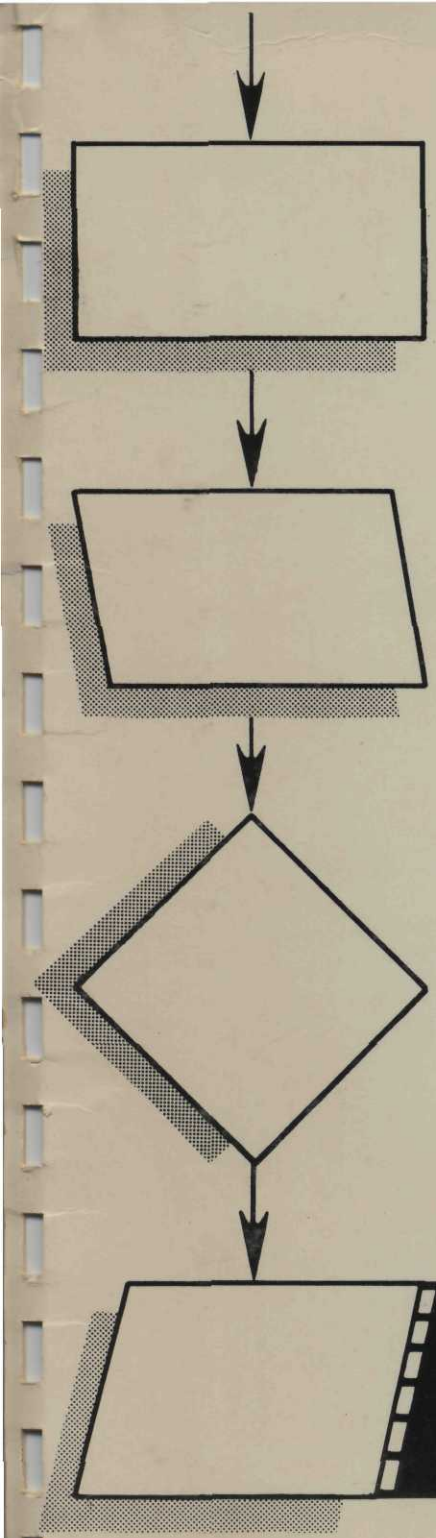
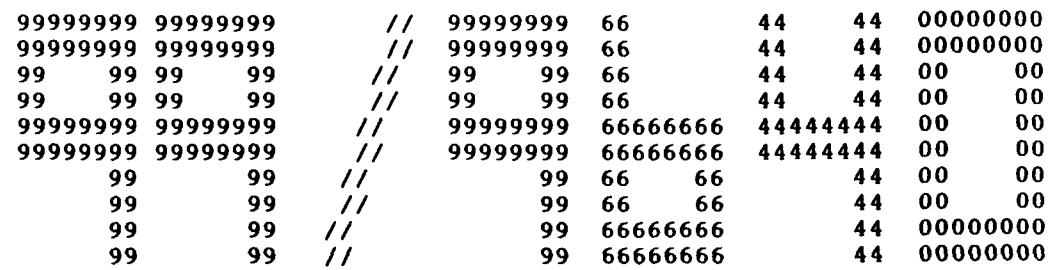
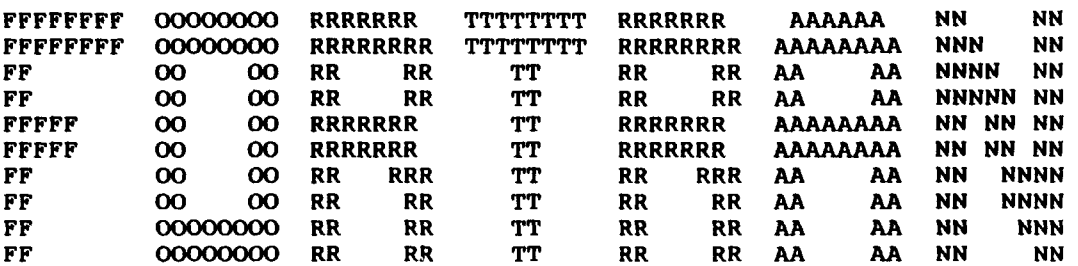


99/9640 FORTRAN

NEW! VERSION 4

LGMA PRODUCTS





USER REFERENCE MANUAL

VERSION 4.3, rev. 0

IMPORTANT NOTICE TO USERS

The user assumes complete responsibility for any decisions made or actions taken on information contained using these programs and book materials which are available solely on an "as-is" basis.

LGMA does not warrant or represent that the programs and book materials will be free from error or will meet the specific requirements of the user.

I would like to thank the following people who made Version 4 of 99/9640 FORTRAN possible, including:

Elmer Clausen, for providing most of the mathematical functions in the FORTRAN library;

Paul Charlton, for his advice and tips on how to use the MYARC Disk Operating System;

Dave Ramsey, for suggesting the MDOS version in the first place;

Jeff Guide, for providing a FORUM (DELPHI) in which he brought many of the technical people involved with the MYARC GENEVE together;

Clint Pulley (author of C99), for giving me permission to use his QDE editor with the 9640 FORTRAN package, and also providing me with tips on how to use the MDOS Operating System;

Dr. Jerry Coffey, who's JUMPBOOT program saved me many hours of reboot time in the early days of MDOS;

Ron Lepine, for helping me get the a99 cross-assembler bugs worked out, and providing me with a forum on Byte Information Exchange (BIX);

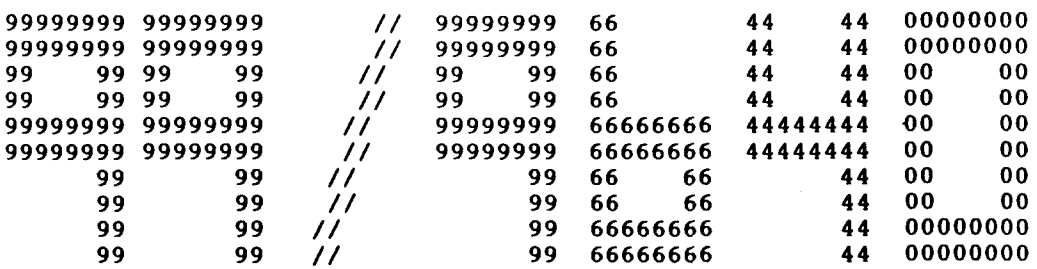
and most importantly, my wife Patti and children Heather and Jeffrey, who put up with my long hours in preparing this software package, and encouraged me to do it.

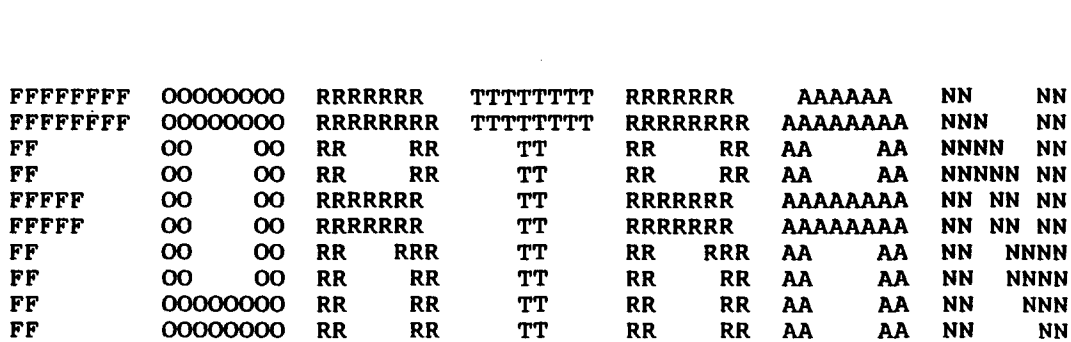
Please note the following trademarks used in this manual:

TI is a trademark of Texas Instruments, Incorporated, Dallas, Texas.

MYARC, MDOS, and GENEVE are trademarks of MYARC, Inc., Basking Ridge, New Jersey.

LGMA, 99 FORTRAN and 9640 FORTRAN are trademarks of LGMA Products, Coopersburg, Pennsylvania.





New! VERSION 4

The latest in programming development for your TI-99/4A or MYARC 9640 GENEVE computer. Requires either a TI-99/4A with 32k memory expansion, at least one disk drive, and one of the cartridges Editor/Assembler, Mini-Memory, Extended BASIC, or TI-Writer; or a MYARC GENEVE computer with at least one DS/SD disk drive.

99/9640 FORTRAN taps into the power of your TI-99 or MYARC GENEVE computer with the following features:

- o Full screen editor, Optimizing FORTRAN compiler, Linker, Symbolic debugger, and over 160 library routines.
- o Six data types, including INTEGER *1, *2, *4; REAL *4, *8, and LOGICAL *2
- o Many FORTRAN 77 extensions, including BLOCK IF
- o Symbolic debugger allows breakpoints by program line number, and FORTRAN labels, memory access by variable name. MDOS version allows source view and disassembly capabilities.
- o Supports access to TMS9900 assembly language subroutines/functions.

ONTENTS: TI-99/4A Version: Boot Disk and Two (2) Library Disks. (SS/SD)
MDOS Version: Boot Disk and Library Disk (DS/SD)
230 Page User Manual

FORTRAN Language Summary

Control Statements

```
GOTO n
GOTO ( n1, n2, ..., nm ), s
IF ( a ) n1, n2, n3
IF ( a ) s
IF ( a ) THEN  /  ELSEIF ( s ) THEN
ELSE           /  ENDIF

DO n I=s1,s2[,s3]
DO WHILE ( a ) / s / ENDDO

PAUSE [i]
STOP  [i]
END
```

Subprograms

```
type FUNCTION f( d1, d2, ..., dn )
SUBROUTINE s( d1, d2, ..., dn )
CALL s( a1, a2, ..., an )          )
RETURN
EXTERNAL sub1,...,subn
```

Compilation Directive Statements

```
INCLUDE 'file-name[/LIST,/NOLIST]'
```

Compilation Options

```
SC - Turn on Subscript Checking
OB - Object Code Listing
DM - Compile Debug Mode Statements
DB - Generate Debugger Symbols
```

Librarian Options:

```
ADD - Add Modules to Library
LIST - List Library
```

Memory Allocation (MDOS)

```
CALL CPMBR/CPMBW
CALL LVMBR/LVMBW
CALL RTFREE/RTPAGE
CALL MPLCPE/RTMAPR
CALL MALLOC
```

* = MDOS Only +=TI/99 Only

Input/Output Statements

```
READ  ( i, n [,keys] ) list
WRITE ( i, n [,keys] ) list
label FORMAT ( S1, S2, ..., Sn )
```

Specification Statements

```
DIMENSION v1, v2, ..., vn
COMMON    s1, s2, ..., sn
EQUIVALENCE s1, s2, ..., sn
INTEGER [*1, *2, *4] s1, s2, ..., sn
REAL [*4, *8]   s1, s2, ..., sn
DOUBLE PRECISION s1, s2, ...
LOGICAL [*2] s1, s2, ..., sn
IMPLICIT type (c1,c2,...,cn)
DATA s1/d1/,s2/d2/,...,sn/dn/
```

Programs

```
PROGRAM name
```

Debugger Commands (GPL/MDOS)

```
W -  Workspace Inspect/Change
H -  Hexadecimal Arithmetic
B -  Remove/Add Breakpoints
M -  Memory Inspect/Change
Q -  Quit Debugger
R -  Inspect/Change WP, PC, or SR
T -  Trade Screen
S -  Select Module
L -  Load Symbol File
```

Debugger Commands (MDOS)

```
D - Disassemble at address
G - Go Program into Execution
P - Parameter Display
V - View Source Module
X,Y,Z - Constant Memory
? - Display Help
```

Input/Output

```
CALL OPEN        CALL CLOSE
CALL DELETE      CALL FILES
*CALL BREAD      *CALL BWRITE
```

FORTRAN Language Summary

Math

IABS/ABS/DABS/KIABS/JIABS
IOR/KIOR/JIOR
IAND/KIAND/JIAND
IEOR/KIEOR/JIEOR
NOT/ISHFT
IFIX/FLOAT/SNGL/DBLE/DFLOAT
MOD/AMOD/DMOD
SQRT/DSQRT
SIN/COS/TAN
DSIN/DCOS/DTAN
SIN/COS/TAN/DSIN/DCOS/DTAN
ATAN/DATAN/ATAN2/DATAN2
COTAN/DCOTAN
EXP/EXP2/DEXP/DEXP2/EXP10/DEXP10
ERF/DERF/ERFC/DERFC
GAMMA/DGAMMA/ALGAMA/DLGAMA
ALOG/DLOG/ALOG2/DLOG2/ALOG10/DLOG10
ARSIN/DASIN/ARCOS/DACOS
IDIM/DIM/DDIM
ISIGN/SIGN/DSIGN

Sprites

CALL SPRITE	CALL SPCHAR
CALL MOTION	CALL POSITI
CALL DELSPR	CALL MAGNIF

Date/Time Library (MDOS)

*CALL CHETIM	*CALL CHEDAT
*CALL CONTTS	*CALL CONDTS
*CALL CONSTT	*CALL CONSTD
*CALL CONJUL	*CALL RETDOW

Execution Errors

AE - Argument Error
BC - Bad Character on Input
BF - Bad # of Files (1-9)
BM - Bad Video Mode
CL - Bad Color (1-16)
CO - Bad Column (1-32, 1-40, 1-80)
CS - Bad Character Set (1-28)
CV - Bad Column Velocity
DC - Bad Dot Column (1-255)
DR - Bad Dot Row (1-192)
EC - Illegal FORTRAN function
IC - Illegal FORMAT character
II - Input for Output Item
IO - Input/Output Error
IR - Input Integer/Real FORMAT

Graphics

CALL GCHAR	CALL HCHAR
CALL VCHAR	CALL SCREEN
CALL COLOR	CALL CHAR
CALL CHARPA	CALL CLEAR
CALL SET32	CALL SET40
CALL SET80	CALL SETMOD
CALL SOUND	CALL SOUSTA
CALL JOYST	CALL KEY
CALL GVIDTB	CALL VWTR
CALL VRFR	CALL GETMOD
CALL SETPOS	CALL GETPOS
*CALL SETVPG	*CALL GETVPG
*CALL SCRLUP	*CALL SCRLDN
*CALL SETBRD	*CALL SETPAL
*CALL SETPIX	*CALL GETPIX
*CALL SETVEC	*CALL CLRSRC
*CALL HBLKMV/CP	*CALL LBLKMV/CP
*CALL SETTWN	*CALL BLKSUP/DN
*CALL GETTWN	*CALL RESCHA
*CALL SETMSE	*CALL GETMSE/MSR
*CALL SCRLLE/RI	*CALL BLKSLE/RI

Miscellaneous

CALL WAIT	CALL QUIT
CALL CLOCK	IRAND
IVAL/VAL/DVAL	CALL CHAIN
CALL PRINTC	*CALL CMDSTR
*CALL LOCK	

Memory Access

PEEK/I/J/K	CALL LOADM
PEEKV/I/J/K	

I/O Errors

0 - No Error
1 - Bad Device Name
2 - Write Protected
3 - Bad Open Attribute
4 - Illegal Operation
5 - Out of Buffer Space
6 - Read Past End of File
7 - Device Error
8 - File Error

Important Addresses

+XMLLNK	203C
+KSCAN	2040
+VSBW	2044

Execution Errors (continued)

IV - Illegal Character Value
KE - Bad Keyboard Unit (0-5)
MF - Bad Magnify Factor (1-4)
*MO - MDOS Function ONLY
NE - Nesting Error in FORMAT
NR - N Processing Error
OB - Bad Byte Count/OPEN (1-255)
OD - Bad DISPLAY on OPEN (0/1)
OI - Bad INPUT on OPEN (0-3)
OR - Bad Relative on OPEN (0/1)
OV - Bad Variable on OPEN (0/1)
O# - Bad Device Number/OPEN
*PA - Bad Pallette Number/SETPAL
RI - Real Item for Integer
*RG - Bad Red/Green/Blue Color
RO - Bad Row Number (1-24)
RP - Bad # Repititions
S4 - 40 column mode error
SA - Bad Shift Amount (-16-+16)
SC - Subscript Error
SD - Bad Sound Duration
SE - Illegal Subroutine
SF - Bad Sound Frequency
SR - Bad Sprite Number
SV - Bad Sound Volume
VE - VDP Access Error (address)

* = MDOS Only + = TI/99 Only

Important Addreses (continued)

+VMBW	2048	
+VSBR	204C	
+VMBR	2050	
+VWTR	2054	
+DSRLNK	2058	
+MENU	2060	
+PRINTF	2002	
+SET40F	2004	
+LOGSTR	200E	
+LOGEND	2010	
+DATSTR	2012	
+DATEND	2014	
+COMEND	2016	
+CLOCK1	2028	
+CLOCK2	202A	
+CRTXY	2036	
+NUMLIN	2036	
+CHAPPL	2038	
+EXCDEV	208E	
+PRTDOP	209C	
+DSKSS	20A4	
+BOTSHE	20A6	
WSP	+8300	*F000
+FAC	834A	
+STFAC	8354	
+ARG	835C	
+KEYUNT	8374	

1.0 Introduction

This programming development package allows you to edit, compile, link, and execute programs written in the FORTRAN language. The FORTRAN language provided is a powerful subset of the FORTRAN 77 standard, and has the following major features:

* Integer *1 (byte), Integer *2 (word), Integer *4 (longword), Real *4, Real *8, and Logical *2 Data Types

* IF/THEN/ELSE/ELSEIF/ENDIF/DOWHILE/ENDDO Structured Programming Statements

* Extended formatting commands for screen output

* Extended FORTRAN library for access to graphics, sprites, and sound

* Optimizing FORTRAN compiler

* Two screen modes; Graphics (32 x 24) and Text (40 x 24) in TI-99 implementation, Nine Screen modes in MDOS implementation.

* True 9900 object code generation (not a P-code compiler)

This manual describes the TI-99 implementation of the 99 FORTRAN compiler, and also describes the MYARC GENEVE 9640 implementation of the 9640 FORTRAN compiler. Differences between the two implementations are noted in this manual.

The TI-99 implementation of 99 FORTRAN (sometimes called the GPL, or Graphics Programming Language implementation) is supplied with three single sided/single density disks, as follows:

1. The first disk (called the boot disk) contains the menu, compiler, linker, execution support, and a save/load utility.

2. The second disk (called the Non-Math Library Disk) contains the object for the non-Math related and integer math library functions. It also has the example program source as discussed in section 8.

3. The third disk (called Math Disk) contains the objects for the full complete extended math library.

The MYARC Disk Operation System (MDOS) implementation of 9640 FORTRAN is supplied with two double sided/single density diskettes, which contain the following:

1. The first disk (called the boot disk) contains the editor, the compiler, the linker, the system FORTRAN library, the graphics FORTRAN library, and the symbolic debugger.

2. The second disk (called the library/demonstration disk) contains the the math library, the FORTRAN librarian, and several

example program sources.

This manual provides details on creating, editing, compiling, linking, running, and debugging FORTRAN programs on your TI-99 or MYARC 9640 computer. It is divided into ten sections, as follows:

SECTION 1 - Contains this introduction, and how to get started using the FORTRAN disks.

SECTION 2 - Describes the full screen editor used to prepare your FORTRAN program source.

SECTION 3 - Contains a reference manual which details the syntax of the FORTRAN programming language.

SECTION 4 - Describes the compiler operation, which details how to execute the compiler, and possible error conditions.

SECTION 5 - Contains an operation manual for the LINKER (used to create executable programs from compiled object), how to load and run your FORTRAN programs, and possible execution time error conditions.

SECTION 6 - Contains an operation and reference manual for the FORTRAN symbolic debugger.

SECTION 7 - Describes the extensive FORTRAN library of subroutines and FUNCTION subprograms.

SECTION 8 - Describes the FORTRAN programming example programs.

SECTION 9 - Describes the PREFERENCES utility, and how to setup the 99 FORTRAN environment for you personal configuration and tastes.

APPENDIX - Has miscellaneous tables and tips on getting the most from the FORTRAN language.

To use the TI-99 implementation of 99 FORTRAN, you need at least the 32k memory expansion, and one disk drive, along with one of the following command cartridges:

1. EDITOR/ASSEMBLER
2. MINI-MEMORY
3. EXTENDED BASIC
4. TI-Writer

To use the MYARC GENEVE implementation, you need a MYARC GENEVE 9640 with at least one double sided/single density disk drive.

A printer with associated interface is recommended for program listings, but it is not required.

1.1 Getting Started

It is recommended before using any of the disks provided that backup disks be created using a disk manager utility. Use only the backup disks during normal operation.

If you have a hard disk using the MYARC HFDCC, you may wish to place the TI-99 implementation of the compiler in the directories:

```
WDS1.DSK.FORTCOMP.
WDS1.DSK.FORTLIBR.
```

Before using the FORTRAN compiler, the correct cartridge must be inserted into the computer, the computer and associated memory expansion must be on, and the boot disk must be in one of the disk drives on the system.

1.1.1 Booting from Editor/Assembler or Mini-Memory (TI-99 GPL Mode)

1. Press any key to make the master selection list appear. Select "1" for BASIC.

2. Type the following in BASIC:

```
OLD "DSK.FORTCOMP.LOAD"
RUN
```

3. Loading is automatic from this point on. After loading two files, the FORTRAN master menu list will be displayed.

1.1.2 Booting from TI-Writer (TI-99 GPL Mode)

1. Press any key to make the master selection list appear. Select "2" for TI-Writer

2. Select item 3 (utility) on the TI-Writer menu. This will cause the messages:

```
ENTER FILE NAME?
DSK1.UTIL1
```

on the screen. If the boot disk is in drive 1, press enter. If not, then edit the disk name (e.g. DSK2, DSK3, etc.) and press enter.

3. Loading is automatic from this point on. After loading two files, the FORTRAN master menu list will be displayed.

1.1.3 Booting from EXTENDED BASIC (TI-99 GPL Mode)

1. Press any key to make the master selection list appear. Select "2" for Extended Basic.

2. If the boot disk is in the first disk drive, the boot will be

automatic. If it was not, type:

```
RUN "DSK.FORTCOMP.LOAD"
```

3. Loading is automatic from this point on. After loading two files, the FORTRAN master menu list will be displayed.

1.1.4 Booting from MDOS Mode

There is no specific BOOT procedure for using the MDOS implementation of 9640 FORTRAN, all functions within 9640 FORTRAN execute as normal MDOS tasks.

Load up MDOS per your normal methods. A RAMDISK may be required by the compiler if your individual source modules exceed about 200 to 300 source lines. Also, you must have at least 128kbytes spare memory to use the FORTRAN linker.

The following is a recommended AUTOEXEC boot file which works well with MDOS 1.14:

```
RAMDISK 110
ASSIGN E=DSK5:
MODE RS232/1:9600
```

This AUTOEXEC file creates a ramdisk of 110k (about 430 sectors), assigns the RAMDISK to disk letter E:, and sets the mode of the RS232 port to 9600 baud.

Of course, this is only an example of how you might set up a RAMDISK. Note, however, that certain compiler operations require a RAMDISK to be set up as DSK5. Also, defining too large a RAM disk will create problems when attempting to compile, link, or debug a FORTRAN program.

1.1.5 Description of MENU (TI-99 GPL Only)

After 99 FORTRAN has been booted, the menu list will be displayed on the screen. Nine options are displayed as numbered selections:

```
                99 FORTRAN

        1 Edit          6 Librarian
        2 Compile       7 Load
        3 Link          8 USER
        4 Run           9 Utilities
        5 Run/Debug
```

On the bottom of the screen is a blinking underscore ("_"), which is the cursor.

To execute a function, depress the number of the function desired and press the ENTER button. For example, to edit a program, press "1" and

the ENTER button. If you make a mistake in entering the function desired, the backarrow (fctn/s) key can be used to delete the previous bad entry.

Functions 1 (edit), 2 (compile), 3(link), 4(run), 5(run/debug), 6(librarian), 8(USER), and 9(Utilities) require that the boot disk be present in one of the disk drives, if a different function was previously executed. Failure to insert the boot disk before the option is selected results in the message:

```
Input/Output Error
Press ENTER to Continue
```

To recover, press the "ENTER" button, and the function menu list will be redisplayed.

1.2 Special Keys

The following key buttons are recognized by the FORTRAN system whenever input is requested:

Button	Name	Description
Fctn 6	PROCD	Same function as ENTER.
Fctn 8	REDO	Returns to the FORTRAN main menu.
Fctn 9	BACK	Returns to the FORTRAN main menu.
Fctn =	QUIT	Returns to the master title (color bars) display.
Fctn S	Back Arrow	Deletes the previous character (if one was entered).
ENTER	-	Enters the current data typed on the screen.
CTRL/C	-	MDOS - Aborts the program and returns to MDOS prompt

During any output to the screen, depressing control/S (XOFF) will stop any output to the screen. Pressing control/Q (XON) will continue the output.

When editing programs, other function keys are enabled. These keys are described in section 2 of this manual.

2.0 Introduction

The EDITOR allows you to prepare source modules for input to the FORTRAN Compiler. The TI-99 implementation of the editor is similar to a reduced in scope Editor/Assembler Editor. The MDOS editor supplied is a version of Clint Pulley's QDE Editor.

2.1 TI-99 (GPL Implementation) Editor

To execute the editor, select option number 1 on the main menu screen. Selecting this option requires that the boot disk be present in one of the disk drives, if the editor was not the last function executed.

After the editor is loaded, the editor menu selection list is displayed:

```
                        99 Editor

        Press:

            1 To Load Program
            2    Edit Program
            3    Save Program
            4    Purge Workspace
            5    Display Statistics
            6    Print File
```

Item 1 (load) loads an old file from the disk into main memory called the workspace).

Item 2 (edit) allows editing the file which has been loaded, or a new file.

Item 3 (save) saves a file which is in main memory (workspace) to the disk.

Item 4 (purge) purges the file in memory.

Item 5 (statistics) displays statistics about the program currently loaded in the workspace.

Item 6 (print file) prints a formatted listing of the program currently loaded on the default printer or the user specified printer.

2.1.1 Load Program

A program on the disk requires loading before it can be edited. The editor expects all files which are loaded to be of type sequential, display, variable 80 character records.

Press 1 on the editor selection list to load an existing file. The prompt:

File to Load?

will be displayed on the screen. Enter the file name which is to be loaded.

For example, entering:

DSK2.FORTSRC

will load the file FORTSRC located on disk drive 2 into main memory.

After loading, the selection list is redisplayed and you may select another option. Each load removes the previous program in memory.

2.1.2 EDIT Program

The edit option allows you to edit the program currently in memory. When the edit option is selected, the first 24 lines of the file are displayed on the screen. If no program has been loaded, then the edit option clears the screen so that you can start editing a new program. The cursor is positioned at the upper left hand corner of the screen and is followed by an end of data marker (>EOD). Press <ENTER> to create a new line.

You may return from the edit mode back to the master selection list by entering pressing the two keys function and back (FCTN/BACK) simultaneously.

The program in memory will be lost if you exit the editor without saving it.

In the edit mode, the screen is 80 columns wide with three overlapping 40 character windows available for displaying the text. You start in the left most window with the first 40 characters of the display shown. Pressing <next screen> moves the display to window number 2, with columns 21 to 60 shown. Pressing <next screen> again displays window number 3, with columns 41 to 80. Pressing <next screen> again displays the first window.

If you have configured the FORTRAN system for 80 column mode operation (see the Utilities MENU), then the screen is composed of a single window of 80 characters, and the "next window" function key is inoperative.

The edit mode allows you to create, modify, and add text to the program file. When you press a key, that character is placed on the screen in the cursor position and the cursor moves one location to the right. (if the cursor is at the right margin, it will not move to the right). In addition, the edit mode has several special keys which perform special edit functions. The following table shows these special keys:

Key	Description

<ENTER>	Enters the text into the text buffer and places the cursor at the start of the next line. If ENTER is pressed at the end of a file, a blank line is automatically inserted (before the >EOD)
<arrow keys>	The arrow keys (fctn/s, fctn/d, fctn/e, fctn/x) move the cursor around the screen in the direction indicated.

In addition to the above keys, special function keys perform other editing functions:

Fct	Key	Description
f1	<delete char>	Deletes the character at the current cursor position. The remainder of the line is moved one character to the left.
f2	<insert char>	Moves the remainder of the line after the cursor one space to the right, and places a blank character at the current cursor position.
f3	<delete line>	Deletes the line at the current cursor position.
f4	<roll up>	Advances to the next page of 24 lines.
f5	<next screen>	Advances to the next screen in sequence. If on the last screen, returns to the first screen (inoperative in 80-column mode operation.
f6	<roll down>	Scrolls the screen down by 24 lines.
f7	<tab>	Advances the cursor (and screen) to the next column 7 in sequence.
f8	<insert line>	Inserts a blank line at the current cursor position.
f9	<back>	Returns to the editor selection list.

2.1.3 Save Program

After you have edited a file, it must be saved to the disk before it can be compiled. To use the save function, select item 3 on the editor editor selection list, and the prompt:

File to Save?

will be displayed. Enter the file name which will contain the edited program.

For example, entering:

DSK2.FORTSRC

will save the workspace contents to the file named FORTSRC in the second disk drive.

2.1.4 Purge Workspace

The purge function allows you to clear the current workspace contents in preparation for entering a new program. To select the purge function, enter 4 on the editor selection list. The message:

Are you sure (Y/N)?

will be displayed. Entering a Y at this point will cause the workspace to be cleared. Entering an N (or any other character except a Y) will return you to the editor selection list without clearing the workspace.

2.1.5 Display Statistics

The statistics function will display the following statistics about the program currently being edited (workspace):

1. Number of records.
2. Average record size (internal format).
3. Number of free bytes left.
4. Estimated number of records left.
5. Name of file which was last loaded.

Note that the editor has a maximum of 1000 records which can be in internal storage at any time, regardless of the individual record size.

To use this function, select 5 on the editor selection list. After the statistics have been displayed, the message:

Press ENTER to Continue

will be displayed. Depress ENTER to return to the editor selection list.

2.1.6 Print Program

Selecting option 6 will allow you to print a program to a printer device. When the Print Program item is selected, the question:

Enter Printer Name:
PIO

will be displayed. The printer name (PIO) will be whatever printer name you have selected in the UTILITY preferences list (see section 9). If the specified printer is ok, just hit enter. If you want the listing to go to a different printer, type the new printer name over

the old. For example, if you want the listing to go the the RS232 device at 4800 baud, then type:

RS232/1.BA=4800

and press enter. The following message will be displayed:

Printing, Hit any key to abort

and the printer will start printing. If you wish to abort the printing, then press any key on the keyboard. Otherwise, the printout will continue until completion, and the message:

Press ENTER to Continue

will be displayed. Press the ENTER key to finish.

2.1.7 Exiting the Editor

To exit the editor, press FCTN/BACK while on the Editor menu. The message:

Are you sure (Y/N)?

will be displayed. Type a Y to exit. Any other character will return you to the menu display.

Be sure to save your file to the disk before exiting the editor. Any files which are not saved will be lost.

2.2 MDOS Editor (QDE)

Included in the MDOS release of 9640 FORTRAN is Clint Pulley's Editor, QDE. I have chosen to include QDE, rather than write yet another editor.

Clint Pulley is providing good support for the QD Editor, and has given me his kind permission to include it in the release of 9640 FORTRAN. Clint has called the QDE editor the "Quick and Dirty" Editor. The user shall see that there is nothing Quick and Dirty about Clint's Editor, and I prefer to call it the Quality Dynamic Editor.

