NEVADA FORTRAN™

for the Commodore 64

commodore
COMPUTER

NEVADA FORTRAN™

Programmer's Reference Manual

Includes NEVADA ASSEMBLER™

COPYRIGHT

DISCLAIMER

TRADEMARKS

We hope you enjoy using NEVADA FORTRANTMfor the Commodore 64. This software runs under CP/M 2.2 Operating System. Here are several other Commodore software packages which you should know about:

NEVADA COBOLTM
This updated subset of COBOL was designed for small businesses with a Commodore 64. The four divisions of a COBOL program are reviewed, as well as the applicable Reserved Words. Details on using COBOL are described, along with error codes and messages. A glossary of items is also included.

EASY SCRIPT 64
This is a powerful word processor with table producing capabilities, comprehensive printer controls, easy update facilities, easy document handling, the ability to interact with EASY SPELL 64, and more.

THE WORD MACHINE and THE NAME MACHINE
This is an easy-to-learn and easy-to-use word processing package. Perfect for letters, address lists, memos, and notes, these programs let you overtype, insert, and delete text; personalize form letters; and print in draft, formal, or informal formats.

EASY SPELL 64
Easy Spell 64 features the following: the automatic correction of spelling errors, the ability to count the number of words in your manuscript and interact with Easy Script 64, and a built-in 20,000 word dictionary that lets you add words not already stored there.

EASY MAIL 64
With Easy Mail 64, you can easily manage your address files. Label printing is also simplified with Easy Mail's ability to search for specific fields/categories. The program's features include entry, change, or deletion of a file by name or number; the capability to print one or two abreast labels; a HELP screen; and the ability to print a complete printout of all the data in each of your records.

EASY CALC 64
Easy Calc 64 is an easy-touse electronic spread sheet which features editing functions and HELP screens. With Easy Calc 64, you can also print bar charts and individually formatted tables.

THE MANAGER
The Manager is a general data base for handling your files.

THE COMMODORE 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM
This package is designed for experienced Assembly language programmers. Everything you need to create, assemble, load, and execute 6500 series Assembly language code is included.

SCREEN EDITOR

The Screen Editor helps you design software by letting you create and edit your own screens. This programming tool is for users with some computer experience.

SUPER EXPANDER 64

This cartridge is a powerful extension of the BASIC language which gives you the commands needed to easily access and implement Commodore's graphics, music, and sound capabilities. You will be amazed at how quickly and easily you can plot pOints and lines; draw arcs, circles, ellipses, rectangles, triangles, octagons; paint shapes with specified colors; read game paddle and joystick locations; create music and sound; display text; split screens to display both text and graphics; and program the function keys.

THE EASY FINANCE SERIES

Commodore is proud to announce an entire series of EASY FINANCE software packages which may solve many of your business and personal needs. The EASY FINANCE series is called "easy" because all of the programs are simple to operate and require no programming experience. Here is a brief description of each:

EASY FINANCE I - LOANS

Calculates 12 different loan concepts including principal, regular payment, last payment, and remaining balance,

EASY FINANCE II - INVESTMENTS

Includes 16 investment concepts. Functions such as future investment value, initial investment, and internal rate of return can be calculated.

EASY FINANCE 111- ADVANCED INVESTMENTS

This advanced version of EASY FINANCE II, includes 16 more investment concepts. Financial terms are clarified and functions such as discount commercial paper, financial management rate of return, and financial leverage and earnings per share are described.

EASY FINANCE IV - BUSINESS MANAGEMENT

This package helps managers make the right decisions about production, inventory, control, compensation, and much more. Lease purchase analysis, depreciation switch, and optimal order quantity are some of the 21 functions that can be calculated.

EASY FINANCE V - STATISTICS

This includes payoff matrix analysis, regression analysis forecasting, and apportionment by ratios,

Please contact your local Commodore dealer for additional information on other software available for your Commodore computer.

Thank you for owning a Commodore computer. Now that you are a member of the Commodore family, maybe you'd like to expand your computer's family. Here is a list of additional hardware which is compatible with your Commodore computer:

Printers

The 1525E printer is an 80-column, dot-matrix, impact printer for creating printouts and hard-copies from your VIC 20 or Commodore 64. The printer features 30 characters per second print speed and prints graphics and text characters. The 1526 Printer has all of the same features but is bi-directional and has programmable line spacing and a print format interpreter.

1520 Plotter/Printer

This is a four color, high resolution plotter that connects directly to your VIC 20 or Commodore 64 computer. With the 1520 Plotter/Printer you can plot on a piece of paper, the unique color graphics that you have created on your screen using LOGO graphiCS!

Commodore Speech Module

The speech module cartridge comes with a built-in vocabulary of 234 words which are easily programmed into sentences. The module "talks" in a pleasant female or male voice ... it can generate other types of voices with special vocabularies geared to each software package. The speech module works with disk, tape, and also has a slot for accepting plug-in cartridges.

1701/1702 Monitor

This full color monitor is compatible with the VIC 20, Commodore 64, and other computers. The *1701/1702* Monitor features high quality resolution video and a built-in speaker with audio amplifier.

1530 DATASSETTE

The 1530 DATASSETTE is a low cost, highly reliable way to store and retrieve programs and data. It features keys for Play, Record, Fast-Forward, Rewind, and Stop. The 1530 DATASSETTE uses standard audio cassette tapes and allows naming of programs and files, verification of programs, and programmable end of tape marker sensing.

Joystick and Paddles

Controls for games and entertainment.

1600 Modem

This telephone interface lets you communicate with other computer systems over your telephone line! The modem package includes cassette-tape terminal software, a free password and one-hour subscription to the CompuServe System* and software controls for duplex, baud rate, and parity. There is also an optional adapter available for non-modular phones.

1650 AutoModem

This telephone interface features automatic answer and automatic dial. The modem package lets you communication with other computer systems over the phone lines! It includes cassette-tape terminal with software, a free password and one-hour subscription to the CompuServe System TM*and software controls for duplex, baud rate, and parity. You need a modular phone or adaptor to use this product.

PET 64

This unique machine combines many of the Commodore 64 features with the capabilities of the Commodore PET. However, sprites, color, and sound are not featured on this machine.

SX-64 Portable Color Computers

These new computers are Commodore 64's in a convenient portable style. The model SX-64 (Single disk drive) is an excellent investment for executive business people, as well as affordable for today's students.

PREFACE

NEVADA FORTRAN for Commodore 64 is an *8080180851Z80* version of FORTAN IV. It is a powerful subset implementation of this widely used language. The compiler works from disk (also using the assembler) to produce *8080180851Z80* 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. 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 the CPIM Operating System for your Commodore 64.

This reference manual assumes you already have the knowledge to program in FORTRAN and have read the Commodore 64 CPIM Operating System. An additional list of supplementary materials can be found in the back of this book.

This manual is **not** a tutorial and will not teach you "how to" program in FORTRAN. However, for the experienced FORTRAN programmer who is already familiar with the CPIM Operating System, this manual provides the necessary tools for using this 3.0 version of NEVADA FORTRAN on your Commodore 64. The manual includes:

General Concepts and details of FORTRAN programming

Summaries of system functions and subroutines

A list of runtime and compile time errors

We hope you enjoy using NEVADA FORTRAN on your Commodore 64.

Files on the NEVADA FORTRAN Data Disk (Used with the NEVADA FORTRAN program)

FORT.COM	is the FORTRAN compiler
FRUN.COM	is the runtime execution package
CONFIG.COM	is a program to generate the error file and setup compiler and runtime defaults.
ERRORS	is the error text file used by CON FIG.

Files on the NEVADA FORTRAN Data Disk (Used with the NEVADA ASSEMBLER included in back section of this book)

ASSM.COM	is the assembler program
RUNA.COM	is the runtime loader
LD.ASM	is a sample Assembler program

TABLE OF CONTENTS

PREFACE i

FILES ON THE NEVADA FORTRAN DATA DISK ii

1 GETTING STARTED 1

GENERATING THE ERROR FILE FORT.ERR..................... 3

CONFIGURING THE FORTRAN SYSTEM 3

2 COMPILING AND EXECUTING A PROGRAM......................... 4

CREATING A PROGRAM 4

RUNNING THE COMPILER 4

COMPILE OPTIONS 6

EXECUTING A PROGRAM.. 8

CREATING A COM FILE.. 8

3 THE FORTRAN LANGUAGE.. 9

THE FORTRAN CHARACTER SET 9

FORTRAN PROGRAM STRUCTURE 10

FORTRAN STATEM ENTS 11

MULTI-STATEMENTS 11

FORTRAN PROGRAM PREPARATION 12

THE COpy STATEMENT 12

THE OPTIONS STATEMENT.. 13

4 NUMBER SYSTEM 15

NUMBER RANGES 15

CONSTANTS 15

Numerical Constants 15

String Constants 16

Logical Constants 16

VARIABLE NAMES 17

TYPE SPECIFICATION 17

Integer 17

Logical 18

Real 18

Double Precision 18

DATA STATEMENT 19
COMMON BLOCKS 20
IMPLICIT STATEMENT 21

5 EXPRESSIONS 22
HIERARCHY OF OPERATORS 22
EXPRESSION EVALUATION 23
INTEGER OPERATIONS 23
REAL OPERATIONS 24
LOGICAL OPERATIONS 24
MIXED EXPRESSIONS 25

6 CONTROL STATEMENTS 26
UNCONDITIONAL GO TO STATEMENT 26
COMPUTED GO TO STATEMENT 27
ASSIGNED GO TO 27
ASSIGN 27
ARITHMETIC IF STATEMENT 28
LOGICAL IF STATEMENT 28
IF-THEN-ELSE 29
DO-LOOPS 31
CONTINUE STATEMENT 32
ERROR TRAPPING 32
CONTROL/C CONTROL 34
TRACING 35
DUMP STATEMENT 36

7 PROGRAM TERMINATION STATEMENTS 37
PAUSE STATEMENT 37
STOP STATEMENT 38
END STATEMENT 38

8 ARRAY SPECIFICATIONS 39
DIMENSION STATEMENT 39
SUBSCRIPTS 41

9 SUBPROGRAMS 42
SUBROUTINE STATEMENT 42
FUNCTION STATEMENT 43
CALL STATEMENT 43
RETURN STATEMENT 44
NORMAL RETURN 44
MULTIPLE RETURN 44
BLOCK DATA SUBPROGRAM 45

10 INPUT/OUTPUT 46
GENERAL INFORMATION 46
1/0 LIST SPECIFICATION 47
READ STATEMENT 48
WRITE STATEMENT 50
MEMORY TO MEMORY I/O STATEMENTS 50
DECODE Statement 50
ENCODE Statement 51
FORMAT STATEMENT AND FORMAT SPECIFICATIONS 51
A-Type (Aw) 52
D-Type (Dw.d) 52
E-Type (Ew.d) 52
F-Type (Fw.d) 53
G-Type (Gw.d) 53
I-Type (Iw) 54
K-Type (Kw) 55
L-Type (Lw) 55
T-Type (Tw) 55
X-Type (wX) 56
Z-Type 56
I-Type (I) 56
Repeating Field Specifications 57
String Output 57
FREE FORMAT 58
Input. 58
Output. 58
BINARY 110 59
REWIND STATEMENT 59
BACKSPACE STATEMENT 60
ENDFILE STATEMENT 60
GENERAL COMMENTS ON FORTRAN 110 UNDER CP/M 61
SPECIAL CHARACTERS DURING CONSOLE 110 61

11 GENERAL PURPOSE SUBROUTINE/FUNCTION LIBRARY ... 62

Subroutine Names:

BIT 63
CHAIN 63
CIN 64
CLOSE 64
CTEST 65
DELAY 65
DELETE 65
EXIT 66
LOAD 66
LOPEN 67
MOVE 68
OPEN 68
OUT 69
POKE 70
PUT 70
RENAME 70
RESET 71
SEEK 71
SETIO 72

Function Names:

CALL 72
CBTOF 73
CHAR 73
COMP 73
INP 74
PEEK 74

12 INTRODUCTION TO NEVADA ASSEMBLER 76

13 OPERATING PROCEDURES 77
HARDWARE REQUIREMENTS 77
SOFTWARE REQUIREMENTS 77
FILE TYPE CONVENTIONS 77
GETTING STARTED 77
EXECUTING THE ASSEMBLER 78
STARTUP 79
EXECUTING THE .OBJ FILE 80
MEMORY USAGE 80
TERMINATION 80

14 STATEMENTS 81
INTRODUCTION 81
LINE NUMBERS .. 81
LABEL FIELD 82
OPERATION FIELD 82
OPERAND FIELD 82
Register Names 83
Labels 83
Constants 84
Expressions 85
High and Low Order Byte Extraction 85
COMMENT FIELD .. 86

15 PSEUDO-OPERATIONS 87

16 ERROR CODES AND MESSAGES 88

APPENDIX A - STATEMENT SUMMARY 94

APPENDIX B - SUMMARY OF SYSTEM FUNCTION 97

APPENDIX C - SUMMARY OF SYSTEM SUBROUTINES 99

APPENDIX 0 - RUNTIME ERRORS 101

APPENDIX E - COMPILE TIME ERRORS 104

APPENDIX F - ASSEMBLY LANGUAGE INTERFACE 108

APPENDIX G - GENERAL COMMENTS 109

APPENDIX H - COMPARISON OF NEVADA FORTRAN AND ANSI FORTRAN 111

APPENDIX I - 8080 OPERATION CODE 112

APPENDIX J - TABLE OF ASCII CODES (Zero Parity) 116

APPENDIX K - SAMPLE ASSEMBLER LISTING 119

APPENDIX L - SAMPLE PROGRAM OF LOADER SOURCE CODE 121

APPENDIX M - SUGGESTED REFERENCES 132

INDEX 133

1 GETTING STARTED

Required Hardware:

Your Commodore 64 computer

The Commodore Z80 microprocessor (This is your CPIM cartridge.)

