*BEL,-10  multiple bytes.
TABLE DB 'A','B','C','D',0   multiple bytes are defined.

Define Word  \[<\text{label}>\] DW <expression>

This pseudo-operation sets two memory locations to a 16-bit quantity. The least significant (low-order) byte of the value is stored at the current address and the most significant byte (high-order) is stored at the current address + 1.

Examples:

SAVE DW 1234H   1234H is stored in memory, 34H in the low-order byte and 12H in the high-order byte.
YES DW 'OK'  The ASCII value for the letters 'O' and 'K' is stored with the 'K' at the lower memory address.

Define Double Byte  \[<\text{label}>\] DDB <expression>

This pseudo-operation is almost the same as DW, except that the two bytes are stored in the opposite order: high-order byte first, followed by the low-order byte.

Example:

FIRST DDB 1234H   1234H is stored in memory, 12H in the low-order byte and 34H in the high-order byte.
Define ASCII String

\[
\text{[<label>] ASC } \#\langle\text{ASCII string}\rangle\#
\]

\[
\text{[<label>] ASCZ } \#\langle\text{ASCII string}\rangle\#
\]

The ASC pseudo-operation puts a string of characters into successive memory locations starting at the current location. The special symbols \# in the format are "delimiters"; they define the beginning and end of the ASCII character string. The assembler uses the first non-blank character found as the delimiter. The string immediately follows this delimiter, and ends at the next occurrence of the same delimiter, or at a carriage return.

The ASCZ pseudo-operation is the same except that it appends a NUL (00H) to the end of the stored string.

Examples:

WORDS ASC "THIS IS AN ASCII STRING"
ASCZ "THIS IS ANOTHER STRING"

Set ASCII List Flag

ASCF 0
ASCF 1

If the operand field contains a 0, the listing of the assembled bytes of an ASCII string will be suppressed after the first line (four bytes). Likewise, only the first four assembled bytes of a DB pseudo-operation with multiple arguments will be listed. If a program contains many long strings, its listing will be easier to read if the ASCF pseudo-operation is used.

If the operand field contains a 1, the assembled form of subsequent ASCII strings and DB pseudo-operations with multiple arguments will be listed in full. This is the default condition.

See Appendix 3 for an example of the listing format.

Conditional Assembly

\[
\text{IF } \langle\text{expression}\rangle
\]

\[
\text{source code}
\]

\[
\text{ENDF}
\]

The value of the expression in the operand field governs whether or not subsequent code up to the matching ENDF will be assembled. If the expression evaluates to a 0 (false), the code will not be assembled. If the expression evaluates to a non-zero value (true), the code will be assembled. Blocks of code delimited by IF and ENDF ("conditional code") may be nested within another block of conditional code.
Any label that appears in the operand field of an IF...ENDF pseudo-operation must be defined in a statement earlier in the program.

Example:

YES EQU 1  
NO EQU 0  
*  
IF YES  
MVI A,'Y'  
IF NO  
MVI A,'N'  
ENDF  
ENDF

Sets the value of the label 'YES' to 1.
Sets the value of the label 'NO' to 0.
the expression here is true (1), so the code on this line will be assembled.
The expression here is false (0), so the code on this line will not be assembled.
This terminates the NO conditional.
This terminates the YES conditional.

List Conditional code  IFLS

This pseudo-operation enables listing of conditional source code even though no object code is being generated because of a false IF condition. The assembler will not list such conditional source code if this pseudo-operation is not used.

Copy file  COPY [<unit:>]<file-name>

This pseudo-operation copies source code from a disk file into a program being assembled. The code from the copied file will be assembled starting at the current address. When the copied file is exhausted, the assembler will continue to assemble from the original file. The resulting object code will be exactly like what would be generated if the copied source code were part of the original file, but the COPY pseudo-operation does not actually alter any source file.

A copied file may not copy another file. And, all files that are accessed by the COPY pseudo-operation must be of the same format as the main source file, i.e., either having or not having line numbers. The files must be type (.ASM).

EXAMPLES:

COPY FILE1
COPY B:FILE2
Listing Control

NLST
LST

The NLST pseudo-operation suppresses all output to the listing file. Object code will still be output to the object code file and the lines containing errors will still be output to the error file. The LST pseudo-operation re-enables output to the listing file.

Listing Title

TITL <first line>"<second line>

If the P option is specified in the ASM command, the one- or two-line title specified by this pseudo-operation will be printed centered at the top of each page of the listing.

Page Eject

PAGE

If the P option is specified in the ASM command, this pseudo-operation causes a skip to the top of the next page of the listing.

End of Source file

END

This pseudo-operation terminates each pass of the assembly. Only one END statement should be in the file or files to be assembled, and it should be the last statement encountered by the assembler. Since an end-of-file on the source code input file will also terminate each pass, the END statement is unnecessary in most cases.
SECTION 5

ERROR CODES AND MESSAGES

ASSEMBLER COMMAND ERRORS

A number of console messages may be generated in response to errors in the ASM command. When an error of this sort occurs, the assembly is aborted and control returns to CP/M.

EXPECTED NAME The source code input file name is missing.

ILLEGAL OPTION An unrecognized option specifier follows $.

91 ERROR IN EXTENDING THE FILE
92 END OF DISK DATA - DISK IS FULL
93 FINE NOT OPEN
94 NO MORE DIRECTORY SPACE - DISK IS FULL
95 FILE CANNOT BE FOUND
96 FILE ALREADY OPEN
97 READING UNWRITTEN DATA

ASSEMBLY ERRORS

If a statement contains one of the following errors, there will be a single letter error code in column 19 of the line output to the listing and/or error files. An error detected during both the first and the second pass of the assembler will be flagged twice in the listing(s). If the error is not an opcode error, NULs will be output as the second and, if appropriate, third bytes of object code for that instruction. If the error is an opcode error, the instruction will be assumed to be a three-byte instruction, and three NULs will be written to the listing and/or error files. The error codes are:
ARGUMENT ERROR  An illegal label or constant appears in the operand field. This might be 1) a number with a letter in it, e.g., 2L, 2) a label that starts with a number, e.g., 3STOP, or 3) an improper representation of a string, e.g., ''A'' in the operand field of a statement containing the ASCII pseudo-operation.

DUPLICATE LABEL  The source code contains multiple labels whose first five characters are identical.

LABEL ERROR  The symbol in the label field contains illegal characters, e.g., it starts with a number.