The reader should note that QDE is written in C99, not assembly or 9640 FORTRAN, and hopefully is the start of cooperative language developments for C99 and 99/9640 FORTRAN using the TI-99 and MYARC GENEVE.

The supplied version of QDE has been modified by LGMA Products for standard FORTRAN tab stops as the alternate tab stop set (control/9) instead of the Assembler tab stops.

Of course, the user can substitute any available editor for QDE, the only requirement is that it produce variable/80 card image formats.

2.2.1 Introduction

QDE is an 80 column screen editor for native MDOS on the Myarc 9640 computer. It is based on a public domain editor of unknown authorship which was written in C for MS-DOS on the IBM PC. The source code for that editor has been converted to c99, all assembly language functions have been rewritten, and numerous new features have been added. The result is QDE, a text editor (NOT a word processor) which is well suited for source program entry and the preparation of short documents.

QDE differs in many ways from the "standard" editor/word processor on the 9640, MYWORD. Wherever possible similar control keys have been used, but the internal design of QDE has led to a different approach for many operations. The best way of learning QDE may be to try everything on a test file, taking care to rename the output file or remove the diskette.

Implementation of QDE on the 9640 required the use of direct screen output (the "old-fashioned" way). The screen is completely rewritten after each keystroke, and the inherent speed of the 9640 hardware results in a flicker-free screen presentation. Unfortunately, the screen output functions in MDOS are far too slow to be used in this manner. As a result, QDE runs with interrupts disabled and may not coexist with future multiprogrammed applications. Testing with MDOS 1.14 has not revealed any problems.

2.2.2 Using QDE

QDE v1.8 requires 10 pages (81,920 bytes) of free memory. If a dual mode system has been configured (the AUTOEXEC file contains a TIMODE statement), set the RAMDISK size to allow sufficient free memory. The CHKDSK command will display available memory.

Assuming that the diskette containing QDE and SD is in the default drive, the editor is invoked from 80 column mode by the command line :

FQDE [filename] (any legal MDOS filename may be used)

(note: QDE is callable via the command FQDE instead of QDE. The reason is that the version of QDE included on the FORTRAN distribution disk has some slight modifications over the standard QDE for FORTRAN tab stops).

If invoked without a filename, FQDE presents a blank screen for editing. An output file name may be provided by using the "name output file" function key. If invoked with a filename, FQDE checks for the existence of the file. If it is not found, the user is queried for permission to create. A negative reply results in an exit to MDOS (allowing the user to recover from a misspelled filename), while a positive reply causes the filename to be retained for subsequent saves (the file is not created at this time). If the file is found it is loaded into the text buffer and the first screen of the file is presented for editing. The warning message "Input file truncated" is displayed if the file is too long for the buffer.

2.2.3 Acknowledgement

Ralph Ford of Auburn, Alabama was kind enough to send me a copy of the original editor some time ago, but QDE had to wait for the availability of the Myarc 9640 computer and the MDOS implementation of c99.

2.2.4 QDE Function Key Usage

The ^ character is used to indicate pressing the Control key together with the indicated key.

The arrow keys are used for cursor movement.

F1 - Delete character under the cursor (also Del key)
^F1 - Oops! Recovers line altered by in-line editing
F2 - Start insert mode (terminated by any function key) (also Ins key)
^F2 - Paragraph pack (stops at next line with space in the first position)
F3 - Delete current line
^F3 - Undelete line (see note re use of delete buffer)
F4 - Page screen up (also Pg Up key)
^F4 - Roll screen down, retaining cursor position on screen
F5 - Display current editor status
^F5 - Duplicate current line
F6 - Page screen down (also Pg Dn key)
^F6 - Roll screen up, retaining cursor position on screen

F7 - Split line at cursor position
^F7 - Join next line to current line
F8 - Insert blank line before current line
^F8 - Swap current and next line
F9 - Find string (prompts for search string)
^F9 - Find next string (uses previously entered search string)
F10- Replace string (prompts for search and replacement strings)
^F10- Replace next string (uses previously entered strings)

^9 - Toggle tab mode between normal (every 4 columns) and FORTRAN (columns 7,37,73,80) tab stops. (Note this is different than the standard QDE).

^0 - Change screen color (4 sets)

^A - Append marked lines to end of capture buffer
^B - Move cursor to beginning of line
^D - Delete marked lines to capture buffer (rewrites buffer)
^E - Move cursor to end of line
^F - Show diskette directory (prompts for drive number [1-5])
^G - Get file (prompts for filename)
^H - Backspace (deletes character under cursor)
^I - Tab to next tab stop (also tab key)
^K - Clear from cursor to end of line
^L - Set left margin at cursor position
^N - Name output file (prompts for filename)
^O - Home cursor to screen upper left
^R - Rewrite capture buffer with marked lines (does not delete)
^T - Transfer capture buffer to text buffer at current line
^U - Enter control character (the next character is biased by -64)
^V - Move cursor to left margin
^W - Wipe text buffer and clear filename (prompts for confirmation)
^X - Set mark for capture buffer functions (X marks the spot!)
^Y - Reset left margin to position 1
^Z - Move to end of text buffer

Esc - If followed by second Esc, terminates editing and prompts for file save (uses filename from command line or ^N) If output file errors occur, returns to editing

2.2.5 QDE Usage Notes

1. The TEXT buffer - This is the visible work area for the editor. Files are loaded into and saved from the text buffer.

2. The DELETE buffer - QDE has a circular delete buffer. Deleted

lines are copied into this buffer and may be retrieved (in reverse order) with the

Undelete function key. This provides a useful method of moving a few lines to another location in the text buffer.

3. The CAPTURE buffer - QDE has a linear capture buffer which can be used to move larger blocks of text within a file or between files. The function keys used for capture buffer operations are Set Mark (^X), Append to Buffer (^A), Delete Marked (^D), Rewrite Buffer (^R) and Transfer Buffer to Text (^T). A block of text is delimited by moving the cursor to the first line in the block, pressing ^X, then moving the cursor to the last line of the block.

For single line operations, use of ^X is not required. The operations which move the marked block to the capture buffer are :

^A - Appends the block to the end of text already in the capture buffer.
This operation does not delete the marked block from the text buffer.

^D - Replaces the contents of the capture buffer with the marked block and deletes the block from the text buffer.

^R - Replaces the contents of the capture buffer with the marked block. This operation does not delete the marked block from the text buffer.

The mark set by ^X is cleared by ^A, ^D or ^R. Any capture buffer operation which would exceed the buffer's capacity is not executed.

The contents of the capture buffer are inserted into the text buffer by moving the cursor to the desired location and pressing ^T. The capture buffer is not cleared by this operation, so multiple transfers are possible. The capture buffer can be used for moving text between files (ie. capture a block of text, get another file, and transfer text to the new file).

4. Non-printing characters are generated by pressing ^U which results in the next character being placed in the text buffer biased by 64 (this results in the same character mapping as in MYWORD). If multiple non-printing characters are required, ^U must be pressed before each character. The symbols displayed on the screen when non-printing characters are present are those provided by MDOS's character pattern definitions.

5. When entering large files, the status display should be checked periodically to ensure that the text buffer does not become full. It is good practice to leave 30-50 lines available to allow for future expansion of the file. When the text buffer is full insertion of new lines is inhibited so existing text is not lost.

6. A printed listing can be obtained by using ^N to change the output

filename to the printer and then doing ^S. The output filename should be reset afterwards to avoid unwanted listings.

7. The paragraph pack function provides a very limited document preparation capability. All lines from the current cursor location to the line preceding the occurrence of a space in the first character position are packed with as many complete words on a line as possible. Lines at the end of the paragraph which become empty are cleared but not deleted in this version of QDE.

8. QDE v1.8 buffer capacities are : Text - 560 lines, Delete - 5 lines, Capture - 64 lines.

9. QDE v1.8 uses the 26 row screen mode and multitasking capability of MDOS and may not function correctly with system versions prior to 1.14. The show directory function requires that the program image file "SD" be on the default device.

10.The "display line numbers" screen mode of MYWORD could not be implemented in QDE because of its text buffer/screen write design.

3.0 Introduction

The FORTRAN Compiler compiles the program prepared with the EDITOR into an object module. A listing with interspersed object code, allocation summaries, and label summaries can also be produced.

3.1 FORTRAN Statements

The basic unit of FORTRAN is a FORTRAN statement. A statement consists of one or more lines of valid FORTRAN code.

3.1.1 Types of Statements

Each statement within a FORTRAN program or subprogram is one of the following types:

1. Assignment statements assign values to variables. The values can be numeric (of three precisions) or logical.

2. Input/Output statements direct information from/to your program to a peripheral device, such as the screen or a disk file.

3. Control statements control the flow of your program, either by making decisions and executing conditional code, or unconditional branches.

4. Declaration statements define the variables you will use in your program, their sizes and attributes.

5. Identification statements identify a unit of compiled code, and the parameters which are passed to and from the unit.

6. Comment statements are interspersed throughout the program, and describe the flow and structure of the program.

3.1.2 Statement Format

The FORTRAN compiler expects certain items to be in certain columns of the source line, as follows:

Column 1-5: Contains an optional positive integer number used to label a statement so that it can be referred to by other statements.

The statement label must be an integer value between 1 to 99999.

Statement labels may appear in any order, although it is recommended for ease of debugging that they be placed in ascending order.

Column 1: The letter 'C' in column 1 indicates that the line is a comment line. It will be printed in the source listing

but will have no other effect on the program.

Column 1: The letter 'D' in column 1 indicates that the statement is to be conditionally compiled.

If the option 'DM' was selected on the compile time options menu, the statement will be compiled as a regular FORTRAN statement.

If the option 'DM' was not selected on the compile time options menu, the statement will be treated as a comment line.

Column 6: Any character in column 6, other than a blank or a zero, indicates that this line is a continuation of the previous line. All statements, including comments and conditionally compiled statements, may be continued in this manner.

Columns 7-72: Specify the FORTRAN statement. Statements may have blanks inserted, as desired, to improve readabilty except within quoted or hollerith fields.

An exclamation mark (!) encountered in the field means that the rest of the line is to be ignored (treated as a comment), as long as it is not contained within a Hollerith field.

If you have a preference for coding in lower case, FORTRAN will translate all characters not within quotes to upper case. For example, the statements:

```
IMPLICIT INTEGER(A-Z)
      and
implicit integer(a-z)
```

are equivalent.

Columns 73-80: Are ignored by the compiler.

Blank lines encountered by the FORTRAN compiler are treated as comment lines, i.e. as if they had the character 'C' in column 1. The first AND last lines of a program or subprogram may not be blank. The last line of a program or subprogram may not be blank, or an I/O error results during compilation.

3.1.3 Constants

Unlike BASIC, which has two types of constants (numeric and string), FORTRAN has six types of constants:

A. Numeric Constants

1. Integer *1 (1 byte)
2. Integer *2 (2 bytes)
3. Integer *4 (4 bytes)
4. Single Precision (real *4, 4 bytes)
5. Double Precision (real *8, 8 bytes)

B. Logical Constants (2 bytes)

C. Hollerith Constants

D. Hexadecimal Constants

E. Octal Constants

F. Binary Constants

3.1.3.1 Integer Constants

Integer constants are written as a string of decimal digits, optionally preceded by a sign. Leading zeros are ignored. The value of an integer constant must lie in the range of -2147483648 to +2147483647. An integer constant occupies two or four bytes of memory according to its value, as follows:

```
     -32768 to +32767      : Integer *2 Constant
-2147483648 to +2147483647 : Integer *4 Constant
```

Examples:

```
240
+16
-100
1000000
0
```

3.1.3.2 Single Precision Constants

Single precision constants are written as a string of decimal digits, optionally preceded by a sign, containing a decimal point and/ or followed by a decimal exponent. A decimal exponent for a single precision constant is written as the letter E followed by an integer constant of the form described above.

Examples:

```
5.
15.4
+5E0
1.5E3
1.5E-3
-3.14159
```

The decimal exponent indicates the power of ten by which the number is to be multiplied (scientific notation). This value may be zero. The decimal point may be omitted if the decimal exponent is specified. It is then assumed to lie to the right of the number.

A single precision constant has a precision of 5 digits, and a magnitude of 1E-127 to 1E127. Any number of digits may be specified, but only the most significant digits will be retained. A single

precision constant occupies four bytes (two words) of memory.

3.1.3.3 Double Precision Constants

Double precision constants are written as a string of decimal digits, optionally preceded by a sign, containing a decimal point and/or followed by a decimal exponent (scientific notation). A decimal exponent for a double precision constant is written as the letter D followed by an integer constant of the form described above.

Examples:

```
5.D0
15.4D0
+5D0
1.5D3
1.5D-3
-3.14159D0
37D2
```

The decimal exponent indicates the power of ten by which the number is to be multiplied. This value may be zero. The decimal point may be omitted if the decimal exponent is specified. It is then assumed to lie to the right of the number.

A double precision constant has a precision of 13 digits, and a magnitude of 1D+127 to 1D-127. Any number of digits may be specified, but only the most significant digits will be retained. A double precision constant occupies eight bytes (four words) of memory.

3.1.3.4 Logical Constants

There are only two logical constants, represented by the values TRUE and FALSE. They take two bytes (one word) of memory, and are written as:

.TRUE. and .FALSE.

3.1.3.5 Hollerith Constants

A Hollerith constant is a string of ASCII characters. Any ASCII character can be used. A Hollerith constant may be specified either by placing the character string within quotes ('), or by preceding it with nH, where n is the number of characters, including blanks, in the character string.

Within a quoted string (but not an H string), the quote character itself is represented by two consecutive quote characters. Note that quote characters are not considered consecutive if they are seperated by a blank.

A Hollerith constant occupies one byte of memory for each character, and is stored left justified in its field. If necessary, a blank is

added to the last byte to fill the last word.

Hollerith constants are evaluated to a constant type according to the number of characters specified in the Hollerith string, according to the following table:

```
1 character       : Integer *1
2 characters      : Integer *2
3 or 4 characters : Integer *4
5 to 8 characters : Double Precision (Real *8)
```

Examples:

```
'GA'
2HGA
'XYZ'
4HINT4
8HD.P.TYPE
'D.P.TYPE'
6H**YES*
'A''B'
```

3.1.3.6 Hexadecimal Constants

A hexadecimal constant is a string of up to 16 hexadecimal digits preceded by a quote and ended by a quote and the letter X ('X). Alternately, the FORTRAN 77 form of the letter Z, followed by a quote, the constant, and an ending quote may be used. A hexadecimal digit is one of the characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F

A hexadecimal digit occupies 2 bytes of memory for each four digits, and is stored right justified in its field. If necessary, zeros are added on the left to fill the first word.

Hexadecimal notation is just a different way of specifying a numeric constant, and may be used anywhere a numeric constant is used.

A hexadecimal constant is evaluated to a constant type according to the number of digits specified in the constant, as follows:

```
1 or 2 digits : Integer *1
3 or 4 digits : Integer *2
5 to 8 digits : Integer *4
8 to 16 digits: Double Precision (Real *8)
```

Examples:

```
'123456'X (or) Z'123456'
'1BF'X    (or) Z'1BF'
'0123456789ABCDEF'X (or) Z'0123456789ABCDEF'
```

3.1.3.7 Octal Constants

Octal constants are specified as the letter O, followed by a quoted string containing valid octal disits zero (0) through seven (7). An octal constant is evaluated to a constant type according to the number of digits specified, as follows:

```
1 to 3 digits   : Integer *1
3 to 6 digits   : Integer *2
6 to 11 digits  : Integer *4
11 to 22 digits : Double Precision (Real *8)
```

Valid octal constants would be:

```
O'1734'
O'7124316'
```

3.1.3.8 Binary Constants

Binary constants are specified as the letter B, followed by a quoted string containing valid binary digits zero (0) and one (1). A binary constant is evaluated to a constant type according to the number of digits specified in the constant, as follows:

```
1 to 8 digits   : Integer *1
9 to 16 digits  : Integer *2
17 to 32 digits : Integer *4
33 to 64 digits : Double Precision (Real *8)
```

Valid binary constants would be:

```
B'10011'
B'10010001111001010001'
```

3.1.4 FORTRAN Names

FORTRAN names are used to identify variables, main programs, subprograms, statement functions, and dummy arguments.

A FORTRAN name is one or more characters in length, but must begin with a letter, and consist of string of letters, digits, dollar signs, and underscores. Other characters are invalid. Only the first nine characters of a variable are used. Imbedded blanks are ignored.

Examples:

```
GET_POINT
SAVEFILE
TI99_4A
BAD DATA
lower_case
```

Note that although lower case letters are allowed, 99/9640 FORTRAN simply translates the statement to upper case (except for quoted strings). Therefore, the variables "LOWER_CASE" and lower_case are exactly the same.

FORTRAN variables are either explictly declared (using an explicit declaration statement such as REAL, INTEGER, etc.) or if no explicit declaration exists, implicitly declared according to the following rules:

1. If the name begins with a letter declared in an IMPLICIT statement, the variable will be assigned the type declared in that statement.

2. If the name begins with letter I, J, K, L, M or N, the variable will be assigned the type of integer.

3. If the name begins with any other letter, the variable will be assigned the type of Single Precision.

3.1.5 FORTRAN Variables

All variables must be given a name. A variable's value is allowed to change during program execution (although it does not necessarily have to). The data type of the variable describes the type of data the variable represents. For example, an integer variable represents integer data.

A variable name can represent a single value or an array of values.

Single variables need not be declared as single; they are defined from context when they are used. Array variables must be declared using the INTEGER, REAL, DOUBLE PRECISION, or DIMENSION.

3.1.6 Arrays and Subscripts

An "array" is a set of data elements which are identified by a single variable name. The array name and its bounds must be declared in a DIMENSION statement, in a dimensional COMMON statement, or a type specification statement.

Subscripts are used to reference particular elements of the array. When used, the subscript list must follow the array name and be enclosed in parenthesis. The subscript list is a sequence of integer expressions seperated by commas. A subscript expression must have one of the following forms:

```
i
k
j+/-k
j*i
j*i+/-k
```

where:

i is an integer scalar variable
j and k are constants

3.1.7 Subscripted Variables

A single element of an array are referred to by forming a subscripted variable. This variable consists of the array name followed by a subscript list. The value of the subscripts determine which element of the array is being referenced. The number of subscripts associated with the variable must equal the number of subscripts defined in the DIMENSION statement for the array, or a compilation error will result.

Examples:

```
A(2)
K(I,I+2,I+3,6)
PI(J+2,7*K)
```

A subscript expression MUST be evaluated to type integer *2 in the current version of 99/9640 FORTRAN.

3.1.8 Expressions

Expressions appear on the right hand side of an equals sign (=), and consist of constants, operators, parentheses, variable names, and function references. There are three types of expressions, numeric, relational, and logical.

3.1.8.1 Numeric Expressions

Numeric expressions consist of numeric constants, numeric variables, function references, and the numeric operators for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**).

The minus sign (-) and plus sign (+) may be used in their unary form as well as with addition and subtraction. As unary operators, the plus sign (+) has no effect, while the minus sign (-) reverses the sign of the expression that follows it.

All expressions yield a single numeric value.

Numeric operations are performed in a hierarchal order. The highest order operators (those which are performed first) are the unary operators (+ and -), followed by exponentiation, followed by multiplication and division, and then by addition and subtraction. Parentheses may be used at any time to alter the order of hierarchal operation.

Operands in a numeric expression may be of type integer, single precision, or double precision.

The following are examples of numeric expressions:

```
4/2
4.0/2.0
2.0*PI*R
```

```
A**2+B**2-C
```

FORTRAN does allow mixed types of numeric modes within the same expression, such as mixing integer values with single precision values, integer with double precision, etc. When this situation occurs, a warning is issued by the compiler (mixed mode arithmetic), and the lower type items in the expression are converted to the higher types. For example, the expression:

```
1/2.0
```

would be evaluated as type single precision, not integer, due to the single precision constant 2.0.

3.1.8.2 Relational Expressions

Relational expressions are used in IF statements, DO WHILE statements, and logical expressions. They consist of numeric expressions seperated by relational operators. The result of a relational expression is always a single value of TRUE or FALSE.

The relational operators are:

Operator	Definition
.GT.	Greater than (>)
.GE.	Greater than or Equal To (>=)
.LT.	Less Than (<)
.LE.	Less Than or Equal To (<=)
.EQ.	Equal To (=)
.NE.	Not Equal To (<>)

The enclosing periods are part of the relational operators and must appear.

Examples:

3.LT.10	is true
A.EQ.2.*B	could be true or false, depending on the values of A and B

Logical expressions are evaluated from left to right, after all numeric operations have been completed.

3.1.8.3 Logical Expressions

Logical expressions are used with relational expressions. The logical operators are:

Operator	Definition
.AND.	True if left and right operands are both true.
.OR.	True if left and right operands are both true.
.EOR.	True if either left or right operand is true, but

not both
.NOT. True if operand following is not true.

Examples:

```
.NOT.  K      is true if K is false
 L.AND.N     is true if L and N are true
```

When parenthesis are omitted, logical expressions are evaluated in the following order:

1. arithmetic expressions
2. relational expressions
3. Logical operators in the order of .NOT., .AND., .OR., and .EOR..

Operations of the same order are performed left to right.

Parenthesis always override the hierarchal order of operations.

For example, the statement:

```
N.EOR..NOT.M.AND.R.LT.S
```

is the same as writing:

```
N.EOR.((.NOT.M).AND.(R.LT.S))
```

3.2 Assignment Statements

An assignment statement assigns a value to a variable. Assignment statements include arithmetic and logical type assignments.

3.2.1 Arithmetic Assignment Statements

General Form:

```
a = b
```

where:

a is a variable (scalar or array) whose type is integer, real, or double precision, and

b is an arithmetic expression

The expression on the right is evaluated and the resulting value is put into the variable on the left. The value of b replaces the value currently associated with variable a. For example:

```
L = L + 1
```

specifies that the new value of L is to be the current value of L incremented by 1.

If the data type associated with b differs from that associated with a, the value of b is converted automatically to the type associated with a. For example, a real value can be rounded to the nearest integer by the following:

```
I=D+.5
```

where D is a single precision variable, and I is an integer variable, .5 will be added to D, and the fractional part is truncated and stored into the integer value I.

Examples:

```
K = K + J
D = P(I,J)+SIN(T)
ARRAY(K) = 2.0*-R+(P/3.0)
```

3.2.2 Logical Assignment Statements

General Form:

```
a = b
```

where:

a is a variable (scalar or array element) whose type is logical, and

b is a logical expression

Examples:

```
LOGICV = .FALSE.
L=B.GT.100.
V(I)=(R.EQ.T).AND.(B.LT.100.)
```

3.2.3 Control Statements

Control Statements allow you to govern the order of execution of the program statements.

3.2.3.1 GO TO Statements

GO TO statements transfer control forwards or backwards within a program. FORTRAN has two forms of the GO TO statement, unconditional and conditional.

3.2.3.2 Unconditional GO TO Statement

General Form:

```
GO TO n
```

where n is a statement label

This statement transfers control to statement labeled n.

For example:

```
GOTO 1000
```

will cause control to be transferred to the statement labeled 1000.

3.2.3.3 Computed GO TO Statement

General Form:

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

where:

n1, n2, ..., nm are statement labels and

e is any integer, single precision, or double precision expression.

The arithmetic expression a is converted to integer (truncated), and becomes a pointer as to which statement label n1,n2,...,nm is branched to. If a is 1, then statement label n1 is performed next. If a is 2, then statement label n2, etc. If the value of a is outside the number

of labels specified (less than 1 or greater than m), then the next statement is sequence is performed.

Examples:

```
GO TO ( 1000, 1100, 1200 ), L
```

would cause a branch to statement 1000 if L was 1, a branch to statement 1100 if L was 2, a branch to statement 1200 if L was 3. If L was neither of these values, the next statement would be performed.

```
GO TO ( 1000, 1100, 1200 ), X
```

would cause a branch to statement 1000 if X was greater than or equal to 1.0 and less than 2.0, a branch to statement 1100 if X was greater than or equal to 2.0 and less than 3.0, a branch to statement 1200 if X was greater than or equal to 3.0 but less than 4.0. If X was outside any of these ranges, then the next statement would be performed.

Note that real values are truncated, not rounded.

3.2.4 IF Statements

IF statements are conditional transfer statements used to change the normal path of program execution based on an arithmetic or logical expression. There are three forms of the IF statement, arithmetic, logical, and structured.

3.2.4.1 Arithmetic IF Statement

General Form:

```
IF ( a ) n1,n2,n3
```

where:

a is an arithmetic expression

n1, n2, n3 are statement labels

Control is transferred to the statement labeled n1,n2, or n3 if the value of the expression a is negative, zero, or positive, respectively.

Example:

```
IF ( B(I) - PI ) 100, 400, 300
```

If the value of the above expression is negative, control will transfer to statement label 100, if it is zero it will transfer to statement label 400, if it is positive it will transfer to statement label 300.

In an arithmetic IF statement, all three statement labels must be supplied, however, any two may be the same.

3.2.4.2 Logical IF Statement

General Form:

```
IF ( a ) s
```

where:

a is a logical expression, and

s is any valid executable FORTRAN statement, except a DO statement or another logical IF statement.

If the value of a is true, statement s is executed, and the computer will jump to the next sequential statement. If the value of a is false, the computer will jump to the next sequential statement.

Examples:

```
IF ( LOGIC1 .AND.  LOGIC2 ) PI=3.14159
IF ( I .EQ.  J ) GOTO 200
IF ( (A.NE.B) .AND.  (J.LT.2)) GOTO (100,200,300),K
IF ( I ) CALL DELSPR(1)
```

3.2.4.3 Structured IF Statement

General Form:

```
IF ( a ) THEN
    s1
ELSEIF ( b ) THEN
    s2
ELSE
    s3
ENDIF
```

where:

a and b are logical expressions

s1,s2,s3 are valid FORTRAN executable statements

The IF statement, as shown above, must be followed by the single keyword THEN. If the value of a is true, control is transferred to the statement or group of statements following the IF statement.

After the IF .. THEN and associated group of statements, is an optional keyword ELSEIF. If the value of the logical expression b is true, control is transferred to the next statement or group of statements.

After the ELSEIF .. THEN and associated group of statements, follows an optional keyword ELSE. If the value of the logical expression b is false, control is transferred to the next statement or group of statements.

Following the ELSE keyword, is the required statement ENDIF. This keyword terminates the IF .. THEN .. ELSEIF .. ELSE .. structure.

One restriction in the structured if, is that do loops are not allowed to extend to the middle of the IF .. ENDIF structure. For example:

```
      DO 1000 I=1,10
      IF ( I .EQ.  0 ) THEN
 1000 CONTINUE
      ENDIF
```

is invalid, and will cause a compilation error (illegal doloop).

Example 1:

```
     IF ( I .GT.  0 ) THEN
        I = I + 1
        X = SIN(Y)
     ENDIF
```

In the above example, if I is greater then 0, then control is transferred to the next group of insructions. If I is not greater than 0, then control is transferred to the ENDIF statement.

Example 2:

```
      IF ( I .GT.  0 ) THEN
        I = I + 1
      ELSEIF ( J .LT.  2 ) THEN
        J = J + 1
      ELSE
        K = K + 1
      ENDIF
```

3.2.5 DO Statement

General Form:

```
       DO n v=a1,a2[,a3]
```

where:

n is a statement label

v is a non-subscripted variable, and

a1,a2,a3 are arithmetic expressions

The DO statement allows repetitions of all of the statements up to and including the statement n, until the value s is greater than or equal to that of the value a2. The value of v is initially set to that of a1, the body of the DO statements execute, the value of v is incremented by a3 (or 1 if a3 is not specified) and compared to that of a2. If the value of a is less than that of a2, then the loop repeats with the new value of v, else the loop terminates

Note that in this version of 99/9640 FORTRAN, the variable s and arithmetic expressions a1, a2, and a3 can be any type of integer or real.

For example, the statements:

```
      J=0
      DO 10 I=1,100
10    J=J+I
```

will perform a summation of the numbers from 1 to 100, and leave the result in the variable J.

Throughout the range of the DO, the index variable v is available for computation, and can be used as a value either for calculation or, if it is an integer *2 variable, as a subscript variable. If you change its value in the DO loop, the number of loop iterations will be affected.

When a DO loop is exited, either when the value is satisfied or when a control transfer statement in range of the DO is executed, the index variable will have the last value attained. If the loop has been satisfied, this will be greater than that of a2.

The expressions a1, a2, a3 are the controls of the DO loop and should not be altered within the range of the DO.

You may branch out of a DO loop using any control transfer instruction, but you may not branch directly into a DO loop using these statements. The results of doing this are unpredicable since the index is not properly initialized.

The statement which terminates a DO loop cannot be one of the following:

```
DO
GO TO
Arithmetic IF
Structured IF
DO WHILE/ENDDO
STOP
PAUSE
RETURN
```

3.2.6 CONTINUE Statement

General Form:

```
CONTINUE
```

This statement is a dummy statement which merely transfers control to the next executable statement in sequence. It is primarily used as a target point for transfers, particularly as the last statement of a DO loop which would otherwise end with a prohibited statement.

3.2.7 DO WHILE/ENDDO/REPEAT Statements

General Form:

```
DO WHILE ( a )
    s
ENDDO
```

or

```
DO WHILE ( a )
    s
REPEAT
```

where:

a is a logical expression, and

s is a valid FORTRAN statement or group of statements.

The structured DO WHILE/ENDDO or DO WHILE/REPEAT constructs execute the body of the DO loop while the logical expression a is true. For example:

```
I= 0
DO WHILE ( I .LT.  100 )
   K = I * K
   I = I + 1
REPEAT
```

In the example above, the DO WHILE statement governs a loop. The loop will execute 100 times until i equals 100.

The test on logical expression a occurs at the start of the loop, so if the logical expression a is false at the start of the DO WHILE loop, the loop will not be executed. For example:

```
I = 0
ENDLOOP = .FALSE.
DO WHILE ( ENDLOOP )
   I = 1
ENDDO
```

In the above example, the value of I at the end of the do loop will be 0, not 1, since the value of ENDLOOP was false at the start of the loop.

Note that the REPEAT and ENDDO statements perform exactly the same function, they are the end of the range for the DO WHILE loop.

3.2.8 PAUSE Statement

General Form:

```
PAUSE [i]
```

where i is an integer constant.

The PAUSE statement causes temporary suspension of the program execution and causes the messages:

```
* PAUSE i

Press ENTER to Continue
```

to be displayed on the screen when the statement is executed. The value i is displayed in decimal.

The constant i need not be specified. If it is omitted, a value of 0 is assumed.

3.2.9 STOP Statement

General Form:

```
STOP [i]
```

where i is an integer constant.

The STOP statement is used to indicate the logical end of a program when it does not coincide with the physical end. This statement terminates the execution of a running program. When execution terminates, the message:

```
* 9640 FORTRAN Stop
        or
* 99 FORTRAN Stop
Press ENTER to Continue
```

will be displayed on the screen. The optional value i is displayed in decimal. In 99 FORTRAN, pressing ENTER will cause a return to the FORTRAN Menu. In 9640 FORTRAN, the program automatically returns to the MDOS prompt.

To eliminate the STOP message when a FORTRAN program terminates, use the CALL EXIT library subroutine call (see chapter 7).

3.2.10 END Statement

General Form:

```
END
```

The END statement indicates the physical end of the program. Each program, subroutine subprogram, or function subprogram must contain an end statement as the last statement. If control passes to the END statement, it will behave in the same manner as a STOP or RETURN statement.