A Commodore 1541 single disk drive

A video display monitor such as the Commodore Color Monitor Model *1701/1702*

Required Software:

Commodore's CPIM Operating System disk

A text editor. ED. COM is found on your Commodore CPIM Operating System disk.

Throughout our discussion, we will be referring to the following disks:

NEVADA FORTRAN Data disk
Included in your NEVADA FORTRAN software package, this disk should only be read. A listing of the files contained on this disk can be found at the front of this manual. (Consider this to be disk B.)

CPIM Operating System disk
This is your Commodore CPIM Operating System disk that you use with your ZOOcartridge.

CP/M NEVADA FORTRAN Operations disk
This is a disk which you create (Consider this to be disk A.)

Note that you should NEVER write on your NEVADA FORTRAN Data disk. To prevent mistakes from occurring, be sure that this disk is write protected. (Place a standard protection label over the "square cornered" notch on the disk.) Before continuing, consult your Commodore 64 CPIM Operating System User's Guide if you are not familiar with the DIR, ERA, PIP, and STAT commands.

Follow these steps to get started using NEVADA FORTRAN:

1. Use one of your CPIM Operating System disk backup copies to create your CPIM-NEVADA FORTRAN Operations disk. If you don't have a backup copy of the CPIM Operating System disk, see Section 4.2 The Copy Utility in your Commodore CPIM Operating System User's Guide.

2. Insert the newly created CPIM NEVADA FORTRAN Operations disk into the disk drive and boot CPIM. After CPIM is booted, the computer automatically displays an 'A>' prompt.

3. Use the CPIM ERA command to erase all of the files except the PIP.COM, STAT.COM and ED.COM files from your newly created NEVADA FORTRAN Operations disk.

4. Now, we will add the FORTRAN files to our newly created disk. Remember, we will refer to the NEVADA FORTRAN Data disk as disk 'B' and the CPIM-NEVADA FORTRAN Operations Disk as disk 'A'.

   Use the PIP command to copy the files from your NEVADA FORTRAN Data disk to the CPIM-NEVADA FORTRAN Operations disk. PIP will prompt you throughout the entire copy process. To invoke the PIP program, input:

PIP

After RETURN is pressed, an asterisk (*) is displayed on the following line. Now, copy and verify the entire FORTRAN disk:

*A: = B: *.* +V1

The following prompt will then be displayed:

Insert disk B into drive 0, press return

Insert the NEVADA FORTRAN Data disk and press RETURN. The PIP program will read the first file from the disk. After a short period of time, the following prompt will be displayed:

Insert disk A into drive 0, press return

Insert the CPIM-NEVADA FORTRAN Operations disk and press **RETURN.** The PIP program will now write onto the disk. This process will continue until the entire NEVADA FORTRAN Data disk is copied onto the CPIM-NEVADA FORTRAN Operations disk. Upon completion, an asterisk will appear.

PIP can be terminated at any time by pressing RETURN after any asterisk (*) prompt. We suggest now placing your NEVADA FORTRAN Data disk in a safe place. You will not need it unless something happens to your Operations disk. Depending on how much program development you do, it may be wise to backup your CPIM-NEVADA FORTRAN Operations disk at least once a day.

NOTE: The NEVADA ASSEMBLER, ASSM.COM, must be on the NEVADA FORTRAN Operations disk.

GENERATING THE ERROR FILE FORT_ERR

In case a problem occurs in your program, error messages are very important to help you understand what the problem is. The program CONFIG reads the text file ERRORS which contains the compiler error messages. These messages may be changed but can only be one line long. Also, the first two characters are the error number followed by a blank and the text of the error message. To generate the error file, just enter CONFIG at the CPIM 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 for one disk systems). 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.

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 CPIM console *110* can be specified as either CPIM function 1 & 2, CPIM function 6 (for CPIM rev 2.0 only) or direct BIOS calls. Specifying CPIM 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 during program execution. Enter Y if it can or N if it cannot.

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

CREATING A PROGRAM

A program is created using the text editor ED.COM. The name of the FORTRAN source program should have the filename extension of .FOR, such as PROG.FOR. Refer to section 3 for a detailed description of the format of each source statement. After the program is created with the text editor, it is later, in a separate step, compiled.

RUNNING THE COMPILER

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

```
FORT U:PGM.LAO SOPTIONS
```

where:

FORT is FORT.COM, the FORTRAN compiler

PGM is the FORTRAN source program to compile and has the extension .FOR. However, the .FOR extension should not be included when specifying the program name in this command.

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 CPIM console
- Y listing to CPIM 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 usually deleted by the FORTRAN compiler.

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

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

The object program will have the same filename as the source file but with the extension .OBJ.

Notes:

If Z is 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 0 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.

COMPILE OPTIONS

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

FORT U:PGM.LAO $NP2

Here is a summary of Compile Options:

Option	Description
N	No assembly or object file will be produced.
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
H	Used in conjunction with the P option to suppress the heading in the listing.
C = XXXX	Specifies the maximum number of COMMON blocks that may be defined in the program to be compiled; default is 15.
B = XXXX	Specifies the size of the input statement buffer; default is 530 characters and the buffer must be large enough to contain a complete statement (first record plus all continuations).
M = XXXX	Specifies the memory address at which blank COMMON will end, i.e., blank COMMON will be allocated downward in memory from the specified address. The address specified must be in hexadecimal. This option is useful for forcing blank COMMON to be allocated at the same address in memory for passing data between routines that CHAIN to each other.

* 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 the statement). For example:

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

Produces: THIS IS A TEST
without blank padding and

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.

Here are some examples using Compile Options:

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 CPIM list (LST:) device.

FORT UPDATEXBZ $PH

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

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.FOR, the commands would be:

```
FORT GRAPH  (listing, object to default disk)
FRUN GRAPH  (will execute the program)
```

CREATING A COM FILE

A CPIM .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

THE FORTRAN CHARACTER SET

The FORTRAN character set is composed of the following characters:

The letters:

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

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 letters 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:

If used in a FORMAT statement, the character following the ampersand is interpreted as a control character (unless it is also an &).

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.

FORTRAN PROGRAM STRUCTURE

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

1) The first 5 columns of the statement are reserved for a statement label. This is a 1 to 5 digit number and is optional; however, if the statement is branched to from another part of the program, a label must be present.
2) The sixth column is used to indicate a continuation of the previous statement. Continuation is indicated by placing any character except a BLANK or ZERO in this column.
3) Columns 7 through 12 are used for the body of the statement. This is anyone 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) Columns 73 through 80 are used for identification purposes and are ignored.
5) A comment line is indicated by placing a C in column 1. A comment line has no effect on the program and is ignored. It is only used for documentation purposes.

Example:

```
Columns    0123456789
                 WRITE (1,2)
                 2 FORMAT ('THIS IS AN
                   * EXAMPLE CHARACTER STRING')
```

This outputs: 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 are in the H FORMAT specification in a FORMAT statement. Otherwise, they will be converted to uppercase internally. Thus, the variable QUANTITY, quantity, and QuAnTiTy present 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.

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 specifications. 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.

MULTI-STATEMENTS

Statements may be compacted with 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

```
   1= 1
3  CONTINUE
   I = 1+ 1
   TYPE I
   GOT03
   STOP
   END
```

could be written as:

```
   1= 1;3:CONTINUE;1 = 1+ 1;TYPE I;GOTO 3; STOP;END
```

FORTRAN PROGRAM PREPARATION

A FORTRAN source program is prepared using one of the available CPIM 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.

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
```


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 the 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.

Here is a summary of available options:

Options	**Descriptions**
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.
B	The FORTRAN source statement is included in the assembly file as a comment.
D = n	n 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 CON FIG 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 the Appendix for a list of error numbers and their meanings.
1= 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.
L = n	n 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.
N	Check for FORTRAN errors only. Do not output an assembly code file.

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 CON FIG 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.

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.

S = n — n 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 CON FIG program.

T = n — n 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.

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.

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.

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

4 NUMBER SYSTEM

The following section details how numbers are handled in NEVADA FORTRAN.

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 .10000000E-127 to 0.99999999E+ 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:

9 9	9 9	9 9	9 9	0 S	F F

BCD NUMBER :Sgn :Exp

NUMBER RANGES

Integer variables and constants can have any value from _ 99999999 to + 99999999. Real variables and constants can take any value between - 0.99999999E - 127 and 0.99999999E + 126. Integer variables and constants are stored internally in the same format.

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.

Here is a more detailed description of each type of constant:

Numerical Constants

Numerical constants can be either integer or real. For example:

Integer	1,3099, - 70
Real	1.34, - 5.98, 1.4E10

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
1=$1000
z= - $CCOO
```

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'

```
#ODOO
i =#127F
```

$805F is stored internally as: 328630000085
#805F is stored internally as: 5F 80 00 00 00 00

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.

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 are usually used as part of a logical expression.

Example

```
1= .TRUE.
1= (J .and. .TRUE.)
```


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 one to six alphanumeric characters the first of which must be a letter. There are four types of variables available: INTEGER, REAL, DOUBLE PRECISION and LOGICAL.

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 a letter 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.

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
```

A more detailed description of each Type Specification follows.

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
```

Logical

The general format of the LOGICAL statement is:

```
LOGICAL v1,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
```

Real

The general format of the REAL statement is:

```
REAL v1,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
```

Double Precision

The general format for DOUBLE PRECISION statement is:

```
DOUBLE PRECISION v1,v2
```

The DOUBLE PRECISION statement is used to explicitly override the default type of the variable. When a variable occurs 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 were REAL. A warning will be issued each time a DOUBLE PRECISION statement is encountered.