MISSING LABEL  An EQU instruction does not have a symbol in the label field.

OPCODE ERROR  The symbol in the operation field is not a valid 8080 instruction mnemonic or an assembler pseudo-operation mnemonic.

REGISTER ERROR  An expression used as a register designator does not have a legal value.

SYNTAX ERROR  A statement is not in the format required by the assembler.

UNDEFINED SYMBOL  A label used in the operand field is not defined, i.e., does not appear in the label field anywhere in the program, or is not defined prior to its use as an operand in an EQU, ORG, DS, RES, or IF pseudo-operation.

VALUE ERROR  The value of the operand lies outside the allowed range.
### MOVE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Accumulator</th>
<th>Load</th>
<th>Stack</th>
<th>Immediate</th>
<th>Immediate*</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVI B C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI C</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI H</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI L</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVI A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### INCREMENT**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Decrement**</th>
</tr>
</thead>
<tbody>
<tr>
<td>INR B</td>
<td>DCR B</td>
</tr>
<tr>
<td>INR C</td>
<td>DCR C</td>
</tr>
<tr>
<td>INR D</td>
<td>DCR D</td>
</tr>
<tr>
<td>INR E</td>
<td>DCR E</td>
</tr>
<tr>
<td>INR H</td>
<td>DCR H</td>
</tr>
<tr>
<td>INR L</td>
<td>DCR L</td>
</tr>
<tr>
<td>INR M</td>
<td>DCR M</td>
</tr>
<tr>
<td>INR A</td>
<td>DCR A</td>
</tr>
<tr>
<td>INX B</td>
<td>DCX B</td>
</tr>
<tr>
<td>INX D</td>
<td>DCX D</td>
</tr>
<tr>
<td>INX H</td>
<td>DCX H</td>
</tr>
<tr>
<td>INX SP</td>
<td>DCX SP</td>
</tr>
</tbody>
</table>

---

**NOTE:**
- * = only carry affected
- ** = all flags except CARRY affected
- = all flags affected
- Address = 16 bit address
- Constant, or logical/arithmetic expression that evaluates to an 8 bit data quantity
- All flags (C Z S P) affected
- D16 = Constant, or logical/arithmetic expression that evaluates to a 16 bit data quantity

---

### Pseudocode

- **APPENDIX I.**
- **NEVADA ASSEMBLER PAGE 25**
- **6-6-83**
00 NOP        28  ...  50 MOV D.B  78 MOV A.B  A0 ANA B  C8 RZ  F0 RP
01 LXI B.D16  29 DAD H  51 MOV D.C  79 MOV A.C  A1 ANA C  C9 RET  F1 POP PSW
02 STAX B    2A LHLD Adr  52 MOV D.D  7A MOV A.D  A2 ANA D  CA JZ  F2 JP ADR
03 INX B      2B DCX H  53 MOV D.E  7B MOV A.E  A3 ANA E  CB  ...  F3 DI
04 INR B      2C INR L  54 MOV D.H  7C MOV A.H  A4 ANA H  CC CZ  Adr  F4 CP ADR
05 DCR B      2D DCR L  55 MOV D.L  7D MOV A.L  A5 ANA L  CD CALL  Adr  F5 PUSH PSW
06 MVI B.D8   2E MVI L.D8  56 MOV D.M  7E MOV A.M  A6 ANA M  CE ACI D8  F6 ORI D8
07 RL C       2F CMA  57 MOV D.A  7F MOV A.A  A7 ANA A  CF RST 1  F7 RST 6
08 ...        30  ...  58 MOV E.B  80 ADD D  A8 XRA B  D0 RNC  F8 RM
09 DAD B      31 LXI SP.D16  59 MOV E.C  81 ADD D  A9 XRA C  D1 POP D  F9 SPHL
0A LDAX B     32 STA Adr  5A MOV E.D  82 ADD D  AA XRA D  D2 JNC Adr  FA JM ADR
0B DCX B      33 INX SP  5B MOV E.E  83 ADD E  AB XRA E  D3 OUT D8  FB EI
0C INR C      34 INR M  5C MOV E.H  84 ADD H  AC XRA H  D4 CRC Adr  FC CM ADR
0D DCR C      35 DCR M  5D MOV E.L  85 ADD L  AD XRA L  D5 PUSH D  FD  ...  G0 NON-PRINTING
0E MVI C.D8   36 MVI M.D8  5E MOV E.M  86 ADD M  AE XRA M  D6 SUI D8  FE CPI D8
0F RFC        37 STC  5F MOV E.A  87 ADD A  AF XRA A  D7 RST 2  FF RST 7
10 ...        38  ...  60 MOV H.B  88 ADC B  B0 ORA B  D8 RC
11 LXI D.D16   39 DAD SP  61 MOV H.C  89 ADC C  B1 ORA C  D9  ...  G1 X-ON
12 STAX D     3A LDA Adr  62 MOV H.D  8A ADC D  B2 ORA D  DA JC ADR
13 INX D      3B DCX SP  63 MOV H.E  8B ADC E  B3 ORA E  DB IN D8
14 INR D      3C INR A  64 MOV H.H  8C ADC H  B4 ORA H  DC CC ADR
15 DCR D      3D DCR A  65 MOV H.L  8D ADC L  B5 ORA L  DD  ...  \n16 MVI D.D8   3E MVI A.D8  66 MOV H.M  8E ADC M  B6 ORA M  DE SBI D8
17 RAL        3F CMC  67 MOV H.A  8F ADC A  B7 ORA A  DF RST 3  \n18 ...        40 MOV B.B  68 MOV L.B  90 SUB B  B8 CMP B  E0 RPO
19 DAD D      41 MOV B.C  69 MOV L.C  91 SUB C  B9 CMP C  E1 POP H
1A LDAX D     42 MOV B.D  6A MOV L.D  92 SUB D  BA CMP D  E2 JPO Adr  00 NULL
1B DCX D      43 MOV B.E  6B MOV L.E  93 SUB E  BB CMP E  E3 XTHL  07 BELL
1C INR E      44 MOV B.H  6C MOV L.H  94 SUB H  BC CMP H  E4 CPO Adr  09 TAB
1D DCR E      45 MOV B.L  6D MOV L.L  95 SUB L  BD CMP L  E5 PUSH H  0A LF
1E MVI E.D8   46 MOV B.M  6E MOV L.M  96 SUB M  BE CMP M  E6 ANSI D8
1F RAR        47 MOV B.A  6F MOV L.A  97 SUB A  BF CMP A  E7 RST 4  0C FORM
20 ...        48 MOV C.B  70 MOV M.B  98 SBB B  C0 RZ  E8 RPE  0D CR
21 LXI H.D16   49 MOV C.C  71 MOV M.C  99 SBB C  C1 POP B  E9 PCHL  11 X-ON
22 SHLD Adr    4A MOV C.D  72 MOV M.D  9A SBB D  C2 JNZ Adr  EA JPE Adr  12 TAPE
23 INX H       4B MOV C.E  73 MOV M.E  9B SBB E  C3 JMP Adr  EB XCHG  13 X-OFF
24 INR H       4C MOV C.H  74 MOV M.H  9C SBB H  C4 CNZ Adr  EC CPE Adr  14 ENT
25 DCR H       4D MOV C.L  75 MOV M.L  9D SBB L  C5 PUSH B  ED  ...  1B ESC
26 MVI H.D8    4E MOV C.M  76 HLT  9E SBB M  C6 ADI D8  EE XRI D8  1D ALT MODE
27 DAA        4F MOV C.A  77 MOV M.A  9F SBB A  C7 RST 0  EF RST 5  1F RUB OUT