3.3 Input/Output Statements

Input/ Output statements are one of three types:

- READ Statements
- WRITE Statements
- FORMAT Statements

READ and WRITE statements define the data which is to be transferred, the format number which is to be used for the transfer, the device number, and what to do in case of error or end of file conditions. FORMAT statements define how the data is to be transferred.

3.3.1 READ Statement

General Form:

READ (i, n [,keys]) variable list

where:

i is an arithmetic expression, of type integer, which designates the input/ output FORTRAN unit number. It must not be zero and must be either 6 (for the screen), or a number used in a CALL OPEN statement.

n is the label of a FORMAT statement or the name of an array which contains the format specification.

keys is an optional list of keyword parameters which define special case information about I/O operation, and

variable list is an optional list of variables to be transferred.

The keyword parameters, which can appear in any order, are:

END=j where j is the label of a statement to which control is to be transferred if an end of file condition occurs on a disk file, or the characters >EOD are typed in on the screen.

ERR=k where k is the label of a statement to which control is to be transferred if an I/O error occurs.

REC=i where i is an arithmetic expression, evaluated as type integer, which indicates the number of the record to be accessed from a random file.

STATUS=m where m is an integer scalar variable into which status information is to be stored if an I/O error occurs.

The READ statement transfers ASCII data from the device specified by i under control of format n. If n specifies an array, the array must contain a valid format specification, beginning with the initial left parenthesis and terminating with the final right parenthesis.

Examples:

```
READ ( 6, 10 ) A,B
READ ( J, FORMT, END=599, ERR=388, STATUS=J) (K(I),I=1,10)
```

3.3.2 WRITE Statement

General Form:

```
WRITE ( i,n [,keys] ) variable list
```

where: i,n,keys and list are as defined above for the READ statement.

The WRITE statement transfers ASCII data to the device specified by i under control of format n. If n specifies an array, the array must contain a valid format specification, beginning with the initial left parenthesis and terminating with the final right parenthesis.

Examples:

```
WRITE ( 6, 20 ) A,B,C
WRITE ( IDEVICE, 9100, REC=I, ERR=1000) A,B,C
WRITE ( 6, FORMT, ERR=100, STATUS=J, END=400 ) (ARRAY(I),I=1,10)
```

3.3.3 Variable Lists

READ and WRITE statements generally require a list of variables to be transmitted.

A variable list can merely contain variables (subscripted or non-subscripted) seperated by commas. For example:

```
READ ( 6, 2 ) A,B,C,J(J),M(2,2),K
```

A variable list can also specify array variables which may be indexed and incremented in the same manner as a DO loop. These array variables and their indexes are enclosed in parenthesis to seperate them from other variables in the list. For example:

```
WRITE ( 6, 400) J, (X(K),K=1,100), L
```

Initial (m1), limit (m2), and step (m3) values for index variables are specified in the DO statement. If the step value is not specified, 1 is assumed. As for DO statements, unlimited nesting of indexing is allowed. For example, if K is a 2x3 array, the statement:

```
READ ( 1,100 ) ((K(I,J),I=1,2),J=1,3)
```

will cause the data values to be transmitted into the elements of the array in the following order:

```
K(1,1)
K(2,1)
K(1,2)
```

```
K(2,2)
K(1,3)
K(2,3)
```

When the entire array is to be transmitted, the indexing may be omitted, and only the array name written. Values are transmitted to or from the the array in order of increasing subscripts, with the leftmost subscript varying most rapidly. Thus, the preceding example could have been written

```
READ ( 1,100 ) K
```

In READ statement I/O lists of the form M, A(M) or M,(A(I),I=1,M), the value of M is read before processing begins for array A. Thus, the subscript calculation or DO limit check uses the newly read value for M rather than its former value. For example, the statements:

```
M = 3
READ ( 1,100 ) A(M), M
```

will read a new value for A(3), and then a new value for M, whereas the statement:

```
READ ( 1,100 ) M,A(M)
```

will read a new value for M, and then use that value to determine which element of A should receive the next value.

3.3.4 Input/Output Device Unit Assignments

The parameter i in input/ output statements specifies the physical device involved in an I/O operation. Physical devices are specified as FORTRAN unit numbers specified in CALL OPEN statements. Also, there are two unit numbers which are opened for you, as follows:

Device 6 - Is always assigned to the CRT (aka SCREEN or Console)
Device 9 - Is assigned to the default printer.

In the MDOS implementation of 9640 FORTRAN, device 9 is assigned to the PRN: device. In the TI-99 GPL implementation, device 9 can be changed to another unit number, and its default printer name assigned using the Preferences Utility (see section 9 of this manual).

The parameter i can also be specified as an asterick (*). In this case, the wild-card unit number in the preferences section is used to determine the device number. In this way, a program can be coded with wild card input and output, and have its input/output assignment changed externally using the Preferences utility. For MDOS, the default device is always assigned to device 6, the CRT.

Examples:

```
WRITE ( 3, 100 ) X
WRITE ( 6, 100 ) X
```

```
READ ( *, 100 ) X
```

specifies input/output on devices 3, 6, and the wild card unit number (*) specified in the Preferences Utility. It is assumed here that device 3 was previously opened in a CALL OPEN statement.

3.3.5 FORMAT Statements

FORMAT statements are used in conjunction with READ and WRITE statements to process ASCII records. Format specifications with the FORMAT statement define record layout and specify the conversions to be performed between internal and external data representations. The FORMAT statement is a non-executable statement, and may appear anywhere in a program.

General Form:

```
label FORMAT ( S1, S2, ..., Sn )
```

where:

Si is a format specification of one of the forms described below, or a repeated group of such specifications in the form:

```
r ( s1, s2, ..., Ss )
```

where:

r is a repeat count of the form described below, and

sj is as described above; in other words, nested repetitions are allowed (up to three levels)

Each FORMAT statement must be given a statement label so that it can be referenced by a READ or WRITE statement.

3.3.6 FORMAT Specifications

A FORMAT specification may have any of the following forms:

rFw.d	rIw	rAw	r's'	rEw.d
rZw	rRw	r/	rCi	rDw.d
rLw	nHs	rX	Mi.j	rQ

The characters F,E,D,I,Z,L,A,R,H, quote('), slash(/),X,C,M and Q define data conversion, editing, and format control.

r is an optional unsigned integer constant which indicates that the specification is to be repeated r times (if omitted, 1 is assumed).

w is an unsigned integer constant which defines the total field width in characters of the external representation of the data being processed (including digits, signs, decimal point, exponent and blanks).

d is an unsigned integer constant which specifies the number of decimal digits which appear to the right of the decimal point in the real data being processed.

i is an unsigned integer constant which qualifies the format

specification (see descriptions of C and M specifications)

n is an unsigned integer constant which specifies the number of characters following an H format specification. s is a string of ASCII characters, and

j is an unsigned integer constant which qualifies the W format specification.

3.3.6.1 Floating Point Data Conversion

The F,E and D Format specifications are used to specify single or double precision data transfer. On input, these specifications are identical, and may be used interchangeably. On output, they differ only in the way that the data is to be represented; either single or double precision data may be output under any of these specifications.

F Format (Fixed Decimal Point) rFw.d

Input - The input field has an overall width of w characters. It may contain an optional sign, a string of digits with or without a decimal point, and an optional exponent. A blank in the input will terminate the field. A blank field will be read as a zero.

An exponent may be specified by one of the characters E, D, + or -, followed by a string of digits. If E or D is used, a sign may appear before the digit string. The exponent specifies the power of 10 by which the number is to be multiplied.

For example, using F8.2 format:

```
802142 on input is converted to 802142.0
 .34 2 on input is converted to .34
-7.E1  on input is converted to -70.0
614D+2 on input is converted to 61400.
```

Output - Internal values are output right justified in the w character field, preceded by blanks. For F conversion, the field width w should provide space for the following:

1. A sign, "-", if the number is negative, blank if positive.

2. At least one digit before the decimal point, with the maximum depending on the magnitude of the number. If the magnitude of the number is less than 1, a zero will appear.

3. A decimal point.

4. d digits after the decimal point

If the field width is not sufficient, characters are truncated from the left. In general, for F conversion w should equal at least d +3.

For example, for the specification F8.3:

```
   273.4  internally appears on output as 273.400
   -.003  internally appears on output as -0.003
442.30461 internally appears on output as 442.305
-92745.0  internally appears on output as 2745.000
```

E Format (Normalized with E Exponent) rEw.d

Input - Same as with F format.

Output - The number, with its exponent, is output right-justified in the w character field, preceded by blanks. For E conversion, the field width w should provide space for the following:

1. A sign
2. A decimal point preceded by zero.
3. d digits following the decimal point.
4. A four character exponent.

If the field width is not sufficient, characters are truncated from the left. In general, for E conversion, w should equal at least d +7.

For example, for the specification E10.3:

```
   283.0 internally appears on output as 0.238E+03
   -.002 internally appears on output as -0.200E-02
   0.0   internally appears on output as 0.000E+00
19732.4  internally appears on output as 0.197E+05
```

D Format (Normalized with D Exponent) rDw.d

Input - Same as E Format

Output - Same as for E Format, except that the exponent is specified by the letter D instead of E.

For example, for the specification D10.3:

```
   283.0 internally appears on output as 0.238D+03
   -.002 internally appears on output as -0.200D-02
   0.0   internally appears on output as 0.000D+00
19732.4  internally appears on output as 0.197D+05
```

3.3.6.2 Fixed Point Data Conversion

The I, L, and Z format codes specify the transfer of fixed point data. I format specifies the transfer of integer data, and L format specifies the transfer of logical data. Z format specifies the transfer of any data type in the ASCII representation of its internal hexadecimal

format.

I Format (Integer) rIw

Input - The optionally signed integer number in the w character input field is converted to internal format. A blank in the input field will terminate the field. For example, under I5 format:

bb325 on input is converted to 325
+12bb on input is converted to 12

(b is ASCII blank)

Output - The integer value is output right-justified in the w character field, preceded by blanks. A negative value is preceded by a minus sign. If the field width w is insufficient, characters are truncated from the left. For example, under I5 format:

312 internally appears on output as 312
-5 internally appears on output as -5

L Format (Logical) rLw

Input - The first T or F encountered in the w character input field will be read as "true" or "false", respectively. All other characters, including blanks, are ignored. If the input field contains neither T or F, a false value will be returned. For example, under L6 processing:

bbTRUE will be read as "true"
FALSE will be read as "false"

(b is ASCII blank)

Output - The values "true" and "false" are represented on output as TRUE or FALSE, right justified in the output field. For example, under L6 format control:

TRUE internally appears on output as TRUE
FALSE internally appears on output as FALSE

Z Format (Hexadecimal) rZw

Input - The ASCII representation of the hexadecimal string in the w character field is converted to the corresponding 4-digit per word hexadecimal number. This number is then stored in the specified input variable. If the input number contains fewer digits than the variable requires, it will be padded on the left with zeros. If the input number contains too many digits, only the rightmost n digits will be used, where n is associated with the variable's data type as follows:

Data Type	Byte Size	n
Integer *1	1	2
Integer *2	2	4
Integer *4	4	8
Single Precision	4	8
Double Precision	8	16

Blanks in the input string are treated as zeros.

For example, under Z2 format control storing to an Integer *2 variable, the value 2A would be stored internally as '002A'X, or decimal 42.

Output - The internal hexadecimal number is converted to ASCII and inserted right justified in the w character field. If w exceeds the number of digits associated with the variable, the output field will be padded with blanks on the left. If the field width w is insufficient, characters will be truncated from the left.

Example:

```
      WRITE ( 6, 70 ) I,X,Y
 70   FORMAT ( 1X,Z2,Z10,Z8)
```

Internal Values:

```
      I=002A, X=402CDF01, Y=0006FD23
```

Output Record:

```
      2A  402CDF010006FD23
```

3.3.6.3 Alphanumeric Data Transfer

The A, R, and H format specifications allow the transfer of data without any conversion. These specifications are generally used with ASCII information. The A and R specifications transfer information to or from the I/O list variables; the H specification transfers information to or from the specification itself.

A Format (Alphanumeric) rAw

Input - The characters in the w character input field are stored in the specified variable without conversion.

If w is less than the number of characters the variable can hold (1 for integer *1, 2 for integer *2, 4 for integer *4, 4 for single precision, and 8 for double precision), the characters are left justified in the variable and padded on the right with blanks. If w is greater, only the right-most characters are used. Input characters are always stored in the variable from left to right.

Example:

```
      READ ( 6, 50 ) I,ALPHA
50    FORMAT ( 2A3 )
```

Input Record: ABCDEFG

Internal Values: I='BC', ALPHA='DEF '

Output - The contents of the specified variable are to be transferred to the w character field without conversion. If w exceeds the number characters contained in the variable, the output characters will be right justified in the field and padded on the left with blanks. If w specifies fewer characters, only the left most characters will be used.

Example:

```
      WRITE ( 6, 60 ) NAME,A,B
60    FORMAT ( 1X,A1,A6,A2 )
```

Internal Values: NAME='AB', A='CDEF', B='GHIJ'

Output Record: A CDEFGH

R Format (Alphanumeric, Right Justified) rRw

The R format specification is similar to the A specification, except that internally data is store right justified, and padded to the left with zeros instead of blanks.

If w is less than the number of characters the variable can hold (1 for integer *1, 2 for integer *2, 4 for integer *4, 4 for single precision, and 8 for double precision), the characters are right justified in the variable and padded on the left with blanks. If w is greater, only the right most characters are used. Input characters are always stored in the variable from left to right.

Example:

```
      READ ( 6, 50 ) I,ALPHA
50    FORMAT ( 2A3 )
```

Input Record: ABCDEFG

Internal Values: I='BC', ALPHA='zDEF' (z='00'X)

Output - The contents of the specified variable are to be transferred to the w character field without conversion. If w exceeds the number of characters contained in the variable, the output characters will be right justified in the field and padded on the left with blanks. If w specifies fewer characters, only the right most characters will be used.

Example:

```
      WRITE ( 6, 60 ) NAME,A,B
```

```
   60  FORMAT ( 1X,A1,A6,A2 )
```

Internal Values: NAME='AB', A='CDEF', B='GHIJ'

Output Record: B CDEFIJ

H Format (Hollerith) nHs

Input - H format specifies that the next n characters are to be read from the input record, and stored in the n character field s; i.e. the input operation is to physically alter the H specification. For example, the statements:

```
       READ ( 6, 50 )
   50  FORMAT ( 8H******** )
```

with input record:

```
       MY NAME
```

will cause the FORMAT statement to be altered as though it was written:

```
   50  FORMAT ( 8H MY NAME )
```

Output - H format specifies that the next n character fields be transferred to the next n positions in the output record. For example, after the read statement in the above example has been executed, the statement:

```
       WRITE ( 6, 50 )
```

will produce the output record:

```
       MY NAME
```

3.3.6.4 Editing Functions

The ' (quote), / (slash) and X format specifications allow editing of output messages through insertion of text, blanks, and vertical spacing. In addition, / and X formats may be used on input to skip and position records.

' Format (quote) r's'

Input - quote format is invalid on input

Output- Performs the same function as the hollerith field, the characters in the string s are inserted in the output record at the next available position.

For example, the statements:

```
      WRITE ( 6, 20 )
   20 FORMAT ( 1X,'A''C',4'*')
```

would produce the output record:

```
A'C****
```

/ Format (Slash) r/

The slash specification causes the current record being processed to be terminated, and processing on the next record is to begin.

In the case of continuous slash specifications (such as ////.../), since no conversion occurs between each of the slash specifications, records are skipped during input, and blank records are generated during output. Slashes serve as a delimiter in FORMAT statements, commas preceding or following them are optional.

On input, a slash will cause a record to skipped. For example, the statements:

```
      READ ( 6, 60 ) I,R
   60 FORMAT ( /I8/F8.2 )
```

would cause the first record to be read and bypassed, the second record read and a value converted from the record and assigned to I; the rest of the record bypassed; the third record read and a value converted from it via F8.2 and assigned to R; and the rest of the third record to be bypassed.

On output, whenever a slash specification is encountered on output, the current record being processed is output and another record is begun. If no other specification has been encountered before the slash, a blank record is produced. For example:

```
      WRITE ( 6, 50 )
   50 FORMAT ( ///5X,'NEW LINE'///)
```

would cause three blank records to be output before the Hollerith record, and three blank records would follow.

X Format (Space) rX

The X format code allows characters to be skipped on input, or blanks to be introduced into a record on output. The X format code differs from others in that the repeat count r is required; it may not be omitted.

On input, the specification rX causes r characters in the input record to be skipped.

Example:

```
      READ ( 6, 50 ) I
   50 FORMAT ( 8X,I2 )
```

Input Record: COUNTb=b10

Internal Value: I=10

On output, the specification rX causes r blanks to be introduced into the output record.

Example:

```
      WRITE ( 6, 50 )
   60 FORMAT (1X,'TIME',5X,'RATE')
```

Output Record: bTIMEbbbbbRATE

(b is ASCII blank)

3.3.6.5 Special Format Codes

The C and M format codes provide special output capabilities. These codes allow you to position the cursor anywhere on the screen, and to output any character (including those defined with the CALL CHAR subroutine) to an output device. The M and C format codes are invalid on input.

C Format (Character) rCi

The C format specification indicates that an ASCII character is to be written to the output device. This is particularly useful for writing to the screen
(device 6), as characters defined using the CALL CHAR subroutine can easily be written. The value of i is the ASCII value to be written to the output device, and may be any value between 0 and 255 ('00'X to 'FF'X).

For example, the format specification:

```
...,C49,C50,C51,...
```

will output the numbers 1,2 and 3 to the output device (e.g. screen).

M Format (Move Cursor) Mi.j

The M format code specifies the position of the CRT cursor during transfer of information to the screen. The value of i is the row address (from 1 to 24), while the value of j is the column address (1 to 32 for 32 column mode, or 1 to 40 for 40 column mode, or 80 for 80 column mode).

For example, the format specification:

```
...,M10.8,'score',...
```

would cause the cursor to be positioned to row 10, column 8, and the message:

```
score
```

to be displayed there.

The M specification should only be used to write to the CRT. It produces cursor positioning coordinate commands compatible with an ADM-3A monitor, which is compatible with the MDOS implementation of the console device.

For example, the cursor position command M10.8 actually sends to the input/output package the following sequence of ascii characters:

```
<esc> = * (
```

which is the ADM-3A cursor postion command in form of <esc>, equals sign, row + space, column + space.

3.3.6.6 N Format Specifications

N FORMAT specifications allow the replacement of any of the quantities m, r, w, d, i or j with the letter N. When N is encountered in the FORMAT specification, its value is obtained from the next variable in the variable list.

For example, the statement:

```
30 FORMAT ( NX,NFN.N,3IN,N(3X,E8.2))
```

is a valid statement, and six values will be taken from the list for the quantities specified by the letter N.

The variable list item for N specifications must be integer.

The following example shows the N format code being used to read a variable number of array elements, based on the first number which was

entered:

```
      READ ( 6, 9100 ) NOITEMS, NOITEMS, (A(I),I=1,NOITEMS)
 9100 FORMAT ( I3, NF8.3 )
```

would correctly process the input record:

```
      3,1.2,6.1,8.5
```

3.3.6.7 Carriage Control

When records are to be printed under format control, the first character of each record is interpreted as a carriage control character; this character is not printed. The following table shows the effect of each carriage control character:

Control Character	Effect
blank	print, new line
0	new line, print, new line
1	new page, print, new line
+	print
other	print, new line

For example, the statements:

```
      WRITE ( 6, 100 )
  100 FORMAT ( '1TITLE'/'0LINE1'/' LINE2')
```

will print at the top of a page:

```
      TITLE

      LINE1
      LINE2
```

The control characters have the same effect on the screen as with a print. A '1' will cause the screen to clear, and the cursor to be positioned to the top of page.

It is suggested that screen formatting using cursor movement (M format) and writes be done using the '+' carriage control, to suppress all carriage control. For example, the statement:

```
 9100 FORMAT ( '+', M10.10, 'Hello' )
```

will cause the cursor to be positioned to row 10, column 10, and the message 'Hello' to be written there.

3.3.6.8 Q Format - I/O Transfer Length

It is occasionally useful to know how many characters were actually read during an input operation. The Q format character will return this value. For example:

```
      READ ( 6, 9100 ) LEN, IVAL
 9100 FORMAT ( Q, I6 )
```

If the user presses enter for the above, then the value of LEN would be zero
(zero bytes or characters read). If the user types one numeric character and enter, then the value of LEN would be 1, etc.

The Q format may appear anywhere in the FORMAT list. It is illegal for output.

3.4 Declaration Statements

Declaration statements define the variables which are used in a program, their dimensions, locations, types, and initial values. Declaration statements are not executable, and must appear before any executable program statement.

3.4.1 Fixed Array Declarations

Fixed array declarations define the dimensions of array variables. An array declaration is made as part of a DIMENSION, COMMON or explicit type statement.

General form:

```
v( d1, d2, ..., dn )
```

where:

v is the FORTRAN name which identifies the array, and

d1,d2,...,dn are integer *2 constants which define the upper bounds of each of the n dimensions.

Examples:

```
A(10)
L(5,5,5)
ARRAY(2,2,2,5)
VALARY(7,23)
```

3.4.2 Variable Array Declarations

Variable array declarations are defined in the same manner as fixed array declarations, with the exception that one or more of the array dimensions can be a variable name passed to a subprogram via a parameter list. A variable used in this manner must be of integer *2 type.

Examples:

```
ARG1(ARG2,ARG3)
ALPHA(BETA,2,7)
```

3.4.3 Array Storage

Elements of an array are stored continuously in memory. If an array has more than one dimension, then the array is stored such that the leftmost dimension varies first, followed by the next leftmost dimension, etc. For example, the integer array K(2,3,2) occupies 24 bytes (12 words) of memory, and is stored as follows:

```
byte 0  K(1,1,1)
```

```
byte 2  K(2,1,1)
byte 4  K(1,2,1)
byte 6  K(2,2,1)
byte 8  K(1,3,1)
byte 10 K(2,3,1)
byte 12 K(1,1,2)
byte 14 K(2,1,2)
byte 16 K(1,2,2)
byte 18 K(2,2,2)
byte 20 K(1,3,2)
byte 22 K(2,3,2)
```

3.4.4 DIMENSION Statement

The DIMENSION statement reserves space in the local data area or COMMON area for an array, and defines the array's bound. You can dimension an array only once in a program, but a single dimension statement can specify the dimensions of multiple arrays.

General Form:

```
DIMENSION v1,v2,...,vn
```

where: each vj is an array declaration.

The DIMENSION statement must precede the first appearance of each subscripted variable in an executable statement or a DATA statement.

Examples:

```
DIMENSION A(1,2)
DIMENSION X(10),Y(5,15),CVAL(3,4,5)
DIMENSION NEXT(I,J,K)
```

3.4.5 COMMON Statement

The COMMON statement specifies variables which are allocated in common data area for your program and subprograms.

General Form:

```
COMMON s1,s2,...,sn
```

where: each sj is a FORTRAN variable name or array declaration.

Variables are allocated common storage in the order in which they are declared in the COMMON statements. For example, the statements:

```
COMMON A,B,C
COMMON I,J,K
```

would cause A, B, and C to be assigned the first three storage units in common, and the variables I, J, and K to be assigned the next three storage units.

A subprograms common declaration need not match the main program's declaration, but it must not be larger. For example, the statements:

```
COMMON X,Y,Z
COMMON L,M,N
```

defined in a subprogram matches the overall allocation size of the previous example, but the variable names are different. If the variable X was accessed in the subprogram, it would actually access the same memory as the variable A from above.

Because you have complete control over the sequence of locations assigned to variables in COMMON, they may be used as the medium for transmitting data between the main program and subprograms, or amoung subprograms. In this way, data is transmitted implicitly, without specifically appearing in the parameter list of a subprogram.

Array declarations, as well as array names, can appear in COMMON statements. For example, the statement:

```
COMMON I(20,20)
```

is valid.

3.4.6 EQUIVALENCE Statement

The EQUIVALENCE statement is used to assign more than one variable to the same storage location or locations.

General Form:

```
EQUIVALENCE s1,s2,s3,...,sn
```

where: each si is an equivalence set of the form (v1,v2,v3,...,vm) and each vj is a subscripted or non-subscripted variable name. If subscripts are used, they must be integer constants.

Each pair of parenthesis in the EQUIVALENCE statement list encloses the names of two or more variables that are to be stored in the same location during execution of the object program.

For example, the statement:

```
EQUIVALENCE (ON,OFF)
```

specifies that the variables ON and OFF are to be assigned the same storage locations.

If an array variable is specified, but the subscripts are not given, it is assumed to be the first element of the array. Thus if XYZ is a single dimensioned array of ten elements,

```
EQUIVALENCE (TEMP,XYZ)
```

is the same as:

```
EQUIVALENCE (TEMP,XYZ(1))
```

Storage allocations specified in COMMON always override those specified in an EQUIVALENCE statement. EQUIVALENCE statements are not allowed to generate conflicting allocations.

For example, the statements:

```
COMMON A,B
EQUIVALENCE (A,B)
```

are invalid, since the EQUIVALENCE declaration attempts to override the allocation specified in the COMMON statement.

3.4.7 Explicit Type Declarations

There are four type statements:

```
INTEGER, REAL, DOUBLE PRECISION, and LOGICAL
```

These statements are used to explicitly define the data type associated with a FORTRAN name.

General Form:

```
type [*bytes] s1,s2,...,sn
```

where: type is one of the four declarations:

```
INTEGER
REAL
DOUBLE PRECISION
LOGICAL
```

[*bytes] is an optional qualifier for valid for INTEGER, REAL, and LOGICAL types. It specifies the number of bytes of storage to be used for each type. The following is the valid types and qualifier combinations:

```
INTEGER *1    -   one byte
INTEGER *2    -   two bytes (default INTEGER)
INTEGER *4    -   four bytes
REAL *4       -   four bytes (default REAL)
REAL *8       -   eight bytes (same as DOUBLE PRECISION)
LOGICAL *2    -   two bytes (default LOGICAL)
```

and si is a FORTRAN name that identifies a non-subscripted variable, an array, or a function name.

The type declaration overrides the implicit type conventions for the identified names and is in effect throughout the program.

Examples:

```
LOGICAL ALPHA,BETA
```

```
DOUBLE PRECISION IVALUE
REAL *8 IVALUE
REAL I,J
INTEGER CONSTANT,SCALE
INTEGER *1 KCHAR
INTEGER *4 JCHAR
```

Optionally, type statements may be used to declare arrays that are not dimensioned in DIMENSION or COMMON statements. For example:

```
REAL *4 I(30)
```

is equivalent to writing:

```
REAL *4 I
DIMENSION I(30)
```

If a name appears in a type statement, but is dimensioned in a COMMON or DIMENSION statement, the type statement must appear first in the program.

3.4.8 IMPLICIT Statement

The IMPLICIT statement is used to define the data types of variables not explicitly declared. Unless explicitly declared, a data type is associated with a variable by the first letter of its name per standard FORTRAN conventions.

General Form:

```
IMPLICIT type [*bytes] (c1,c2,...cn)
```

where:

type is one of four declarations:

```
INTEGER
REAL
DOUBLE PRECISION
LOGICAL
```

[*bytes] is an optional qualifier for valid for INTEGER, REAL, and LOGICAL types. It specifies the number of bytes of storage to be used for each type. The following is the valid types and qualifier combinations:

```
INTEGER *1    -   one byte
INTEGER *2    -   two bytes (default INTEGER)
INTEGER *4    -   four bytes
REAL *4       -   four bytes (default REAL)
REAL *8       -   eight bytes (same as DOUBLE PRECISION)
LOGICAL *2    -   two bytes (default LOGICAL)
```

and ci is a single letter or a range of letters consisting of two letters in alphabetical order seperated by a dash.

Examples:

```
IMPLICIT LOGICAL (A,X-Z)
```

specifies that all variables whose names begin with A, X, Y, or Z will be the LOGICAL type unless another type is explicitly declared for them.

```
IMPLICIT INTEGER *4 (A-Z)
```

specifies that all variables will be of INTEGER *4 type unless another data type is explicitly declared for them.

3.4.9 DATA Statement

Data variables not in COMMON may be given an initial value by means of a DATA statement. If a variable does not appear in a DATA statement, it will be initialized to zero.

General Form:

```
DATA s1/d1/,s2/d2/,...,sn/dn/
```

where:

si is a list of variables to be initialized

di is a list of constants that represent the initial values to be given the variables in the corresponding si.

The commas following the slash characters are optional. However, the elements of both lists must be seperated by commas. For example:

```
DATA A,B/1.4,2.9/C/.012/
```

A single data element of a previously-dimensioned array may be initialized. This is done by forming a subscripted variable with constant subscript expressions. For example:

```
DATA K(4,3) / 3 /
```

A complete array may be initialized by placing the array name unsubscripted in the variable list. For example, if M is an array of five elements, the statement:

```
DATA M/1,1,1,4,5/
```

will initialize the five elements of M.

Any constant in the values list may be preceded by an unsigned integer followed by an asterick (*). The integer specifies the number of times the data item is to be repeated. For example:

```
DATA M/3*1,4,5/
```

specifies the same five values for the array M as the previous example.

Data items may be numeric, logical, hexadecimal, or Hollerith constants. For example:

```
DATA A,B,C,L/3.7,z'123A','ABCD',.TRUE.  /
```

The data items of each data list must correspond in total storage units with the variables in the variable list. For example:

```
DATA K,J,A,B / 4,3,7.2,0 /
```

is invalid because it specifies only five storage units (one per integer constant and two per real constant) in the data list, whereas six storage units are specified in the variable list.

If a string of alphanumeric constants is being assigned to an array, the characters may be strung together without seperating commas. For example:

```
DATA M/'ABCDEFGHIJ'/
```

If an odd number of characters is specified, a blank will be added to fit the string into an even number of bytes. The variable list must specify sufficient storage words for the constant.

Initial values assigned via a DATA statement are done at compilation time, rather than at execution time as with an assignment statement. Therefore, if a variable is given a new value via an assignment statement, and the program is re-run without being reloaded, then the variable will not longer contain the value specified in the DATA statement. As a rule, it is best if variables initialized via DATA statements never appear on the left hand side of an assignment statement.

3.4.10 EXTERNAL Statement

The EXTERNAL statement is used to explicitly specify that a symbolic name is an external subprogram, and not an INTRINSIC function reference. You can use this statement to provide your own library routines with the same names as the provided INTRINSIC library routines.

General Form:

```
EXTERNAL subl [,subn]
```

where:

subl through subn are subprogram names.

For example, the following statement would override the INTRINSIC definition of the IAND expression:

```
EXTERNAL IAND
```

Therefore, an external reference for the IAND routine would be generated, and the linker would expect one to be supplied.

3.5 Subprograms

Subprograms are units of compiled code that can be called by other subprograms or by main programs. The subprograms are loaded with the main program when it is linked, and reside with the main program in memory.

Subprograms can be defined internally within the program (via an inline function) or externally. If the subprogram is defined externally, it may be defined as a library subprogram or compiled as part of your program.

Inline functions (called statement functions) are defined by a single assignment statement before any executable statements in a program or subprogram.

External subprograms (functions and subroutines) consist of a set of FORTRAN statements, the first of which must be a FUNCTION or SUBROUTINE statement, and the last of which must be an END statement. The subprogram can contain any FORTRAN statement, except for a PROGRAM statement, or another FUNCTION or SUBROUTINE statement.

Functions always have a set (one or more) of parameters and generally return a single value in its function name. A function name never appears on the left hand side of an assignment statement.

Subroutines do not necessarily have to have parameters, and need not return a value. Subroutines are called using the FORTRAN 'CALL' statement.