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/n1,n2 ....I,list1/n1,n21
```

where list is a list of variables (or array elements) to be initialized and n1, 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 AI11,BI1,2,31,CI3*01
DATA LIST/'THIS IS A CHARACTER STRING'I
```

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,CI1,2,31
```

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

```
DATA AI11,BI2I,CI31
```

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 Iname111ist1 Iname2/list2
```

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

Example

```
DIMENSION X(100)
COMMON IZZZ/ FIRST,LAST,X
CALL ADDEM


END

SUBROUTINE ADDEM
REAL NUMBER
COMMON IZZZ/ 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 CPIM 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 be different from any SUBROUTINE or FUNCTION name.

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 INTEGER, REAL, LOGICAL, or 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 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 overriden 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 were 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 to
.LE.	Less than or equal to

Logical Operators

.NOT.	Logial negation
.AND.	Logical and
.OR.	Logical or
.x0R.	Logical exclusive or

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

HIERARCHY OF OPERATORS

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

Highest	System and User Functions
	** OR (up arrow)
	* and /
	+ and - (including unary -)
	.LT., .LE., .NE., .EQ., .GE., .GT.
	.NOT.
	.AND.
	.OR. and .x0R.
Lowest	Replacement (=)

EXPRESSION EVALUATION

FORTRAN expressions are evaluated as follows:

1. Parenthesized 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*5 will be evaluated as:
((A+ (1 + (Z*5))

VAL *Z + (T + 4)16*X* *Y will be evaluated as:
((VAL*Z)+ (((T + 4)I6)* (X**Y)))

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

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. For example, 2.9 times 4.8 equals 13.92. However, in integer mode the result is 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

6/3=2
but 7/3 = 2 (NOTE: no fraction is retained) and 7/9 = 0
99999999+ 5 =? integer overflow

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

12/6.0 = 2.0
15.0/2 = 7.5
1.12. = 0.5

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

```
A = 1 .GT. 2 (FALSE)    A would evaluate to 0
A = 1 .EQ. 1 (TRUE)     A would evaluate to 1
A = 1 .LT. 2 (TRUE)     A would evaluate to 1
```

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

.LT.	Less Than
.LE.	Less Than or Equal To
.NE.	Not Equal To
.EQ.	Equal To
.GE.	Greater Than or Equal To
.GT.	Greater Than
.AND.	True only if both operands are true
.OR.	True if either operand is true
.xOR.	True if operands are different

Example

```
IF (A .EQ. B) GO TO 500
IF (A .EQ. B.OR.K .EQ. D) STOP
```

Logical variables can also be used in assignment statements:

```
A=A .AND. B
I= (A .OR. B).XOR.((T .EQ. 35.4).OR.(T .EQ. 39))
```


The following logical operators are also available, as listed in the following truth charts,

R=A .AND. B

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

R= A .OR. B

A	B	R
0	0	0
0	1	1
1	0	1
1	1	1

R=A .XOR. B

A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

R= .NOT. A

A	R
0	1
1	0

MIXED EXPRESSIONS

Here is a summary of the standard FORTRAN rules for mixed mode expressions.

Expression	Description
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	Causes truncation of any fractional part of real and an error if the truncated result is outside the range of integers
Real = Integer	Causes integer to be converted to real

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

Note: <op> represents one of the operators: + - / *

6 CONTROL STATEMENTS

Here are several different statements that control the execution flow of a FORTRAN program.

GO TO statements
- Unconditional GO TO
- Computed GO TO
- Assigned GO TO

IF statements
- Arithmetic IF
- Logical IF
- IF-THEN-ELSE

DO

CONTINUE

PAUSE

STOP

CALL

RETURN
- Explicit RETURN
- Multiple RETURN

UNCONDITIONAL GO TO STATEMENT

The general format of the unconditional GO TO is:

```
      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

```
      GO TO 10
      GO TO 400
10    CONTINUE
400   FORMAT (1X)
```


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:

Computed GO TO Statement	Present Value of Variable	Next Executed Statement
GO TO (1,5,98,167,4),K2	K2=5	4
GO TO (44,28),J	J = 1	44
GO TO (51,6,7,1,46),M	M=4	1
GO TO (1,1,1,1,2,2),LOOT	LOOT = 3	1

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.

ASSIGNED GO TO

The general format of the assigned GO TO is:

```
      GO TO v,(n1,n2, ...)
```

where v is the variable used in an ASSIGN statement and n1,n2 are statement labels.

Example

```
      GO TO LABL,(100,400,500)
      GO TO K,(1,2,3,4,5)
```

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)
```

ARITHMETIC IF STATEMENT

The Arithmetic IF allows the programmer to evaluate an expression which may be any combination of INTEGER, REAL, or LOGICAL operators. Based upon the expression's relationship to zero, control is transferred to one of three specified statements.

The general form of the Arithmetic IF is:

```
IF (e) 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:	Next Statement
<0	n1
=0	n2
>0	n3

Example

```
IF (A) 1,2,3
IF (BETA *SIN(BETAIDEGREE))100,150,432
IF (A-1)1,1,99
IF (.NOT. FLAG)1,5,7
```

LOGICAL IF STATEMENT

The general format of the Logical IF is:

```
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 evaluted 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

```
IF (DEGREE .EQ. 100)WRITE (1,*) RADIAN
IF ((A .EQ. 12).0R.(LOOP .LE. 500))RETURN
IF (SIN(30)/WHERE - .00005 .LT.. 00004)STOP
IF (A .NE. B)GO TO 500
IF (A .EQ. 1)GO TO (1,2,3),J
IF (VALUE .EQ. 6)IF (J)99,33,67
IF (I .GE. 500)J = 1+20/8
IF (FLAG) A = 2* A + 5
```

IF-THEN-ELSE

The general format of the IFTHEN-ELSE statement is:

```
IF (e) THEN
        statement 1
        statement 2

        ELSE
        statement 3
        statement 4

ENDIF
```

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

There can be more than one statement to execute if the IF is true.

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 IFTHEN-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:

```
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

```
IF (I .EQ. 0) THEN
                    L= K+ 1
                    K=I
                    ELSE
                    K=0
                    ENDIF
IF (J .LT. 7) THEN
                    LL= L+ 1
                    ELSE
                    IF (A .EQ. B)THEN
                          Q=0
                          D=N
                          ELSE
                          TYPE 'ERROR'
                          STOP
                          ENDIF
                    ENDIF
```


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:

The i is set to the value of m1.
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 stop 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 the 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 m1. The 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, IFTHEN-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 1=1,5
      DO 20 J=3,99


20    CONTINUE


10    CONTINUE
```

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 1=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.

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 labelled 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. Here is a summary:

Valued v	Runtime Error
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	CHAINILOAD error
12	Illegal FORTRAN logical unit number
13	Unit already open
14	Disk full
15	Unit not open
16	Binary 1/0 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
22	Assigned GOTO error
23	CONTROL/C abort
24	Illegal character input
25	File operation error
26	Seek error

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.

The ERRCLR statement clears the effect of the last executed ERRSET statement. The ERRCLR and ERRSET statements must be in the same routine.

Example

```
      OPTIONSQ

            ERRSET 10,CODE


            ERRCLR


            STOP
 10         TYPE 'ERROR ,ERROR CODE = ',CODE

            END
```

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:

If there is an ERRSET in effect, the error branch will be taken with a CONTROL/C error.
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 1= 1,100
     IF (I .EQ. 51)CTRL DISABLE
1    TYPE I
     END
```

The previous 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.

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 both trace statements is global over the whole program. Therefore, 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 ???? regardless of the state of the X option.

Example

```
     IF (FLAG .EQ. O)TRACE ON
     TRACE OFF

     DO 1 I= 1,100
     IF (I .EQ. 50)TRACE ON
1    TYPE I
```

DUMP STATEMENT

The general format of the DUMP statement is:

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

where ident is up to a ten 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 IAFTER-DIVI 'Index after divide is ',K



      A= K/I


      END
```

will cause the dump statement to output if I is zero.

7 PROGRAM TERMINATION STATEMENTS

The following section details the various program termination statements.

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

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 CPIM. 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 one to five digit integer number that will display. The 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. O)STOP 'ERROR'
IF (FLAG .AND. STOPIT)STOP 'ALL DONE'
```

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, 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. Also, 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.

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:

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.

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

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.

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

The number of dimensions specified for a particular array cannot exceed seven.

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 /DUMMYI 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 ABLE(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.

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 previously explained in the EXPRESSION section.

Example

```
ZIT(8)
A(1 + 2)
ORANGES(I + 5- (K*10)12)
ABLE(5)
```

9 SUBPROGRAMS

Subprograms provide a means to define frequently used sections of code that can be considered 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:

SUBROUTINE subprograms
FUNCTION subprograms
Built in library functions

The major differences between FUNCTIONS and SUBROUTINES are listed below:

FUNCTIONS are used in expressions, while SUBROUTINES must be CALLed.
FUNCTIONS require at least one parameter; SUBROUTINES do not require any.
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.

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
```


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=AI2
RETURN
END

DOUBLE PRECISION VALUE(WHA n
VALUE= WHATI4*7 + 5
RETURN
END
```

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)
CALLXSWAP
```

RETURN STATEMENT

There are two types of RETURN statements:

Normal RETURN
Multiple RETURN

Normal RETURN

The format for a normal RETURN is:

```
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)
1=0
J=0
RETURN
END

FUNCTION ZZ(VAL)
COMMON IAI A,B,C
ZZ= A+ BIC- VAL
RETURN
END
```

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
C THE FOLLOWING RETURN WILL TRANSFER TO THE
C STATEMENT LABELLED '1' IN THE CALLING PROGRAM.
C
      RETURN I

C
C THE FOLLOWING RETURN WILL TRANSFER TO THE
C STATEMENT LABELLED '2' IN THE CALLING PROGRAM.
C
      RETURN J
      END
```

NOTE: Multiple RETURNS are only valid for SUBROUTINES.

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 lONE! NAMES(100) /TWOI FIRST,LAST
DATA FIRST 11/, LASTI101
DATA NAMES 11,2.0,4,5,6,7,8,9,10,90*999991
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

The following is some information concerning NEVADA FORTRAN 110_

GENERAL INFORMATION

Input/Output *(1/0)* under FORTRAN may take one of the following forms:

Standard Formatted *110*
Free Format *1/0*
Binary *1/0*

In standard formatted *1/0,* 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 *1/0* is used as in BASIC. All the values are entered using commas (,) or carriage returns to delimit the numbers.

Binary 110 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 to 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 110 cannot be specified for logical units 0 or 1 and dOing so will result in a runtime error.

There are two special *1/0* statements:

TYPE
ACCEPT

Both of these are followed by a standard 110 list. TYPE is equivalent to WRITE (1,*) and ACCEPT to READ (0,*). This is just a convenient method of dOing console *1/0.*

Example

```
      TYPE I,J,(A,(I),I= 1,10)
      ACCEPT 'INPUT THE MAX COUNT',COUNT
```

A RUNTIME FORMAT can be specified for any formatted *1/0* 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
```

110 LIST SPECIFICATION

The 110 List is used to specify which variables are to be READ or WRITTEN in a particular *1/0* 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:

Simple (non-subscripted) variable
Array element
Array name
Implied DO loop
Literal
Constant (WRITE only)

The above types are combined to form the *1/0* list specification. 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 DO loops can also 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.

READ STATEMENT

READ(unit{ ,format} {,END = end} {,ERR = error}) *1/0* 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 one as the logical unit. 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:

The FORMAT number is the label on the FORMAT statement which is to be used,
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.
Binary input is assumed if the FORMAT entry is left blank (or not specified).
If an array name is specified, that array 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 a decimal point in an integer field). Use ERRSET to handle these input errors. *1/0* 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. 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. If an error occurs, go to statement label 100.

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

WRITE STATEMENT

WRITE (unit{,format} {,END = end} {,ERR = error}) *1/0* 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 *1/0* list is specified exactly the same as for the READ statement with the exception that a string can always be used in the *1/0* 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) (1,I= 1,10)
WRITE (10,*) THIS
WRITE (6,12,END = 99,ERR = 66) LOOP,COUNT
```

MEMORY TO MEMORY 110 STATEMENTS

The ENCODE and DECODE statements allow *1/0* 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. However, with ENCODE, instead of being output to a file, data will be placed in memory at the specified location for further processing.

DECODE statement

The general form of the DECODE statement is:

DECODE (variable,length,format) *1/0* 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 *1/0* 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 (110,13,F10.5)
```


ENCODE statement

The general form of the ENCODE statement is:

ENCODE (variable,length,format) *1/0* 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, 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, *) (1,I= 1,5)
```

FORMAT STATEMENT AND FORMAT SPECIFICATIONS

The general form of the FORMAT statement is:

n FORMAT (s1,s2, ...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 *1/0* power. On FORMATTED input, blanks are treated as zeros except when reading in A format. A constant enclosed in backslashes (i.e. \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 FORTRAN will print a negative number even when there is not enough room to place the negative sign in the field. The negative sign will simply be omitted. In this case, NEVADA FORTRAN will fill the field with asterisks. Using asterisks to fill 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.

A-Type (Aw)

The A-Type specification is used to perform the input of alphanumeric data in ASCII character form. Up to six 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 (A10,110,A6)
```

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.

E-Type (Ew.d)

The E-Type specification is another method of performing *1/0* with floating pOint (real) numbers. It is through this specification that the programmer may perform *1/0* using an exponential format. That is a mantissa followed by an exponent of ten. 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 digit exponent. The d part cannot be zero for output.

Example

Output

E9.2	O.OOE+ 000
E9.2	0.12E + 004
E10.0	invalid

Input

E10.0	1000.
E9.2	1.23E + 004
E9.2	O.

NOTE: Data can be read in the F format using the E or D format specification without causing an error.

When the E format is used for input, the data must be right justified in the field. If it is not, then the blanks appearing in the exponent field will be interpreted as zeros.

F-Type (Fw.d)

The F-Type specification is one of several specifications for performing *1/0* 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

Output

F4.1	32.2
F7.5	0.00001
F3.0	7.
F7.2	bbb4.50

Input

F7.2	b4.5bbb
F2.1	70
F7.5	bbbb001
F4.1	32.2

NOTE: The b is used to indicate a blank position. During input, the F field specifier reads w characters. If a decimal point is not read in the field, a decimal is inserted d digits from the right. A decimal point in the input field overrides the field specifications.

GType (Gw.d)

The G-Type can be used on either input or output and for both integer and real values where wand 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 <= number < 10**d, or

F(w - 5).d,5X	if .1 <= number < 1
F(w - 5).(d -1),5X	if 1 <= number < 10
F(w - 5).1,5X	if 10**(d - 2) <= number < 10**(d -1)
F(w - 5).0,5X	if 10**(d -1) <= number < 10**d

In general, in this range:

F(w - 5).(d - (exponent of number)),5X

Input

If the input element is of type integer: Iw, then the format code used will be Iw.

If the input element is of type real: Ew.d, then the format code used is Ew.d.

Example

```
      A=5.67
      WRITE (1,34) A
34    FORMAT (G10.5)
      READ (0,9) A
9     FORMAT (G9.3)
```

I-Type (Iw)

The I-Type specification is used as a method of performing *1/0* 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 (10110)
```


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

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

will output the line:

100000000081

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 aT or F is found. The T or F can be located anywhere in the field and all characters that follow the Tor F in the remainder of the field are ignored. If the first character found is not a Tor F, an error will be generated. If the input field is completely blank, then a .FALSE. value will be used.