* D6 = constant, or logical/arithmetic expression that evaluates
to an 8 bit data quantity
D16 = constant, or logical/arithmetic expression that evaluates
to a 16 bit data quantity
Adr = 16 bit address

APPENDIX I
# TABLE OF ASCII CODES (Zero Parity)

<table>
<thead>
<tr>
<th>Paper Tape</th>
<th>Upper Octal</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hex</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-6-83 NEVADA ASSEMBLER PAGE 27</td>
<td>0000</td>
<td>000</td>
<td>0</td>
<td>00</td>
<td>ctrl @ NUL</td>
</tr>
<tr>
<td></td>
<td>0004</td>
<td>001</td>
<td>1</td>
<td>01</td>
<td>ctrl A SOH Start of Heading</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>002</td>
<td>2</td>
<td>02</td>
<td>ctrl B STX Start of Text</td>
</tr>
<tr>
<td></td>
<td>0014</td>
<td>003</td>
<td>3</td>
<td>03</td>
<td>ctrl C ETX End of Text</td>
</tr>
<tr>
<td></td>
<td>0020</td>
<td>004</td>
<td>4</td>
<td>04</td>
<td>ctrl D EOT End of Xmit</td>
</tr>
<tr>
<td></td>
<td>0024</td>
<td>005</td>
<td>5</td>
<td>05</td>
<td>ctrl E ENQ Enquiry</td>
</tr>
<tr>
<td></td>
<td>0030</td>
<td>006</td>
<td>6</td>
<td>06</td>
<td>ctrl F ACK Acknowledge</td>
</tr>
<tr>
<td></td>
<td>0034</td>
<td>007</td>
<td>7</td>
<td>07</td>
<td>ctrl G BEL Audible Signal</td>
</tr>
<tr>
<td></td>
<td>0040</td>
<td>010</td>
<td>8</td>
<td>08</td>
<td>ctrl H BS Back Space</td>
</tr>
<tr>
<td></td>
<td>0044</td>
<td>011</td>
<td>9</td>
<td>09</td>
<td>ctrl I HT Horizontal Tab</td>
</tr>
<tr>
<td></td>
<td>0050</td>
<td>012</td>
<td>10</td>
<td>0A</td>
<td>ctrl J LF Line Feed</td>
</tr>
<tr>
<td></td>
<td>0054</td>
<td>013</td>
<td>11</td>
<td>0B</td>
<td>ctrl K VT Vertical Tab</td>
</tr>
<tr>
<td></td>
<td>0060</td>
<td>014</td>
<td>12</td>
<td>0C</td>
<td>ctrl L FF Form Feed</td>
</tr>
<tr>
<td></td>
<td>0064</td>
<td>015</td>
<td>13</td>
<td>0D</td>
<td>ctrl M GR Carriage Return</td>
</tr>
<tr>
<td></td>
<td>0070</td>
<td>016</td>
<td>14</td>
<td>0E</td>
<td>ctrl N SO Shift Out</td>
</tr>
<tr>
<td></td>
<td>0074</td>
<td>017</td>
<td>15</td>
<td>0F</td>
<td>ctrl O SI Shift In</td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>020</td>
<td>16</td>
<td>10</td>
<td>ctrl P DLE Data Line Escape</td>
</tr>
<tr>
<td></td>
<td>0104</td>
<td>021</td>
<td>17</td>
<td>11</td>
<td>ctrl Q DC1 X On</td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td>022</td>
<td>18</td>
<td>12</td>
<td>ctrl R DC2 Aux On</td>
</tr>
<tr>
<td></td>
<td>0114</td>
<td>023</td>
<td>19</td>
<td>13</td>
<td>ctrl S DC3 X Off</td>
</tr>
<tr>
<td></td>
<td>0120</td>
<td>024</td>
<td>20</td>
<td>14</td>
<td>ctrl T DC4 Aux Off</td>
</tr>
<tr>
<td></td>
<td>0124</td>
<td>025</td>
<td>21</td>
<td>15</td>
<td>ctrl U NAK Negative Acknowledge</td>
</tr>
<tr>
<td></td>
<td>0130</td>
<td>026</td>
<td>22</td>
<td>16</td>
<td>ctrl V SYN Synchronous File</td>
</tr>
<tr>
<td></td>
<td>0134</td>
<td>027</td>
<td>23</td>
<td>17</td>
<td>ctrl W ETB End of Xmit Block</td>
</tr>
<tr>
<td></td>
<td>0140</td>
<td>030</td>
<td>24</td>
<td>18</td>
<td>ctrl X CAN Cancel</td>
</tr>
<tr>
<td></td>
<td>0144</td>
<td>031</td>
<td>25</td>
<td>19</td>
<td>ctrl Y EM End of Medium</td>
</tr>
<tr>
<td></td>
<td>0150</td>
<td>032</td>
<td>26</td>
<td>1A</td>
<td>ctrl Z SUB Substitute</td>
</tr>
<tr>
<td></td>
<td>0154</td>
<td>033</td>
<td>27</td>
<td>1B</td>
<td>ctrl \ ESC Escape</td>
</tr>
<tr>
<td></td>
<td>0160</td>
<td>034</td>
<td>28</td>
<td>1C</td>
<td>ctrl \ FS File Separator</td>
</tr>
<tr>
<td></td>
<td>0164</td>
<td>035</td>
<td>29</td>
<td>1D</td>
<td>ctrl ] GS Group Separator</td>
</tr>
<tr>
<td></td>
<td>0170</td>
<td>036</td>
<td>30</td>
<td>1E</td>
<td>ctrl ^ RS Record Separator</td>
</tr>
<tr>
<td></td>
<td>0174</td>
<td>037</td>
<td>31</td>
<td>1F</td>
<td>ctrl _ US Unit Separator</td>
</tr>
<tr>
<td></td>
<td>0200</td>
<td>040</td>
<td>32</td>
<td>20</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>0204</td>
<td>041</td>
<td>33</td>
<td>21</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>0210</td>
<td>042</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
</tr>
<tr>
<td></td>
<td>0214</td>
<td>043</td>
<td>35</td>
<td>23</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>0220</td>
<td>044</td>
<td>36</td>
<td>24</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>0224</td>
<td>045</td>
<td>37</td>
<td>25</td>
<td>%</td>
</tr>
<tr>
<td></td>
<td>0230</td>
<td>046</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>0234</td>
<td>047</td>
<td>39</td>
<td>27</td>
<td>'</td>
</tr>
<tr>
<td></td>
<td>0240</td>
<td>050</td>
<td>40</td>
<td>28</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>0244</td>
<td>051</td>
<td>41</td>
<td>29</td>
<td>)</td>
</tr>
<tr>
<td></td>
<td>0250</td>
<td>052</td>
<td>42</td>
<td>2A</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>0254</td>
<td>053</td>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>0260</td>
<td>054</td>
<td>44</td>
<td>2C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0264</td>
<td>055</td>
<td>45</td>
<td>2D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0270</td>
<td>056</td>
<td>46</td>
<td>2E</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0274</td>
<td>057</td>
<td>47</td>
<td>2F</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>0300</td>
<td>060</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0304</td>
<td>061</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0310</td>
<td>062</td>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0314</td>
<td>063</td>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0320</td>
<td>064</td>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0324</td>
<td>065</td>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0330</td>