3.5.1 Parameter Lists

Within a subprogram, parameters passed to the subprogram are referred to as parameter names. These names may be arbitrarily selected by you, they need not be the same as the names of the parameters passed by the calling program. Parameters are defined in such in the FUNCTION or SUBROUTINE statement of the subprogram. For example, the statement

```
SUBROUTINE XYZ(A,B)
```

defines the start of subroutine XYZ which will be passed two parameters referred to as A and B. When the statement:

```
CALL XYZ(C,D)
```

is executed, each reference to A in XYZ will use the value of C, and each reference to B the value of D. If XYZ is later initiated by the statement:

```
CALL XYZ(E,F+3.14)
```

each reference to A in XYZ will be replaced by the value of E, and each reference to D by the value of F+3.14.

If a subroutine is passed an array as a parameter, it may also be

passed some or all of the dimensions of the array. When the dummy array is dimensioned, the dummy dimension values are used in the declaration.

For example:

```
FUNCTION FUNCT(A,B,C)
INTEGER *2 B,C,A(B,C)
```

Note that only a dummy array can be adjustably dimensioned, and that only parameter list integer scalar variables can be used as variable dimension values. This is because no space need be allocated for the array in the subroutine, it already exists in the calling program. The array declaration only serves to define the array bounds and the structure to the subroutine.

Constants may be passed as parameters. It is important that subroutines do not modify a constant parameter.

3.5.2 Statement Functions

Statement functions are functions which can be defined with a single assignment statement. Both arithmetic and logical functions may be defined. Statement functions are defined within a main program or subprogram, and are known only within that program unit. Their names may not be passed to a subprogram. Statement function definitions must appear after the last specification statement, but before the first executable statement of the program unit.

General Form:

```
f(d1,d2,...,dn)= b
```

where:

f is the FORTRAN name of the function being defined,

di is the FORTRAN name of dummy variable i, and

b is any valid arithmetic or logical expression

Parameters of a statement function may represent only scalar variables within the function definition. However, since dummy names only serve as place holders within the definition, the same names may be used elsewhere in the program to represent actual scalars, arrays, main programs, subprograms, statement functions, or other dummy variables.

The data type of the statement function and its parameters may be declared through the use of type declaration statements. If a parameter name is declared in this fashion, and the name is also used outside the function definition, the two items will have the same data type.

If a name is used in the expression part of a statement function definition which is not a parameter of that function, it represents the

same item as it would represent outside the definition.

Statement functions can refer to other statement functions, as long as those referenced statement functions have been defined previously in the program.

3.5.3 FUNCTION Subprograms

Function subprograms are generally used when a function cannot be defined with a single assignment statement. A function subprogram is compiled seperately from the calling program or subprograms.

Function subprograms consist qf any FORTRAN statements, except the PROGRAM statement. It returns a single value as the function name and is terminated by a RETURN statement.

3.5.3.1 FUNCTION Statement

General Form:

```
type FUNCTION f(d1,d2,...,dn)
```

where:

type is optional and is one of the specifications: INTEGER, REAL, DOUBLE PRECISION, or LOGICAL

[*bytes] is optional, and is only specified if the type qualifier is specified. It is the number of bytes contained within the type specified. The type and bytes qualifier must be one of the following combinations:

```
INTEGER *1
INTEGER *2
INTEGER *4
REAL *4
REAL *8
LOGICAL *2
```

f is the FORTRAN name of the function being defined, and

di is the FORTRAN name of parameter i.

The specification type is optional. It identifies the data type of the value associated with the function. If omitted, a data type will be associated with the name as defined by standard FORTRAN rules. For example, the statement:

```
FUNCTION RATE(T,S,V)
```

defines a function named RATE which has three parameters and returns a single precision (real *4) value to its caller; the statement

```
LOGICAL *2 FUNCTION GAME(X,R,T)
```

defines a similar function which returns a logical *2 value.

3.5.3.2 Function Calls

Function subprograms and statement functions are referenced in the same way: the function name, followed by a list of parameters in parenthesis, appears as part or all of an arithmetic expression. A function subprogram, however, is not defined in the same program unit as the caller. Hence, if the type of value returned by the function differs from that defined by the type association rules, that type must be explicitly declared in the calling program as well as in the function subprogram.

The following example illustrates the double precision sine function included in object form on the library disk:

```
      REAL *8 FUNCTION DSIN ( X )
      IMPLICIT REAL *8 (A-Z)
C SINE FUNCTION OF X
      DATA C0,C1,C2,C3,C4 /
     1      1.570796327D0, -0.645964096D0,
     2      0.0796925827D0, -0.00468126637D0,
     3      0.000158206524D0 /
      DATA PI2 / 6.28318530D0 /
      DATA PID2 / 0.63661977D0 /
      ARG = X                 !REDUCE ARGUMENT BY 2PI
      DO WHILE ( ARG .GE.  PI2 )
        ARG = ARG - PI2
      ENDDO
      ARG = ARG * PID2
      IF ( ARG .LT.  0.0D0 ) THEN
        ARG = ARG + 4.0D0
      ENDIF
      XARG = ARG              !SPECIAL CALC IF SMALL ARG
      IF ( XARG .LT.  0.0D0 ) THEN
        XARG = -XARG
      ENDIF
      IF ( XARG .LT.  1D-16 ) THEN
        DSIN = XARG
        RETURN
      ENDIF
      IF     ((ARG .GT.  1.0D0) .AND.  (ARG .LE.  3.0D0) ) THEN
        ARG =(-ARG) + 2.0D0
      ELSEIF ( ARG .GT.  3.0D0 ) THEN
        ARG = ARG - 4.0D0
      ENDIF
      ARG2 = ARG*ARG
      DSIN = ((((C4*ARG2+C3)*ARG2+C2)*ARG2+C1)*ARG2+C0)*ARG
      END
```

3.5.4 SUBROUTINE Subprograms

A subroutine is a subprogram which may be initiated only by a CALL

statement containing its name and a parameter list if required. A subroutine can have any number or no parameters, and can return any number of values. A subroutine is compiled seperately from its calling program or subprograms.

A subroutine can consist of any valid FORTRAN code, with the exception of the PROGRAM statement. It terminates by executing a RETURN statement. The subroutine always returns to the calling program or subprogram at the statement after it was called.

3.5.4.1 SUBROUTINE Statement

General Form:

```
SUBROUTINE s(d1,d2,...,dn)
```

where:

s is the FORTRAN name of the subroutine being defined, and

di is the FORTRAN name of parameter i. If the subroutine has no parameters, then the parameter list and its enclosing parenthesis are omitted.

The following example illustrates a subroutine to poll the keyboard until a key is read, and return the key value to the calling program:

```
      SUBROUTINE KEYSCAN ( KEYCODE )
      IMPLICIT INTEGER *2 (A-Z)
C
C THIS SUBROUTINE SCANS THE KEYBOARD UNTIL A KEY IS FOUND, THEN RETURNS
C THE KEYCODE TO THE CALLING PROGRAM
C
      STATUS = 0
      DO WHILE ( STATUS .EQ.  0 )
        CALL KEY ( 0,KEYCODE,STATUS)
      ENDDO
      RETURN
      END
```

3.5.4.2 CALL Statement

General Form:

```
CALL s(a1,a2,...,an)
```

where:

s is the FORTRAN name of the subroutine being called, and

ai is the variable name, constant, subprogram name, or expression being passed as parameter i. If the subroutine has no parameters, then the parameter list and its enclosing parenthesis are omitted.

Examples:

```
CALL SUBR(K,ALPHA)
CALL PUNCH
```

3.5.5 RETURN Statement

This statement logically terminates function and subroutine subprograms, and returns control to the calling program. When used in a function subprogram, it returns the value most recently assigned to the function name.

General Form:

```
RETURN
```

3.5.6 Library Subprograms

FORTRAN provides a library of subprograms which include commonly used mathematical computations; linkages to graphics, sounds, and sprites; and other miscellaneous useful functions. All that is required to use these functions is to reference them correctly using the information provided in chapter 7. The subprograms are included in three supplied libraries, the FL (FORTRAN) library, the GL (Graphics) Library, and the ML (Math) Library. To use the subprograms, you must specify the name of the library at link time.

If you provide a subroutine with the same name as a library routine, your subroutine will be used by the linker rather than the library version.

Refer to Chapter 7 for the library subprogram definitions.

3.6 PROGRAM Statement

A program statement identifes the main unit of compiled code. The main unit is the code that initially receives control upon execution. It must be linked first (before any subprograms).

3.6.1 PROGRAM Statement

General Form:

```
PROGRAM name
```

where name is a valid FORTRAN name of 9 or less characters.

The PROGRAM statement is an optional statement which is used to identify a main program. When the main program and its subprograms are processed by the linker, this name is printed with its starting address in memory.

Example:

```
PROGRAM TEST
```

could appear as the first statement in a FORTRAN main program.

This name is also used in the symbolic debugger to specify the module name for the main program.

3.7 Compilation Directive Statements

Compilation directives alter the compilation process. There is only one compilation directive statement, the INCLUDE statement.

3.7.1 INCLUDE Statement

General Form:

```
INCLUDE 'file-name[/LIST or /NOLIST]'
```

where:

file-name - is an up to 24 character file name which indicates a file to include during the compilation process, and

/LIST or /NOLIST - is an optional keyword which specifies that the included block of FORTRAN code is to be listed on the listing file.

When the include directive is read, the file-name specified is opened, and FORTRAN statements are read from the specified file. When an end of file is encountered on the included file, then records are again read from the original source file. You can use as many INCLUDE statements as you like in your program, but INCLUDE statements cannot be nested (i.e., an included file cannot contain an INCLUDE statement).

The optional /LIST and /NOLIST qualifiers allow specification of whether the included code is listed to the listing file. If the qualifier is not specified, then the code will be listed.

INCLUDE statements are especially useful for processing COMMON blocks, as the common block need not be specified repeatedly in each program and subprogram.

For example:

```
INCLUDE 'DSK1.COMMON/NOLIST'
```

would specify including the disk file 'DSK1.COMMON' in the compilation of the routine.

The following are valid TI-99 GPL and MDOS implementation include files:

```
INCLUDE 'DSK1.DBCOMM'
INCLUDE 'DSK1.FORTCOMP.COMMON/NOLIST'
```

Note that with MDOS 9640 FORTRAN, it is not necessary to include the disk volume name. If it is unspecified, then the default drive (current MDOS prompt) will be searched for the specified file.

For example, the following are all valid MDOS file names:

```
INCLUDE 'DSK1.COMMON'
```

```
INCLUDE 'COMMON'
INCLUDE 'A:COMMON'
```

4.0 Introduction

The FORTRAN compiler is initiated differently when using the MDOS compiler implementation or when using the TI-99 GPL compiler implementation.

The TI-99 GPL implementation of the compiler is initiated by selecting item 2 on the FORTRAN main menu. This will cause 4 files to be loaded into main memory, and the execution of the compiler to be initiated.

The MDOS implementation of the compiler is initiated by specifying the task name "F9640" on the MDOS command line. All parameters needed for the compilation are specified on a single command line. Note it is possible to include the command line as part of a regular MDOS batch file.

4.1 Compiler Requirements

The compiler requires that the following items be specified:

1. Input File Name:

The name of the file which contains the source (prepared by the EDITOR).

2. Object File Name:

The optional name of the file which will contain the object module.

3. Listing File Name:

The optional name of the file or device which will contain a formatted listing of the input file, and allocation maps.

4. Scratch Disk Number:

The compiler may require that two scratch files be temporarily created, used, and then deleted. The TI-99 GPL compiler implementation allows you to specify the volume on which the scratch disk files will be created, the MDOS implementation of the compiler always defaults to the RAM disk (device DSK5:). The names of the scratch files are as follows:

SC$1FIL$ - contains a copy of the source code (one module) used for printing purposes.

SC$2FIL$ - contains intermediate object code (one module) used by the compiler.

As an estimate, the scratch files will need to be created if your input file contains a program or subprogram longer than 80 lines (160 if you are compiling using the mini-memory module, 320 if you are compiling using MDOS).

If you have a RAM disk, compilations will be quicker if you assign the scratch files to it. Under MDOS, the scratch files are ALWAYS written to the RAM disk.

5. Compilation Options:

Compilation options are specified as two character identifications, seperated by commas. The following options are available:

OB - Intersperse source listing with the hex object for each statement.

SC - Perform subscript checking on every subscript reference to ensure that the specified array index is within bounds of the array (as defined in a DIMENSION, or type specification statement). Note that you are not required to specify this option if you have arrays in your program, it just provides additional execution time subscript checking. (specification of this option will cause your program to grow larger in memory size)

DM - Compile all statements which begin with the letter "D" as normal statements (these are treated as comment lines if the DM option is not specified).

DB - Generate debugger symbols (see section 6).

4.1.1 TI-99 GPL Compiler Invocation

The TI-99 GPL compiler implementation is MENU driven. You are prompted for each required data item as specified above. If you do not wish to enter an optional item (e.g. the options), simply press enter when requested.

If you wish to automate (batch) the compilation process, you may want to consider using the use the BATCH-IT utility distributed by Asgard Software (not provided as part of this package).

4.1.2 TI-99 GPL Compiler Example:

The following will correctly execute the FORTRAN compiler, assuming the disk files as specified exist:

```
                    99 FORTRAN Compiler

Input File Name?
> DSK1.TEST

Object File Name?
> DSK1.TESTOBJ

Listing File Name?
> CRT
```

```
Scratch Disk Number (1-3)?
> 1

Compilation Options?
> OB,DM

Press ENTER to Continue
```

Where items tagged with a caret (>) are items entered by the user.

4.1.3 MDOS FORTRAN Compiler Invocation

The MDOS implementation of the FORTRAN compiler is completely command line driven, there are no menus. This allows you flexibility to use the "shell" features of MDOS to back up to a previous command line using the "up-arrow" key. It also allows you to include the compilation statement in a batch file, for automated re-compilations.

The syntax to compile a FORTRAN program using the MDOS implementation of the FORTRAN compiler is:

```
F9640 [/List_File] [/POptions] [/Object_File] Source_Filename
```

where:

List_File: Is the optional listing file name, e.g. AUX (for RS232), CRT, DSK2.LISTFILE, etc.

Object_File: Is the optional object file name, e.g. TESTOBJ, DSK5.TOB, etc.

Options: Are the optional compilation options OB, SC, DB, or DM. They are specified as a string, with no imbedded blanks, and each option is seperated by a comma (e.g. DM,OB,SC).

Source_Filename: Is the required name of the FORTRAN source file to compile. It must be last in the list of parameters specified on the command line.

For example, to compile a program called TEST on B: (DSK2.) from the MDOS prompt B>, with the 9640 FORTRAN disk inserted in the A: drive, you might type:

```
A:F9640 TEST
```

Note that this command will only generate an error listing on the CRT, no object file or listing file will be produced as none was specified.

As another example, the following command will compile the same file, but produce a listing file on the AUX device (to a printer that is set up for 8-bit, 4800 baud, no parity), generating an object file to the ram disk DSK5:, with compilation options for OB (Object Generation), SC (subscript checking), and DB (debug code):

```
MODE RS232/1:4800
F9640 /ODSK5.TESTOBJ /POB,SC,DB /LAUX TEST
```

Note that the qualifiers /O, /P and /L are all optional, and may be specified in any order on the command line.

4.2 Compiler Execution

Once the compiler has been initiated, the messages

```
99/9640 FORTRAN Compiler Version xxxxx
Copyright 1989 by LGMA Products
Compilation in Progress
```

are displayed.

Any source errors which may be present in your input file will be listed along with your listing file, and also echoed to the screen.

When the compiler completes, the MDOS implementation of the compiler returns to the MDOS prompt. The TI-99 GPL implementation displays the message:

```
Press ENTER to Continue
```

In this case, press ENTER to return to the FORTRAN main menu.

4.3 Compiler Listing

A listing of the program is produced on the listing device, if one was specified. Each line of the listing contains the line number in decimal, followed by the source statement. If the source statement contains warnings or errors, then the warnings or errors follow the line.

If a source statement has a warning or error message, the statement is also listed on the screen, along with the warning or error message.

The top line of each page contains the FORTRAN version identifier, the program or subprogram name, and a page number.

The source listing is followed by an allocation map which contains allocation, size, and error information about the compiled program.

4.4 Allocation Map

The top line of each page of the allocation map is a header as described in the previous section, except the word "Allocation" is appended to the program or subprogram name. The following sections may appear in the map:

Statement Labels

If there are any statement labels within the program, they are printed under this heading in order of increasing label number. The label's number is listed first, followed by the label's hexadecimal location relative to the beginning of the module. The absolute location can be determined by adding this number to the absolute memory address for the start of the logic in the linker map. An error flag is appended to the label if the label was incorrectly used.

Subprogram References

If any subprograms are referenced, their names are printed under this heading.

Common Area

If any variables were declared to be in blank COMMON, the variables and their locations are printed under this heading. Each variable is printed in the summary along with the following information:

1. The hexadecimal location of the start of the variable, relative to the start of COMMON.

2. A following letter which indicates the variable type, as follows:

b - INTEGER *1	I - INTEGER *2
4 - INTEGER *4	R - REAL *4
D - REAL *8	L - LOGICAL

3. The name of the variable.

Common Errors

Any errors concerning the allocation of variables in blank COMMON are printed under this heading. The name of the variable in error is printed.

Local Data Area

If any local variables were used, the variables and their locations are printed under this heading. Each variable is printed in the summary along with the following information:

1. The hexadecimal location of the start of the variable, relative to the start of local data area.

2. A following letter which indicates the variable type, as follows:

b - INTEGER *1	I - INTEGER *2
4 - INTEGER *4	R - REAL *4

D - REAL *8 L - LOGICAL

3. The name of the variable.

Local Errors

Any errors concerning the allocation of variables in the local area are printed under this heading. The name of the variable in error is printed.

Note that the absence of any header indicates that no such identifiers occured in the program.

The second part of the allocation map contains a summary of the local data section, logic section, and common section sizes. The following items appear in the map:

Data Size - The size of the local data area, in bytes. This includes space for linkage, dummy variables, Hollerith strings, local variables, and temporaries.

Logic Size - Total program logic size including the program's instructions and constants, in bytes.

Module Size - The sum of the data size and logic size.

Common Size - The total byte size of COMMON.

4.5 Compiler Abort Errors

The following errors will cause the compiler to display a text message describing the error, and will cause the compiler to abort:

Input/Output Errors

The following errors are all input/output related errors returned by the operating system:

I/O Error - Bad Device Name
I/O Error - Write Protected
I/O Error - Bad Open Attribute
I/O Error - Illegal Operation
I/O Error - No Buffer Space
I/O Error - Read past end of file
I/O Error - Device Error
I/O Error - File Error

Internal Compiler Errors

The following errors are related to internal problems with the FORTRAN compiler. The first error is caused by your program being too large to compile, and the solution is to "downsize" your program by reducing the

data size, or by splitting the program into smaller modules (subprograms). The second error should never occur and indicates an internal compiler fault.

OVF Error - Compiler Memory Overflow
INT Error - Internal Compiler Fault

MDOS Only Errors

The following errors are related to the MDOS command line processing and memory allocation:

NOP Error - Nothing to DO!
Occurs when no processing is indicated on command line.

BCM Error - Bad Command Line Option Specified (/O, /L, /P)
The "slash" option is not an O, L, or P.

OTL Error - Option too long
An individual option exceeds 20 characters.

CRS Error - Cannot Retrieve Command String
An error was returned by MDOS when retrieving command line.

OST Error - Command Line Option Specified Twice
A "slash" option was specified twice.

MEM Error - Cannot Get Memory, Error=x
Not enough memory to run compiler. Suggest reducing RAM disk, or removing TIMODE in AUTOEXEC.

If the error is an input/output error, then the file name associated with the error will also be displayed.

4.6 Source Statement Warnings

Certain conditions arising in the compilation may not be what you intended. The compiler checks for three of these conditions, and if they arise, prints a warning on the statement. There are three such warnings, as follows:

1. Mixed mode arithmetic

The statement contains an expression on the right side of the equals sign which has mixed mode arithmetic (e.g. real and integer expressions). FORTRAN will supply the necessary conversion for the different types.

2. Name is longer than 9 characters

A FORTRAN name (variable name, subprogram name, or program name) is

longer than 9 characters. FORTRAN will truncate the name to 9 characters.

3. Value is too large or too small

A numeric value has been specified which is too large or too small for the type of variable specified. FORTRAN will truncate the value.

4.7 Source Statement Errors

The compiler checks the syntax of your FORTRAN program, and if incorrect, will print an error message with the source line in error to the specified listing file, and will also display the source line in error and error message on the CRT. You should not attempt to execute a program which contains source errors, as the result is unpredictable.

1. Could not allocate variable

The compiler could not allocate memory to some or all of the variable names specified.

Rules:

a. A name may not appear in more than one COMMON statement per module, or more than once within such a statement.

b. A dummy argument of a subprogram may not appear in a COMMON or EQUIVALENCE statement.

c. Constant dimensions must be greater than 1 in a specification statement

d. Only dummy arguments may be adjustably dimensioned, and only dummy arguments may be used as adjustable dimensions.

2. Storage count is wrong in DATA

The storage requirements of the variables being initialized do not match those of the constants used to initialize them.

For example:

```
      DATA A,B / 1.0, 2.0, 3.0 /
**Error-Storage count is wrong in DATA
```

3. Variable in DATA statement also in COMMON

An attempt was made to initialize variables in COMMON using a DATA statement. Only local variables will be initialized as specified.

For example:

```
      COMMON A
      DATA A / 1.0 /
**Error-Variable in DATA statement also in COMMON
```

4. Variable used incorrectly

An identifier is being used in a construct not permitted by the FORTRAN language.

For example:

```
      FUNCTION FUNCT (A,B,A)
**Error-Variable used incorrectly
```

Rules:

a. Dummy variables must be unique, and subscript variables must be of type integer.

b. Function references must be accompanied by argument lists.

c. An array's type may not be declared after it is dimensioned.

d. Letter ranges in an IMPLICIT statement must be in alphabetical order. Implicit type may be specified only in terms of starting letter rather than starting letter combinations.

5. Illegal compilation option

The specified option is invalid.

For example :

```
    OB,XX
**Error-Illegal compilation option
```

6. Missing or bad statement label

A format statement has either no label or a doubly defined label.

For example:

```
    FORMAT ( 1X,F10.2 )
**Error-Missing or bad statement label
```

7. Illegal statement of logical IF

Illegal argument statement of a Logical IF.

DO, IF, and Computed GOTO statements may not be arguments of a logical IF statement.

For example:

```
      IF ( A .LT. B) IF ( C .LT. D ) GOTO 2
   **Error-Illegal statement of logical IF
```

8. Subscript is out of range

Array name is not properly subscripted.

For example:

```
      DIMENSION A(10)
      A(I,J) = 10.0
   **Error-Subscript is out of range
```

Rules:

a. Subscripts must match array declarations in number except in an EQUIVALENCE statement.

b. All array references must be subscripted except when used as arguments in DATA, READ, WRITE, CALL, or function reference statements.

c. Subscripts are limited to form j*i+/-k.

d. Constant subscripts must refer to an array element between 1 and n for an n-element array.

9. Don't understand this statement

The statement is not understood by the compiler, and is incorrect.

For example:

```
      A = A+
   **Error-Don't understand this statement
```

10. Variable used in incorrect context

The type (integer, real, double precision, or logical) of the indicated variable or expression is different from that required by the syntax of the FORTRAN language.

For example:

```
      REAL A
      A=.FALSE.
```

**Error-Variable used in incorrect context

Rules:

a. An integer *2 variable is required for:

Variable Subscript
Adjustable Dimension
I/O status return variable

b. Logical values may not be assigned to numeric variables. Numeric variables may not be assigned to logical variables.

c. IF statements with numeric conditions must have 3 labels as arguments IF statements with logical conditions must have full statements as arguments, or the keyword "THEN".

d. Numeric and relational operators (+, -, /, **, .LE., .LT., .GE., .GT., .EQ., .NE.) must have numeric arguments, logical operators (.AND., .OR., .EOR., .NOT.) must have logical arguments.

e. Conficting explicit or implicit type declarations are not permitted.

11. ENDIF statement is missing

A program specified an IF ... THEN structured if, but no ENDIF statement was found.

For example:

```
      IF ( I .EQ. 1 ) THEN
      END
   **Error-ENDIF statement is missing
```

12. DO loop is used incorrectly

A program specified a DO loop which extended into the range of a structured IF ... THEN ... ENDIF, or a structured DO WHILE ... ENDDO.

For example:

```
      DO 1000 I=1,10
      IF ( I .EQ. 1 ) THEN
 1000 CONTINUE
      ENDIF
   **Error-DO loop is used incorrectly
```

13. THEN statement is missing

An IF statement was missing the "THEN" keyword.

For example:

```
      ELSEIF ( I .EQ. 1 )
   **Error-THEN statement is missing
```

14. ELSE statement is used incorrectly

An ELSE statement was specified without a preceding IF ... THEN.

For example:

```
      IF ( I .EQ. 1 ) GOTO 100
      ELSE
   **Error-ELSE statement is used incorrectly
```

15. ENDIF statement is used incorrectly

An ENDIF statement was specified without a preceding IF ... THEN.

For example:

```
      IF ( I .EQ. 1 ) GOTO 100
      ENDIF
    **Error-ENDIF statement is used incorrectly
```

16. Compiler workspace memory overflow

An internal memory overflow occurred at the statement specified. The only solution to this error is to reduce the size of individual modules within your program, or to reduce the number and/or size of the variables specified in your program.

4.8 Allocation Errors

After all statements of a program have been compiled, the compiler assigns storage to all variables as specified by COMMON and EQUIVALENCE statements. When such assignment is impossible, the compiler assigns storage for the variables in error as if they had not appeared in these statements. As part of the allocation map, each variable name appears in an error summary.

4.9 Label Errors

In the label summary, the listing of the label in error is followed by one of the letter 'E' if the label is in error. A label error may be caused by:

1. Label is undefined. All references to an undefined label are

resolved to an error exit.

2. Label is doubly defined. All references to a doubly defined label are resolved to the first appearance of the label.

3. Label has been misused. Possible violations include:

a. A labelled DO, GOTO, Computed GOTO, Arithmetic IF, STOP, PAUSE, RETURN, or END statement terminates the range of a DO loop.

b. A FORMAT statement label was used other than as a format specification a READ or WRITE statement.

c. A non-FORMAT label was used in a READ or WRITE statement's format specification.

d. A DO loop terminating on the specified label is improperly nested.

e. A labeled specification statement was referenced.

All references to a misused label are resolved to an error exit.

4.10 Program Size Restrictions

During program compilation, the compiler may require that two scratch files be created, as described in section 4.0. The disk specified for the scratch files must contain enough spare area to hold them. If an overflow on the disk takes place, then an input/output error (out of room on device) will take place on one of the files SC$1FIL$ or SC$2FIL$. To recover, use a disk which has more room available, or reduce the size of individual modules within your program/subprograms.

For the MDOS implementation, these files are created automatically on the RAM disk. If you run out of room for either of these files, either reduce the size of individual modules within your program, or increase the size of the RAM disk in your AUTOEXEC file (and reboot MDOS).

The compiler also uses fixed table space. This area primarily contains working tables for unique symbol names (e.g. scalars, arrays, labels, etc.) for each program or subprogram which is being compiled. When this table is exceeded, then the error message:

OVF - Memory Overflow

is displayed on the screen, and the compiler aborts. The only solution at this point is to reduce the number of unique symbol names in your program (perhaps by segmenting the program into multiple subprograms).

The compiler also restricts each program or subprogram to 127 names of each type (e.g. 127 labels, 127 integer *2 variables, etc.).

5.0 Introduction

After your program has been compiled, producing an object module, it must be linked with other object modules, and library subroutines, to produce an executable program. The linker allows your to collect together (or link together) these seperately compiled object modules to form an executable program in main memory. The linker also produces a "link map", which indicates where in memory the various object modules reside.

Both implementations (TI-99 GPL and MDOS) of the linker produce an executable file directly from the linker. The MDOS implementation produces program files which are initiated by typing the FORTRAN task name on the MDOS command line. the TI-99 GPL implemenation produces a FORTRAN program which can be initiated using item 7 (Load) on the FORTRAN menu, or directly executed from item 5 (LOAD and RUN) on the Editor/Assembler Menu page.

5.1 LINKER

The linker is initiated differently when using the MDOS implementation of the linker or when using the TI-99 GPL Implementation.

The TI-99 GPL implementation of the linker is called by selecting item 3 on the main FORTRAN menu. After loading a file, the LINKER is initiated.

The MDOS implementation of the linker is called by entering the task name "FLINK" on the MDOS command line. All parameters needed for the link are specified on a single command line. Note it is possible to include the command line as part of a regular MDOS batch file.

5.1.1 Operation

The linker requires that the following information be specified:

1. Executable File Name:

   This is a required item, and is the starting name of the executable program module to be produced. Note that FORTRAN produces as many modules as are needed to generate the entire program object. The different modules are created by incrementing the last letter of the file name, as with the Editor/Assembler SAVE routine.

   For example, if you specified an executable file name of:

   DSK1.TESTEXE

   and the FORTRAN linker actually required three modules to complete the link, then the three files created on the disk would be:

   DSK1.TESTEXE

```
DSK1.TESTEXF
DSK1.TESTEXG
```

Note when using the TI-99 GPL linker implementation, the preceding volume name (e.g. DSK1.) is required. When using the MDOS implementation, it is optional, and if omitted, the files will be created on the current MDOS default volume specified on the command prompt (e.g. if your command prompt is currently A>, then the files will be created on disk drive DSK1.).

2. Listing File Name

The optional name of the file or device which will contain a listing of the FORTRAN link map. If omitted, the link map is sent to the CRT.

3. Object File Name(s):

At least two object files MUST be specified as being loaded with the linker.

The FIRST object file must start with your FORTRAN main program. It may also contain FORTRAN subroutines or FUNCTION subprograms.

Intermediate object files may contain FORTRAN subroutines and/or FORTRAN function subprograms, and/or assembly language subroutines prepared in the proper format. You may specify these in any order.

4. Scan FORTRAN Library

If unresolved references still exist after all object modules have been loaded, then the specified object libraries will be scanned. The first library to be scanned should always be the main FORTRAN library FL. This may be followed by supplied alternate FORTRAN libraries (e.g. GL and ML), and finally any user generated FORTRAN libraries.

The FL library MUST always be specified.

5. Symbol File Name

If any modules within the object programs were compiled using the "DB" (debug) option, you may optionally specify a symbol file name to be used in conjunction with the symbolic debugger (see chapter 6).

5.1.2 TI-99 GPL LINKER Operation

The TI-99 GPL implementation of the linker is entirely menu driven. The user is requested for each needed file or device for the linker.

Optional items can be skipped by pressing ENTER for the menu item response.

For example, the following will correctly execute using the TI-99 GPL implementation of the linker, assuming that the disk files as specified exist:

```
                        99 LINKER

        Executable File Name?
        > DSK1.TESTEXE

        Listing File Name?
        > RS232/1.BA=4800

        Object File Name?
        > DSK1.TESTOBJ

        Object File Name?
        > DSK1.TESTOBJ1

        Object File Name?
        > DSK1.OBJE

        Object File Name?
        >

        Unresolved References
        Scan Library Name?
        > FL

        No More Unresolved References
        Symbol File Name?
        > DSK1.TESTSYM

        Saving file >TESTEXE
        Saving file >TESTEXF
```

5.1.3 MDOS FORTRAN LINKER Operation

To use the FORTRAN linker FLINK, you must have at least 128k of memory available.