Example

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

T-Type (Tw)

The T-Type code can be used on both input and output. It is used to move to an 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

```
     WRITE (1,56) I,LOT
56   FORMAT (110,T50,14)

     J = 1234
     WRITE (1,34) J
34   FORMAT (,$$$$$$$$$$',T5,14)
```

will produce:

```
$$$$$1234$
```

X-Type (wX)

The X-Type specification is used to space over any number of columns with a maximum of 255 character positions. The 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,110,3X,15)
99   FORMAT (1X,'THIS IS A LITERAL',5X,'$$$$')
```

Z-Type

Only used for output, the Z-Type specification indicates 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
```

I-TYPE (I)

The I-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 (1101)

     WRITE (1,100) 1,20,45
100  FORMAT (13/213)
```

will generate

```
 1
20   45
```


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 4110 is the same as 110,110,110,110The following FORMATS are equivalent:

```
10   FORMAT (314,2F10.4)
10   FORMAT (14,14,14,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 (14,2(13,F4.1))
19   FORMAT (14,13,F4.1,13,F4.1)
```

The FORMATS:

```
10   FORMAT (15,2(13,F5.1))
10   FORMAT (15,13,F5.1,13,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 (151(F10.5))
```

The 15specification 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(15,3(110))            is ok, while
10   FORMAT (2(15,3(110,2(11)))      is not legal.
```

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 CON FIG 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

```
     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!

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

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

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

generates: This is an ampersand: &

FREE FORMAT 110

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 *1/0* list specifies more variables than there are in an input record, succeeding records will be read until the last 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.

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 (0,*) A,B,C

TYPE THE RESULTING VALUE IS ',VALUE
WRITE (0,*) THE RESULTING VALUE IS',VALUE
TYPE '1 + 1 = ',2
```


Free format *1/0* 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 110 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)
```

BINARY 110

BINARY *1/0* 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 *1/0* list. WRITE causes the item in the *1/0* 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 *1/0* 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 six bytes from the file specified directly to the variable in the *1/0* 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.

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 INFILE
REWIND OUTF
```

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.

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, ENDFILE is equivalent to closing the file.

Example

```
ENDFILE 4
ENDFILE FILE
```


GENERAL COMMENTS ON FORTRAN 110 UNDER CP/M

The OPEN or LOPEN subroutine is used to associate a file with a FORTRAN logical unit. Eight files are available, numbers 0 through 7 with 0 being permanently open and associated with input from the CPIM console, logical file 1 also is permanently open and associated with output to the CPIM console. Logical files 0 and 1 cannot be opened or closed. Additionally, any logical unit associated with the CPIM console (through the use of the filename CON:) cannot have binary *110* 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 first 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 *1/0* wi II 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 CPIM end-of-file indicator (control-Z) if the file specified in the ENDFILE has been written to and no binary *1/0* was done to the file. If binary *1/0* has been done to the file, then an end-of-file of six 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 to 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.

SPECIAL CHARACTERS DURING CONSOLE 110

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

End-of-file from the CPIM console is indicated by a CONTROL-Z being entered as the first character of an input line during console *1/0.*

Entering a DELETE (7F hex) or CONTROL-H will erase the last character entered.

11 GENERAL PURPOSE SUBROUTINE/FUNCTION LIBRARY

The following list of FORTRAN subroutines and functions are available:

SUBROUTINE Name

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

FUNCTION Name

CALL
CBTOF
CHAR
COMP
INP
PEEK

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 CPIM error should occur, return will be to the statement following the call and error will contain the appropriate error code as listed below. If an 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 be returned through the optional error parameter.

0= OK
1 = Specified file not found
2 = Disk is full
3 = End-of-file encountered
4 = New filename for RENAME already exists
5 = Seek error
6 = Seek error (but file is closed)
7 = Format error in CHAIN or LOAD file

AVAILABLE FORTRAN SUBROUTINES

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,O,'S')
CALL  BIT (ZAPIT,O,'T',VALUE)
```

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 the program name specified does not exist, and an 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 assummed 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)
```

CIN

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,#ODOO,1) .NE. O)GO TO 80
```

In the above example, #DOOOmust 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 (OD) to be placed in the left most byte for the COMP routine.

CLOSE

CALL CLOSE(unit)

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

Example

```
CALL CLOSE(3)
CALL CLOSE (FILE)
```


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 (OAH) BEFORE CONTINUING.
        ARAND= .3478
  10    ARAND = RAND (ARAND)
        CALL CTEST (STATUS)
        IF (STATUS .EQ. O)GO TO 10
C
C   CHARACTER HIT, READ IT
C
        CALL CIN(CHAR)
        IF (COMP(CHAR,#OAOO,1) .NE. O)GO TO 10
```

DELAY

CALL DELAY(wait time)

The DELAY routine enables the user to implement a time DELAY of 11100 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 11100 less than O. This time is based on a 2 MHZ 8080 processor.

Example

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

DELETE

CALL DELETE ('file' {,error})

The DELETE routine is used by the FORTRAN user to remove a file from the CPIM 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('OUTFI LE',ERROR)
CALL DELETE (FILE)
```

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
```

LOAD

CALL LOAD('file to 10ad',load-type{ ,error})

The LOAD routine is used to load either a standard CPIM .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 routine package occupies memory from 100H to 4000H. If the file to load does not exist and an 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',O)
CALL LOAD ('ASMOBJ',1,ERROR)
```


LOPEN

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

This subroutine is functionally the same as OPEN in that it associates a FORTRAN unit with a CPIM 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:

First Character	Action
+	overprint the last record
blank (space)	single skip
0	double skip
-	triple skip
1	page eject

If none of the above characters are 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 (OCH).

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

ODH (carriage return)	-	return to beginning of this line
OAH (line feed)	-	space 1 line, do not return to beginning of line
OCH (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 (,1THIS SHOULD BE ON THE TOP OF A NEW
      PAGE')
C
      WRITE (2,2)
2     FORMAT ('OONE BLANK LINE ABOVE THIS ONE'I
 *    '+ ','THIS LINE WILL OVERPRINT THE ONE ABOVE'I
 *    '- ',5X,'THIS LINE WILL HAVE 2 LINES ABOVE IT')
      STOP
      END
```

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,$CCOO,- 1)
This MOVES 2 bytes from the address specified by A to address CCOO(HEX).

CALL MOVE(6,'STRING',0,$CCOO, - 1)
This MOVES 6 bytes of the string 'STRING' to address CCOO(HEX).

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

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

OPEN

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

The OPEN routine is used to open a CPIM file the user may wish to access. The unit and file are required entries. If the CPIM 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 CPIM error code will be returned and the file will not be opened.

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

CON: Used to specify either CPIM console input or output

LST: Used to specify CPIM 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,*) 'INPUT QUANTITY ',QUANT
```

Output can be directed to the CPIM LST device by opening the file LST: as in the following example:

```
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.O)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:

13 characters are encountered
A NULL is encountered

OUT

CALL OUT (port,value)

This routine allows access to the *8080180851Z80* 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)
```

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.

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',O))    will output an A
```

RENAME

```
CALL RENAME('old file','new file',{ error})
```

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 (2A612A6)
      CALL RENAME (OFILE,NFILE,ERROR)
```

RESET

```
CALL RESET
```

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

Example

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

SEEK

```
CALL SEEK (unit,position {,error} )
```

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 five indicates a seek to a part of the file that doesn't exist, and a six indicates a seek to an extent of the file that does not exist. The difference between the two is that if error code six 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)
```

SETIO

CALL SETIO(new 1/0)

This routine allows changing how the runtime package performs console 1/0. The default method is setup using the CONFIG program, however it can be changed as follows:

new 1/0 = 0 to use direct BIOS 1/0
new 1/0 = 2 to use CPIM function 1&2
new 110 <> 0 or 2 to use CPIM function 6 (should be used with CPIM 2.x only).

The use of CPIM functions 1&2 permits the use of the control-p ability of CPIM 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)
```

Available FORTRAN Functions

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 ($DEOO,VALUE)
```


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 a 16 bit value (stored 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 CPIM BIOS jump table by reading the 16 bit address at location 0001 and subtracting 3 from it.

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,O)        returns 61
B = CHAR(A,1)        returns 62
B = CHAR(A,5)        returns 66
```

COMP

A = COM P(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

INP

A = INP(port)

This routine allows access to the *8080180851Z80* 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

```
     1= INP(10)
10   IF (INP(CONSOLE) .NE. O)GO TO 10
     VALID = INP(CLOCK) .AND. 48
```

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.

NEVADA ASSEMBLER™

User's Reference Manual

12 INTRODUCTION TO NEVADA ASSEMBLER

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 seven 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 **CTRL** 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.

13 OPERATING PROCEDURES

HARDWARE REQUIREMENTS

Your Commodore 64 Computer

The Commodore Z80 microprocessor

A Commodore 1541 single disk drive

A video display monitor such as the Commodore color monitor model 1701/1702

SOFTWARE REQUIREMENTS

Commodore's CP/M Operating System disk

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

Refer to the Getting Started section in your Commodore 64 NEVADA FORTRAN manual.

EXECUTING THE ASSEMBLER

The assembler is invoked by a CP/M command with the following formats.

FORMAT-1:

ASSM file <CR>

FORMAT-2:

ASSM file[.uuu u$#LP0] <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.

<$options>
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) begins 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>
NEVADA ASSEMBLER (C) COPYRIGHT 1982
ELLIS COMPUTING, INC.
REV 2.1 ASSEMBLING

NO ASSEMBLY ERRORS.      4 LABELS WERE DEFINED.
A>
```

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 (assembled with proper origin), 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 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 returned to CP/M.

```
NO ASSEMBLY ERRORS.        4 LABELS WERE DEFINED.
A>
```

The assembly process can be interrupted at any time by pressing the CTRL and C keys.

14 STATEMENTS

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 (usng 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 asembler 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, Hand L. SP stands for the stack pointer, while M refers to 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 $
TABLE DB ENTRY
*
*

LAST EQU $
TABLN EQU LAST-FIRST
```

The label FIRST is set to the address 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 correct, even if later additions or 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 0 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:

MVI A,128	Move 128 decimal to register A.
MVI C,10D	Move 10 decimal to register C.
LXI H,2FH	Move 2F hexadecimal to registers HL.
MVI B,303Q	Move 303 octal to register B.
MVIA,'Y'	Move the ASCII value for Y to register A.
MVI A,101B	Move 101 binary to register A.
JMP OFFH	Jump to address FF hexadecimal.

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:

```
MVI  A,255D/10H- 5
LDA  POTTS/256*OFFSET
LXI  SP,30*2+ STACK
```

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 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      AN I 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
```


15 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 <label> 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 [<label>] ORG <expression>

This pseudo-operation sets the asembler'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:

```
        ORG 4000H     Determines that the object code generated
*                     by subsequent statements must be loaded
*                     in locations beginning at 4000H.
START   ORG 100H      Determines that the object code generated
*                     by subsequent statements must be loaded
*                     in locations beginning at 100H.
```

Set Execution Address XEQ < 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 XEQ pseudo-operation, the object code file will contain a starting address of 100H. If more than one 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:

```
        XEQ 100H      The entry point address for the assembled
*                     program is set to 100H.
```

Define Storage [<label>] DS <expression>
[< label>] RES < 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:

```
    SPEED DS 1      Reserves one byte.
      DS400         Reserves 400 bytes.
      RES 177Q      Reserves 177 (octal) bytes.
```

Define byte [< 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:

```
    DB1          one byte is defined.
  DB OFFH,303Q,100D,11010011B,3*BELL,-10          multiple bytes.
  TABLE DB 'A','B','C','D',0          multiple bytes are defined.
```


Define Word [< 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 [<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 [<label>] ASC #<ASCII string>#
[< label>] ASCZ #< ASCII string >#

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 ASCF0
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 IF <expression>

source code

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        Sets the value of the label 'YES' to 1.
NO EQU 0         Sets the value of the label 'NO' to 0.
*
 IFYES           The expression here is true (1), so the
 MVIA,'Y'        code on this line will be assembled.
 IF NO           The expression here is false (0), so the
 MVIA, 'N'       code on this line will not be assembled.
 ENDF            This terminates the NO conditional.
 ENDF            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.