<td>066</td>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0334</td>
<td>067</td>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>0340</td>
<td>070</td>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>0344</td>
<td>071</td>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0350</td>
<td>072</td>
<td>58</td>
<td>3A</td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>0354</td>
<td>073</td>
<td>59</td>
<td>3B</td>
<td>;</td>
</tr>
<tr>
<td></td>
<td>0360</td>
<td>074</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td>0364</td>
<td>075</td>
<td>61</td>
<td>3D</td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>0370</td>
<td>076</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>0374</td>
<td>077</td>
<td>63</td>
<td>3F</td>
<td>?</td>
</tr>
<tr>
<td>Paper tape</td>
<td>Upper Octal</td>
<td>Octal</td>
<td>Decimal</td>
<td>Hex</td>
<td>Character</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------</td>
<td>---------</td>
<td>-----</td>
<td>-----------</td>
</tr>
<tr>
<td>123 4 5 6 7 8</td>
<td>0400 100</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0404 101</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0410 102</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0414 103</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0420 104</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0424 105</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0430 106</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0434 107</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0440 108</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0444 109</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0450 110</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0454 111</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0460 112</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0464 113</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0470 114</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0474 115</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0500 116</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0504 117</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0510 118</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0514 119</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0520 120</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0524 121</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0530 122</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0534 123</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0540 124</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0544 125</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0550 126</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0554 127</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0560 128</td>
<td>92</td>
<td>5C \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0564 129</td>
<td>93</td>
<td>5D ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0570 130</td>
<td>94</td>
<td>5E ^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0574 131</td>
<td>95</td>
<td>5F _</td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0600 132</td>
<td>96</td>
<td>60</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0604 133</td>
<td>97</td>
<td>61</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0610 134</td>
<td>98</td>
<td>62</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0614 135</td>
<td>99</td>
<td>63</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0620 136</td>
<td>100</td>
<td>64</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0624 137</td>
<td>101</td>
<td>65</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0630 138</td>
<td>102</td>
<td>66</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0634 139</td>
<td>103</td>
<td>67</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0640 140</td>
<td>104</td>
<td>68</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0644 141</td>
<td>105</td>
<td>69</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0650 142</td>
<td>106</td>
<td>6A</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0654 143</td>
<td>107</td>
<td>6B</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0660 144</td>
<td>108</td>
<td>6C</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0664 145</td>
<td>109</td>
<td>6D</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0670 146</td>
<td>110</td>
<td>6E</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0674 147</td>
<td>111</td>
<td>6F</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0700 148</td>
<td>112</td>
<td>70</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0704 149</td>
<td>113</td>
<td>71</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0710 150</td>
<td>114</td>
<td>72</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0714 151</td>
<td>115</td>
<td>73</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0720 152</td>
<td>116</td>
<td>74</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0724 153</td>
<td>117</td>
<td>75</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0730 154</td>
<td>118</td>
<td>76</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0734 155</td>
<td>119</td>
<td>77</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0740 156</td>
<td>120</td>
<td>78</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0744 157</td>
<td>121</td>
<td>79</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0750 158</td>
<td>122</td>
<td>7A</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0754 159</td>
<td>123</td>
<td>7B</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>. . . . .</td>
<td>0760 160</td>
<td>124</td>
<td>7C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### NEVADA ASSEMBLER