The syntax for the linker command line is:

```
FLINK /Oobject_file,...,object_file [/Ilibrary_file,...,library_file]
[/SSymbol_file] [/Llisting_file] Executable_file
```

(note that command MUST appear on a single MDOS command line).

where:

object_file: are the required names of the object files to be linked together, starting with the main FORTRAN program. Each file name is seperated from the next by a comma.

library_file: If you use the extended routines from the FL or ML math libraries, or provide your own FORTRAN library, you need to specify the names of those files here. The FL library MUST always be specified to resolve linkages to the execution support package necessary to allow your MDOS program to run.

symbol_file: If you are using the symbolic debugger, you would specify the name of the file to be created to contain the debugger information.

executable_file: The REQUIRED name of the executable program file to be created. Note that FORTRAN always creates at least two modules, one for logic and one for data.

For example, lets assume that you want to link a program called TESTOBJ, and that the program does not include any references to any libraries other than the required system library FL. The command line would be:

```
A:FLINK /OTESTOBJ /IA:FL TESTEXE
```

This will link the object file DSK2.TESTOBJ, and produce the resulting program files DSK2.TESTEXE and DSK2.TESTEXF.

A more complex link consisting of:

- a main program TESTOBJ and two sub-programs TESTOBJ0 and TESTOBJ1
- scan of library FL
- scan of library ML
- scan of user library UL
- listing to the AUX device
- execution file on the RAM disk, DSK5, of TESTEXE TESTEXF

would have a command line of:

```
A:FLINK /OTESTOBJ,TESTOBJ0,TESTOBJ1 /LAUX
   /IA:FL,A:ML,UL DSK5.TESTEXE
```

Note that the entire command MUST fit on a single MDOS command line.

5.1.4 LINKER Map

The LINKER Map describes where in memory the FORTRAN main program, the main program data area, the FORTRAN subprograms, the subprogram data areas, and the assembly language subroutines have been loaded.

The TI-99 GPL implementation of the FORTRAN linker loads the FORTRAN

task into the 24kbyte segment of high ram starting at memory location A000, and ending at location FFC0.

The MDOS implementation of the FORTRAN linker loads the FORTRAN task into the 60kbyte segment of ram starting at memory location >0400, and ending at memory location >EFFE. The memory area from >F000 to >FFFE is used for workspace registers, I/O address space, I/O buffers, and other general work areas.

The LINKER map describes the loaded structure of the FORTRAN program. The map is divided into four sections, as follows:

Logic Area - The size of the FORTRAN logic

Spare Area - The size of the spare area (unused)

Data Area - The size of the FORTRAN data area, including any assembly language subprograms.

Common Area - The size of the FORTRAN blank common area.

Along with the above sizes, the starting locations of each program/subprogram logic and data areas are shown. For example, the map:

```
FORTRAN Map 4.2

Logic Area     Size = 1686

   TEST        A000
   ERRORPR     A582

Spare Area     Size = 1FA4

Data Area      Size = 18D6

   OPEN        D68E
   ERRORPR   * D782
   TEST      * DE9E

Common Area    Size=0200

   $COMMON     F100
```

would be produced for a program named TEST which has a user subroutine named ERRORPR, calls a library routine named OPEN, and has a common block declared of 200 bytes.

In the data area map, assembly language subroutines are shown without an asterick (*), while data areas for FORTRAN programs are shown with an asterick (*).

5.1.5 LINKER Errors

If an error is encountered by the linker while reading an object

file, the linker will abort, and a message will be displayed on the screen as follows:

Input/Output Errors:

The following errors are all input/output related errors returned by the operating system:

I/O Error - Bad Device
I/O Error - Write Protect
I/O Error - Bad Open Attribute
I/O Error - Illegal Operation
I/O Error - No Buffer Space
I/O Error - Read Past EOF
I/O Error - Device Error
I/O Error - File Error

Internal LINKER Errors:

The following errors are detected by the linker during processing of the object or library files:

IVF Error - Internal Overflow Error
The module caused an overflow of the internal Ref/Def table or the debugger symbol file. Reduce the number of references to subroutines, or reduce the size of the debugger symbol file (perhaps by selectively compiling FORTRAN modules with the DB option).

IMO Error - Illegal Binary Module
The module being loaded was not a binary object module. Possible causes include assembly language objects compiled in compress mode, and a non-object (source) file being loaded.

MIS Error - Object File Mismatch
The object file being loaded contains an opcode not recognized by the linker.

COM Error - Common Size Mismatch
A subroutine being loaded has a larger common size than main program.

DMA Error - Double Main Program
An object module being loaded contains a main program, but a main program has already been loaded.

MMA Error - Missing Main Program

The first object module loaded does not contain a main program.

OVF Error - Overflow Segment

The object module being loaded overflowed the overlay area ('A000'X to 'FFD6'X for TI-99 GPL, or '0400'X to 'EFFE'X for MDOS)

DDE Error - Doubly Defined Module
A module name has been used more than once.

LIB Error - Bad Object Library
An error was detected in the processing of a FORTRAN library, the format of the library has been corrupted, or this is not a library file.

Command Processing Errors

The following errors are related to errors in command string processing when running under MDOS only:

CRS Error - Command String Error
An error was returned by MDOS when returning the command string.

NTD Error - Nothing to Do!
The command line did not specify a required option parameter, such as the executable file name, or an object file name (/O option).

BOP Error - Bad Option Letter in command
A slash option had a bad option letter (not /O, /I, /L or /S).

TLO Error - Option String Too Long (>60 Characters)
The specified option string was greater than 60 characters.

OST Error - Option Specified Twice
A slash option (e.g. /L) was specified twice (e.g. /LAUX /LCRT).

MEM Error - Cannot Get Memory, Error=x
Not enough memory to run LINKER. Suggest reducing RAM disk, or removing TIMODE in AUTOEXEC.

DCO Error - Double Comma in Command
An option string was specified with a double comma,
e.g. /OTO,TP,,TD,OBJE

5.2 LOAD Utility

When using the TI-99 GPL implentation of the FORTRAN compiler, once a program has been saved to a disk from the linker, it can be reloaded at a later time without relinking by using the LOAD item on the main menu (item 6).

When the LOAD item is selected, the following screen is displayed:

```
99 FORTRAN Load
File Name?
```

Enter the file name which contains the executable image. For example, if the file TESTEXE was previously saved using item 5 on the main menu, it will be reloaded if the file name:

```
DSK1.TESTEXE
```

is entered (assuming that the file is on a disk in disk drive 1).

After a program is loaded, it can then be executed using item 4 (RUN), or run with the debugger (item 5) on the menu.

The program may also be loaded and run by exiting the 99 FORTRAN package, and specifying the file name in the Editor/Assembler option 5 "RUN PROGRAM FILE

5.3 RUN, RUN/DEBUG Operation

This section discusses how to run a FORTRAN program after it has been compiled and linked. It also discusses how to run the program under the symbolic debugger.

Running a FORTRAN program is quite different when using the MDOS GENEVE implementation of the FORTRAN compiler, and when using the TI-99 GPL implementation.

5.3.1 TI-99 GPL Invocation

Items 4 and 5 on the menu specifies running of a FORTRAN program which has been previously loaded (item 6) or linked (item 3).

Selecting item 4 on the menu will cause one file (FORTRAN execution support) to be loaded, the screen to be cleared, and the execution of your program to begin.

You can also run your FORTRAN program by using item 5 on the Editor/Assembler main menu, RUN PROGRAM FILE. To use this option, select item 5, and enter the name of the FORTRAN execution file specified in the linker as the file to load and run.

Selecting item 5 on the FORTRAN main menu will cause two files (FORTRAN execution support and the symbolic FORTRAN debugger) to be loaded, and the debugger to be initiated. In order to use the debugger, you must be using the mini-memory module, or have at least 4096 bytes of free memory in ram (the spare area on the link map musts be at least '1000'X).

Refer to section 6 for more information about the debugger.

5.3.2 MDOS GENEVE Invocation

When using the MDOS implementation, a FORTRAN program is initiated by typing the executable program name at the command line prompt, just as you would initiate any MDOS command or task. The FORTRAN program will be loaded, and execution will begin.

For example, if your executable file name you specified when linking the program was "A:TESTEXE", then you could type the following at the A: prompt to run the program:

```
TESTEXE
```

To run the program under the debugger, you would specify the debugger task, followed by any command line options you may have for your FORTRAN program, as follows:

```
FDEB (command line options)
```

You would then load your task using the "L" command of the debugger, i.e.:

```
]L TESTEXE
```

Refer to section 6 for more information on using the debugger.

5.3.3 Execution Errors

The following types of execution errors are detectable by the FORTRAN execution support package:

1. Bad input/output (such as typing in a bad character, or attempting to read a non-existent file).

2. Subscript error checking (when your program is compiled using the SC option).

3. Attempting to run a program which has compilation errors.

4. Illegal usage of FORMAT statements.

5. Bad values being passed to library routines.

When an error is detected by the FORTRAN execution support package, an error message is displayed on the screen in the following format:

```
* FORTRAN Error xx @yyyy
* 1=aaaa 2=bbbb 3=cccc
```

where:

xx - is a two character error identifier as described below,

@yyyy - is the absolute hexadecimal memory location within your program where the error occured,

aaaa - contains the absolute hexadecimal memory address start of the data area for the module in which the error occurred (refer to the link map), and

bbbb and cccc - may or may not be useful, depending on the type of error. These two fields are described in detail for each error message.

The absolute memory address can be translated into a location within your program by the following procedure:

1. Look at the link map for the location previous to the address where the error occured. This defines the module, and starting address of the module, in which the error occurred.

2. Subtract the address given in the error message from the starting address of the module. This gives the offset within the module where the error occurred.

3. Look at the listing of the module, in the allocation labels offset for an address lower than that in (2), and greater than that in (2). This gives you the range of where the error occurred.

4. If you have an OB (hexadecimal object) listing of your program, the statement in error can be located precisely by looking at the starting offset of each statement and comparing it to (2) above.

5. For example, the error message:

```
* FORTRAN Error NE @A362
* 1=FD60 2=0000 3=0000
```

was encountered. Inspecting the link map, it was determined that module ERRORPR started at location A300, and therefore the error occurred within that module. Subtracting the error address given (A362) from the starting address (A300), it was found that the error occurred at offset 62 in the ERRORPR module.

Inspecting the Label allocation map for module ERRORPR, it was determined that label 100 started at location A35A, and label 120 was at location A380. Therefore, the error occured between these two labels.

Note that since this was a FORMAT statement nest error (NE), the error must have occurred on a READ or WRITE statement.

5.3.4 Debugger Handled Errors

If your program was initiated using item 5 (run/debug), then the debugger will be called after the error is displayed. If a symbol file has been loaded into the debugger (see section 6), then a traceback of the error will be displayed as follows:

```
!FORTRAN Error xxxx, WP=yyyy, SR=zzzz

Locn    Module      Line  Label
nnnn mmmmmmmmmmm    llll  aaaaa
```

where:

xxxx - is the hexadecimal location at which the error occurred.

yyyy - is the hexadecimal workspace pointer (usually 8300).

zzzz - is the hexadecimal status register.

nnnn - is the hexadecimal location in this module.

mmmmmmmmmm - is the module name in which the error occurred.

llll - is the decimal line number at which the error occurred.

aaaaa - is the decimal label (if any), at which the error occurred.

If the error occurred in a low level subroutine within the program, then a traceback (up to ten levels) is displayed which shows where each subroutine was called.

Using the previous example, the following traceback is typical of what would be displayed on an execution time error:

```
!FORTRAN Error A362, WP=8300, SR=0002

Locn     Module    Line  Label
A362 ERRORPR         25   0000
A2A0 TEST           102   0100
```

The traceback always terminates after 10 levels, or when the main program is displayed.

Quitting from the debugger will force an unconditional return to the main menu. You cannot continue your program from a debugger handled error.

5.3.5 Execution Error Codes

The following are the two letter error codes displayed when an execution error occurs, their meanings, the corrective action taken for each error, and the meanings of the values bbbb and cccc displayed.

AE - Argument Error
An argument was needed by the called subroutine, and it was not provided by the calling program/subprogram.

Check the calling sequence for the called subroutine carefully with this manual.

Registers bbbb and cccc are useless. The request is ignored.

BC - Bad Character
An illegal character was encountered during input string processing. The valid characters (for each type of input) are:
- A or R: any ASCII character
- Z: numbers 0 to 9, letters A to F
- I: numbers 0 to 9
- F,E,D: numbers 0 to 9, decimal pt, E or D, + or -

Values bbbb and cccc are useless.

If the ERR= label is present, the error action address is taken. Otherwise, execution continues with the next statement.

BF - Bad Number of Files Specified.
In the CALL FILES routine, the number of files specified was not in the range of 1 to 9.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

CL - Bad Color Specified
The color value passed to the COLOR, SCREEN, or SPRITE subroutine was not a number between 1 and 16.

Alternately, under MDOS mode, the color value passed to the SETPIX routine was not in the range of 0 to 255.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

BM - Bad Mode
The graphics MODE specified in the CALL SETMOD subroutine is not in the range of 0 to 9 (inclusive).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

CO - Bad Column Value Specified
The column value passed to the GCHAR, HCHAR, or VCHAR subroutine was not a number between 1 and 32 (for 32 column mode), or 1 and 40 (for 40 column mode), or 1 and 80 (for 80 column mode).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

CS - Bad Character Set Number
The character set number passed to the COLOR subroutine was not a value between 1 and 28.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

CV - Bad Column Velocity Value
The column velocity passed to the SPRITE or MOTION subroutine was not between -128 and 127.

Values bbbb and cccc contain the return address and bad value. The request is ignored.

DC - Bad Dot Column
The dot column value passed to the SPRITE subroutine was not a value between 1 and 255.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

DR - Bad Dot Row
The dot row value passed to the SPRITE subroutine was not a value between 1 and 192.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

EC - Error Call
An error condition detected by the compiler has occured. This is usually caused by your program containing source errors. Check your source for compile time errors and correct them.

Values bbbb and cccc are useless.

This error forces an unconditional return to the main menu or to the MDOS prompt.

IC - Illegal FORMAT Character
An illegal character was found in the specified FORMAT statement.

Values bbbb and cccc are useless.

If ERR= label present, the error action is taken. Otherwise, continues on to the next statement.

II - Input for Output item

A READ statement had a FORMAT code which is only for output, such as string (' '), C, M.

Values bbbb and cccc are useless.

This error forces an unconditional return to the main menu or to the MDOS prompt.

IO - I/O Error

An input/output error occurred, and the program had no error transfer statement label (ERR=label) in the associated read/write statement. The error number is returned in status, as follows:

0. No error
1. Bad device name
2. Device is write protected
3. Bad open attribute such as incorrect file type, incorrect record length, incorrect I/O mode, or no records in a relative file.
4. Illegal operation, an operation not supported on the peripheral or a conflict with the open attributes.
5. Out of table or buffer space
6. Attempt to read past end of file. Also given for non-existant records in a relative record file.
7. Device Error. Covers all hard device errors such as parity and bad medium errors.
8. File error such as program/data file mismatch, non-existing file opened in INPUT mode, etc.
9. CRT only. Fctn/Back was depressed.
10. CRT only. Fctn/Redo was depressed.

Value bbbb is the error number (in hexadecimal) from above.

Value cccc is the device number (in hexadecimal) on which the error occurred.

IR - Input Item Integer for Real Format

The argument in a read or write statement was integer, but the FORMAT statement specified F, D, or E processing. The argument item must be of type single precision or double precision for F, D, or E processing.

Values bbbb and cccc are useless.

The error forces an unconditional return to the main menu, or

to the MDOS prompt.

IV - Illegal Character Value
The character value passed to the CHAR or CHARPA subroutine is not between 1 to 255; or the character value passed to the SPCHAR or SPRITE subroutine is not between 128 and 255.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

KE - Bad Keyboard Unit Number
The keyboard unit number passed to the KEY subroutine is not between 0 and 5; or the keyboard unit number passed to the JOYST subroutine is not 1 or 2 (for left or right keyboard).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

MF - Bad Magnification Factor
The magnification factor passed to the MAGNIF subroutine is not 1, 2, 3, or 4.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

MO - MDOS Only
The requested subroutine call is not available in the TI-99 GPL implementation of 99 FORTRAN, it is only available in the MDOS GENEVE implementation.

Values bbbb and cccc are useless.

The request is ignored.

NE - Nest Error
The FORMAT statement parenthesis were nested too deeply. A maximum of 3 left /right parenthesis are allowed.

Values bbbb and cccc are useless.

This error forces and unconditional return to the main menu or to the MDOS prompt.

NR - N Processing Error
An error occured during the processing of an "N" FORMAT specification. The following error checks are performed:

- The n value cannot be negative
- I/O list item must be integer

Values bbbb and cccc are useless.

This error forces an unconditional return to the main menu or to the MDOS prompt.

OB - Bad Byte Count in OPEN
A bad byte count (not between 1 and 255) was passed to the OPEN subroutine.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

OD - Bad Display Flag in OPEN
The display/internal flag passed to the OPEN subroutine was not 0 (display) or 1 (internal).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

OI - Bad Inflg in OPEN
The input flag passed to the OPEN subroutine was not 0 (update), 1 (output), 2 (input), or 3 (append).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

OR - Bad Relflag Flag in OPEN
The relflag passed to the OPEN subroutine was not 0 (sequential file) or 1 (relative file).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

OV - Bad Varflag Flag in OPEN
The varflag passed to the OPEN subroutine was not 0 (fixed length records) or 1 (variable length records).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

O# - Bad Device Number in OPEN
The device number passed to the OPEN subroutine was zero.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

PA - Bad Pallette Number
The pallette number passed to the SETPAL subroutine was not in the range of 0 to 15 (inclusive).

Values bbbb and cccc contain the return address and the value

passed.

The request is ignored.

RG - Bad Red, Green, or Blue Color

The color value passed to the SETPAL subroutine was not in the range of 0 to 7 (inclusive).

Values bbbb and cccc contain the return address and the bad color value.

The request is ignored.

RI - Real item for integer

A FORMAT statement specified integer (I) processing, but the list item was real (single or double precision).

Values bbbb and cccc are useless.

This error forces an unconditional return to the main menu or to the MDOS main prompt.

RO - Bad Row Number

The row number passed to the GCHAR, HCHAR, VCHAR, or SETPOS subroutine was not between the values of 1 to 24 (inclusive).

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

RP - Bad Number of Repetitions

The number of repetitions specified in the HCHAR or VCHAR subroutine caused the specified character to go off the screen.

Values bbbb and cccc contain the return address and the bad value. The subroutine terminates when the off-screen condition is detected.

RV - Bad Row Velocity

The row velocity value passed to the SPRITE or MOTION subroutine was not between the values of -128 to 127.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

S4 - Bad Subroutine Call in 40 Column Mode

A call was made to one of the following subroutines:

SPRITE, DELSPR, SPCHAR, POSITI, MOTION, MAGNIF, or COLOR

while the screen was in 40 column mode. Sprites are illegal in

40 column mode, as well as character color groups.

Value bbbb contains the return address. Value cccc is useless. The request is ignored.

SA - Bad Shift Amount
The shift amount passed to the ISHFT function was not in the range of -16 to +16.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

SC - Subscript Error
An error occurred during subscript array processing where the specified subscript was out of bounds with the dimensioned range of the array. Will only occur if the program was compiled with the 'SC' option.

Value bbbb is the base address of the array (for non-adjustable array processing only).

SD - Sound Duration
The sound duration argument passed to the CALL SOUND routine is not in the range of 1 to 4250, or -1 to -4250.

Values bbbb and cccc are useless. The request is ignored.

SE - Subroutine Error
A linkage to an unresolved (in link) or badly coded assembly language subroutine was attempted. This could be the result of the routine not being linked properly during the linkage step (check for unresolved references), the subroutine being written over by errant code, or a bad assembly language subroutine (first word not zero or negative).

Value bbbb contains the offending first word of the subroutine. Value cccc is useless. The request is ignored.

SF - Bad Sound Frequency
The frequency argument passed to the CALL SOUND routine is not in the range of -8 to -1 (for NOISE arguments) or 110 to 32767 (for frequency).

Values bbbb and cccc are useless. The request is ignored.

SR - Bad Sprite Number
The specified sprite in CALL SOUND, CALL MOTION, CALL POSITI, or CALL DELSPR is not between 1 and 32, inclusive.

Value bbbb and cccc are useless. The request is ignored.

SV - Bad SOUND Volume

The volume argument for the CALL SOUND routine is not in the range of 0 to 30.

Values bbbb and cccc are useless. The request is ignored.

VE - VDP Access Error

The VDP location argument passed to the VMBR, VMBW, LVMBR, or LVMBW subroutine is out of range.

The allowable range for VMBR and VMBW is from z'0000' to z'3fff'. The allowable range using LVMBR and LVMBW is from z'00000000' to z'0001ffff'.

Value bbbb contains the return address. Value cccc contains either the bad value (for VMBR/VMBW) or the bad bank code number (for LVMBR/LVMBW). The bank code is the high four digits of the passed integer *4 address (e.g. 2 for z'2ffff').

The request is ignored.

XC - Bad X Coordinate

The passed X coordinate value is not in the range of 0 to 511, for the SETPIX or GETPIX subroutine.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

YC - Bad Y Coordinate

The passed Y coordinate value is not in the range of 0 to 1023, for the SETPIX or GETPIX subroutine.

Values bbbb and cccc contain the return address and the bad value. The request is ignored.

6.0 Introduction

After you have edited, compiled, linked, and run your FORTRAN program, it is likely that the program does not do what you want it to. You must now "debug" the program, or remove the errors.

Before using the debugger, there are several steps which may be useful in debugging your program. These steps include:

1. Check the variable allocation map at the end of each module. Do you recognize all of the variables listed? If not, it is likely that you misspelled a variable somewhere.

2. A well-commented program will assist you in debugging. Comments add very little to the compilation time of a program, and unlike BASIC, add nothing to the execution time.

3. Temporary debug WRITE statements (perhaps compiled with the "D" mode so they do not have to be removed later) will help you follow the flow of the program.

4. You may want to sit down with the listing away from the computer and "walk-through" your program.

The above basic techniques will allow you to solve most of your programming bugs. If you still cannot find the problem, you need to use the debugger.

The debugger allows you to find errors in your program while it is actually running. You can read and write memory values, inspect and modify variable values, inspect and modify the program's workspace, and place breakpoints in your program which halt the program during execution.

The MDOS implementation of the debugger has been extended to allow you to inspect (list) selected portions of your source file while in the debugger, to disassemble your object code in normal Assembly language, to display and modify memory in byte, word, and longword formats, and to display help information.

6.1 Debugger Preparation

Before using the debugger, compile the routines which you want to debug with the "DB" option. This will cause the compiler to produce symbol definitions for variables names, line numbers, and labels as part of the program's object.

When the program is linked, you will be requested to supply a file name for the symbol file. For example, if you have a program called TEST, you may want to call the files as follows:

```
Input File:      DSK1.TEST
Object File:     DSK1.TESTOBJ
Symbol File:     DSK1.TESTSYM
Execution File: DSK1.TESTEXE
```

After linking your program, load and run your program by selecting item 5 (RUN/DEBUG) on the FORTRAN Main Menu (TI-99 GPL implementation), or by entering the following command on your MDOS command line (MDOS implementation):

```
FDEB <command line options>
```

Note the command line options you should enter here are NOT for the debugger, but instead are for your FORTRAN program.

Once the debugger has been initiated, the screen will clear, and the following line will be displayed:

```
99/9640 FORTRAN Symbolic Debugger V4.2
]
```

The symbolic debugger is now ready to accept commands. The first step at this point would be to use the "L" command to load your symbol file, and if you are using the MDOS implementation, to load your FORTRAN task and optionally your source files.

When using the MDOS implementation of the debugger, you may enter a question mark to obtain help information. For example, to obtain information on the available commands, you could type:

```
?
```

and a list of commands would be displayed. To obtain help on the MEMORY command, you could type:

```
M ?
```

and specific information about the usage of the memory command would be displayed. This is NOT available when using the TI-99 implementation of the debugger.

To execute your program, use the "G", GO command.

6.2 Debugger Memory Usage

Both the TI-99 GPL and MDOS implementations of the debugger require additional memory to run. The TI-99 GPL implementation uses memory space that either your task did not need, or that is available due to the mini-memory module. The MDOS implementation of the symbolic debugger pages in additional memory as needed.

6.2.1 TI-99 GPL Memory Usage

The TI-99 GPL symbolic debugger uses additional memory in CPU and VDP ram. It needs at least four Kbytes ('1000'X bytes) of storage in CPU ram, and a variable amount of storage in VDP ram, depending on the size of the symbol file.

If you are running with the Mini-Memory module, the debugger will

locate within the module (memory locations '7000'X to '7FFF'X). This area is not normally used by your FORTRAN program unless you are doing your own LOAD calls. If you are not running with the Mini-Memory module, then the debugger will be located between the logic area end and data area start in high ram (in the spare area on the Linker map).

The debugger uses '400'X bytes of VDP memory just below the start of the I/O buffer spaces. To determine what address this is, examine location '201A'X in cpu ram. Subtract '400'X bytes from this. The result is the start of the debugger save area.

The debugger uses a variable amount of space (depending on the size of the symbol file) in VDP ram starting at address '1000'X. If the symbol file cannot fit between address '1000'X and the start of the debugger save area, an error message will be displayed when the symbol file is loaded.

6.2.2 MDOS Memory Usage

The MDOS implementation of the debugger uses at least 128kbytes of memory for the debugger and user task. If you are also including a symbol file, then you may need more than 128kbytes of memory. If you are also including source files, you will need more than 128kbytes of memory, up to a maximum of 256kbytes.

This means to run the symbolic debugger, you may need to reduce the size of your RAM disk in your AUTOEXEC file, or remove the TIMODE statement from the AUTOEXEC file.

When initially loaded, the debugger is placed into the first 64kbyte memory bank (addresses z'00000' to z'0ffff'). When you use the "L" (Load) command, your FORTRAN task is loaded into the second 64kbyte memory bank (addresses z'10000' to z'1ffff'). The symbol file is loaded either in spare memory located in the second memory bank (z'10000' to z'1ffff') or into the third 64kbyte memory bank (z'20000' to z'2ffff'). The source files are always loaded into the third 64k memory bank (z'30000' to z'3ffff').

If your source files exceed the 128kbytes allotted in the third and fourth memory banks, then an error message will be displayed when loading the source files. This would only occur on extremely large programs, and the solution would be to only load source files for the sources you are debugging.

When your user task is started using the G (GO) command, then the debugger and user task are "swapped" in the first two memory banks. Your program then executes in the first memory bank until a breakpoint, and execution error, or a stop or call exit statement is executed. The "swap" is again performed, and the debugger is reentered.

The debugger also uses the area from z'1e000' to z'1ffff' in VDP memory for its screen image (note that the debugger defines its own screen outside of MDOS). MDOS does not normally use this area.

6.3 General Syntax

The debugger syntax is a single letter command, followed by one or more optional arguments. Each argument must be seperated by at least one space character.

If you specify too many arguments or too few arguments in a command line, then an error will be displayed.

The debugger accepts as arguments hexadecimal constants, integer constants and symbol constants. The MDOS implementation of the debugger also accepts string constants.

Hexadecimal constants are specified as a normal hexadecimal number, e.g.:

```
ABC
12AB
1
```

Integer constants can be specified by preceding them with the letter 'I', i.e.:

```
I10
I-16
```

The MDOS implementation of the debugger supports longword forms of the hexadecimal and integer constants, i.e.:

```
ABCDEF123
I12848623
```

The MDOS implementation of the debugger also supports string constants, specified as a quote character, an ascii string, and ended by a quote character:

```
'GENEVE'
'LGMA'
```

All commands require that the ENTER button be pressed after the end of the command. This is different than the debugger in the TI-99 GPL Editor/Assembler package.

As an example of value specification, the command:

```
H ABCD EF01
```

would be valid, as ABCD and EF01 are both hexadecimal numbers. The command:

```
H I1234 I6789
```

would also be valid, as 1234 and 6789 are both valid decimal numbers. The command:

```
H IABCD
```

would not be valid, however, as ABCD is not a valid decimal number.

6.4 Specifying Symbols

Memory locations can be referred to by a symbol name. A symbol can be a variable name, a line number, or a FORTRAN statement label. A symbol can be used on a command line in the same way as a value, but the symbol name must be preceded by a special character, depending on the symbol type, as follows:

&mmmmmmmmmm - Specifies a variable name mmmmmmmmmm.

%llll - Specifies a line number llll.

*aaaaa - Specifies a FORTRAN label aaaaa.

For example, the following are all valid symbol specifications (assuming that the symbols exist in the symbol file):

```
M &ICHAR          (specifies FORTRAN variable ICHAR)
B %20              (specifies line number 20)
B *1000            (specifies statement label 1000)
```

6.5 Debugger Commands

The following is a summary of the available commands:

Letter	Description
B	Remove/Add Breakpoints
M	Memory Inspect/Change
G	GO Program into Execution
Q	Quit Debugger
R	Inspect/Change WP, PC, or SR
T	Trade Screen
W	Inspect/Change Workspace Registers
H	Hexadecimal Arithmetic
L	Load File
S	Select Module Scope
X	Set X Bias
Y	Set Y Bias
Z	Set Z Bias

The following are commands which are only available in the MDOS implementation of the FORTRAN symbolic debugger

Letter	Description
D	Disassemble at Address
P	Parameter Display
V	View Source Module
?	Display Help Information

6.5.1 Load Task/Symbol/Source Files

The command letter L allows you to load a new symbol file into the debugger. For the MDOS implementation, it also allows you to load in the FORTRAN task and optionally source files.

This command is generally only executed once per debugging session, at the beginning of the session.

There are different syntaxes depending on whether you are using the TI-99 GPL implementation of the debugger or the MDOS implementation of the debugger.

TI-99 GPL Implementation

Using the TI-99 GPL implementation, the L command is used strictly to load in the symbol file. Since the symbol file is stored in volatile VDP ram, it must be reloaded with each exit from the program (back to the main menu). After typing the command letter L, and ENTER, the following prompt will be displayed:

```
Symbol File Name?
>
```

Enter the disk file name of the symbol file. For example, if the symbol file name specified in the linker was DSK1.TESTSYM, the the following command sequence will load the symbol file:

```
]L
Symbol File Name?
>DSK1.TESTSYM
```

If there is not enough room for the symbol file in vdp ram, an input/output error will result.

MDOS Implementation

The command letter L under MDOS is extended to not only allow the loading of a symbol file, but also to load the FORTRAN task to be debugged, and optionally load any source files which may then be viewed using the "V" command during the debug session.

The file name for the FORTRAN task MUST be specified in the command line as the first parameter. The optional file name for the symbol file MUST be specified next, followed by any source files you may wish to include.

The format for the command is:

```
L Task [symbol_file] [source file 1]...[source file n]
```

where:

```
Task          : is the first file name of the FORTRAN
```

task to be debugged, produced by the FORTRAN linker.

symbol_file : is the file name of the symbol file produced by the FORTRAN linker, and

source_file_x : is up to eight additional source files which you may specify for the debugger to read in for use with the VIEW command.

The FORTRAN task is loaded in its entirety by the debugger. For example, if you linked a FORTRAN task, which produced three executable files: TESTEXE, TESTEXF, and TESTEXG, you could specify a command line of:

```
L TESTEXE
```

and all three files would be loaded automatically.