16 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 FILE 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 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:

A ARGUMENT ERROR — An illegal label or constant appears in the operand field. This might be
- a number with a letter in it, i.e. 2L
- a label that starts with a number, i.e.,3STOP
- an improper representation of a string, i.e. "'A"' in the operand field of a statement containing the ASCII pseudo-operation.

D DUPLICATE LABEL — The source code contains multiple labels whose first five characters are identical.

L LABEL ERROR — The symbol in the label field contains illegal characters, e.g., it starts with a number.

M MISSING LABEL — An EQU instruction does not have a symbol in the label field.

O OPCODE ERROR — The symbol in the operation field is not a valid 8080 instuction mnemonic or an assembler pseudo-operation mnemonic.

R REGISTER ERROR — An expression used as a register designator does not have a legal value.

S SYNTAX ERROR — A statement is not in the format required by the assembler.

U 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.

V VALUE ERROR — The value of the operand lies outside the allowed range.

APPENDIX A - STATEMENT SUMMARY

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 = n1 ,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 labelled 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.

GOTO n

Transfer control to the statement labelled n.

GO TO v,(n1,n2,_._),v

The COMPUTED GOTO transfers control to n1 if v= 1, n2 if v= 2, etc.

GO TO v,(n1,n2, ..)

The ASSIGNED GOTO transfers control to statement n1, n2,.. depending on the value of v. V must have appeared in an ASSIG N statement.

IF (e)n1,n2,n3

The arithmetic IF transfers control to n1 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.

STOPn
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.

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

Write (unit,format {,ERR = }) output list
Writes the values of the variable in output list to unit according to format.

APPENDIX B - SUMMARY OF SYSTEM FUNCTIONS

Name	Function	Arg	Result	Argument
ABS	Absolute (x)	1	real	real
ALOG	Log Base e (x)	1	real	real
ALOG10	Log base 10 (x)	1	real	real
AMAX0	Maximum	<255	either	either
AMAX1	Maximum	<255	either	either
AMINO	Minimum	<255	either	either
AMIN1	Minimum	<255	either	either
AMOD	Remainder (x/y)	2	real	real
ATAN	Arctangent(x)	1	real	real
ATAN2	Arctangent(y/x)	2	real	real
BIT	Bit handling	3/4	either	either
CALL	Execute asm pgm	2	either	either
CBTOF	Convert to real	2/3	real	both
CHAR	Character value	1	either	character
COMP	Compare strings	3	either	either
COS	Cosine(x)	1	real	real
DIM	Positive difference	2	real	real
EXP	e**(x)	1	real	real
FLOAT	Make real (x)	1	real	integer
IABS	Absolute (x)	1	integer	integer
101M	Positive difference	2	real	real
IFIX	Truncate (x)	1	integer	real
INP	Input from a port	1	either	either
ISIGN	Transfer of sign	2	integer	integer
MAX0	Maximum	< 255	either	either
MAX1	Maximum	<255	either	either
MIN0	Minimum	< 255	either	either
MIN1	Minimum	<255	either	either
MOD	Remainder (x/y)	2	integer	integer
PEEK	Examine mem loc.	1	either	either
RAND	Random Number (x)	1	real	real 0.0<R<1.0
SIGN	Transfer of sign	2	real	real
SIN	Sine(x)	1	real	real
SQRT	Square Root (x)	1	real	real
TAN	Tangent(x)	1	real	real

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.	**Value returned for RAND**
0	The next entry in the table
-1	The first entry in the table. Also, the pointer for the next entry (arg = 0) is reset to the second entry in the table.
n	Returns the table entry following n.

APPENDIX C - SUMMARY OF SYSTEM SUBROUTINES

CALL BIT (variable,disp,code)
Set, resets or flips bit 0+ disp of variable according to the code.

CALL CBTOF(loc1,disp1,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 1/100ths of a second.

CALL DELETE('filename'{,error})
Delete the specified filename from the disk.

CALL EXIT
Terminates program execution.

CALL MOVE(n,loc1 ,disp1 ,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)
The value is converted to an 8 bit number and output to 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 disk at runtime and then being able to write on the changed disk. Without using this function, an attempt to write on a disk 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 110)
Allows changing the way that console 110 is performed during program execution.

APPENDIX 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. YYYYH

pgm was executing line LLLL in routine NNNN

where: XXXXXXXX is the ERROR, YYYY 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.

SUMMARY OF RUNTIME ERRORS

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

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.

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).

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.

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

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 were specified.

CON BIN
BINARY 110 TO CONSOLE: binary 110 is not supported to the system console.

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

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 110 (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.

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

DSK FULL
DISK FULL: either the disk is full or the directory is full.

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.

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.

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

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

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.

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

110 ERR
110 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.

110 LIST
INVALID 110 LIST: this error indicates an error in the I/Olist specification of a formatted WRITE or READ. This error will not normally occur.

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.

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

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

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.

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

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

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

APPENDIX 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* poped off bottom of operand stack
37 expecting end of statement, not found
38 statement too complex; increase P and/or 0 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
50 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 ERRSETIERRCLR
67 Hex constant of zero (0) invalid in *1/0* statement
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
60 Hex constant of 27H (') invalid in FORMAT
6E Invalid number following STOP or PAUSE
6F invalid TRACE statement (operand not ONIOFF)
70 invalid 10STAT = 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 ENDIF
78 invalid ELSE or ENDIF
79 missing ENDIF
7A initialization of non-COMMON variable
7B "DOUBLE PRECISION" not supported, treated as "REAL"
7C UNUSED
70 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 0=
8C *FATAL* stack overflow (compiler error)
80 *FATAL* stack overflow (compiler eror)
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 1=
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