**APPENDIX 3**

**ASSEMBLER LISTING**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>ASSEMBLED CODE</th>
<th>ERROR FLAG</th>
<th>LINE NO.</th>
<th>LABEL</th>
<th>OPERATION</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SEARCH TABLE FOR MATCH TO STRING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>EACH TABLE ENTRY IS FOLLOWED BY A TWO-BYTE DISPATCH ADDRESS.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>TABLE MUST HAVE AT LEAST ONE ENTRY AND IS TERMINATED BY A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>*ZERO BYTE.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td>*ON ENTRY: HL POINTS TO STRING</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td>DE POINTS TO TABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007</td>
<td>C IS NUMBER OF CHARACTERS IN TABLE ENTRIES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>*ON RETURN: ZERO FLAG SET IF NO MATCH, ELSE DE POINTS TO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0009</td>
<td>* dispatch address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```assembly
0100 E5 0111 TSRCCH PUSH H             SAVE STRING ADDRESS
0101 41 0112 MOV B,C INITILIZE CHARACTER COUNT
0102 1A 0113 TS1 LOADX D COMPARE CHARACTERS
0103 BE 0114 CMF M
0104 C2 11 01 0115 JNZ TS3
0107 23 0116 INX H CHARACTERS MATCH, GO ON TO NEXT
0108 13 0117 INX D
0109 05 0118 DCR B
010A C2 02 01 0119 JNZ TS1
010B F6 01 0120 CRI 1 MATCHING ENTRY FOUND
010F E1 0121 TS2 POF H
0110 C9 0122 RET
0111 B7 0123 TS3 ORA A TEST FOR END OF TABLE
0112 CA OF 01 0124 JLE TS2
0115 13 0125 TS4 INX D SKIP TO NEXT ENTRY
0116 05 0126 DCR B
0117 C2 15 01 0127 JNZ TS4
011A 13 0128 INX D
011B 13 0129 INX D
011C E1 0130 POF H
011D C3 00 01 0131 JMP TSRCCH
0132 *                          |
0133 * EXAMPLE OF TSRCH USE:     |
0134 *                             |
0135 * ASSUME HL POINTS TO A FOUR-CHARACTER COMMAND STRING |               |
0120 11 35 01 0136 LXI D,CTABL DE POINTS TO COMMAND TABLE |
0123 0E 04 0137 MVI C,4 TABLE ENTRIES ARE FOUR CHARACTERS LONG |
0125 CD 00 01 0138 CALL TSRCCH |
0126 CA 00 00 01 0139 JZ ERROR COMMAND NOT IN TABLE |
0128 EB 0140 XCHG |
012C 11 00 00 0141 LXI D,COMMAND |
012F D5 0142 PUSH D |
0130 7E 0143 MOV A,M DISPATCH TO APPROPRIATE COMMAND ROUTINE |
0131 23 0144 INX H |
0132 66 0145 MOV H,M |
0133 6F 0146 MOV L,A |
0134 E9 0147 PCHL |
0135 43 4F 4D 31 0148 |
0139 43 4F 4D 32 0149 COMMAND TABLE |
013F 00 00 0150 CTABL ASCII 'COM1' FIRST ENTRY |
0141 00 00 0151 SUB1 ADDRESS OF SUB1 |
0143 43 4F 4D 32 0152 DW SUB1 ADDRESS OF SUB2 |
014F 00 00 0153 ASCII 'COM2' SECOND ENTRY |
0155 DB 00 END OF TABLE MARK |
```

### SYMBOL TABLE LISTING

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CTABL</td>
<td>0135</td>
<td>TS1</td>
<td>0102</td>
<td>TS2</td>
</tr>
<tr>
<td>TS4</td>
<td>0115</td>
<td>TSRCCH</td>
<td>0100</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX 4

This is a sample program. The loader source code.