The symbol file is the name of the symbol file produced by the linker. For example, if you linked a FORTRAN task, as above, and produced a symbol file with the /S option of TESTSYM, you could specify:

```
L TESTEXE TESTSYM
```

If the original source file was made up of several source files called TESTA, TESTB, and TESTC, you could specify:

```
L TESTEXE TESTSYM TESTA TESTB TESTC
```

Special notes concerning the use of the L command:

1. Do NOT put any INCLUDED files on this command line (those files you may have specified in your program source with the FORTRAN INCLUDE statement). The INCLUDE files will be read in automatically as called for in your program source.

2. Your MAIN program must be the first source file specified on the command line. It MUST be included if your are including any source files, and it MUST be first.

3. It is important that your FORTRAN program being debugged not attempt to access memory locations less than z'40000', or else the debugger will not operate properly.

4. As the source files are loaded, the FORTRAN statements are parsed by the debugger in the same manner as the FORTRAN compiler parses the statements. The parser within the debugger is somewhat simplified, however, and it is important that a PROGRAM, SUBROUTINE, FUNCTION, or INCLUDE statement be totally contained on a single source line. For example, the statement:

```
PROGRAM TEST
```

would be parsed correctly by the debugger, whereas the statement:

```
      PROGRAM
+          TEST
```

would not, even though both are perfectly valid FORTRAN statements.

5. Loaded SOURCE modules must match the loaded symbol file. Do not attempt to load any source modules which are from a different program than you are debugging, or an error message will result.

6. Assembly language source cannot be included in the debugger, since assembly object contains no debug information. Note that the disassembly feature of the MDOS debugger provides you with much of the same capability.

6.5.2 Select Module

The single command letter S allows you to select the module which you wish to place breakpoints or examine/modify variables.

After a symbol file has been loaded using the L command, the debugger automatically selects the FORTRAN main program module. If you wish to debug other modules, you must select them using the S command.

For example, the program you are debugging has two parts, a main program called TEST and a subroutine called ERRORPR. To select the subroutine for symbol file access, you would use the statement:

```
S ERRORPR
```

After issuing the command, you can then access all of the symbols related to the ERRORPR subroutine, including local and common variables, line numbers, and statement labels.

To reselect the main program for access, you would type:

```
S TEST
```

6.5.3 Remove/Add Breakpoints

The command letter B allows you to add a breakpoint, remove a breakpoint, or list the current open breakpoints. When a breakpoint is encountered by the computer, the debugger is called, and the breakpoint is removed. At this time the contents of the workspaces, memory, program variables, and registers can be inspected or altered using any of the other debugger commands. In addition, other breakpoints can be entered.

It is important that you always place a breakpoint on an executable instruction, rather than a data item. The best way to do this is to always place the breakpoint on the first word of the start of a FORTRAN statement, which will always be an instruction.

Adding a breakpoint:

To add a breakpoint, you enter the command "B", followed by an absolute memory address or symbol name. For example, the command:

```
B A360
```

would set a breakpoint at memory address A360.

The commands:

```
B %25
B *9100
```

would cause breakpoints to be emplaced on line number 25 and statement label 9100.

Removing a breakpoint:

To remove a breakpoint, you enter the command "B", followed by an absolute memory address or symbol name, and the letter -. For example:

```
B A360-
```

would remove the breakpoint from address A360, while the commands:

```
B %25-
B *9100-
```

would remove the breakpoints at line number 25 and statement label 9100.

Listing the current breakpoints

To list the current active breakpoints, you would enter the letter "B" with no arguments. For example, the command:

```
B
```

might list the following:

```
Locn   Module    Line  Label

A360   TEST         0   0000
A420   TEST        25   0000
A560   ERRORPR     18   9100
```

where:

Locn - is the absolute memory location at which the breakpoint exists,

Module - is the module in which the breakpoint is contained,

Line - is the line number at which the breakpoint is emplaced (0 if no symbol file present, or the line number cannot be located), and

Label - is the FORTRAN statement label at the breakpoint (0000 if no label can be located at the breakpoint address).

Deleting All Breakpoints:

To delete all open breakpoints, enter the command letter "B", followed by the minus sign (-). For example, the command:

```
B -
```

would remove all open breakpoints. (note the space between the B and the minus sign, the space delimiter is required)

Breakpoint Execution:

When a breakpoint which you have set is encountered, the following message will be displayed:

```
!FORTRAN Break @xxxx, WP=yyyy, SR=zzzz
```

The hexadecimal number @xxxx is the location at which the breakpoint occurred. The hexadecimal numbers yyyy and zzzz are the workspace pointer and status register, respectively. A debugger (]) prompt also appears, and you can enter debugger commands.

If a symbol file is present, the breakpoint will be listed in the same format as the breakpoint list command.

6.5.4 Memory Inspect/Change

The command letter M allows you to display or modify cpu or vdp memory, or display or modify local and common program variables.

If a single address is given, then the debugger enters the inspect/modify mode. If a range of addresses are given, then the debugger only displays the memory locations.

If a variable name symbol (one which begins with an ampersand, e.g. &ICHAR) is given, then the current value of the variable is displayed in its corresponding type (Integer *1, Integer *2, Integer *4, Single Precision, Double Precision, or Logical), and a new value can be entered.

Inspect/Change Memory

If a single address is given, then the debugger displays the specified memory address. If a number is then entered, the debugger modifies the memory location specified. If "ENTER" is depressed with no value, then the debugger displays the next location. If "FCTN/BACK" (f9 under MDOS) is entered or a period (.) is entered, then the debugger returns to command mode.

For example, the commands:

```
M A360
A360 = 1690
A362 = C155 C260
A364 =                  <FCTN/BACK or F9 pressed>
```

would modify location A362 to a C260 (was C155). If the program line number 25 was located at memory location A360, then the following command would perform exactly the same operation:

```
M %25
```

If the memory address is specified with the letter V following, then VDP memory is accessed rather than cpu memory. For example:

```
M 0000V
0000 = 3120 2020
```

would change the home position on the current TI-99 GPL screen from '31'X (ASCII 1) to '20'X (ASCII blank).

MDOS Extensions to MEMORY Command

The MDOS implementation of the MEMORY command allows you to display or modify memory in alternate byte and longword formats. The format of the MEMORY command under MDOS with these qualifiers is:

```
M.B     - Memory BYTE format
M.W     - Memory WORD format
```

M.L - Memory LONGWORD format

For example, to display the same memory locations as before in byte format, then the command:

```
M.B A360
A360 = 16
A361 = 90
A362 = C2
A363 = 60 .
```

Other extensions to the MDOS implementation of the MEMORY command include:

1. The memory display command now displays sixteen bytes per line.

2. The addresses to display or modify can be specified in longword format. This allows you to access or display the full 128kbytes of VDP memory. For example, you may specify an address of:

```
M 121ACV
```

when using the MDOS implementation of the debugger. This address would be invalid in the TI-99 GPL implementation.

3. The <ESC> key can be used to abort long displays of memory.

4. Values may be entered as ASCII text strings (using quoted strings), symbol locations, and integer values as well as hexadecimal values.

5. A new value response qualifier ("^") allows you to "back-up" to the previous modified value.

Inspect Memory

If a range of addresses are specified, then the memory locations specified are displayed on the screen, along with the character translation of the addresses. For example,

```
M 0000V 0300
```

would display the contents of VDP memory locations '0000'x to '0300'x, which are the current screen contents. To display cpu memory, omit the V in the range of examples, as follows:

```
M A000 A300
```

Displaying cpu memory can also be performed using the statement label and the line number symbol arguments. For example, to display the memory between statements 25 and 30, the following command could be given:

```
M %25 %30
```

or to display from labels 9100 to 9200:

```
M *9100 *9200
```

Inspect/Alter Variables

To inspect or alter a variable, give as an argument to the M command a variable name, preceded by an ampersand (&). For example, to display the value of the integer variable ICHAR, the command:

```
M &ICHAR
```

could be given. This would cause the following to be displayed:

```
ICHAR(    1) =  16706
```

The variable appears with a subscript (1), even though the variable may not have been declared as an array. At this point, you may enter a new value, press enter to see the next variable in the data area, or press Fctn/Back (f9 using the MDOS implementation of the debugger) to return to the debugger command mode.

Variables are normally displayed according to type. You can also modify the display/alter mode by typing a comma, followed by the single letter A or Z (for alphanumeric or hexadecimal) after the variable name. For example, the command:

```
M &ICHAR,Z
```

would produce the display line:

```
ICHAR(    1) = 4142
```

while the command line:

```
M &ICHAR,A
```

would produce the command line:

```
ICHAR(    1) = AB
```

showing the results of the Z and A qualifiers. The following table shows the display format for each type, with each qualifier:

Type	No Qualifier	Z Format	A Format
Integer *1	I6	Z2	A1
Integer *2	I6	Z4	A2
Integer *4	I12	Z8	A4
Single Precision	E13.6	Z8	A4
Double Precision	D13.6	Z16	A8

Caution: no subscript checking is performed by the debugger. Therefore

it is possible to for you to enter a value in an array location which is outside the bounds of the array. Check the bounds of the array very closely before changing the array contents.

To save room in the symbol file, only the main program's definition of the blank common block is saved. This requires that the main program be compiled with the "DB" option, if you want to access variables in common.

FORTRAN dummy names (arguments in subroutines and function subprograms) are not saved in the symbol file, and as such are not accessable using the dummy name. To examine/ alter the arguments in the subroutine, select the calling program/subprogram and access it via the variable name.

You cannot use a symbol value in entering a new value for a variable. The new value must be the same in type as the displayed value. For example, the command:

```
M &ICHAR
ICHAR(     1) =     2  23
```

would be valid since the new value entered (23) is of the same type as displayed (integer). However, the command:

```
M &ICHAR
ICHAR(     1) =     2  1D2
```

would not be valid since the value entered (1D2) is hexadecimal and the display type is integer.

The following are some examples of the extended MDOS modification formats:

1. The following example modifies VDP memory mapped at location z'400' in byte format. The user presses enter until location z'402' is displayed, he/she then "backs-up" by typing a caret (^) and enter, redisplaying location 402. He/she then exits using a period (.).

```
M.B 400
400 AC '.'
401 16 '.'
402 41 'A' 'C'
403 42 'B' ^
402 43 'C' .
```

2. The following example shows integer format of modification, using the 'I' format:

```
M 400
400 AC16 '..' I1231
402 4342 'CB' .
```

6.5.5 Quit Command

The command letter Q causes the debugger to exit and return either to the FORTRAN main menu (TI-99 GPL implementation), or return to the MDOS command prompt (MDOS implementation)..

6.5.6 Inspect or Change WP, PC, SR

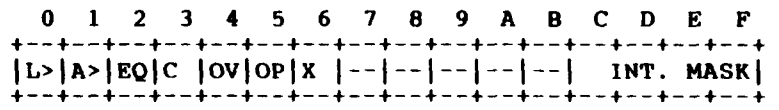
The command letter R shows the workspace pointer, program counter, and status register and allows you to change their values. After you display a register, you can alter it by entering a new value, followed by ENTER.

The workspace pointer points to the program workspace. This value should always be z'8300' under TI-99 GPL mode or z'f000' under MDOS mode. It should never be changed.

The program counter points to where the program was executing when the debugger was entered, and represents the breakpoint location.

The status register contains the status (logical greater than, arithmetic greater than, equal, carry, overflow, odd parity, extended operation, and interrupt mask) at the time of the breakpoint. These are passed back to your program when the program is resumed.

Status Register:



where:

Name	Bit Number	Description
L>	0	Logical Greater Than
A>	1	Arithmetic Greater Than
EQ	2	Equal To
C	3	Carry
OV	4	Overflow
OP	5	Odd Parity
X	6	Extended Operation
-	7-11	Reserved
INT. MASK	12-15	Interrupt Mask

For example, the command:

```
R
WP = FC60          <ENTER pressed>
PC = A632 A600     <ENTER pressed>
SR = 8000          <ENTER pressed>
```

would modify the program counter to A600 (was A632). When the program continues execution (by the Q or Quit command), it will resume execution at location A600 rather than A632.

In the MDOS implementation of this command, the status register is decoded and active flags within the status are displayed. For example, a status register of z'3402' would be displayed as:

```
SR = 3402   /  /EQ/C /  /OV/  /  /  , level=2
```

which says that the EQual status bit is set, the Carry status bit is set, the OVerflow status bit is set, and the current interrupt level is level 2.

6.5.7 Trade Screen

The command letter T trades the debugger screen for the screen as it was when you entered the debugger. The current contents of the debugger screen are lost.

For example, the command:

```
T
```

would remove the debugger screen, and replace it with the screen when you entered the debugger. To return to the debugger screen, depress any key.

6.5.8 Inspect/Change Workspace Registers

The command letter W displays all of your workspace registers, and their values if no workspace register number is given. If a workspace register number is given, it and its value are displayed and the value can be changed. After changing the value, you can press ENTER to enter the value and display the next workspace register. Pressing FCTN/BACK (f9 under MDOS) will return you to command mode without changing its value.

For example, the command:

```
W
```

would display all 16 registers, while:

```
W 1
```

would display only register 1, and also allow you to modify the register.

The following describes FORTRAN's usage of the sixteen registers:

```
R0, R1, R4, R9, R12  :  Temporaries
R2                   :  Dynamic Pointer
```

```
R3                  : Points to start of data area/this module.
R5, R6, R7, R8      : Extended Accumulator:
                        Integer *1 : R5 MSB
                        Integer *2 : R5
                        Integer *4 : R5, R6
                        Real *4    : R5, R6
                        Real *8    : R5, R6, R7, R8
R10                 : Points to execution support package
R11                 : Return for subroutine calls
R13, R14, R15       : Used for BLWP
```

6.5.9 Hexadecimal Arithmetic

The command letter H allows you to to add, subtract, multiply, divide and display the decimal equivalent of the two values entered. For example, the command:

```
H A 6
```

would display the following:

```
H1=000A H2=0006 H1+H2=0010
H1-H2=0004 H1*H2=0000 003C
H1/H2=0001 R 0004
H1(I)=    10 H2(I)=     6
```

which represent the two values entered (H1 and H2) in hexadecimal, the sum, the difference, the product, the quotient, the remainder, and the equivalent values in integer decimal.

This command can be used to perform hexadecimal to decimal conversions, and vice versa. For example, the command:

```
H I23
```

would display the values:

```
H1=0017 H2=0000 H1+H2=0017
H1-H2=0017 H1*H2=0000 0000
H1/H2= 0000 R 0000    .
H1(I)=  23  H2(I)=  0
```

converting the decimal value 23 to its equivalent hexadecimal 17.

You can also use this command to translate symbols to their hexadecimal and decimal memory locations. For example, the command:

```
H %25
```

would display the values:

```
H1=A360 H2=0000 H1+H2=A360
H1-H2=A360 H1*H2=0000 0000
H1/H2= 0000 R 0000
```

```
H1(I)= -23712 H2(I)= 0
```

MDOS extensions to this command allow for integer *4 (longword) arguments, and symbol/string formats of constant expressions.

6.5.10 GO Program into Execution

The G command letter specifies that the debugger is to start the FORTRAN task into execution. If an address was specified in the command line, then the program will start executing at that execution address. If an address has NOT been specified, then the program will start at the address specified in the current program counter PC (see R command).

Command Format:

```
G [address]
```

where: address is an optional execution address.

Return from the GO command is via one of the three following conditions:

1. A user emplaced breakpoint was encountered.
2. A FORTRAN execution error was detected, or
3. A FORTRAN STOP or CALL EXIT statement was executed.

The following are examples of the GO command usage:

```
G                start execution at PC
G 0480           start execution at address >480
G %50            start execution at line 50
G 00          start execution at label 1000
```

6.5.11 X, Y, and Z Bias

Three new commands allow you to specify an offset in a specified constant. These three commands are analogous to a "memory" button on a calculator they remember a 16-bit (TI-99 GPL implementation) or a 32-bit (MDOS implementation) value which can be recalled later.

The command syntaxes are:

```
X value
Y value
Z value
```

For example, using the MDOS implementation of the symbolic debugger, the following sets the X constant value to z'12341234':

```
X 12341234
```

The following are other examples:

```
Z I125          -  set Z to integer 125
Y %15           - set Y to location of line 15
X 00        - set X to location of label 9100
X &XYZ           - set X to location of variable XYZ
```

You can then use these constants in any expression:

```
M 0X 20X       - display memory from location 0+X
                   to location 20+X

B 0X           - put breakpoint on location 0+X

H 123X 124YZ   - hexadecimal arithmetic on the
                   values 123+X, and 124+Y+Z.
```

6.5.12 Disassemble at Address

The command letter D allows you to disassemble code starting at the specified start hexadecimal address, and continuing until the end address specified. The command format is:

```
D start_address [end_address]
```

If the "end_address" parameter is not specified, then only ten bytes of the code will be disassembled.

Note that the start_address and end_address can also be specified as symbols. For example, to disassemble starting at line number 25 to line number 30, you could enter:

```
D %25 %30
```

or to disassemble starting at FORTRAN label 100:

```
D 0
```

or to disassemble string at location >0480, and continuing to location >7FF:

```
D 480 7FF
```

The disassembler will replace hexadecimal values with symbol names, as follows:

a) line numbers are replaced by the % symbol, and the line number. For example, if line number 143 was mapped at location z'0464', then the following code:

```
BL @>0464
```

would be replaced by:

```
BL @%143
```

b) Variable names are replaced by the & symbol, and the variable name itself. For example, if the variable "ITEST" were mapped at location >EF10, then the code statement:

```
MOV @>EF10,R5
```

would be replaced by:

```
MOV &ITEST,R5
```

6.5.13 Viewing Source Files

The command letter V allows you to view the FORTRAN source files which make up the FORTRAN program you are debugging. This is useful to determine where breakpoints are to be placed, or where breakpoints have occurred.

The VIEW command format is:

```
V start_line_number [end_line_number]

                or

V.D start_line_number [end_line_number]
```

The command letter V alone specifies displaying line numbers, line locations, and source lines only. The option command V.D specifies the source listing is to be interspersed with disassembled object code.

Both the starting and ending line numbers are specified in decimal. The ending line number argument is optional, and if omitted, only the starting line will be displayed.

Only source lines within the currently selected source module can be displayed. For example, to display lines 100 through 102, inclusive, in module SYMLOAD, you might specify:

```
S SYMLOAD
V 100 110

100 04C2        if ( numbyte .le.  0 ) then
101 04D0          call prnerr ( -1 )
102 04DA          stop
```

To display the same lines with interspersed disassembly code, you might specify:

```
V.D 100 102

100 04C2        if ( numbyte .le.  0 ) then
    04C2 04C5           CLR R5
    04C4 1501           JGT @>04C8
    04C6 0705           SETO R5
```

```
     04C8 0505          NEG R5
     04CA 1602          JNE @>04D0
     04CC 0460          B   @>04DE
     04CE 04DE
101  04D0         call prnerr ( -1 )
     04D0 06A0          BL @PRNERR
     04D2 84B8
     04D4 0140
     04D8 FFFF
102  04DA         stop
     04DA 06A0          BL @STOP*
     04DC 8078
```

6.5.14 Display Program Parameters

The command letter P allows you to display information about the current debugging session. The letter P with no parameters is used to display overall debugger information, whereas the command letter P followed by a FORTRAN main program, subprogram, or assembly language DEF symbol name will display known information about that particular module.

The following is the syntax of the Parameter command:

```
P                - displays overall debugger information
P module_name    - displays information about the module
```

The following information is displayed in response to a command letter P:

a) The name of the FORTRAN task loaded on command line

b) The load address of the symbol file (0 if none)

c) The name and module pointer of the currently selected module (module scope)

d) The number of modules currently loaded in symbol/source files, the name of each module, and position within the symbol/source files.

The modules are shown in ascending location order, the order in which they are loaded in memory. The main program module is always first, followed by any FORTRAN function subprograms or subroutines, and then followed by any assembly language subroutines, including library routines.

The following information is displayed concerning an individual module:

a) The module name, and the starting location

b) FORTRAN variables, their starting address, the area in which they are located (common or local to module), and the variable

type.

c) FORTRAN line numbers and starting locations. Note that FORTRAN does not save line numbers of comment lines, or line numbers of continuation lines.

d) FORTRAN labels and starting locations.

The following are examples of using the command letter "P":

```
P                - display overall information
P TESTPROG       - display information about main
                   program called TESTPROG
P SUB1           - display information about sub-
                   program called SUB1
```

7.0 Introduction

A FORTRAN library is made up of a number of pre-compiled FORTRAN subroutines, FORTRAN function subprograms, and assembly language subroutines. These routines are combined into a single library file using the FORTRAN Librarian, FLIB.

The FORTRAN librarian is called differently when using the MDOS implementation and when using the TI-99 GPL implementation.

The TI-99 GPL implementation of the FORTRAN librarian is called from item 6 on the main FORTRAN menu (Librarian). After loading in several files, the execution of the librarian is begun.

The MDOS implementation of the FORTRAN librarian is called by the MDOS command FLIB. All parameters for FLIB are passed on the command line.

Three libraries are supplied for your use. These are:

FL - Main FORTRAN Library
ML - Single and Double Precision Math Library
GL - Graphics Library

The FL library is the main FORTRAN library. It provides many of the basic necessary routines which allows your program to execute, including interfaces for the STOP statement, the CALL statement, and the like. You MUST reference the FL library in the linkage step to resolve basic FORTRAN references made by your program.

The ML library contains single and double precision mathematical functions, such as SIN, COS, DSIN, DCOS, etc. You will only need to reference this library if you make reference to one of these higher math routines. Note that the ML library makes reference to several routines within the FL library. Therefore, it is most efficient to reference the FL library FIRST when library files are called for in the link step.

The GL library contains graphic and sprite routines. You will only need to reference this library if you make reference to an extended graphics or sprite routine.

The FORTRAN library consists of a number of pre-compiled subroutines which can be automatically linked with your FORTRAN program. These FUNCTION subprograms and subroutines provide frequently used mathematical functions, and linkages to the sprite, sound, and graphics capabilities in your computer.

The FORTRAN library routines are very similar to the routines in BASIC and EXTENDED BASIC. This allows easy conversion of BASIC programs into FORTRAN.

To use a routine in the FORTRAN library, reference the routine as specified. When you link your program using the LINKER (item 3 on the FORTRAN main menu for the TI-99 GPL implementation, or FLINK for the MDOS version), you will be asked to scan the library. The MDOS implementation scans the referenced library automatically, the TI-99

GPL implementation will request the library names to scan. Simply enter each library name in sequence (e.g. FL, then GL, then ML).

Be careful with declaring your function subprograms of the proper type in your calling routine. For example, if you use the statement:

```
IMPLICIT INTEGER (A-Z)
```

in your program, and then used the statement:

```
X = VAL (IARRAY)
```

then the VAL function, as well as the variable X must be declared of type REAL explicitly, as follows:

```
REAL X,Y,VAL
```

7.1 FORTRAN Librarian

The FORTRAN Librarian allows you to create your own object library files, which may then be used in conjunction with the FORTRAN linker to link your program. The linker has two major functions:

```
LIST - List a library file
ADD  - Create a new library file
```

To run the linker, then enter item 6 on the FORTRAN main menu, for the TI-99 GPL implementation, or type the command FLIB if using the MDOS implementation.

7.1.1 TI-99 GPL Invocation

After selecting item 6 on the FORTRAN main menu, the screen will clear, and the menu:

```
            FORTRAN Librarian V4.2

     Press:

           1.  To List Library

           2.  To Create a New Library

           3.  Exit Librarian
```

Press 1, 2, or 3 depending on the desired function

Listing a Library

After pressing 1, to list a library, the following prompts will be displayed:

```
Enter Library File Name:
```

Enter the name of the library file to list. For example, if the FORTRAN library disk is in disk drive DSK1, then you may enter:

```
DSK1.FL
```

to list the default FORTRAN library.

The next prompt will request the device to print to, such as the CRT or the default printer, or a specified printer:

```
Enter Device to List To:
```

If you wish to list to the screen, type in a device name of CRT. If you wish to list to the default printer as specified in the preferences menu, then just press enter. You may also enter a printer or file name such as:

```
RS232/1.BA=4800
PIO
DSK1.LISTFILE
```

A listing of the file will then be produced.

Creating a New Library

When item 2, create a new library, is selected, then the screen will clear and the prompt:

```
Enter Library File Name:
```

Enter the name of the library file to create. The next prompt:

```
Enter Name of Command File:
```

will be displayed. Enter the name of the file which contains a list of the object files to be produced. Note that you may use a name of CRT to indicate that the list is to be entered interactively on the screen. Usually, this is a disk file which contains a list of other disk files, such as:

```
DSK2.TESTOBJ
DSK2.OBJS
```

etc. Once this prompt has been entered, then the library will automatically be created.

7.1.2 MDOS Invocation

To execute the FORTRAN librarian under MDOS, then use the command FLIB. The following is the syntax of the FLIB command:

```
FLIB L[IST] /Iinput_file [/PPR] FLIB A[DD]
  /Ooutput_file /C[command_file]
```

where:

LIST - lists the input_file specified (optionally to the printer if the /PPR and

ADD - creates a new output_file library, given a command_file which has a list of file names to be contained in the new library.

Listing Libraries

The LIST subcommand allows you to list the contents of already created libraries, either to the screen or to the printer. For example:

```
FLIB LIST /IFL
```

will list the non-math FORTRAN library. To list the same library to the printer, use the command:

```
FLIB LIST /IFL /PPR
```

Creating New Libraries

The ADD subcommand allows you to create entirely new object libraries. For example, to create a new FORTRAN library called DL, given a command file called BUILD, the following command could be used:

```
FLIB ADD /OB:DL /CA:BUILD
```

The file called BUILD is a display/var/80 file which contains a list of file names to be included in the FORTRAN library. For example:

```
DSK5.ABS
DSK2.TESTOBJ2
A:PEEK
C:PEEKV
SCREEN
```

are all valid file names which might make up a BUILD file.

The files to be included may be assembly language modules (coded in accordance with the methods described in this manual), or FORTRAN modules (subroutines or function subprograms).

FORTRAN object files which contain multiple object modules can be included in a library, and will be listed seperately on the library listing. For example, a FORTRAN object file SUBS, which contains three subroutines SUB1, SUB2, and SUB3, and inserted into a library using the statement:

```
FLIB ADD /OUSERLIB /ICRT
SUBS
>EOD
```

could be listed using the command:

```
FLIB LIST /IUSERLIB
```

and would show the individual FORTRAN subroutines as:

```
Listing of FORTRAN Library File: USERLIB
        On: 12-08-88 Time: 12:57:56

Record   Size   Identifier   Definitions

   0       4    SUB1         SUB1, SUB1
   4       8    SUB2         SUB2, SUB2
  12       7    SUB3         SUB3, SUB3
```

Once a user FORTRAN library has been created, it can be included in your FORTRAN linker command line, e.g.:

```
FLINK /OTESTOBJ,OBJE /IFL,USERLIB DSK5.TESTEXE
```

7.1.3 FORTRAN Librarian Listing Example

The following is an example of a librarian listing:

```
Listing of FORTRAN Library File: FL
        On: 11-27-88 14:58:00

Record Size Identifier Definitions

   0       4    ABS          ABS
   4       8    AINT         AINT
  12       7    AMAMI0       AMAX0, AMIN0
                  .
                  .
                  .
 387       9    WCHAR        HCHAR, VCHAR

Total Number of Records in Library are: 397
```

The "Record" field indicates the record number (in 80-character record lengths) of where this library module starts in the library file; the "Size" field indicates the number of 80-character records contained; the "Identifier" field is the identifier on the "IDT" source record of an assembly language subroutine or subprogram, or the FORTRAN subroutine or function name; and the "Definitions" field displays all DEF type records in the object module. Note that assembly language subprograms may have multiple definitions included in their module.

7.2 Mathematical Functions

The mathematical functions provide a wide range of commonly used trigonometric, logorithmic, type conversion, logicial, hyperbolic, memory access, and other miscellaneous functions. The following pages describe the available routines, their definition, the number of calling arguments, the argument type, the resulting type, and whether or not the function reference generates inline code.

The routines are grouped by general function. These general functions are:

a. Absolute Value
b. Error Function
c. Maximums and Minimums
d. Truncation
e. Random Number Generator
f. Type Conversions
g. Remaindering
h. Logical Functions
i. Memory Access
j. Logical Shift
k. Logorithm
l. Trigonometric Functions
m. Hyperbolic Functions
n. Exponential Functions
o. Gamma Function
p. Positive Difference
q. Transfer of Sign

All of the routines following are INTRINSIC. That means that the routines are known by the compiler, as well as routine type, the type of the arguments, and whether the routine is generated as inline code or not. Your program must agree with respect to the number of arguments and the argument type. If you have a routine coded with the same name as an INTRINSIC function, you must override the compiler defaults by using the EXTERNAL statement.

The following are some examples of using the INTRINSIC Mathematical Functions:

Example	Result
J = ISHFT('5555'X,-8)	'0055'X
J = IAND ('AAAA'X, '5555'X)	0
J = IOR ('AAAA'X, '5555'X)	'FFFF'X
J = IEOR ('AAAA'X, '5555'X)	'FFFF'X
J = NOT ('AAAA'X)	'5555'X
J = IABS (-2)	2
J = IFIX (2.5)	2
X = FLOAT (2)	2.0
J = MOD (5, 2)	1
X = SQRT (2.0)	1.4142
X = SIN (1.5708)	1.0
X = ATAN (0.0)	0.4140506
X = ALOG (2.718)	1.0

The following are special notes regarding the following subprograms:

1. All angles are expressed in RADIANS.

2. All arguments of an intrinsic function must be of the same type.

3. Per normal FORTRAN conventions (and as contrasted with BASIC), the floating point to integer conversion routines do not round, they truncate. Therefore, the expression:

```
I = IFIX ( 0.9 )
```

will produce a value of 0 for I, not 1.

4. The intrinsic function calls must agree with the number of arguments shown. In most cases, this is a fixed number of arguments (e.g. 1 or 2). In the case of the MINIMUM and MAXIMUM functions, then a variable number of arguments, from 2 to 10, are allowed.

5. Many of the intrinsic functions actually generate inline code, rather than calling a function subprogram. In this case, the FUNCTION name (e.g. IAND) will be shown in the SUBPROGRAM SUMMARY in the allocation map, but no reference will be generated. The function name will not appear in the link map.

6. You can override the INTRINSIC definition of any function, as long as you supply your own function subprogram module, by using the EXTERNAL statement.