APPENDIX 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 ALWAY 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
```


APPENDIX 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 six characters will be retained for any variable and if less than six, will be zero filled in the low order bytes of the variable.

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

   - the first 13 characters,
   - 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=$E060
   A= -$CCOO

   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 # 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. Eight FORTRAN files may be open at anyone time (file numbers 0-7). Remember that files 0 and 1 are permanently open.

APPENDIX H - 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 *1/0.*
19. Multiple returns from subroutines.
20. K format specification.

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,CI1 *,2/31*

5. The P format specifications.
6. Statement functions.
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.

APPENDIX 1- 8080 Operation Code

JUMP

C3	JMP	
C2	JNZ	
CA	JZ	
D2	JNC	
DA	JC	Adr
E2	JPO	
EA	JPE	
F2	JP	
FA	JM	
E9	PCHL	

CALL

CD	CALL	
C4	CNZ	
CC	CZ	
D4	CNC	
DC	CC	Adr
E4	CPO	
EC	CPE	
F4	CP	
FC	CM	

RETURN

C9	RET
CO	RNZ
C8	RZ
DO	RNC
D8	RC
EO	RPO
E8	RPE
FO	RP
F8	RM

RESTART

C7	RST 0
CF	RST 1
D7	RST 2
DF	RST 3
E7	RST 4
EF	RST 5
F7	RST 6
FF	RST 7

MOVE IMMEDIATE

06	MVI B,	
OE	MVI C,	
16	MVI D,	
1E	MVI E,	D8
26	MVI H,	
2E	MVI L,	
36	MVI M,	
3E	MVI A,	

Acc IMMEDIATE*

C6	ADI	
CE	ACI	
D6	SUI	
DE	SBI	
E6	ANI	D·8
EE	XRI	
F6	ORI	
FE	CPI	

LOAD IMMEDIATE

01	LXI B	
11	LXI D,	
21	LXI H,	D16
31	LXI SP,	

DOUBLEADDt

09	DAD B
19	DAD D
29	DAD H
39	DAD SP

STACK OPS

C5	PUSH B
D5	PUSH D
E5	PUSH H
F5	PUSH PSW
C1	POP B
D1	POP D
E1	POPH
F1	POP PSW*
E3	XTHL
F9	SPHL

INCREMENT* *

04	INR B
OC	INR C
14	INR D
1C	INR E
24	INR H
2C	INR L
34	INR M
3C	INR A
03	INX B
13	INX D
23	INX H
33	INX SP

DECREMENT* *

05	DCR B
OD	DCR C
15	DCR D
10	DCR E
25	DCR H
2D	DCR L
35	DCR M
3D	DCR A
OB	DCX B
1B	DCX D
2B	DCX H
3B	DCX SP

LOAD/STORE

OA	LDAX B	
1A	LDAX D	
2A	LHLD	Adr
3A	LDA	Adr
02	STAX B	
12	STAX D	
22	SHLD	Adr
32	STA	Adr

SPECIALS

EB	XCHG
27	DAA*
2F	CMA
37	STC t
3F	CMCt

INPUT/OUTPUT

D3	OUT D8
DB	IN D8

D8 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.

t = only CARRY affected

** = all Flags except CARRY affected; (exception: INX and DCX affect no Flags)

ROTATEt

07	RLC
OF	RRC
17	RAL
1F	RAR

CONTROL

00	NOP
76	HLT
F3	DI
FB	EI

MOVE

40	MOV B,B
41	MOV B,C
42	MOV B,D
43	MOV B,E
44	MOV B,H
45	MOV B,L
46	MOV B,M
47	MOV B,A
48	MOV C,B
49	MOV C,C
4A	MOV C,D
4B	MOV C,E
4C	MOV C,H
4D	MOV C,L
4E	MOV C,M
4F	MOV C,A
50	MOV D,B
51	MOV D,C
52	MOV D,D
53	MOV D,E
54	MOV D,H
55	MOV D,L
56	MOV D,M
57	MOV D,A

MOVE (Cont.)

58	MOV E,B
59	MOV E,C
5A	MOV E,D
5B	MOV E,E
5C	MOV E,H
5D	MOV E,L
5E	MOV E,M
5F	MOV E,A
60	MOV H,B
61	MOV H,C
62	MOV H,D
63	MOV H,E
64	MOV H,H
65	MOV H,L
66	MOV H,M
67	MOV H,A
68	MOV L,B
69	MOV L,C
6A	MOV L,D
6B	MOV L,E
6C	MOV L,H
6D	MOV L,L
6E	MOV L,M
6F	MOV L,A
70	MOV M,B
71	MOV M,C
72	MOV M,D
73	MOV M,E
74	MOV M,H
75	MOV M,L

77	MOV M,A
78	MOV A,B
79	MOV A,C
7A	MOV A,D
7B	MOV A,E
7C	MOV A,H
7D	MOV A,L
7E	MOV A,M
7F	MOV A,A

ACCUMULATOR*

80	ADD B
81	ADD C
82	ADD D
83	ADD E
84	ADD H
85	ADD L
86	ADD M
87	ADD A
88	ADC B
89	ADC C
8A	ADC D
8B	ADC E
8C	ADC H
8D	ADC L
8E	ADC M
8F	ADC A
90	SUB B
91	SUB C
92	SUB D
93	SUB E
94	SUB H
95	SUB L
96	SUB M
97	SUB A
98	SBB B
99	SBB C
9A	SBB D
9B	SBB E
9C	SBB H
9D	SBB L
9E	SBB M
9F	SBB A
AO	ANA B
A1	ANA C
A2	ANA D
A3	ANA E
A4	ANA H
A5	ANA L
A6	ANA M
A7	ANA A

A8	XRA B
A9	XRA C
AA	XRA D
AB	XRA E
AC	XRA H
AD	XRA L
AE	XRA M
AF	XRA A
BO	ORA B
B1	ORA C
B2	ORA D
B3	ORA E
B4	ORA H
B5	ORA L
B6	ORA M
B7	ORA A
B8	CMP B
B9	CMP C
BA	CMP D
BB	CMP E
BC	CMP H
BD	CMP L
BE	CMP M
BF	CMP A

PSEUDO INSTRUCTION

ORG	Adr
END	
Eau	D16
DS	D16
DB	D8
DW	D16

Adr = 16 bit address

00	NOP	
01	LXI	B,D16
02	STAX	B
03	INX	B
04	INR	B
05	DCR	B
06	MVI	B,D8
07	RLC	
08	- - -	
09	DAD	B
OA	LDAX	B
OB	OCX	B
OC	INR	C
00	OCR	C
OE	MVI	C,D8
OF	RRC	
10	- - -	
11	LXI	0,D16
12	STAX	0
13	INX	D
14	INR	D
15	DCR	D
16	MVI	0,08
17	RAL	
18	-	
19	DAD	D
1A	LOAX	D
1B	DCX	0
1C	INR	E
10	DCR	E
1E	MVI	E,D8
1F	RAR	
20	-	
21	LXI	H,D16
22	SHLD	Adr
23	.INX	H
24	INR	H
25	OCR	H
26	MVI	H,08
27	DAA	

28	- - -	
29	DAD	H
2A	LHLD	Adr
2B	DCX	H
2C	INR	L
2D	DCR	L
2E	MVI	L,D8
2F	CMA	
30	-	
31	LXI	SP,D16
32	STA	Adr
33	INX	SP
34	INR	M
35	OCR	M
36	MVI	M,D8
37	STC	
38	- - -	
39	DAD	SP
3A	LDA	Adr
3B	DCX	SP
3C	INR	A
3D	OCR	A
3E	MVI	A,D8
3F	CMC	
40	MOV	B,B
41	MOV	B,C
42	MOV	B,O
43	MOV	B,E
44	MOV	B,H
45	MOV	B,L
46	MOV	B,M
47	MOV	B,A
48	MOV	C,B
49	MOV	C,C
4A	MOV	C,D
4B	MOV	C,E
4C	MOV	C,H
40	MOV	C,L
4E	MOV	C,M
4F	MOV	C,A

50	MOV	D,B
51	MOV	D,C
52	MOV	D,D
53	MOV	D,E
54	MOV	D,H
55	MOV	D,L
56	MOV	D,M
57	MOV	D,A
58	MOV	E,B
59	MOV	E,C
5A	MOV	E,O
5B	MOV	E,E
5C	MOV	E,H
50	MOV	E,L
5E	MOV	E,M
5F	MOV	E,A
60	MOV	H,B
61	MOV	H,C
62	MOV	H,O
63	MOV	H,E
64	MOV	H,H
65	MOV	H,L
66	MOV	H,M
67	MOV	H,A
68	MOV	L,B
69	MOV	L,C
6A	MOV	L,D
6B	MOV	L,E
6C	MOV	L,H
60	MOV	L,L
6E	MOV	L,M
6F	MOV	L,A
70	MOV	M,B
71	MOV	M,C
72	MOV	M,O
73	MOV	M,E
74	MOV	M,H
75	MOV	M,L
76	HLT	
77	MOV	M,A
78	MOV	A,B

79	MOV	A,C
7A	MOV	A,D
7B	MOV	A,E
7C	MOV	A,H
7D	MOV	A,L
7E	MOV	A,M
7F	MOV	A,A
80	ADD	B
81	ADD	C
82	ADD	D
83	ADD	E
84	ADD	H
85	ADD	L
86	ADD	M
87	ADD	A
88	AOC	B
89	AOC	C
88	ADC	B
89	ADC	C
8A	AOC	0
8B	AOC	E
8C	AOC	H
80	AOC	L
8E	ADC	M
8F	AOC	A
90	SUB	B
91	SUB	C
92	SUB	0
93	SUB	E
94	SUB	H
95	SUB	L
96	SUB	M
97	SUB	A
98	SBB	B
99	SBB	C
9A	SBB	0
9B	SBB	E
9C	SBB	H
90	SBB	L
9E	SBB	M
9F	SBB	A

D8 = 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.

AO	ANA	B
A1	ANA	C
A2	ANA	D
A3	ANA	E
A4	ANA	H
A5	ANA	L
A6	ANA	M
A7	ANA	A
A8	XRA	B
A9	XRA	C
AA	XRA	D
AB	XRA	E
AC	XRA	H
AD	XRA	L
AE	XRA	M
AF	XRA	A
BO	ORA	B
B1	ORA	C
B2	ORA	D
B3	ORA	E
B4	ORA	H
B5	ORA	L
B6	ORA	M
B7	ORA	A
B8	CMP	B
B9	CMP	C
BA	CMP	0
BB	CMP	E
BC	CMP	H
BD	CMP	L
BE	CMP	M
BF	CMP	A
CO	RNZ	
C1	POP	B
C2	JNZ	Adr
C3	JMP	Adr
C4	CNZ	Adr
C5	PUSH	B
C6	ADI	D8
C7	RST	0

C8	RZ	
C9	RET	
CA	JZ	
CB	- - -	
CC	CZ	Adr
CD	CALL	Adr
CE	ACI	D8
CF	RST	1
DO	RNC	
D1	POP	D
02	JNC	Adr
D3	OUT	D8
04	CNC	Adr
D5	PUSH	D
D6	SUI	D8
07	RST	2
D8	RC	
D9	- - -	
DA	JC	Adr
DB	IN	D8
DC	CC	Adr
DD	- - -	
DE	SBI	08
DF	RST	3
EO	RPO	
E1	POP	H
E2	JPO	Adr
E3	XTHL	
E4	CPO	Adr
E5	PUSH	H
E6	ANI	D8
E7	RST	4
E8	RPE	
E9	PCHL	
EA	JPE	Adr
EB	XCHG	
EC	CPE	Adr
ED	- - -	
EE	XRI	08
EF	RST	5

FO	RP	
F1	POP	PSW
F2	JP	Adr
F3	DI	
F4	CP	Adr
F5	PUSH	PSW
F6	ORI	D8
F7	RST	6
F8	RM	
F9	SPHL	
FA	JM	Adr
FB	EI	
FC	CM	Adr
FD	- - -	
FE	CPI	08
FF	RST	7

HEX·ASCII **TABLE**

Non·Printing

00	NULL
07	BELL
09	TAB
OA	LF
OB	VT
OC	FORM
00	CR
11	X·ON
12	TAPE
13	X·OFF
14	
1B	ESC
7D	ALT MODE
7F	RUB OUT

Adr = 16 bit address

APPENDIX J - TABLE OF ASCII CODES (Zero Parity)

Upper Octal	Octal	Decimal	Hex	Character	
0000	000	0	00	ctrlONUL	
0004	001	1	01	ctrl A SOH	Start of Heading
0010	002	2	02	ctrl B STX	Start of Text
0014	003	3	03	ctrl C ETX	End of Text
0020	004	4	04	ctrl D EOT	End of Xmit
0024	005	5	05	ctrl E ENa	Enquiry
0030	006	6	06	ctrl F ACK	Acknowledge
0034	007	7	07	ctrl G BEL	Audible Signal
0040	010	8	08	ctrl H BS	Back Space
0044	011	9	09	ctrl1 HT	Horizontal Tab
0050	012	10	OA	ctrl J LF	Line Feed
0054	013	11	OB	ctrl K VT	Vertical Tab
0060	014	12	OC	ctrl L FF	Form Feed
0064	015	13	OD	ctrl M CR	Carriage Return
0070	016	14	OE	ctrl N SO	Shift Out
0074	017	15	OF	ctrlO SI	Shift In
0100	020	16	10	ctrl POLE	Data Line Escape
0104	021	17	11	ctrl a OC1	XOn
0110	022	18	12	ctrl R DC2	Aux On
0114	023	19	13	ctrl S DC3	X Off
0120	024	20	14	ctrl T OC4	Aux Off
0124	025	21	15	ctrl U NAK	Negative Acknowledge
0130	026	22	16	ctrl V SYN	Synchronous File
0134	027	23	17	ctrl W ETB	End of Xmit Block
0140	030	24	18	ctrl X CAN	Cancel
0144	031	25	19	ctrl Y EM	End of Medium
0150	032	26	1A	ctrl Z SUB	Substitute
0154	033	27	1B	ctrl [ESC	Escape
0160	034	28	1C	ctrl \ FS	File Separator
0164	035	29	1O	ctrl) GS	Group Separator
0170	036	30	1E	ctrl RS	Record Separator
0174	037	31	1F	ctrl_ US	Unit Separator
0200	040	32	20	Space	
0204	041	33	21	!	
0210	042	34	22	"	
0214	043	35	23	#	
0220	044	36	24	$	
0224	045	37	25	%	
0230	046	38	26	&	
0234	047	39	27		
0240	050	40	28	(	
0244	051	41	29	)	
0250	052	42	2A	*	
0254	053	43	2B	+	
0260	054	44	2C		
0264	055	45	2D	-	
0270	056	46	2E		

APPENDIX J - TABLE OF ASCII CODES (Zero Parity) (Continued)

Upper Octal	Octal	Decimal	Hex	Character
0274	057	47	2F	/
0300	060	48	30	0
0304	061	49	31	1
0310	062	50	32	2
0314	063	51	33	3
0320	064	52	34	4
0324	065	53	35	5
0330	066	54	36	6
0334	067	55	37	7
0340	070	56	38	8
0344	071	57	39	9
0350	072	58	3A	
0354	073	59	3B	,
0360	074	60	3C	<
0364	075	61	3D	=
0370	076	62	3E	>
0374	077	63	3F	?
0400	100	64	40	@
0404	101	65	41	A
0410	102	66	42	B
0414	103	67	43	C
0420	104	68	44	0
0424	105	69	45	E
0430	106	70	46	F
0434	107	71	47	G
0440	110	72	48	H
0444	111	73	49	I
0450	112	74	4A	J
0454	113	75	4B	K
0460	114	76	4C	L
0464	115	77	40	M
0470	116	78	4E	N
0474	117	79	4F	O
0500	120	80	50	P
0504	121	81	51	a
0510	122	82	52	R
0514	123	83	53	S
0520	124	84	54	T
0524	125	85	55	U
0530	126	86	56	V
0534	127	87	57	W
0540	130	88	58	X
0544	131	89	59	Y
0550	132	90	5A	Z
0554	133	91	5B	[
0560	134	92	5C	\

APPENDIX J - TABLE OF ASCII CODES (Zero Parity) (Continued)

Upper Octal	Octal	Decimal	Hex	Character	
0564	135	93	5D	)	
0570	136	94	5E	t	
0574	137	95	5F		
0600	140	96	60		
0604	141	97	61	a	
0610	142	98	62	b	
0614	143	99	63	c	
0620	144	100	64	d	
0624	145	101	65	e	
0630	146	102	66	f	
0634	147	103	67	g	
0640	150	104	68	h	
0644	151	105	69	i	
0650	152	106	6A	j	
0654	153	107	6B	k	
0660	154	108	6C	l	
0664	155	109	6D	m	
0670	156	110	6E	n	
0674	157	111	6F	o	
0700	160	112	70	p	
0704	161	113	71	q	
0710	162	114	72	r	
0714	163	115	73	s	
0720	164	116	74	t	
0724	165	117	75	u	
0730	166	118	76	v	
0734	167	119	77	w	
0740	170	120	78	x	
0744	171	121	79	y	
0750	172	122	7A	z	
0754	173	123	7B	}	
0760	174	124	7C		
0764	175	125	7D	}	
0770	176	126	7E		Prefix
0774	177	127	7F	DEL	Rubout

APPENDIX K - SAMPLE ASSEMBLER LISTING