```
0001  ****************************************************
0002 *                                                 *
0003 *                                                 *
0004 *                                                 *
0005 *                                                 *
0006 *                                                 *
0007 *                                                 *
0008 * An .OBJ file consists of one or more segments that *
0009 * have the format:                                  *
0010 *                                                 *
0011 * BYTE(S)   DESCRIPTION                           *
0012 *                                                 *
0013 *                                                 *
0014 * The run time package will load each segment at the *
0015 * specified address until a starting address is     *
0016 * encountered. A starting address is represented as *
0017 * load address with a zero byte count.              *
0018 *                                                 *
0019 * **************************************************
0020 *
0021 RELOC EQU 0 ;4200H FOR TRS-80 MOD 1
0022 BDOS EQU 5+RELOC ;CP/M
0023 BLKSIZ EQU 128
0024 OFCB EQU 5CH+RELOC ;IN CP/M
0025 OEX EQU OFCB+12
0026 OCR EQU OFCB+32
0027 OBUF EQU 80H+RELOC ;IN CP/M
0028 *
0029 CSTART EQU $                                
0030 LXI SP,STK                                   
0031 MVI C,0CH ;RETURN VERSION #                
0032 CALL BDOS                                   
0033 MOV A,L                                     
0034 ORA A                                       
0035 JNZ VER2X                                   
0036 LDA 4+RELOC ;CPM 1.4 DEFAULT DRIVE         
0037 CPI 5                                       
0038 JC SETDF                                    
0039 XRA A                                       
0040 SETDF EQU $ ;11-30-81 FOR MP/M II
0041 STA ODRIVE ;DEFAULT DRIVE                  
0042 * GET OPTIONS FROM TYPE FIELD
0043 LXI H,5CH+8                                 
0044 MVI C,4                                     
0045 NEXT EQU $                                 
0046 INX H                                     
```
6-6-83  NEVADA ASSEMBLER  PAGE 31

0047  DCR C
0048  JZ NOOPTIONS
0049  MOV A,M
0050  CPI ','
0051  JZ NOOPTIONS
0052  CPI 'Z'
0053  JZ ZEROFIL
0054  CPI 'C'
0055  JZ COMFILE
0056  CPI 'L'
0057  JZ NOEXEC
0058  * ERROR ILLEGAL OPTION
0059  LXI H,MESGA
0060  CALL DISPLAY
0061  JMP 0+RELOC
0062  * GET SIZE OF INSTRUCTION
0063  GETSZ LXI H,TBL-1
0064  AGAIN MOV A,C
0065  INX H
0066  MOV B,M
0067  ANA B
0068  JZ BYTE1
0069  INX H
0070  MOV B,M
0071  XRA B
0072  INX H
0073  JNZ AGAIN
0074  MOV A,M
0075  RET . EXIT
0076  BYTE1 MVI A,1
0077  RET . EXIT
0078  *
0079  REL EQU $ RELOCATION
0080  PUSH H
0081  PUSH D
0082  PUSH PSW
0083  INX H
0084  MOV E,M
0085  INX H
0086  MOV A,M
0087  ORA A WE DON'T RELOCATE BELOW 100+RELOC
0088  JZ NOREL
0089  MOV D,A
0090  PUSH H
0091  LHLDBASE
0092  DAD D ADDRESS IS NOW ADJUSTED
0093  XCHG
0094  POP H
0095  MOV M,D PUT IT BACK
0096  DCX H
0097  MOV M,E
0098  NOREL EQU $
0099  POP PSW
0100  POP D
0101 POP H
0102 RET
0103 *
0104 VER2X EQU $
0105 MVI C,19H ;GET CPM 2.X DEFAULT DRIVE
0106 CALL BDOS
0107 JMP SETDF
0108 *
0109 MESGA ASC 'ILLEGAL OPTION'
0110 DB 0DH,0AH
0111 ASC 'RUN D:FILE.ZCL<CR>'
0112 DB 0DH,0AH,0
0113 *
0114 TBL DB -1,11101001B,1
0115 DB -1,11001101B,3
0116 DB 11000111B,11000100B,3
0117 DB -1,11000011B,3
0118 DB 11000111B,11000010B,3
0119 DB 11000111B,11000111B,1
0120 DB -1,11001001B,1
0121 DB 11000111B,11000000B,1
0122 DB 11001111B,1,3
0123 DB 11100111B,00100010B,3
0124 DB 11110111B,11010011B,2
0125 DB 11000111B,6,2
0126 DB 11000111B,11000110B,2
0127 DB 0 END OF TABLE
0128 *
0129 BASE DW 0 BASE ADJ TO ADD TO ADDRESS TO BE RELOCATED
0130 START DW 0 STARTING ADDR OF RELOCATED CODE
0131 *
0132 ZEROFILL EQU $
0133 STA ZX
0134 JMP NEXT
0135 *
0136 COMFILE EQU $
0137 STA CX
0138 JMP NEXT
0139 *
0140 NOEXEC EQU $
0141 STA LX
0142 JMP NEXT
0143 *
0144 OSET EQU $
0145 LXI D,OBUF
0146 MVI C,26 ;SET DMA
0147 CALL BDOS
0148 LDA ODRIVE
0149 MVI D,0
0150 MOV E,A
0151 MVI C,14 ;SET DRIVE
0152 CALL BDOS
0153 LXI D,OF CB
0154 RET
0155 *  
0156 NOOPTIONS EQU $  
0157 LXI H,080H+3+RELOC  
0158 MOV A,M  
0159 CPI ":" ;WAS DRIVE REQUESTED?  
0160 JNZ DEFDRIVE ;DEFAULT IS SET  
0161 DCX H  
0162 MOV A,M  
0163 CPI 'A'  
0164 JC DEFDRIVE  
0165 SUI 'A'  
0166 STA ODRIVE  
0167 DEFDRIVE EQU $  
0168 CALL SETFCB  
0169 MVI M,'O'  
0170 INX H  
0171 MVI M,'B'  
0172 INX H  
0173 MVI M,'J'  
0174 CALL OSET  
0175 MVI C,15 ;OPEN  
0176 CALL BDOS  
0177 CPI -1  
0178 JZ OERR ;OPEN ERROR  
0179 XRA A  
0180 STA OCR  
0181 *  RELOCATE CODE TO JUST BELOW CP/M  
0182 LHLD 6+RELOC  
0183 DCX H HIGHEST ADDR  
0184 LXI B,LAST-LOADFILE SIZE OF CODE TO BE RELOCATED  
0185 MOV A,L  
0186 SUB C  
0187 MOV L,A  
0188 MOV A,H  
0189 SBB B  
0190 MOV H,A  
0191 ; H&L= STARTING ADDRESS  
0192 SHLD START  
0193 PUSH H  
0194 LXI D,LOADFILE  
0195 MOV A,L  
0196 SUB E  
0197 MOV L,A  
0198 MOV A,H  
0199 SBB D  
0200 MOV H,A  
0201 SHLD BASE  
0202 POP H  
0203 LXI B,CONSTANTS-LOADFILE SIZE OF INSTRUCTION MOVE  
0204 XCHG  
0205 NXTI EQU $  
0206 PUSH H  
0207 PUSH D  
0208 PUSH B
MOV C,M GET OPCODE
CALL GETSZ GET SIZE OF INSTRUCTION
POP B
POP D
POP H
CPI 3
JC SKPREL
CALL REL RELOCATE ADDR IN THIS 3 BYTE INST
SKPREL EQU $
PUSH B
PUSH PSW
MOV C,A SIZE
NXTM EQU $
MOV A,M
STAX D
INX H
INX D
DCR C
JNZ NXTM
POP PSW
POP B
NXTD EQU $
DCX B
DCR A
JNZ NXTD
MOV A,C
ORA B
JNZ NXTC
MOV A,C
ORA B
JNZ NXTC
MOV A,C