7. The following are the meanings of the letter designations under argument and result types:

K	Integer *1 (1 byte) argument
I	Integer *2 (2 byte) argument
J	Integer *4 (4 byte) argument
R	Real *4 (Single Precision 4 byte) argument
D	Real *8 (Double Precision 8 byte) argument

General Function	Routine Name	Definition	No. Arguments	Argument Type	Result Type	Inline?
Absolute Value	KIABS	y = |x|	1	K	K	
	IABS			I	I	Y
	JIABS			J	J	Y
	ABS			R	R	Y
	DABS			D	D	Y
Error Function	ERF	$y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$	1	R	R	N
	DERF			D	D	
	ERFC	$y = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-u^2} du$	1	R	R	N
	DERFC			D	D	
Maximum and Minimum	MAX	y=max(x1, ...,xn)	2 to 10	I	I	N
	MAX0			I	I	
	AMAX0			I	R	
	MAX1			R	I	
	AMAX1			R	R	
	DMAX1			D	D	
	MIN	y=min(x1, ...,xn)		I	I	
	MIN0			I	I	
	AMIN0			I	R	
	MIN1			R	I	
	AMIN1			R	R	
	DMIN1			D	D	
Truncation	AINT	y=(sign of x) *n, where n is the largest integer <= to x	1	R	R	N
	DINT			D	D	N
	INT			R	I	Y
	IDINT			D	I	Y
Random Number Generator	IRAND	y = random(x) where y is in range: 0 <= y < x	1	I	I	N
Type Conversions	IFIX	Real to Int-	1	R	I	Y
	FLOAT	Int to Real		I	R	
	SNGL	Dbl to Real		D	R	
	DBLE	Real to Dbl		R	D	
	DFLOAT	Int to Dbl		I	D	
Remaindering	MOD	y = x1-int (x1/x2) * x2	2	I	I	N
	AMOD			R	R	
	DMOD			D	D	

General Function	Routine Name	Definition	No. Arguments	Argument Type	Result Type	Inline?
Logical Functions	KIAND	y = a & b	2	K	K	Y
	IAND			I		
	JIAND			J	J	
	KIOR	y = a | b		K	K	
	IOR			I	I	
	JIOR			J	J	
	KIEOR	y = xor(a,b)		K	K	
	IEOR			I	I	
	JIEOR			J	J	
	NOT	y = ~x		I	I	
Memory Access	PEEK	y = contents of address x (cpu or vdp)	1	I	I	N
	PEEKI			I	I	
	PEEKK			I	K	
	PEEKJ			I	J	
	PEEKV			I	I	
	PEEKVI			I	I	
	PEEKVK			I	K	
	PEEKVJ			I	J	
Square Root	SQRT DSQRT	y = x ** 0.5	1	R D	R D	N
Logorithm	ALOG	y = ln(x)	1	R	R	N
	DLOG			D	D	
	ALOG2	y = $\log_2(x)$		R	R	
	DLOG2			D	D	
	ALOG10	y = $\log_{10}(x)$		R	R	
	DLOG10			D	D	
Trigono-metric Functions	SIN	sine (x)	1	R	R	N
	DSIN			D	D	
	COS	cosine (x)	(radians)	R	R	
	DCOS			D	D	
	TAN	tangent (x)		R	D	
	DTAN			D	D	
	COTAN	co-tangent(x)		R	R	
	DCOTAN			D	D	
	ARSIN	arc-sine(x)	1	R	R	
	DASIN			D	D	
	ARCOS	arc-cosine(x)	result in radians	R	R	
	DACOS			D	D	
	ATAN	arc-tangent(x)		R	R	
	DATAN			D	D	
	ATAN2	arc-tangent(x1/x2)	2	R	R	
	DATAN2			D	D	

General Function	Routine Name	Definition	No. Arguments	Argument Type	Result Type	Inline?
Hyperbolic Functions	SINH DSINH	hyperbolic-sine (x)	1	R D	R D	N
	COSH DCOSH	hyperbolic-cosine(x)		R D	R D	
	TANH DTANH	hyperbolic-tangent(x)		R D	R D	
	ASINH DASINH	hyperbolic-arc sine(x)	1	R D	R D	
	ACOSH DACOSH	hyperbolic-arc cosine(x)		R D	R D	
	ATANH DATANH	hyperbolic-arc-tangent(x)		R D	R D	
Exponential Functions	EXP DEXP	$y = e^x$	1	R D	R D	N
	EXP2 DEXP2	$y = 2^x$		R D	R D	
	EXP10 DEXP10	$y = 10^x$		R D	R D	
Gamma Function	GAMMA DGAMMA	y = r (x)	1	R D	R D	N
	ALGAMA DLGAMA	y = log(r(x))	1	R D	R D	
Positive Difference	IDIM DIM DDIM	y=x1-x2 if x1>x2 y= 0 if x1<=x2	2	I R	I R	N
Transfer of Sign	ISIGN SIGN DSIGN	y=abs(x1) if x2>0 -abs(x1) if x2<0	2	I R D	I R D	N
Logical Shift	ISHFT	y=ishft(i,j) shift i by j bits, right if j negative, left if j positive.	2	I 1 to 16 or -1 to-16	I	N

7.3 Input/Output Routines

The input/output routines allow you to open files, assign the files to FORTRAN logical unit numbers, and to close and delete files.

7.3.1 CALL OPEN

CALL OPEN allows you to open a new file which accesses any Device Service routine (excluding the cassette tape unit interface) under TI-99 GPL, or any valid MDOS device name under MDOS, and assign the file to a FORTRAN logical unit number so that it may be accessed using standard FORTRAN READ/WRITE statements.

Calling Sequence:

```
CALL OPEN (Unit, Name, inflag, disflag, varflag,
           relflag, bytecount, error)
```

where:

unit is a one word integer variable or constant which contains the file number (must be non-zero).

name is a hollerith field containing the file name. It must be terminated with an ASCII blank (' ' or '20'X).

A special name exists for accessing the screen. If you specify a name of "CRT", then the unit number specified will be opened as the screen. Note that unit number 6 is assigned to the screen by the FORTRAN execution support package.

inflag is a one word integer variable or constant which defines the input/output mode of the file as follows:

0 - update (input and output)
1 - output only
2 - input only
3 - append (output to end of file)

disflag is a one word integer variable or constant which is set as:

0 - display type file
1 - internal type file

varflag is a one word integer variable or constant which is set as:

0 - fixed length records
1 - variable length records

relflag is a one word integer variable or constant which is set as:

0 - sequential file
1 - random access file

bytecount is a one word integer variable or constant which defines the byte size of a logical record in the file. It must be between 1 and 255 bytes.

error is a one word variable that returns the open error status, as follows:

0 - no error
1 - bad device name
2 - write protected
3 - bad open attribute
4 - illegal operation
5 - out of table or buffer space
6 - read past end of file
7 - device error
8 - file error

For example, the statements:

```
INTEGER FILEID(8),ERROR
DATA FILEID / 16HRS232/1.BA=4800 /
CALL OPEN ( 3, FILEID, 1, 0, 1, 0, 96, ERROR )
```

will open the file "RS232/1.BA=4800" as FORTRAN unit number 3, an output file, with display attributes, variable length records, a maximum byte count of 96 bytes/record. If an error occurs, the variable ERROR will be non-zero on return.

If you pass the OPEN subroutine a blank file name, it will open a null file. Anything written to a null file will be discarded with no error message, and anything read from a null file will be returned as a blank record (Ascii blanks).

Devices opened under MDOS are parsed to their proper full name. For example, if you open a file such as:

```
CALL OPEN ( 3, 'LGMA', 1, 0, 1, 0, 80, ERROR )
```

And the currently selected disk drive is drive A:, then the actual file which is opened will be:

```
DSK1.LGMA
```

7.3.2 CALL CLOSE

The CALL CLOSE subroutine allows you to close a file previously opened by the CALL OPEN statement. All files which you have opened in your program should be closed (or deleted) before exiting your program

Calling Sequence:

```
CALL CLOSE ( unit [,error] )
```

where:

unit is a one word integer variable or constant which describes the unit number to close (must be non-zero).

error is an optional one word integer variable which will contain the error status of the close operation. The meaning of the error status is the same as with the OPEN subroutine.

For example, the statement:

```
CALL CLOSE ( 3 )
```

will close the file opened in the CALL OPEN example.

7.3.3 CALL DELETE

The CALL DELETE subroutine allows you to delete a file previously opened by the CALL OPEN statement. All files which you have opened in your program should be closed (or deleted) before exiting your program.

Calling Sequence:

```
CALL DELETE ( unit [,error] )
```

where:

unit is a one word integer variable or constant which describes the unit number to delete (must be non-zero).

error is an optional one word integer variable which will contain the error status of the delete operation. The meaning of the error status is the same as with the OPEN subroutine.

For example, the statement:

```
CALL DELETE ( 2 )
```

will delete file number 2.

7.3.4 CALL FILES

The CALL FILES subroutine allows you to change the number of disk files which can be opened for access by your program. When you execute your program, the menu routine sets the number of disk files which you can access to 3. This statement allows you to change the number from 1 to 9.

```
CALL FILES ( number files )
```

where:

number files is a one word integer variable or constant which defines the number of files to open.

For example:

```
CALL FILES(5)
```

will allow your FORTRAN program to open up to 5 disk files simultaneously.

This subroutine should not be used if your program has disk files currently open. If this situation occurs, the resulting condition of the open disk files is unpredictable.

This subroutine should only be used when using the TI-99 GPL implementation of the FORTRAN compiler. If you call this subroutine from a program running under MDOS, an execution error message (BF) will result.

7.3.5 CALL BREAD/BWRITE (MDOS Only)

The BREAD and BWRITE subroutines are only available when using the MDOS implementation of the FORTRAN compiler. The subroutines provide low level sector access to disk files and disk volumes. The subroutines call the equivalent BREAD and BWRITE I/O Functions to allow the user to read and write individual disk sectors of a file, regardless of the file type.

Note that the user must be aware of how the TI-99/4A and MDOS manage disk files to use this function.

Calling Sequence:

```
CALL BREAD  ( file, sector, no_sectors, buffer, error)
CALL BWRITE ( file, sector, no_sectors, buffer, error)
```

where:

file is an array which contains the name of the file to read or write (e.g. DSK1.TEST, A:TEST, TEST, A:, DSK1., etc.).

sector is a one-word integer variable which contains the sector offset within the file to access, from 0 to n-1, where n is the number of sectors used by the file.

no_sectors is a one-word integer variable which contains the number of sectors to read or write. If this variable is zero, then the file header is returned.

buffer is an array which contains the data to write, or will contain the data read. It should be at least 18 bytes long to read or write the file header, or at least no_sectors * 256 bytes long for sector access.

For example, the following will return the file header information into the array "filehdr":

```
integer *1 filehdr(18), filename(10)
data filename / 10hTESTFILE     /
call bread ( filename, 0, 0, filehdr, error )
```

7.4 Graphics Interface

These subroutines allow access to graphics, including reading and writing to the screen, defining your own character shapes, setting the screen width (32 column, 40 column, or 80 column), changing the color of the screen and characters, reading character patterns, and clearing the screen.

7.4.1 CALL GCHAR

The GCHAR subroutine reads a character from anywhere on the screen.

Calling Sequence:

```
CALL GCHAR ( row, column, character [,f_color, b_color])
```

where:

row is a one word integer constant or variable which defines the row number to read. It ranges from 1 (top of screen) to 24 (bottom of screen)

column is a one word integer constant or variable which defins the column number to read. It ranges from 1 (left of screen) to 32 (right of screen)

character is a one word integer variable which will contain the ASCII character number at the location specified. It ranges from 1 to 255.

f_color is an optional one word integer variable which is only valid using the MDOS implementation of the subroutine. It returns the foreground color of the character.

b_color is an optional one word integer variable which is only valid using the MDOS implementation of the subroutine. It returns the background color of the character.

For example, assuming the character "0" is at row 1, column 20, the statement:

```
CALL GCHAR ( 1, 20, ICHAR )
```

will return the value '30'X in ICHAR.

7.4.2 CALL HCHAR

The HCHAR routine displays a character anywhere on the screen and optionally repeats it horizontally.

Calling Sequence:

```
CALL HCHAR ( row, column, character [,repetitions] )
```

where:

row is a one word integer variable or constant which contains the starting row number, from 1 (top) to 24 (bottom).

column is a one word integer variable or constant which contains the starting column number from 1 (left) to 32 (right), or 40, or 80 depending on the screen mode.

character is a one word integer variable or constant which contains the ASCII character number to display, from 1 to 255.

repetitions is an optional one word integer variable or constant which describes the number of times to repeat the character horizontally across the screen. If this parameter is not specified, the character is only drawn once.

For example, the statement:

```
CALL HCHAR ( 4, 10, '0030'X, 10 )
```

will draw ten ASCII zero's across the screen starting at row 4, column 10.

7.4.3 CALL VCHAR

The VCHAR routine displays a character anywhere on the screen and optionally repeats it vertically.

Calling Sequence:

```
CALL VCHAR ( row, column, character [,repetitions] )
```

where:

row is a one word integer variable or constant which contains the starting row number, from 1 (top) to 24 (bottom).

column is a one word integer variable or constant which contains the starting column number from 1 (left) to 32 (right), or 40, or 80 depending on screen mode.

character is a one word integer variable or constant which contains the ASCII character number to display, from 1 to 255.

repetitions is an optional one word integer variable or constant which describes the number of times to repeat the character vertically down the screen. If this parameter is not specified, the character is only drawn once.

For example, the statement:

```
CALL VCHAR ( 4, 10, '0030'X, 10 )
```

will draw ten ASCII zero's down the screen starting at row 4, column 10.

7.4.4 CALL SCREEN

The SCREEN subroutine allows you to change the background color of the screen (normally set to the color specified in the Preferences Utility under TI-99 GPL mode, or the default MDOS screen color in MDOS mode).

Calling Sequence:

```
CALL SCREEN ( background color [,foreground color] )
```

where:

background color is a one word integer variable describing the background color of the screen.

foreground color is an optional one word integer variable describing the foreground color of the characters on the screen.

The color definitions are:

1 - Transparent	9 - Medium Red
2 - Black	10 - Light Red
3 - Medium Green	11 - Dark Yellow
4 - Light Green	12 - Light Yellow
5 - Dark Blue	13 - Dark Green
6 - Light Blue	14 - Magenta
7 - Dark Red	15 - Gray
8 - Cyan	16 - White

For example, the statement:

```
CALL SCREEN ( 7, 2 )
```

will change the screen color to dark red and the characters to black.

7.4.5 CALL COLOR

The CALL COLOR subroutine allows you to change the foreground /background color attributes of a character set.

Calling Sequence:

```
CALL COLOR ( character set, foreground color, background color )
```

where:

character set is a one word integer variable defining the character set number to modify, as follows:

1 - 32 to 39	11 - 112 to 119	21 - 192 to 199
2 - 40 to 47	12 - 120 to 127	22 - 200 to 107
3 - 48 to 55	13 - 128 to 135	23 - 208 to 215
4 - 56 to 63	14 - 136 to 143	24 - 216 to 223
5 - 64 to 71	15 - 144 to 151	25 - 224 to 231
6 - 72 to 79	16 - 152 to 159	26 - 232 to 239
7 - 80 to 87	17 - 160 to 167	27 - 240 to 247
8 - 88 to 95	18 - 168 to 175	28 - 248 to 255
9 - 96 to 103	19 - 176 to 183	
10 - 104 to 111	20 - 184 to 191	

For example, the statement:

```
CALL COLOR ( 3, 10, 1 )
```

would define the characters 48 to 55 (hex '30'x to '37'x), which are the characters 0,1,2,3,4,5,6 and 7 as foreground color of light red (10), and background color of transparent (1).

This subroutine is NOT available under MDOS, due to MDOS restrictions

7.4.6 CALL CHAR

The CHAR subroutine allows you to define your own character shapes by passing an eight byte pattern identifier.

Calling Sequence:

```
CALL CHAR ( character code, pattern identifier )
```

where:

character code is a one word integer variable or constant which contains the ASCII character number to modify, from 0 to 255.

pattern identifier is an eight byte integer array or double precision constant which defines the pattern.

For example, the statement:

```
CALL CHAR ( 48, '1898FF3D3C3CE404'X )
```

will place the pattern identifier as character number 48 ('30'x, or ASCII zero). The character can then be written to the screen using the HCHAR, VCHAR, or a standard FORTRAN WRITE statement (e.g. C48).

Note that the standard character set (characters '20'X to '7F'X) are loaded only when FORTRAN is initially booted. Therefore, the standard character set should not be modified by your program.

7.4.7 CALL CHARPA

The CHARPA routine allows you to retrieve the character pattern of the

specified character.

Calling Sequence:

CALL CHARPA (character code, pattern identifier)

where:

character code has the same meaning as with CALL CHAR, and

pattern identifier is an eight byte integer array or double precision variable which will contain the pattern identifier.

For example, the statement:

CALL CHARPA (48, DCHAR)

would return a value of 1898FF3D3C3CE404 in the double precision variable DCHAR if executed after the example in CALL CHAR.

7.4.8 CALL CLEAR

The CLEAR subroutine clears the screen, and homes the cursor.

Calling Sequence:

CALL CLEAR

There are no arguments. This statement provides the same function as writing a top of form ('1') using a WRITE statement.

7.4.9 CALL SET32

The SET32 subroutine sets the screen width to 32 column mode. In this mode the screen is 32 columns wide by 24 rows long. You can use sprites. This is the default mode of a FORTRAN program.

Calling Sequence:

CALL SET32

There are no parameters. When the command is executed, the screen is cleared, and the column width is set to 32 columns.

When this statement is executed when using the MDOS implemenation, it puts your program into "TI-99 compatibility mode" with graphics mode 1. The normal MDOS interface to graphics mode 1 does not operate the same as the TI-99 (e.g. there is no equivalent of CALL COLOR within MDOS). If you wish to code a program which uses graphics mode 1 (32 column graphics mode), and is compatible with 99 FORTRAN using the TI-99, then use a CALL SET32 statement. If you wish to use the MDOS implementation of graphics mode 1, you can issue a CALL SETMOD(3) statement.

7.4.10 CALL SET40

The SET40 subroutine sets the screen width to 40 column mode. In this mode the screen is 40 columns wide by 24 rows long. You cannot use sprites or the COLOR subroutine.

Calling Sequence:

```
CALL SET40
```

There are no parameters. When the command is executed, the screen is cleared, and the column width is set to 40 columns.

7.4.11 CALL SET80

The SET80 subroutine can be called using the MDOS implementation of the FORTRAN compiler, and also may be called in the TI-99 GPL implementation if used with a MYARC GENEVE running under GPL. It sets the screen width to 80 column mode. In this mode, the screen is 80 columns wide by 24 rows long. You cannot use sprites or the COLOR subroutine while in this mode.

Calling Sequence:

```
CALL SET80
```

There are no parameters. When the command is executed, the screen is cleared, and the column width is set to 80 columns.

7.4.12 CALL PRINTC

The PRINTC subroutine allows you to enable (flag=.TRUE.) or disable (flag=.FALSE) the checking done by the screen output routine on control/S or control/Q.

Normally, after any screen output completion, the screen routine within the execution support package checks the keyboard to see if the user has depressed control/S (XOFF). If he has, then all output is halted waiting for the user to press control/Q (XON). However, this slows screen output, and is not useful for certain types of application programs (e.g. interactive games and menu driven utilities).

This subroutine allows you to enable and disable this checking.

Calling Sequence:

```
CALL PRINTC ( flag )
```

where:

flag is a one-word logical variable which defines whether control/S checking is enabled (.TRUE.) or not (.FALSE.).

Note that FORTRAN programs are always started with control/S, control/Q checking enabled. Also, disabling control/S processing will only remain in effect as long as the current program is executing, i.e. new

programs start out with the default of control/S checking.

For example:

```
CALL PRINTC ( .FALSE.  )
```

will disable control/S checking.

7.4.13 CALL CMDSTR (MDOS Only)

The CMDSTR subroutine is available under MDOS only. It returns the command line string specified when the task name was used on the MDOS command line.

If an error is detected (e.g. the command is too long for the user array) an error number is returned as -1.

Calling Sequence:

```
CALL CMDSTR ( array, ierror )
```

where:

array is an integer *1 array of which the first byte is the length-1 of the array, and the remaining bytes will contain the command line string upon return.

ierror is an integer *2 variable which will be set to zero if the command was returned into the array, it will be set to -1 if the command line exceeded the length of the array.

Upon return from the subroutine, the first byte of the array will contain the number of bytes actually retrieved. The second through n bytes will contain the actual command string.

For example, the following returns the command string:

```
      integer *1 command(81)
      command(1) = z'50'       !length of array (80bytes)
      call cmdstr ( command, ierror )
      if ( ierror .ne.  0 ) then
        write ( 6, 9100 ) ierror
      else
        n = command(1)+1
        write ( 6, 9110 ) (command(i),i=2,n)
      endif
9100  format ( ' Error returned, error is ',i6)
9110  format ( ' The command array is ',80a1)
      end
```

7.5 Sprites

FORTRAN allows access to the sprite interface of your computer. The subroutines provided allow you to define sprites, set them in motion, delete sprites, determine the position of sprites, and control the sprite magnification factor.

7.5.1 CALL SPRITE

The SPRITE subroutine creates sprites. Sprites are graphic characters which have a color, a shape, a location on the screen, and optionally a velocity. Sprites move smoothly across the screen since their motion is controlled by an interrupt routine in your computer.

Calling Sequence:

CALL SPRITE (sprite number, character value, sprite color, dot row, dot column [,row velocity, column velocity])

where:

sprite number is a one word integer variable or constant which contains the sprite number, from 1 to 32. If the value is the same as one that has already been defined, the old sprite is replaced by the new.

character value is a one word integer variable or constant which contains the character number associated with the sprite. This number can range from 128 to 255. Note that this number is not the same as used with the CHAR subroutine. Use the SPCHAR subroutine to define sprite shapes.

sprite color is a one word integer variable or constant which can vary from 1 to 16. It defines the same colors as in the SCREEN subroutine.

dot row and dot column are one word integer variables or constants which specify the starting row, column of the sprite. Dot row can vary from 1 to 192 (top to bottom) and dot column can vary from 1 to 255 (left to right). The position of the sprite is the upper left hand corner of the sprite.

row velocity and column velocity are optional one word integer variables or constants which define the velocity of the sprite. If row velocity and column velocity are both zero, than the sprite is stationary. A positive row velocity moves the sprite down and a negative row velocity moves it up. A positive column velocity moves the sprite to the right, while a negative column velocity moves it to the left. If both row velocity and column velocity are non zero, the sprite moves at an angle in a direction determined by the values. Row and column velocity may vary between -128 and 127. A value close to zero is very slow. A value far from zero is very fast. When a sprite comes to the edge of the screen, it disappears and reappears on the opposite side of

the screen.

For example, the statements:

```
CALL SPCHAR ( 96, 'FFFFFFFFFFFFFFFF'X )
CALL SPRITE ( 1, 96, 5, 92, 1, 24 )
```

defines sprite number 1 to be associated with sprite character 96, with a color of 5 (dark blue), a dot row and column address of 1, 24. The sprite defined is stationary as the row, column velocity is not specified.

7.5.2 CALL SPCHAR

The SPCHAR subroutine allows you to define your own sprite character shapes by passing an eight byte pattern identifier.

Calling Sequence:

```
CALL SPCHAR ( character code, pattern identifier )
```

where:

character code is a one word integer variable or constant which contains the sprite character number to modify, from 128 to 255.

pattern identifier is an eight byte integer array or double precision constant which defines the pattern.

For example, the statement:

```
CALL SPCHAR ( 128, '1898FF3D3C3CE404'X )
```

will place the pattern identifier as sprite character number 128, or hex '80'X. The sprite can then be displayed using the CALL SPRITE subroutine.

7.5.3 CALL MOTION

The MOTION subroutine allows you to set a previously defined sprite in motion, by defining the row, column velocity.

Calling Sequence:

```
CALL MOTION ( sprite number, row velocity, column velocity )
```

where:

sprite number is a one word integer variable or constant which contains the sprite number, from 1 to 32. The sprite must have already been defined using the CALL SPRITE subroutine.

row velocity and column velocity are optional one word integer variables or constants which define the velocity of the sprite. If row velocity and column velocity are both zeroes, then the sprite is stationary. These mean the same as with the SPRITE subroutine.

For example, the statement:

```
CALL MOTION ( 1, -20, 10 )
```

will set sprite number 1 in motion, using the row and column velocity of -20 (going up) and 10 (going right).

7.5.4 CALL POSITI

The POSITI subroutine is used to return the current position of the referenced sprite.

Calling Sequence:

```
CALL POSITI ( sprite number, dot row, dot column )
```

where:

sprite number is a one word integer variable or constant which contains the sprite number, from 1 to 32. The sprite must have already been defined using the CALL SPRITE subroutine.

row velocity and column velocity are optional one word integer variables which will contain the position of the sprite.

For example, the statement:

```
CALL POSITI ( 1, IDOTR, IDOTC )
```

will return the current dot row and dot column of sprite number 1 in the variables IDOTR, IDOTC.

7.5.5 CALL DELSPR

The DELSPR subroutine is used to delete the definition for the referenced sprite.

Calling sequence:

```
CALL DELSPR ( sprite number )
```

where:

sprite number is a one word integer variable or constant which contains the sprite number, from 1 to 32. The sprite must have already been defined using the CALL SPRITE

subroutine.

For example:

```
CALL DELSPR ( 1 )
```

will delete the definition for sprite number 1. The sprite will disappear from the screen.

7.5.6 CALL MAGNIF

The MAGNIF subroutine is used to control the magnification factor of the sprites.

Calling Sequence:

```
CALL MAGNIF ( magnification factor )
```

where:

magnification factor is a one word integer value or constant which defines the current magnification factor for the sprites. It can vary from 1 to 4. If no CALL MAGNIF statement is in your program, the default value used is 1.

Magnification factor 1 causes all sprites to be single size and unmagnified.

Magnification factor 2 causes all sprites to be single size and magnified.

Magnification factor 3 causes all sprites to be double size and unmagnified.

Magnification factor 4 causes all sprites to be double size and magnified.

For example, the statement:

```
CALL MAGNIF ( 4 )
```

would set all sprites to be double sized and magnified.

7.6 Sound Routines

The CALL SOUND subroutine interfaces with the sound generator in your TI-99 GPL or MYARC GENEVE computer.

Calling Sequence:

CALL SOUND (duration, frequency1, volume1 [,...,frequency 4, volume 4])

where:

duration is a one word integer variable or constant which contains the duration of the sound or noise in thousandths of a second. It ranges from 1 to 4250 (to cause a wait until the previous sound is completed) or -1 to -4250 (to terminate the previous sound and start the new sound)

frequency is a one word integer variable or constant which contains the frequency of the note being played from 110 to 32767 hz. Alternately, frequency contains the noise specification, as follows:

-1 Periodic Noise type 1
-2 Periodic Noise type 2
-3 Periodic Noise type 3
-4 Periodic Noise that varies with the frequency of the third tone specified

-5 White Noise type 1
-6 White Noise type 2
-7 White Noise type 3
-8 White Noise that varies with the frequency of the third tone specified.

volume1 to volume4 are one word integer variables or constants which specify the loudness of the note or noise. Zero is the loudest and 30 is the softest. Arguments 2 through 4 are optional.

For example, the statement:

```
CALL SOUND ( 1000, 110, 0 )
```

plays A below low C loudly for one second.

7.6.1 CALL SOUSTA

The SOUSTA subroutine returns the status of the sound generator, whether it is currently playing or not. It returns the status into a single logical variable.

Calling Sequence:

CALL SOUSTA (flag)

where:

flag is a logical variable which will be set .TRUE. if a sound or noise is currently being played, and will be set .FALSE. otherwise.

For example:

```
logical *2 flag1, flag2
call sound ( 1000, 440, 2)
call sousta ( flag1 )
call delay ( 20 )
call sousta ( flag2 )
```

will return .TRUE. in the variable flag1, and .FALSE. in the variable flag2.

7.7 Keyboard and Joystick Subroutines

The following subroutines allow access to the keyboard and the joystick.

7.7.1 CALL KEY

The KEY subroutine provides the interface to the keyboard.

Calling Sequence:

```
CALL KEY ( unit, character, status )
```

where:

unit is a one word integer variable or constant which specifies the key unit number as follows:

- 0 - Read entire keyboard
- 1 - Read left side of keyboard
- 2 - Read right side of keyboard
- 3 - See BASIC user manual
- 4 - See BASIC user manual
- 5 - See BASIC user manual
- 7 - Break Key Check (MDOS Only)
- 8 - Raw Scan Code (MDOS Only)

character is a one word integer variable or constant which will contain the depressed key upon return from the subroutine. The ASCII character returned varies from 1 to 127.

status is a one word integer variable which will contain the status upon return from the KEY subroutine. Status is set as follows:

- 0 No key depressed
- 1 Key depressed, ASCII value in character

For example, the statement:

```
CALL KEY ( 0, KEYC, ISTATUS )
```

will read the entire keyboard, and return a value between 1 and 128 in KEYC if a key was depressed.

Functions 7 (break check) and 8 (raw scan code) are available in the MDOS implementation of the compiler only. Mode 7 returns status =1 and a key code of 255 if the break key is currently pressed. Mode 8 returns a raw scan code of the keyboard in the returned KEYC variable. If there is no keyboard code in the keyboard buffer, then a keycode of 255 is returned.

7.7.2 CALL JOYST

The JOYST subroutine reads the current position of the joystick.

Calling Sequence:

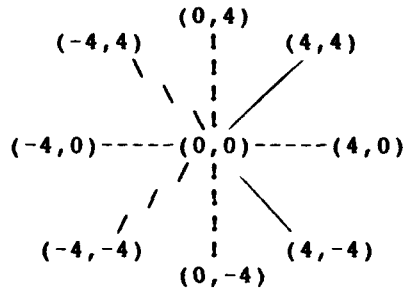
CALL JOYST (key unit, x-return, y-return)

where:

key unit is a one word integer variable or constant which specifies the joystick to read, as follows:

1 - Joystick number 1
2 - Joystick number 2

x-return, y-return are one word integer variables which return the x-y position of the joystick, as follows:



The first value in the parenthesis is returned as x-return, and the second as y-return.

For example, the statement:

CALL JOYST (1,IX,IY)

will read the current position of joystick number 1, and return the x position in variable IX, and the y in variable IY.

7.8 Memory Access Subprograms

The following subprograms allow access to memory areas outside the normal address space of your FORTRAN variables. The PEEK function subprogram has already been described in the intrinsic function summary (see section 7.2)

7.8.1 CALL VMBR/CALL VMBW

The VMBR routine allows reading multiple bytes from VDP RAM into a local variable or array. The VMBW routine reverses this and writes the local variable or array into VDP RAM.

Calling Sequence:

```
CALL VMBR ( VDP location, variable, number bytes )
CALL VMBW ( VDP location, variable, number bytes )
```

where:

VDP location is a one word integer variable or constant which specifies the VDP RAM location to start access from,

variable is an integer variable or array where the read information will be put, and

byte count is the number of bytes to read (VMBR) or write (VMBW) to/from VDP RAM.

For example, the statements:

```
INTEGER IARRAY(484)
CALL VMBR ( 0, IARRAY, '300'X )
```

will read the entire screen image into array IARRAY.

7.8.2 CALL LVMBR/CALL LVMBW (MDOS Only)

The LVMBR and LVMBW VDP access subroutines are only available when using MDOS. They access the extended (up to 128kbytes) of VDP memory available on the MYARC GENEVE. They are similar to the VMBR/VMBW subroutines, except that the VDP location to access is an integer *4 variable, and may range from z'00000' to z'1ffff'.

Calling Sequence:

```
CALL LVMBR ( VDP location, variable, number bytes )
CALL LVMBW ( VDP location, variable, number bytes )
```

where:

VDP location is an integer *4 variable or constant which specifies the VDP RAM location to start access from, from z'00000' to z'1ffff' inclusive,

variable is an integer variable or array where the read information will be written, and

number bytes is the number of bytes to read (LVMBR) or write (LVMBW) to VDP RAM.

For example, the statements:

```
integer *1 array(100)
integer *2 nobytes
integer *4 vdploc
vdploc = z'10040'
nobytes = 100
call lvmbr ( vdploc, array, nobytes )
```

will read 100 decimal bytes starting at VDP location z'10040' into the array "array".

7.8.3 CALL LOADM

The LOADM subroutine is used to put up to ten values into sequential CPU RAM locations.

Calling Sequence:

```
CALL LOADM ( location, value1 [,...,value10] )
```

where:

location is the CPU RAM memory address to write to, and

value1, ,...,value10 are 1 to 10 values (one word integer variables) in which to write to the memory location.

Note that full words (not bytes) are written.