```
ADDRESS  ASSEMBLED CODE  ERROR FLAG  LINE NO.  LABEL  OPERATION  OPERAND  COMMENT

                            0000  *
                            0001  *SEARCH  TABLE  FOR MATCH  TO STRING
                            0002  * EACH TABLE ENTRY IS FOLLOWED BY A TWO·BYTE
                                    DISPATCH  ADDRESS.
                            0003  *TABLE  MUST HAVE AT LEAST ONE ENTRY AND IS
                            0004  *TERMINATED  BY A ZERO  BYTE.
                            0005  *ON  ENTRY:      HL POINTS  TO STRING
                            0006  *                DE POINT  TO TABLE
                            0007  *                C IS NUMBER  OF CHARACTERS  IN
                                                   TABLE  ENTRIES
                            0008  *ON  RETURN:ZERO  FLAG  SET IF NO MATCH,  ELSE DE
                            0009  *              POINTS  TO DISPATCH  ADDRESS
                            0010  *
0100  E5                    0011  TSRCH   PUSH   H      SAVE  STRING  ADDRESS
0101  41                    0012          MOV    B,C    INITIALIZE  CHARACTER
                                                        COUNT
0102  1A                    0013  TS1     LDAX   D      COMPARE  CHARACTERS
0103  BE                    0014          CMP    M
0104  C211  01              0015          JNZ    TS3
0107  23                    0016          INX    H      CHARACTER  MATCH,  GO ON
                                                        TO NEXT
0108  13                    0017          INX    D
0109  05                    0018          DCR    B
010A  C2 02 01              0019          JNZ    TS1
010D  F601                  0020          ORI    1      MATCHING  ENTRY  FOUND
010F  E1                    0021  TS2     POP    H
0110  C9                    0022          RET
0111  B7                    0023  TS3     ORA    A      TEST  FOR  END  OF  TABLE
0112  CA OF 01              0024          JZ     TS2
0115  13                    0025  TS4     INX    D      SKIP  TO  NEXT  ENTRY
0116  05                    0026          DCR    B
0117C21501                  0027          JNZ    TS4
011A  13                    0028          INX    D
011 B  13                   0029          INX    D
011C  E1                    0030          POP    H
0110  C3 00 01              0031          JMP    TSRCH
                            0032  *
                            0033  * EXAMPLE  OF TSRCH  USE:
                            0034  *
                            0035  *(ASSUME  HL  POINTS  TO A FOUR·CHARACTER
                                    COMMAND  STRING)
0120  11 3501               0036          LXI,   D,CTABL  DE POINTS  TO COMMAND
                                                   TABLE
0123  OE  04                0037          MVI    C,4 TABLE  ENTRIES  ARE  FOUR
                                                 CHARACTERS  LONG
0125  CD  00  01            0038          CALL   TSRCH
0128  CA  00  00          U 0039          JZ     ERROR  COMMAND  NOT  IN TABLE
012B  EB                    0040          XCHG     SET UP STACK  FOR RETURN  TO
                                                 MAIN  ROUTINE
```

ADDRESS	ASSEMBLED CODE	ERROR FLAG	LINE NO.	LABEL	OPERATION	OPERAND	COMMENT
012C	1100 00	U	0041		LXI	D,COMMAND	
012F	05		0042		PUSH	D	
0130	7E		0043		MOV	A,M	DISPATCH TO APPROPRIATE COMMAND ROUTINE
0131	23		0044		INX	H	
0132	66		0045		MOV	H,M	
0133	6F		0046		MOV	L,A	
0134	E9		0047		PCHL		
			0048	*			
			0049	*COMMAND TABLE			
			0050	*			
0135	43 4F 4D 31		0051	CTABL	ASC	'COM1'	FIRST ENTRY
0139	00 00	U	0052		DW	SUB 1	ADDRESS OF SUB1
013B	43 4F 40 32		0053		ASC	'COM2'	SECOND ENTRY
013F	00 00	U	0054		DW	SUB2	ADDRESS OF SUB2
0141	00		0055		DB	O	END OF TABLE MARK

SYMBOL TABLE LISTING

Label	Addr.	Label	Addr.	Label	Addr.	Label	Addr.
CTABL	0135	TS1	0102	TS2	010F	TS3	0111
TS4	0115	TSRCH	0100				

APPENDIX L - SAMPLE PROGRAM OF LOADER SOURCE CODE

```
**************************************************
*
*                  RUNA file-name [.ZCL]
*
*  An .OBJ file consists of one of more segments that
*  have the format:
*       #BYTES        DECRIPTION
*         2           Number of code and data bytes in
*                     segment
*         2           Load address of code and data
*                     belonging to the segment.
*       Variable      Code and/or data.
*
*  The run time package will load each segment at the
*  specified address until a starting address is
*  encountered. A starting address is represented as
*  load address with a zero byte count.
*
**************************************************
*
RELOC EQU 0        ;4200H FOR TRS-80 MOD 1
BOOS EQU 5+RELOC        ;CPIM
BLKSIZ EQU 128
OFCB EQU 5CH + RELOC        ;IN CPIM
OEX EQU OFCB + 12
OCR EQU OFCB +32
OBUF EQU 80H + RELOC        ;IN CPIM
*
CSTART EQU $
   LXI SP,STK
   MVI C,OCH        ;RETURN VERSION #
   CALL BOOS
   MOV A,L
   ORA A
   JNZ VER2X
   LOA 4+RELOC        ;CPM 1.4 DEFAULT DRIVE
   CPI5
   JC SETDF
   XRA A
SETDF EQU $        ;11-30-81 FOR MPIM II
   STA ODRIVE        ;DEFAULT DRIVE
*      GET OPTIONS FROM TYPE FIELD
   LXI H,5CH +8
   MVI C,4
NEXT EQU $
   INX H
```

```
      OCR C
      JZ NOOPTIONS
      MOV A,M
      CPI"
      JZ NOOPTIONS
      CPI 'Z'
      JZ ZEROFIL
      CPI 'C'
JZ COMFILE
CPI'L'
      JZ NOEXEC
*       ERROR ILLEGAL OPTION
      LXI H,MESGA
      CALL DISPLAY
      JMP O + RELOC
*       GET SIZE OF INSTRUCTION
GETSZ LXI H,TBL-1
AGAIN MOV A,C
      INX H
      MOV B,M
      ANA B
      JZ BYTE1
      INX H
      MOV B,M
      XRA B
      INX H
      JNZ AGAIN
      MOV A,M
      RET.  EXIT
BYTE1 MVI A,1
      RET.  EXIT
*
REL EQU $ RELOCATION
      PUSH H
      PUSH O
      OPUSH PSW
      INX H
      MOV E,M
      INX H
      MOV A,M
      ORA A WE DON'T RELOCATE BELOW 100+ RELOC
      JZ NOREL
      MOV D,A
      PUSH H
      LHLD BASE
      DAD O ADDRESS IS NOW ADJUSTED
      XCHG
      POP H
      MOV M,D        PUT IT BACK
      DCX H
      MOV M,E
NOREL EQU $
      POP PSW
      POP O
      POP H
      RET
*
VER2X EQU $
      MVI C,19H        ;GET CPM 2.X DEFAULT DRIVE
      CALL BOOS
      JMP SETDF
*
MESGA ASC 'ILLEGAL OPTION'
      DB ODH,OAH
      ASC 'RUN D:FILE.ZCL<CR>'
      DB ODH,OAH,O
*
TBL DB -1,11101001B,1
      DB -1,11001101B,3
      DB 11000111B,11000100B,3
      DB -1,11000011B,3
      DB 11000111B,11000010B,3
      DB 11000111B,11000111B,1
      DB -1,11001001B,1
      DB 11000111B,11OOOOOOB,
      DB 11001111B,1,3
      DB 11100111B,00100010B,3
      DB 11110111B,11010011B,2
      DB 11000111B,6,2
      DB 11000111B,11000110B,2
      DB 0          END OF TABLE
*
BASE OW 0 BASE ADJ TO ADD TO ADDRESS TO BE
 RELOCATED
START OW 0 STARTING ADDR OF RELOCATED CODE
*
ZEROFILL EQU
      STA ZX
      JMP NEXT
*
COMPILE EQU $
      STA CX
      JMP NEXT
*
NOEXEC EQU $
      STA LX
      JMP NEXT
```

```
*
OSET  EQU  $
   LXID,OBUF
   MVI  C,26        :SET  DMA
   CALL  BOOS
   LOA  ODRIVE
   MVID,O
   MOV  E,A
   MVI  C,14        :SET  DRIVE
   CALL  BOOS
   LXI  D,OFCB
   RET
*
NOOPTIONS   EQU  $
   LXI  H,080H +3 + RELOC
   MOV  A,M
   CPI  ':'      ;WAS  DRIVE  REQUESTED?
   JNZ  DEFDRIVE        ;DEFAULT   IS SET
   DCX  H
   MOV  A,M
   CPI'A'
   JC  DEFDRIVE
   SUI  'A'
   STA  ODRIVE
DEFDRIVE  EQU  $
   CALL  SETFCB
   MVI  M,'O'
   INX  H
   MVI  M,'B'
   INX  H
   MVI  M,'J'
   CALL  OSET
   MVI  C,15       ;OPEN
   CALL  BOOS
   CPI-1
   JZ OERR        ;OPEN   ERROR
   XRA  A
   STA  OCR
*       RELOATE  CODE  TO JUST  BELOW  CPIM
   LHLD  6 + RELOC
   DCX  H HIGHEST   ADDR
   LXI  B,LAST-LOADFILE    SIZE  OF  CODE  TO  BE
    RELOCATED
   MOV  A,L
   SUB  C
   MOV  L,A
   MOV  A,H
   SBB  B
```



```
   MOV  H,A
;      H&L=   STARTING   ADDRESS
   SHLD  START
   PUSH  H
   LXI  D,LOADFILE
   MOV  A,L
   SUB  E
   MOV  L,A
   MOVA,H
   SBB  O
   MOVE  H,A
   SHLD  BASE
   POP  H
   LXI  B,CONSTANTS-LOADFILE      SIZE  OF  INSTRUCTION
    MOVE
   XCHG
NXTI  EQU  $
   PUSH  H
   PUSH  O
   PUSH  B
   MOV  C,M   GET OPCODE
   CALL  GETSZ        GET SIZE  OF  INSTRUCTION
   POP  B
   POP  0
   POP  H
   CPI3
   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  0
   INX  H
   INX  0
   DCRC
   JNZ  NXTM
   POP  PSW
   POP  B
NXTD  EQU  $
   DCX  B
   DCRA
   JNZ  NXTD
   MOV  A,C
   ORA  B
   JNZ  NXTI
```

```
*      RELOCATE  CONSTANTS
   LXI  B,LAST-CONST  ANTS       SIZE  OF  CONSTANTS
NXTC  EQU  $
   MOV  A,M
   STAX  O
   INX  H
   INX  0
   DCX  B
   MOV  A,C
   ORA  B
   JNZ  NXTC
   LH LD  START
   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
   LOA  ZX       ZERO  FILL  MEMORY?
   ORA  A
   JZ  SKPCLR
   LXI  D,LOADFILE-1
   MVI  H,1          STARTING  ADDR    + RELOC
   MVI  L,O
CLEAR  EQU  $
   XRA  A
   MOV  M,A
   INX  H
   MOV  A,L
   SUB  E
   MOV  A,H
   SBB  O
   JC  CLEAR
SKPCLT  EQU  $
   CALL  ORO       ;GET  1ST  RECORD  OF  .OBJ  FILE
OLOAD  EQU  $
   CALL  GETOP
MAO  MOV  A,M        ;MOVE  4  BYTES  FROM  BUF  TO  WORK
   STAX  O
   INX  H
   INX  O
   OCR  C
   CZ  ORO
   OCR  B
   JNZ  MAO
;   H&L  =  BUFFER  C = COUNT
   XCHG
   LHLD  OWRK        ;SIZE  OF  NEXT  READ
   MOV  A,L
   ORA  H
   JZ  CLOSE
   SHLD  OSIZE
   LHLD  OWRK+  2
   XCHG
MAOA  MOV  A,M        ;MOVE  FROM  BUF  TO  OBJ  ADDR
   STAX  O
   INX  H
   INX  O
   OCR  C
   CZ  ORO
   PUSH  H
   LHLD  OSIZE
   DCX  H
   SH LD  OSIZE
   MOV  A,L
   ORA  H
   POP  H
   JNZ  MAOA
   CALL  SAVOP
   JMP  OLOAD
*
GETOP  EQU  $       ;GET  O  POINTERS
   LXID,OWRK
   LHLD  OCBA        ;BUF  AD DR
   MVI  B,4       :LENGTH  OF  WRK
   LOA  OCBC        ;BUF  CNT
   MOV  C,A
   RET
*
ORO  EQU  $
   PUSH  B
   PUSH  O       ;OPNT
   LXI  D,OFCB
   MVI  C,20        ;READ
   CALL  BOOS
   POP  O
   POP  B
   ORA  A
   JNZ  RERR
   LXI  H,OBUF
   MVI  C,BLKSIZ
   RET
```

```
*
SAVOP  EQU $
    SHLD  OCBA        ;BUFF  ADDR
    MOV  A,C
    STA  OCBC       ;BUF  CNT
    LOA  HIGH
    CMP  O
    RNC
    MOV  A,D
    STA  HIGH
    RET
*
CLOSE  EQU $
    LXI  D,OFCB
    MVI  C,16       ;CLOSE
    CALL  BOOS
    LOA  CX
    ORA  A
    JNZ  GENCOM
    LOA  LX
    ORA  A
    JNZ  0+  RELOC       LOAD  BUT  DON'T  EXECUTE
    LHLD  OWRK+  2       ;STARTING  ADDRESS
    PCHL
*
SETFCB  EQU $
    XRA  A
    STA  OFCB
    STA  OCR
    LXI  H,OEX
    MVI  C,4
EXLUP  EQU $       ;10-2-81       ZERO  CPM  EXT AREA
    MOV  M,A
    INX  H
    OCR  C
    JNZ  EXLUP
    LXI  H,OBUF
    SHLD  OCBA
    MVI  A, BLKSIZ
    STA  OCBC
    LXI  H, 5CH + 9 + RELOC     ;CPIM  FILE  TYPE
    RET
*
CREATE  EQU $
    LXI  D,OFCB
    MVI  C,22   CREATE
    CALL  BOOS
    CPI  -1
```