ORA B
JNZ NXTC
MOV A,C
ORA B
JNZ NXTC
LXI B,LAST-CONSTANTS SIZE OF CONSTANTS
NXTC EQU $
MOV A,M
STAX D
INX H
INX D
DCX B
MOV A,C
ORA B
JNZ NXTC
LXI D,LOADFILE-1
LXI SP,STK SET STACK AFTER RELOCATION
LDA ZX ZERO FILL MEMORY?
ORA A
JZ SKPCLR
LXI D,LOADFILE-1
PCHL . CODE HAS BEEN RELOCATED NOW GO TO IT
RUNA A:FILE.OBJ<CR>
MOVE PARAMETERS AND CHECK
LOADFILE EQU $
LXI SP,STK SET STACK AFTER RELOCATION
LDA ZX ZERO FILL MEMORY?
0263 MVI H,1 STARTING ADDR +RELOC
0264 MVI L,0
0265 CLEAR EQU $
0266 XRA A
0267 MOV M,A
0268 INX H
0269 MOV A,L
0270 SUB E
0271 MOV A,H
0272 SBB D
0273 JC CLEAR
0274 SKPCLR EQU $
0275 CALL ORD ;GET 1ST RECORD OF .OBJ FILE
0276 OLOAD EQU $
0277 CALL GETOP
0278 MAO MOV A,M ;MOVE 4 BYTES FROM BUF TO WORK
0279 STAX D
0280 INX H
0281 INX D
0282 DCR C
0283 CZ ORD
0284 DCR B
0285 JNZ MAO
0286 ; H&L = BUFFER C=COUNT
0287 XCHG
0288 LHLD OWRK ;SIZE OF NEXT READ
0289 MOV A,L
0290 ORA H
0291 JZ CLOSE
0292 SHLD OSIZE
0293 LHLD OWRK+2
0294 XCHG
0295 MAOA MOV A,M ;MOVE FROM BUF TO OBJ ADDR
0296 STAX D
0297 INX H
0298 INX D
0299 DCR C
0300 CZ ORD
0301 PUSH H
0302 LHLD OSIZE
0303 DCX H
0304 SHLD OSIZE
0305 MOV A,L
0306 ORA H
0307 POP H
0308 JNZ MAOA
0309 CALL SAVOP
0310 JMP OLOAD
0311 *
0312 GETOP EQU $ ;GET 0 POINTERS
0313 LXI D,OWRK
0314 LHLD OCBA ;BUF ADDR
0315 MVI B,4 ;LENGTH OF WRK
0316 LDA OCBC ;BUF CNT
0317  MOV C,A
0318  RET
0319  *
0320  ORD EQU $
0321  PUSH B
0322  PUSH D ;OPNT
0323  LXI D,OFCB
0324  MVI C,20 ;READ
0325  CALL BDOS
0326  POP D
0327  POP B
0328  ORA A
0329  JNZ RERR
0330  LXI H, OBUF
0331  MVI C, BLKSIZ
0332  RET
0333  *
0334  SAVOP EQU $
0335  SHLD OCBA ;BUFF ADDR
0336  MOV A,C
0337  STA OCBC ;BUF CNT
0338  LDA HIGH
0339  CMP D
0340  RNC
0341  MOV A,D
0342  STA HIGH
0343  RET
0344  *
0345  CLOSE EQU $
0346  LXI D, OFCB
0347  MVI C,16 ;CLOSE
0348  CALL BDOS
0349  LDA CX
0350  ORA A
0351  JNZ GENCOM
0352  LDA LX
0353  ORA A
0354  JNZ 0+RELOC LOAD BUT DON'T EXECUTE
0355  LHLD OWRK+2 ;STARTING ADDRESS
0356  PCHL
0357  *
0358  SETFCB EQU $
0359  XRA A
0360  STA OFCB
0361  STA OCR
0362  LXI H, OEX
0363  MVI C, 4
0364  EXLUP EQU $ ;10-2-81 ZERO CPM EXT AREA
0365  MOV M,A
0366  INX H
0367  DCR C
0368  JNZ EXLUP
0369  LXI H, OBUF
0370  SHLD OCBA
0371 MVI A, BLKSIZ
0372 STA OCBC
0373 LXI H, 5CH + 9 + RELOC ; CP/M FILE TYPE
0374 RET
0375 *
0376 CREATE EQU $
0377 LXI D, OFCB
0378 MVI C, 22 CREATE
0379 CALL BDOS
0380 CPI -1
0381 RNZ
0382 OERR EQU $
0383 LXI H, MESGO OPEN ERROR
0384 CALL DISPLAY
0385 JMP OXT1
0386 *
0387 GENCOM EQU $ GENERATE .COM FILE
0388 CALL SETFCB
0389 MVI M, 'C'
0390 INX H
0391 MVI M, 'O'
0392 INX H
0393 MVI M, 'M' .COM IN FCB
0394 *OPEN
0395 LXI D, OFCB
0396 MVI C, 15 OPEN .COM FILE
0397 CALL BDOS
0398 CPI -1
0399 CZ CREATE
0400 XRA A
0401 STA OCR
0402 *WRITE
0403 LDA HIGH
0404 DCR A
0405 MOV H, A
0406 MVI L, OFFH
0407 SHLD SIZ OF THIS WRITE
0408 MVI D, 1 STARTING ADDRESS + RELOC
0409 MVI E, 0
0410 LXI H, OBUFF BUFFER ADDRESS
0411 MVI C, BLKSIZ BUFFER SIZE
0412 NXTW EQU $
0413 LDAX D
0414 MOV M, A
0415 INX H
0416 INX D
0417 DCR C BUFF COUNT
0418 CZ WRITE
0419 PUSH H
0420 LHLD SIZ
0421 DCX H
0422 SHLD SIZ
0423 MOV A, L
0424 ORA H
0425    POP H
0426    JNZ NXTW
0427    CALL WRITE LAST BLOCK
0428    * CLOSE
0429    LXI D, OFCB
0430    MVI C, 16 CLOSE
0431    CALL BDOS
0432    JMP 0+RELOC
0433    *
0434    WRITE EQU $
0435    PUSH D
0436    LXI D, OFCB
0437    MVI C, 21 WRITE
0438    CALL BDOS
0439    POP D
0440    ORA A
0441    JNZ ERRW
0442    LXI H, OBUF
0443    MVI C, BLKSIZ
0444    RET
0445    *
0446    **+**************************************************************************+
0447    $$ DISPLAY A MESSAGE TO THE CONSOLE
0448    * ENTRY H&L CONTAIN STARTING ADDRESS OF THE MESSAGE
0449    * THE MESSAGE TEXT IS TERMINATED BY 0 HEX
0450    * CALL DISPLAY
0451    **-**************************************************************************
0452    *
0453    DISPLAY EQU $
0454    MOV A, M
0455    ORA A
0456    RZ . EXIT TO CALLING ROUTINE **
0457    MOV E, A
0458    MVI C, 2
0459    PUSH H
0460    CALL BDOS ;PUT THE CHAR TO THE CONSOLE
0461    POP H
0462    INX H
0463    JMP DISPLAY
0464    *
0465    ERRW EQU $
0466    LXI H, MESGW WRITE ERROR
0467    CALL DISPLAY
0468    JMP OXT1
0469    *
0470    RERR EQU $
0471    LXI H, MESGR READ ERROR
0472    CALL DISPLAY
0473    OXT1 EQU $
0474    LXI H, OFCB+1 ;FILE NAME
0475    CALL DISPLAY
0476    JMP 0+RELOC RETURN TO CP/M
0477    ************************************************************
0478    CONSTANTS EQU $
0479 HIGH DB 0 HIGHEST PAGE USED FOR .COM
0480 ZX DB 0 DEFAULT NO CLEAR ;Z=ZERO FILL BEFORE LOADING
0481 CX DB 0 DEFAULT NO .COM ;C=.COM FILE
0482 LX DB 0 DEFAULT EXECUTE ;L=LOAD BUT NO EXECUTION
0483 ODRIVE DB 0
0484 OWRK DB 0,0,0,0
0485 OCBA DW OBUF ;CURRENT BUFFER ADDRESS
0486 OCBC DB BLKSIZ ;CURRENT BUFFER COUNTER
0487 OSIZE DW 0 ;SIZE OF NEXT OBJ BLOCK
0488 SIZ DW 0 SIZE OF COM FILE CODE
0489 MESGO ASC 'OPEN ERROR '
0490 DB 0
0491 MESGR ASC 'READ ERROR '
0492 DB 0
0493 MESSW ASC 'WRITE ERROR '
0494 DB 0
0495 DS 30
0496 STK DB 'S'
0497 LAST DB 0
APPENDIX 5

REFERENCES


SOFTWARE LICENSE AGREEMENT

IMPORTANT: All Ellis Computing, Inc. programs are sold only on the condition that the purchaser agrees to the following License.

Ellis Computing, Inc. agrees to grant and the Customer agrees to accept on the following terms and conditions nontransferable and nonexclusive Licenses to use the software program(s) (Licensed Programs) herein delivered with this Agreement.

TERM:

This Agreement is effective from the date of receipt of the above-referenced program(s) and shall remain in force until terminated by the Customer upon one month's prior written notice, or by Ellis Computing, Inc. as provided below.

Any License under this Agreement may be discontinued by the Customer at any time upon one month's prior written notice. Ellis Computing, Inc. may discontinue any License or terminate this Agreement if the Customer fails to comply with any of the terms and conditions of this Agreement.

LICENSE:

Each program License granted under this Agreement authorizes the Customer to use the Licensed Program in any machine readable form on any single computer system (referred to as System). A separate license is required for each System on which the Licensed Program will be used.

This Agreement and any of the Licenses, programs or materials to which it applies may not be assigned, sublicensed or otherwise transferred by the Customer without prior written consent from Ellis Computing, Inc. No right to print or copy, in whole or in part, the Licensed Programs is granted except as hereinafter expressly provided.

PERMISSION TO COPY OR MODIFY LICENSED PROGRAMS:

The customer shall not copy, in whole or in part, any Licensed Programs which are provided by Ellis Computing, Inc. in printed form under this Agreement. Additional copies of printed materials may be acquired from Ellis Computing, Inc.

The NEVADA ASSEMBLER-FORTRAN Licensed Programs which are provided by Ellis Computing, Inc. in machine readable form
may be copied, in whole or in part, in machine readable form in sufficient number for use by the Customer with the designated System, for back-up purposes, or for archive purposes. The original, and any copies of the Licensed Programs, in whole or in part, which are made by the Customer shall be the property of Ellis Computing, Inc. This does not imply that Ellis Computing, Inc. owns the media on which the Licensed Programs are recorded.

The NEVADA ASSEMBLER-FORTRAN Licensed Programs called "FRUN.COM" and "ARUN.COM" which are provided by Ellis Computing, Inc. may be distributed to third parties.

The Customer agrees to reproduce and include the copyright notice of Ellis Computing, Inc. on all copies, in whole or in part, in any form, including partial copies of modifications, of Licensed Programs made hereunder.

PROTECTION AND SECURITY:

The Customer agrees not to provide or otherwise make available the NEVADA ASSEMBLER-FORTRAN Program including but not limited to program listings, object code and source code, in any form, to any person other than Customer or Ellis Computing, Inc. employees, without prior written consent from Ellis Computing, Inc., except with the Customer's permission for purposes specifically related to the Customer's use of the Licensed Program.

DISCLAIMER OF WARRANTY:

Ellis Computing, Inc. makes no warranties with respect to the Licensed Programs.

LIMITATION OF LIABILITY:

THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL Ellis Computing, Inc. BE LIABLE FOR CONSEQUENTIAL DAMAGES EVEN IF Ellis Computing, Inc. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

GENERAL:

If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted. This is the complete and exclusive statement of the Agreement between the parties which supercedes all proposals, oral or written, and all other communications between the parties relating to the subject matter of this Agreement. This Agreement will be governed by the laws of the State of California.
CORRECTIONS AND SUGGESTIONS

All suggestions and problems must be reported in writing. Please include samples if possible.

ASSEMBLER-FORTRAN VERSION ________ SERIAL # ________

Operating system and version ____________________________

Hardware configuration ____________________________

ERRORS IN MANUAL:

SUGGESTIONS FOR IMPROVEMENTS TO MANUAL:

ERRORS IN ASSEMBLER-FORTRAN:

SUGGESTIONS FOR IMPROVEMENT TO ASSEMBLER-FORTRAN:

MAIL TO: Ellis Computing, Inc.
3917 Noriega Street
San Francisco, CA 94122

FROM: NAME________________________ DATE______________

ADDRESS________________________________________

CITY, STATE, ZIP_________________________________

PHONE NUMBER___________________________________

If you wish a reply include a self-addressed postage-paid envelope. Thank you.
A
ASC 20
ASCF 20
ASCZ 20
Assembly Errors 23
Assembler Command Errors 23

C
Comment Field 16
Conditional Assembly 20
Console 6
Constants 14
COPY file 21
Corrections 43
CP/M 4, 5, 6

D
DB 19
DDB 19
Define Double Byte 19
Define Byte 19
Define Word 19
Define Storage 18
Define ASCII String 20
DS 18
DW 19

E
END of Source file 22
ENDF 20
EQU 17
Equate 17
Error (.ERR) file 7
Error Codes 23
Executing the Assembler 6
Executing the .OBJ file 9
Expressions 15

F
FILES 4-8, 22

G
Getting Started 5

H
Hardware Requirements 5
High-Order Byte Extraction 15

I
IF 20
IFLS 21
Introduction 4
L
Label Field 12
Labels 13
License Agreement 41
Line Length 11
Line Numbers 8 11
List Conditional Code 21
Listing Title 22
Listing Control 22
Listing File 6, 8
Loader Source Code 30
Low-Order Byte Extraction 15
LST 22

M
Memory Usage 9
Messages 23

N
NLST 22

O
Object (.OBJ) file 7
OP-Codes 25
Operand Field 13
Operation Field 12
Operating Procedures 5
Options 8
ORG 17

P
Page Eject 22
Paginate Output 8
Pseudo-Operations 17

R
References 40
Register Names 13
RES 18
RUNA 9, 30
Sample Program 30
Set ASCII List Flag 20
Set Execution Address 18
Set Origin 17
SID 8
Software Requirements 5
Source Code 11, 30
Startup 8
Statements 11
Suggestions 43
Symbol (.SYM) file 8
Termination 10
Text Files 4
TITL 8, 22

XEQ 18