```
    RNZ
OERR  EQU $
    LXI  H,MESGO    OPEN  ERROR
    CALL  DISPLAY
    JMP  OXT1
*
GENCOM   EQU $ GENERATE   .COM  FILE
    CALL  SETFCB
    MVI  M,'C'
    INX  H
    MVI  M,'O'
    INX  H
    MVI  M,'M'   .COM  IN  FCB
*  OPEN
    LXI  D,OFCB
    MVI  C,15  OPEN  .COM  FILE
    CALL  BOOS
    CPI  -1
    CZ CREATE
    XRA  A
    STA  OCR
*  WRITE
    LOA  HIGH
    OCR  A
    MOV  H,A
    MVI  L, OFFH
    SHLD  SIZ  OF  THIS  WRITE
    MVID,1    STARTING   ADDRESS     + RELOC
    MVI   E,O
    LXI  H,OBUF   BUFFER  ADDRESS
    MVI  C,BLKSIZ   BUFFER  SIZE
NXTW  EQU $
    LDAX  O
    MOV  M,A
    INX  H
    INX  0
    OCR  C    BUFF  COUNT
    CZ  WRITE
    PUSH  H
    LH LD  SIZ
    DCX  H
    SHLD  SIZ
    MOV  A,L
    ORA  H
    POP  H
    JNZ  NXTW
    CALL    WRITE  LAST  BLOCK
*   CLOSE
```

```
   LXI D,OFCD
   MVI C,16  CLOSE
   CALL  BOOS
   JMP  0 + RELOC
*
WRITE  EQU $
   PUSH  0
   LXI D,OFCB
   MVI C,21   WRITE
   CALL  BOOS
   POP  O
   ORA  A
   JNZ  ERRW
   LXI H,OBUF
   MVI  C,BLKSIZ
   RET
*
* + + ***********************************************
*$$      DISPLAY  A MESSAGE  TO THE  CONSOLE
*        ENTRY H&L CONTAIN  STARTING  ADDRESS  OF
         THE  MESSAGE
*        THE  MESSAGE  TEXT IS TERMINATED   BY 0 HEX
*        CALL  DISPLAY
* -  - ***********************************************
*
DISPLAY  EQU $
   MOV  A,M
   ORA  A
   RZ.  EXIT TO CALLING   ROUTINE  * *
   MOV  E,A
   MVI  C,2
   PUSH  H
   CALL  BOOS   ;PUT THE  CHAR  TO THE  CONSOLE
   POP  H
   INX  H
   JMP DISPLAY
*
ERRW  EQU $
   LXI H,MESGW    WRITE  ERROR
   CALL  DISPLAY
   JMP  OXT1
*
RERR  EQU $
   LXI H,MESGR   READ  ERROR
   CALL  DISPLAY
OXT1  EQU $
   LXI H,OFCB  + 1   ;FILE  NAME
   CALL  DISPLAY
```



```
JMP  0+  RELOC   RETURN  TO CPIM
*****.*********************************************
CONSTANTS   EQU $
HIGH  DB 0 HIGHEST   PAGE  USED  FOR .COM
ZX DB 0 DEFAULT   NO CLEAR   ;Z=  ZERO  FILL  BEFORE
 LOADING
CX  DB 0 DEFAULT   NO .COM   ;C = .COM  FILE
LX DB 0 DEFAULT   EXECUTE   ;L = LOAD  BUT NO
 EXECUTION
ODRIVE  DB 0
OWRK  DB 0,0,0,0
OCBA  OW OBUF    ;CURRENT  BUFFER  ADDRESS
OCBC  DB BLKSIZ   ;CURRENT  BUFFER  COUNTER
OSIZE  OW 0    ;SIZE OF  NEXT OBJ  BLOCK
SIZ  OW 0 SIZE  OF  COM  FILE CODE
MESGO  ASC  'OPEN  ERROR'
   DB 0
MESGR  ASC  'READ  ERROR'
   DB 0
MESGW  ASC  'WRITE  ERROR'
   DB 0
   OS 30
STK  DB'S'
LAST  DB 0
```

APPENDIX M - SUGGESTED REFERENCES

A GUIDE TO FORTRAN PROGRAMMING
Daniel D. McCracken
Addison-Wesley Publishing Co., 1961.

CP/M USER'S GUIDE
T. Hogan
OSBOURNE,1981.

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

FORTRAN IV WITH WATFOR AND WATFIV
Cress, Dirksen, and Graham
Prentice-Hall, Inc., 1970.

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

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

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

8080/8050 ASSEMBLY LANGUAGE PROGRAMMING MANUAL
INTEL CORPORATION
SANTA CLARA, CA., 1977.

8080A/8050 ASSEMBLY LANGUAGE PROGRAMMING
Lance A. Leventhal
OSBORNE,1978.

NEVADA FORTRAN and NEVADA ASSEMBLER INDEX

A
A-Type, 52
A= n, 92,107
ABS, 97
ACCEPT,94
ALOG,97
ALOG10,97
AMAXO,97
AMAX1,97
AMINO, 97
AMIN1,97
AMOD,97
ARG CNT, 101
Arithmetic IF, 28
Arithmetic Operators, 22
Array, 105
Assembly Errors, 92
Assembler Command Errors, 92
ASSIGN, 27, 94, 104
ASSIGNED GOTO, 27, 101, 106
ASSM.COM, 91
Asterisk, 108
ATAN,97
ATAN2,97

B
Backslash, 58, 109
BACKSPACE, 60, 94
Binary *1/0,* 59
BIT, 62-63, 97, 99
Blank COMMON, 104
Blank padding, 10
BLOCK DATA, 45,94,106

C
C=XXXX, 107
CALL, 42-44, 62, 72, 94, 97, 104, 111
CALL POP, 101
CALL PSH, 101
CBTOF, 62,73,97,99
CHAIN, 6, 21, 33, 62-63,99
CHAIN FL, 63, 66,102
CHAR, 62, 73, 97
CIN,34,62,64,99
CLOSE, 60, 62, 64, 99
COM GO TO, 102

Comment Field, 86
COMMON, 20, 39-40, 94,106,111
COMP, 62, 73, 97
Comparison Operators, 22
COMPUTED GO TO, 27,104
CON BIN, 102
Conditional Assembly, 90
CONFIG, 3, 13-14,72, 109
Constant, 15·16, 84
CONTINUE, 26, 31-32, 94
CONTROL-C, 33-34, 76, 80, 94, 102
CONTROL-H, 61
CONTROL-X, 61
CONTROL-Z, 61
CONVERT,102
COPY, 12,91,94,106-107,111
COpy file, 91
CPIM OPERATING SYSTEM, 1
COS, 97
CTEST, 62, 65, 99
CTRL DISABLE, 34, 94
CTRL ENABLE, 34, 94

0
D-Type,52
0= n, 92,107
DATA, 19,94,106-107,111
DECODE, 50, 111
Define ASCII String (ASC), 99
Define Byte (DB), 88
Define Double Byte (DDB), 89
Define Word (OW), 89
Define Storage (OS), 88
DELAY, 62, 65, 99
DELETE, 61-62, 65, 99
DIM, 97
DIMENSION, 39-40, 94,104,106
Disk is full, 92,102
DIV ZERO, 102
00,28,31,94,104
DOUBLE PRECISION, 17-18,21,39,43,94,106
DUMP, 36, 94, 106

E
E-Type,52
ENCODE, 50-51, 111
END, 19,28,31,38,91,95,105,107
END =,49-50, 105
END of Source File, 91

ENDF,90
ENDFILE, 59-61, 95
ENDIF, 29-30, 95, 106
EQU, 87
Equate, 87
ERR =,49, 105
ERRCLR, 32, 34, 95, 106
Error Codes, 63, 92
Error (.ERR) file, 3, 77
ERRORS, 3
ERRSET, 32-34, 49, 95, 106
Executing the Assembler, 77
Executing the .OBJ File, 8, 80
EXIT, 38, 62,66, 99
EXP, 97
Expressions, 22, 23, 41, 85

F
F-Type,53
FILE OPR, 102
FLOAT, 97
FORMAT, 51, 95,102
Formatted *1/0,* 46, 51, 58-59
FORT.COM,3
FORT.ERR,3
Free Format *1/0, 58*
FUNCTION, 62,72,95,105
FUNCTION statement, 43

G
G,104
G-Type,53
Getting Started, 1
GO TO, 26-27, 31-33, 97

H
H,57
Hardware Requirements, 1, 77
Hexadecimal, 109
High-Order Byte Extraction, 85

I
I-Type, 54
1/0 ERR, 103
110 LIST, 46-47, 103
IABS, 97
IDIM,97
IF,28,95

IF-THEN-ELSE, 29-31, 95,106,111
IFIX, 97
IFLS,91
ILL CHAR, 102
ILL UNIT, 102
1MPLICIT, 21, 95, 106, 111
Implied DO loop, 105
INP, 62, 74, 97
INPT ERR, 103
INT RANG, 23, 103
INTEGER, 17,21,23,39,43,95,106,109
ISIGN,97

K
K-Type,55

L
L-Type,55
L=, 92,107
Label Field, 82
Labels, 83, 107
Line Length, 103
Line Numbers, 81
List Conditional Code, 91
Listing Control, 91
Listing Title, 91
LOAD, 33, 62, 66, 99
LOG NEG, 103
LOGICAL, 17-18,21,39,43,95,106
Logical IF, 28, 105
Logical Operators, 22, 24-25
LOPEN, 59, 61-62, 64, 67, 99
Low-Order Byte Extraction, 85

M
M=XXXX,92
MAXO,97
MAX1,97
Memory Usage, 80
MINO,97
MIN1,97
Mixed mode expressions, 25
MOD,97
MOVE, 62, 68, 99
Multiple RETURN, 44

N
NLST,91
Normal return, 44

O
0= n, 93,105
Object (.OBJ) Code, 76
OPERATION FIELD, 82
OPEN, 59, 61-62, 64, 68, 99, 107
Operand Field, 82
Options, 3, 13, 32
ORG,87
OUT, 62, 69, 99
OVERFLOW, 103

P
P= n, 79,105,111
Page Eject, 91
PAUSE, 37, 95, 106
PEEK, 62,74,97
POKE, 62, 70, 99
Pseudo-Operations, 87
PUT, 62,70,100

Q
Q,32

R
R=,93
RAND,97
READ, 47-49, 96, 111
REAL, 17-18,23-24,39,43,96, 106
Register Names, 83
RENAME, 62, 70,100
RESET, 62,71,100
RETURN, 31,44,96
REWIND, 59,61,96
RUNA, 76, 80
Runtime error, 33, 101
Runtime Format, 21, 46

S
S =, 79, 93, 107
Sample Program, 119-131
SEEK, 61-62,71,100
SEEK ERR, 103
Set ASCII List Flag, 90
Set Execution Address, 88

Set Origin, 87
SETIO, 62, 72, 100
SIGN,97
SIN,97
Software Requirements, 1, 77
Source Code, 76
SQRT, 97, 103
Statements, 81
STOP, 31, 38, 66, 94, 96,106
Strings, 57
Subprograms, 42
SUBROUTINE, 42-43, 62, 96, 99-100
Subscripts, 41

T
T-Type,55
TAN,97
Termination, 37, 80
TITL,91
TRACE OFF, 35, 96,106
TRACE ON, 35, 96,106
TYPE, 17,96

U
U=,93
Unconditional GO TO, 26
UNIT CLO, 103
UNIT OPN, 103

V
V,93
Variable, 17,21-22,33,45,47,50-51,63,96

W
WRITE, 36, 47, 50, 96, 111

X
X-Type, 56
XEQ,88

Z
Z-Type,56

#, 79, 109
$, 78, 106, 109
*,108
I-Type, 56
.AND., 22, 24
.EQ., 22, 24
.FALSE.,28
.GE., 22, 24
.GT., 22, 24
.LE., 22, 24
.LT., 22, 24
.NE., 22, 24
.NOT.,22
.OR., 22, 24
.TRUE., 28
.XOR., 22, 24

Commodore Business Machines. Inc.
1200 Wilson Drive West Chester. PA 193BO

Commodore Business Machines. Limited
3370 Pharmacy Avenue Agincourt. Ontario. M1W 2K4