For example, the statement:

```
CALL LOADM ( '834A'X, '0026'X, '1280'X )
```

will load memory locations '834A'X and '834C'X with the values '0025'X and '1280'X.

7.8.4 CALL VWTR/CALL VRFR

The VWTR and VRFR subroutines provide access to the 9918 (TI-99 GPL mode) or 9938 (MDOS GENEVE) Video Chip Registers. The VWTR subroutine writes a byte value to a single register, where as the VRFR subroutine reads a byte value from a video register.

Since both the 9918 and 9938 Video registers are WRITE-ONLY, the value read back using the VRFR functions are actually the saved values at the last time the VWTR function was called. If you chose to bypass the VWTR function and write to the video registers within your own assembly

language subroutine, then FORTRAN cannot retrieve the current register values.

Calling Sequence:

```
CALL VWTR ( register#, value )
CALL VRFR ( register#, value )
```

where:

register# is a one word integer variable which contains the register number to read or write. For the TI-99 GPL mode, it must be from 0 to 7, inclusive. For the MDOS mode it must be the register number from 0 to 46, inclusive.

value is a one word integer variable which contains the value to write to the specified register, or the value read from the register. The value is right justified, is eight bits wide in the sixteen bit field.

For example, the following statement will set video register 2 to a >01:

```
CALL VWTR ( 2, 1 )
```

The value written to video register 2 can later be retrieved via the statement:

```
CALL VRFR ( 2, IVALUE )
```

7.8.5 CALL GVIDTB

The GVIDTB subroutine returns the addresses in Video RAM of the following tables:

Color Table
Pattern Table
Screen Image Table
Sprite Attribute Table
Sprite Pattern Table
Sprite Color Table

Calling Sequence:

```
CALL GVIDTB ( array )
```

where:

array is a two-word integer array of six elements which will contain the integer *4 addresses of the color table, the pattern table, the screen image table, the sprite attribute table, the sprite pattern table, and the sprite color table.

For example, the following will return the starting video RAM addresses of the specified six tables in the array "vaddr":

```
integer *4 vaddr(6)
call gvidtb ( vaddr )
```

7.8.6 CALL CPMBR/CALL CPMBW (MDOS Only)

The CPMBR and CPMBW subroutines provide access to the extended memory of the MYARC GENEVE when running under MDOS only. To use the subroutine, you must first allocate additional memory using the CALL MALLOC subroutine.

The CPMBR subroutine reads data at the given logical cpu memory location into the local variable or array. The CPMBW routine writes data from the local array or variable starting at the given CPU location.

Note that since the cpu location is an integer *4 varaible, addresses above the 64kbyte memory limit of a user task may be addressed.

If you use this subroutine in conjunction with the symbolic debugger, you must only access memory locations from z'40000' and above.

Calling Sequences:

```
CALL CPMBR ( cpu_location, variable, number_bytes )
CALL CPMBW ( cpu_location, variable, number_bytes )
```

where:

cpu_location is an integer *4 variable which is the memory location to access, from z'00000' to z'fffff'. MDOS currently supports 24 bit addressing on local pages. Note that this is NOT the physical memory address, it is the logical memory address based on which pages you have mapped with the CALL MALLOC subroutine.

variable is a variable or array which must be at least number_bytes in length, and is the data to write, or where the data will be placed on a read,

number_bytes is an integer *2 variable or constant which specifies the number of bytes to read or write.

For example:

```
integer *1 array(100)
integer *2 nobytes
integer *4 cpuloc
cpuloc = z'11400'
nobytes = 100
call malloc ( 8, 1, 0, error, noapages, nofpages )
call cpmbr ( cpuloc, array, nobytes )
```

will read 100 bytes of data into the local array "array", from local memory address z'11400'.

Note the subroutine call is preceeded by a call to the MALLOC memory allocation subroutine. This call allocates a single memory page located at logical memory addresses z'10000' to z'11fff' which is accessable using the CALL CPMBR and CALL CPMBW subroutines.

7.9 Miscellaneous Routines

The following subroutines provide useful miscellaneous functions.

7.9.1 CALL QUIT

The QUIT subroutine forces an exit back to the master title screen (when using TI-99 GPL mode) or the MDOS command prompt (when using MDOS mode). All files are closed before exit.

Calling Sequence:

```
CALL QUIT
```

There are no arguments.

7.9.2 CALL WAIT

The WAIT subroutine displays the message:

Press ENTER to Continue

on the screen. When the enter button is depressed, the subroutine returns to the calling program.

Calling Sequence:

```
CALL WAIT
```

There are no arguments.

7.9.3 IRAND Function

IRAND returns a random number.

Calling Sequence:

```
value = IRAND ( top value )
```

where:

value is a one word integer variable which will contain the random number.

top value is a one word integer variable or constant which specifies the range of the random number. The range will be:

0 .LE. value .LT. top value

For example, the statement:

```
IVALUE = IRAND ( 100 )
```

will return a random number between 0 and 99 to the value IVALUE.

Note that the name IRAND must be declared type INTEGER in your program, either implicitly or explicitly.

7.9.4 IVAL/VAL/DVAL Functions

IVAL/VAL/DVAL return an integer, single precision real, or double precision number based on a passed ASCII string of digits, decimal point, and sign character (+ or -).

Calling Sequence:

```
integer variable           = IVAL ( string )
real variable              = VAL ( string )
double precision variable  = DVAL ( string )
```

where:

integer variable is an integer variable which will contain the integer result of the value string, and

real variable is a single precision real variable which will contain the single precision RADIX 100 value of the string, and

double precision variable is a double precision variable which will contain the double precision RADIX 100 value of the string, and

string is an array containing the string to be passed. It must be terminated with a non-number, non-decimal point, or non-sign character.

For example, the statements:

```
DOUBLE PRECISION DVALUE, DVAL
INTEGER STRING(5)
DATA STRING / '12', '34', '.5', '67', '8' /
DVALUE = DVAL ( STRING )
```

Upon execution of the DVAL function, the variable DVALUE will contain the floating point number 1234.5678. Note that the character is used to terminate the string. Note also, since the IVAL/VAL/DVAL routines are not INTRINSIC functions, their type must be declared properly.

7.9.5 CALL EXIT

The EXIT subroutine performs the same function as the STOP statement, except that an immediate return is performed to the main menu routine, with no STOP message or WAIT message on the screen.

Calling Sequence:

```
CALL EXIT
```

There are no arguments.

7.9.6 CALL DELAY

The DELAY subroutine provides specific time delays in your program. Delays are specified in tenths of a second.

Under TI-99 GPL mode, the MENU routine maintains a double word real time clock in memory locations '2028'X and '202A'X. This clock is incremented every sixth VDP interrupt, which is every 1/10th of a second. Note that interrupts are inhibited during Input/ Output, so that the clock will lose time during disk I/O operations.

Delay clears the second word of the clock (location '202A'X), and waits for the value in that cell to be incremented to the requested value (tenths of a second). When it has, control is returned to the calling program.

Under MDOS mode, no real-time clock is maintained since MDOS does not allow for user interrupt routines. Instead, the real-time clock last digit is examined in a polling loop, and when it increments by the desired amount, control is returned to the user. In this mode, time will not be lost during disk I/O operations.

Calling Sequence:

```
CALL DELAY ( Time )
```

where:

Time is a one word integer variable, which specifies the amount of time to delay, in tenths of a second.

For example, the statement:

```
CALL DELAY ( 15 )
```

would cause the program to delay 1.5 seconds (15 tenths of a second).

7.9.7 CALL CHAIN

The CHAIN subroutine allows you to call one FORTRAN program from another.

Calling Sequence:

```
CALL CHAIN ( file name, error )
```

where:

file name - is an array which contains the disk file name of program to be called. Its length must be less than 40 characters, and it must be terminated by a blank.

error - is a one word integer variable which is set if an

error condition was detected during the operation. It is set to -1 if the the file name was too long, or it is set to the I/O error code if an I/O error was detected during the load, or the MDOS error code when using the MDOS implementation.

In the TI-99 GPL implementation, not all errors cause a return to the calling program, some will cause a return to the MENU.

In the MDOS implementation, the called program need not be a FORTRAN program, it can be a C99 or assembly program. Also, in MDOS the CHAIN subroutine always returns to the calling program. The called program multi-tasks with the calling program (unless the called program sets the multi-task lock via the CALL LOCK subroutine).

Refer to the appendix for tips on data passing between programs. Note also that all files are closed between program calls. You cannot leave a file open and expect to use it in the called routine.

7.9.8 CALL LOCK

The LOCK subroutine allows you to disable or reenable multi-tasking in the MDOS implementation only. Refer to the MDOS programming documentation concerning the concept of multi-tasking.

Calling Sequence:

```
CALL LOCK ( flag )
```

where:

flag is a one-word logical *2 variable which is set to .TRUE. to disable multi-tasking, and is set to .FALSE. to reenable multi-tasking.

This subroutine is especially useful for a child task (a program which has been called via the CHAIN subroutine). By setting the lock via CALL LOCK (.TRUE.), the child task will complete before control is returned to the calling task (or until the child task does a CALL LOCK (.FALSE.)). Otherwise, the called routine and the calling routine will multi-task (share program execution).

7.10 Extended Graphics Library

All of the following subroutines apply to the the 9640 GENEVE computer, running under MDOS. Some of the following subroutines (e.g. SETMOD, GETMOD) may be used in the TI-99 GPL implementation.

The functions are available by linking to the 'GL' (Graphics Library) library on the FORTRAN distribution disk.

7.10.1 CALL SETMOD

Sets the current video mode according to the mode variable, from 0 to 9 MDOS mode), or 0 to 3 (TI-99 GPL mode), as follows:

#	description	Cols	Rows
0	Text 1 Mode	40	24
1	Text 2 Mode	80	26
2	Multi-Color	--	--
3	Graphics 1	32	24
4	Graphics 2	32	24
5	Graphics 3	--	--
6	Graphics 4	40	26
7	Graphics 5	80	26
8	Graphics 6	80	26
9	Graphics 7	40	26

Calling Sequence:

```
CALL SETMOD ( mode )
```

where:

mode is a one-word integer variable that specifies the graphics mode, from 0 to 9.

For example, the statement:

```
CALL SETMOD ( 2 )
```

Sets the current graphics mode to 80 column text mode 2, the default mode of MDOS (and the same as using the CALL SET80 statement).

Note that graphics mode 1 invoked by CALL SETMOD(3) is not the same as the graphics mode used by the TI-99. If you wish true compatibility with TI-BASIC graphics mode 1, then you should use a CALL SET32 statement to invoke this mode, not a CALL SETMOD(3). 9640 FORTRAN provides a special compatibility mode with a CALL SET32 statement.

7.10.2 CALL GETMOD

GETMOD returns the current video mode, from 0 to 9 (MDOS implementation) or 0 to 3 (TI-99 GPL implemenation). Optionally, the

number of columns and number of rows are returned also.

Calling Sequence:

```
CALL GETMOD ( mode [,irow, icolumn )
```

where:

mode is a one-word integer variable that will contain the graphics mode, from 0 to 9.

irow is an optional one-word integer variable that will contain the number of rows on the screen, and

icolumn is an optional one-word integer variable that will contain the number of columns on the screen.

For example, the statement:

```
CALL GETMOD ( MODE )
```

would return a 3 in the MODE variable, if the screen mode had been set to 3 via the SETMOD statement, or if the SET32 subroutine had been called.

7.10.3 CALL SETPOS

The SETPOS subroutine sets the current row and column on the screen.

Calling Sequence:

```
CALL SETPOS ( irow, icolumn )
```

where:

irow is a one-word integer variable specifying the row number to position to, and

icolumn is a one-word integer variable specifying the column number to position to.

The arguments irow and icolumn are specified from 0 to n, where n is one minus the number of rows or columns which can be displayed in the current video mode. For example, in mode 1 (text 2 mode), irow must be in the range of 0 to 25, and icolumn must be in the range of 0 to 79. An execution time error will result if the arguments are not in the specified range.

For example, the statements:

```
call setpos ( 10, 10 )
write ( 6, 9100 )
9100 format ( '+text string')
```

would cause the cursor to be positioned to row 10, column 10, and the text string 'text string' to be written there.

7.10.4 CALL GETPOS

The GETPOS subroutine returns the current row and column where the cursor is positioned.

Calling Sequence:

```
CALL GETPOS ( irow, icolumn )
```

where:

irow is a one-word integer variable which will contain the row number positioned to, and

icolumn is a one-word integer variable which will contain the column number positioned to.

For example, the statement:

```
CALL GETPOS ( IROW, ICOLUMN )
```

would return 10 for IROW, and 10 for ICOLUMN, if issued directly after the CALL SETPOS statement from above.

7.10.5 CALL SETVPG (MDOS Only)

The SETVPG subroutine causes the specified video page number to be displayed on the screen.

Calling Sequence:

```
CALL SETVPG ( ipage )
```

where:

ipage is a one-word integer variable specifying the video page number to display, from 0 to n.

For example, the statement:

```
CALL SETVPG ( 0 )
```

would set the current displayed video page to page 0.

7.10.6 CALL GETVPG (MDOS Only)

The GETVPG subroutine causes the current displayed video page number to be returned in the specified variable.

Calling Sequence:

```
CALL GETVPG ( ipage )
```

where:

ipage is a one-word integer variable which will contain the current displayed video page number, from 0 to n.

For example, the statement:

```
CALL GETVPG ( IPAGE )
```

would return 0 in the variable IPAGE, unless the video page has been changed via the SETVPG subroutine.

7.10.7 CALL SCRLUP, CALL SCRLDN, CALL SCRLLE, CALL SCRLRI (MDOS Only)

The window scroll functions provide scrolling in any of four directions (up, down, left, or right), given the coordinates of the window to scroll, the number of lines to scroll, the character to blank lines with, and the color to blank lines with.

Calling Sequences:

```
CALL SCRLUP ( nolines, row_ul, col_ul, row_lr,
    col_lr, character, foreground, background )

CALL SCRLDN ( nolines, row_ul, col_ul, row_lr,
    col_lr, character, foreground, background )

CALL SCRLRI ( nolines, row_ul, col_ul, row_lr,
    col_lr, character, foreground, background )

CALL SCRLLE ( nolines, row_ul, col_ul, row_lr,
    col_lr, character, foreground, background )
```

where:

nolines is a one-word integer variable which is the number of lines to scroll,

row_ul is a one-word integer variable which is the row number of the upper left corner,

col_ul is a one-word integer variable which is the column number of the upper left corner,

row_lr is a one-word integer variable which is the row number of the lower right corner,

col_lr is a one-word integer variable which is the column number of the lower right corner,

character is a one-word integer variable containing the character code number to written for blank lines,

fore_cl is the foreground color to use for writing the blank character, and

back_cl is the background color to use for writing the blank character.

The functions operate as follows:

```
CALL SCRLUP - Scroll window up
CALL SCRLDN - Scroll window down
CALL SCRLLE - Scroll window left
CALL SCRLRI - Scroll window right
```

For example, the statement:

```
CALL SCRLUP ( 5, 10, 10, 20, 20, Z'0030', 2, 3 )
```

will scroll the window at upper left 10,10 and lower right 20,20 up five lines. Lines left behind will be blanked with the character z'0030' (ASCII 0), and colored foreground and background with colors 2,3.

7.10.8 CALL SETBRD (MDOS Only)

The SETBRD subroutine sets the border color to the value specified.

Calling Sequence:

```
CALL SETBRD ( color )
```

where:

color is a one-word integer variable specifying the color to render the border.

For example, the statement:

```
CALL SETBRD ( 3 )
```

will set the border color to light green, when in 32-column mode.

7.10.9 CALL SETPAL (MDOS Only)

The SETPAL subroutine is used to set the pallette colors in the graphics chip registers to the specified color combination.

Calling Sequence:

```
CALL SETPAL ( register, red, blue, green )
```

where:

register is a one-word integer variable which contains the pallette register number to modify from 0 to 15,

red is a one-word integer variable which specifies the intensity of the color red, from 0 to 7,

blue is a one-word integer variable which specifies the intensity of the color blue, from 0 to 7, and

green is a one-word integer variable which specifies the intensity of the color green, from 0 to 7.

For example, the statement:

```
CALL SETPAL ( 4, 7, 0, 0 )
```

will turn the background color of the screen dark red in 80 column mode.

7.10.10 CALL SETPIX (MDOS Only)

The SETPIX subroutine sets the pixel at the specified x and y coordinates to the specified foreground color.

Calling Sequence:

```
CALL SETPIX ( ixcord, iycord, ifore )
```

where:

ixcord is a one-word integer variable which specifies the x-coordinate of the pixel, from 0 to 511,

iycord is a one-word integer variable which specifies the y-coordinate of the pixel, from 0 to 1023,

ifore is a one-word integer variable which specifies the foreground color number, from:

```
graphics mode 4,6 : 0 to 15
graphics mode 5 : 0 to 3
graphics mode 7 : 0 to 255
```

Notes:

1. This subroutine should only be called in graphics modes 4 to 7.

2. The color byte should be specified according to the grqphics mode above.

3. In the interests of speed, the only error checks on the passed data is the x coordinate value (0 to 511), the y coordinate value (0 to 1023) and the color (0 to 255).

For example, the following statements would set a dark red pixel on the screen in coordinates 100,120:

```
call setmod ( 9 )                ! graphics mode 7
call setpix ( 100, 120, z'00e0' )
```

7.9.11 CALL GETPIX (MDOS Only)

The GETPIX subroutine returns the foreground, and possibly background colors of the specified pixel (background in graphics modes 2 and 3 only).

Calling Sequence:

```
CALL GETPIX ( ixcord, iycord, ifore [,iback] )
```

where:

ixcord is a one-word integer variable which specifies the x-coordinate of the pixel,

iycord is a one-word integer variable which specifies the y-coordinate of the pixel,

ifore is a one-word integer variable which will con- tain the foreground color number, and

iback is an optional one-word integer variable which will contain the background color number (only returned in video modes 4 and 5, graphic modes 2 and 3).

7.10.12 CALL SETVEC (MDOS Only)

The SETVEC subroutine effectively draws lines on the screen, by specifying two pixel coordinates, and a foreground and optional background color to render the vector between the coordinates.

Calling Sequence:

```
CALL SETVEC ( 1st_x, 1st_y, 2nd_x, 2nd_y,
    foreground [,background] )
```

where:

1st_x is a one-word integer variable which contains the first X pixel coordinate,

1st_y is a one-word integer variable which contains the first Y pixel coordinate,

2nd_x is a one-word integer variable which contains the second X pixel coordinate,

2nd_y is a one-word integer variable which contains the second Y pixel coordinate,

foreground is a one-word integer variable which contains the

foreground color to render vector, and

background is an optional one-word integer variable which contains the background color to render vector (only needed for graphics modes 2 and 3)

For example, the statement:

```
CALL SETVEC ( 10,10,200,200,2 )
```

will draw a green line diagonally on the screen say in screen mode 7 (see SETMOD subroutine).

7.10.13 CALL CLRSRC (MDOS Only)

The CLRSRC subroutine searches the pixels on the screen for the specified color. Both LEFT and RIGHT searches are supported, starting at a given pixel coordinate.

Calling Sequence:

```
CALL CLRSRC ( x-cord, y-cord, color, direction,
     x1_cord, y1_cord )
```

where:

x-cord is a one-word integer variable which specifies the source x pixel coordinate,

y-cord is a one-word integer variable which specifies the source y pixel coordinate,

color is a one-word integer variable which specifies the color to search for,

direction is a one-word integer variable, which is set to the value -1 for a RIGHT search, and 0 for a LEFT search,

x1_cord is a one-word integer variable which will contain the found x-coordinate (-1 if not found), and

y1_cord is a one-word integer varaible which will contain the found y-coordinate (-1 if not found).

For example, the statement:

```
CALL CLRSRC ( 10, 10, 2, -1, NXCORD, NYCORD )
```

will search to the right of pixel coordinates 10,10 for color 2, and if found will return the new pixel coordinates in the variables NXCORD and NYCORD. If the color is not found, then the variables NXCORD and NYCORD will be set equal to -1.

7.10.14 CALL HBLKMV, HBLKCP, LBLKMV, LBLKCP (MDOS Only)

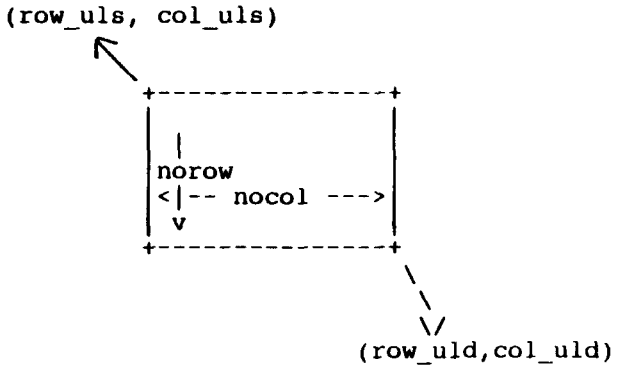
The CALL HBLKMV, HBLKCP, LBLKMV, LBLKCP perform moves on blocks of pixel data on the screen. The moves are actually performed by the 9938 video chip, and so are very fast.

The CALL HBLKMV and HBLKCP subroutines perform a high speed move or copy of the pixel data to the specified coordinates. These subroutines destroy the pixel data where the data is being moved "to".

The CALL LBLKMV and LBLKCP subroutines also perform a high speed move or copy of the specified pixel data to the specified cooridinates. In this case, however, a logical operation of the "moved to" data is performed with original pixel data on the screen according to the "logic" argument.

The CALL HBLKMV and LBLKMV move subroutines also require a color argument which specifies the color to blank the pixels from where the data is being moved "from".

A block of area to be moved is specified by six arguments, in terms of number of pixels, as follows:



(row_uld,col_uld)

Calling Sequence:

```
CALL HBLKMV ( row_uls, col_uls, row_uld, col_uld,
    norow, nocol )

CALL HBLKCP ( row_uls, col_uls, row_uld, col_uld,
     norow, nocol, color, logic )

CALL LBLKMV ( row_uls, col_uls, row_uld, col_uld,
     norow, nocol, logic )

CALL LBLKCP ( row_uls, col_uls, row_uld, col_uld,
     norow, nocol, color )
```

where:

row_uls is a one-word integer variable which contains the row number of the upper left corner of source, (same as

y-coordinate)

col_uls is a one-word integer variable which contains the column number of the upper left corner of source, (same as x-coordinate)

row_uld is a one-word integer variable which contains the row number of the upper left corner of destination, (same as y-coordinate)

col_uld is a one-word integer variable which contains the column number of the upper left corner of destination, (same as x-coordinate)

norow is a one-word integer variable which contains the number of rows,

nocol is a one-word integer variable which contains the number of columns,

color is a one-word integer variable which contains the pixel color for blank pixels, and

logic is a one-word integer variable which contains the logic operation to be performed on the destination.

An example of using some of the subroutines is:

```
      call setmod ( 9 )
      call clear
c build a block on the screen
      do 1000 i=50,79
1000  call setvec ( 10, i, 39, i, 120 )
c move the built block to coordinates 100,100
      call delay ( 40 )
      call hblkmv ( 50, 10, 100, 100, 30, 30, 64 )
c copy the built block to coordinates 70,70
      call delay ( 40 )
      call hblkcp ( 50, 10, 70, 70, 30, 30 )
      end
```

This program sets the graphics mode to 9; clears the screen; builds an orange block at coordinates 50,10; moves the orange block to coordinates 100, 100 and turns the orange block green; and copies the green block to coordinates 70,70.

7.10.15 CALL BLKSUP, BLKSDN, BLKSLE, BLKSRI (MDOS Only)

The block scroll functions provide scrolling of pixel based graphics in any of four directions (up, down, left or right), given the pixel row and columns, the number of pixels to scroll, and the pixel color for blank pixels.

Calling Sequences:

```
CALL BLKSUP ( nopixels, row_ul, col_ul, row_lr,
    col_lr, pixel_color )

CALL BLKSDN ( nopixels, row_ul, col_ul, row_lr,
    col_lr, pixel_color )

CALL BLKSLE ( nopixels, row_ul, col_ul, row_lr,
    col_lr, pixel_color )

CALL BLKSRI ( nopixels, row_ul, col_ul, row_lr,
    col_lr, pixel_color )
```

where:

nopixels is a one-word integer variable which is the number of pixels to scroll

row_ul is a one-word integer variable which is the row number of the upper left corner,

col_ul is a one-word integer variable which is the column number of the upper left corner,

row_lr is a one-word integer variable which is the row number of the lower right corner,

col_lr is a one-word integer variable which is the column number of the lower right corner,

pixel_ is a one-word integer variable which is the color pixel color for blank pixels

The functions operate as follows:

```
CALL BLKSUP - Scroll Block Up
CALL BLKSDN - Scroll Block Down
CALL BLKSLE - Scroll Block Left
CALL BLKSRI - Scroll Block Right
```

For example, the statement:

```
CALL BLKSUP ( 5, 10, 10, 20, 20, 2 )
```

will scroll the window at upper left 10,10 and lower right 20,20 up five lines. Pixels left behind will be blanked with the color 2.

7.10.16 CALL SETTWN (MDOS Only)

The SETTWN subroutine sets up a text window, at the specified four points on the screen.

Calling sequence:

```
CALL SETTWN ( top, left, bottom, right )
```

where:

top is a one-word integer variable which specifies the top row of the window,

left is a one-word integer variable which specifies the leftmost column of the text window,

bottom is a one-word integer variable which specifies the bottom row of the window, and

right is a one-word integer variable which specifies the rightmost column of the window.

For example, the statement:

```
CALL SETTWN ( 5, 8, 20, 25 )
```

would open a window between columns 8 and 25, and rows 5 and 20.

Subsequent writes using WRITE statements will be within this window.

7.10.17 CALL GETTWN (MDOS VIDEO OPCODE >26)

The GETTWN subroutine returns the current coordinates of the text window, as specified by four points on the screen.

Calling sequence:

```
CALL GETTWN ( top, left, bottom, right )
```

where:

top is a one-word integer variable which will contain the top row of the window,

left is a one-word integer variable which will contain the leftmost column of the text window,

bottom is a one-word integer variable which will contain the bottom row of the window, and

right is a one-word integer variable which will contain the rightmost column of the window.

For example, the statement:

```
CALL GETTWN ( ITOP, ILEFT, IBOTTOM, IRIGHT )
```

would return the values 5, 8, 20, and 25 in the specified variables ITOP, ILEFT, IBOTTOM, and IRIGHT if this statement followed the CALL SETTWN statement in SETTWN example.

7.10.18 CALL RESCHA (MDOS Only)

The RESCHA subroutines restores the character set definition (reloads the character pixels), given a starting character number, an ending character number, and an optional character set pattern array.

Calling Sequence:

```
CALL RESCHA ( istart, iend [,charse] )
```

where:

istart is a one-word integer variable specifying the starting character number to restore, from 0 to 255,

iend is a one-word integer variable specifying the ending character number to restore, from 0 to 255, and

charse is an optional array which contains the pattern definitions for the specified characters.

7.10.19 CALL SETMSE (MDOS Only)

The SETMSE subroutine sets the x and y positions of the mouse, and also the mouse speed in eight increments.

NOTE: This routine MUST be called before using CALL GETMSE or CALL GETMSR, otherwise MDOS will hang and you will have to cold boot your GENEVE to continue (as of MDOS 1.14).

Calling Sequence:

```
CALL SETMSE ( x_pos, y_pos, scale )
```

where:

x_pos is a one-word integer variable which contains the new desired x position for the mouse,

y_pos is a one-word integer variable which contains the new desired y position for the mouse,

scale is a one-word integer variable which contains the scale factor for the mouse speed, a number from 0 to 7, where 0 is the fastest speed.

7.10.20 CALL GETMSE, CALL GETMSR (MDOS Only)

The GETMSE subroutine returns the x and y positions of the mouse, and also returns three logical variables representing the button positions of the left, middle, and right pushbuttons.

The GETMSR subroutine returns the relative x and y positions of the mouse, since the last call to GETMSE or GETMSR.

NOTE: You MUST call SETMSE before calling GETMSE or GETMSR, otherwise

MDOS will hang and you will need to cold boot your GENEVE to continue (as of MDOS 1.14).

Calling Sequence:

```
CALL GETMSE ( x_pos, y_pos, left, middle, right )
CALL GETMSR ( x_pos, y_pos, left, middle, right )
```

where:

x_pos is a one-word integer variable which will contain the current x position for the mouse, or the displacement since the last call to GETMSE or GETMSR,

y_pos is a one-word integer variable which will contain the current y position for the mouse, or the displacement since the last call to GETMSE or GETMSR,

left is a one-word logical variable which will be .TRUE. if the leftmost button is pressed, .FALSE. otherwise,

middle is a one-word logical variable which will be .TRUE. if the middle button is pressed, .FALSE. otherwise,

right is a one-word logical variable which will be .TRUE. if the right button is pressed, .FALSE. otherwise.

7.11 DATE/TIME Library (MDOS Only)

The DATE/TIME Library routines provide access to the real-time clock in the GENEVE. Routines are provided to check the current time and date for validity, to return the time and date to text strings, to set the time and date from text strings, and to return the day of the week.

These subroutines call the utility XOP library within MDOS, and as such are not available under GPL mode. Any attempt to call the following routines within MDOS will result in an error 'MO' (MDOS only) to be displayed.

All of these routines are included in the 'FL' non-math FORTRAN library (for MDOS only).

7.11.1 CALL CHETIM/CHEDAT (MDOS Only)

The CHETIM routine checks the current time in the real-time clock for validity, and returns a flag .TRUE. or .FALSE. depending on whether the time and data is valid (.TRUE.) or invalid (.FALSE.).

The CHEDAT routine operates in the same manner, except the date is checked instead of the time.

Calling Sequences:

```
CALL CHETIM ( flag )
CALL CHEDAT ( flag )
```

where:

flag is a logical *2 argument which is .FALSE. if the time or date is incorrect, or .TRUE. if the time or date is correct.

For example, the sequence:

```
logical *2 datflag
call chedat ( datflag )
```

will return a .TRUE. in datflag if the date is currently valid, will return a

7.11.2 CALL CONTTS/CONDTS (MDOS Only)

The CONTTS and CONDTS subroutines convert the time and date to strings, suitable for display or printing.

Calling Sequences:

```
CALL CONTTS ( string )
CALL CONDTS ( string )
```

where:

string is at least a ten byte integer array to hold the current time, or at least an eight byte integer array to hold the date.

For example:

```
integer *1 date(8)
call condts ( date )
```

would return an eight byte string representing the date in the array "date". If today was Nov. 27, 1988, then the string would contain: 11-27-88.

7.11.3 CALL CONSTT/CONSTD (MDOS Only)

The CONSTT/CONSTD subroutines convert a given string representing either the time or date into the system time, and changes the system time to match. Error checking is performed on the passed string, and a flag is returned to indicate whether the string was accepted.

Calling Sequences:

```
CALL CONSTT ( string, flag )
CALL CONSTD ( string, flag )
```

where:

string is an array which contains the text string representing the time (at least 10 bytes) or the date (at least 8 bytes),

flag is a logical *2 variable which is set .TRUE. if the new date or time was accepted by the system, .FALSE. if it was not.

For example:

```
      logical *2 flag
      integer *2 date(4)
      data date / 8h11-27-88 /
      call constd ( date, flag )
      if ( flag ) write ( 6, 9100 )
9100  format ( ' date was accepted')
```

would set the date to November 27, 1988; and display the statement 'date was accepted' on the CRT.

7.11.4 CALL CONJUL (MDOS Only)

The CONJUL subroutine is passed the month, the day, and the year as three integer *2 values. It returns the two word "internal" format date as an integer *4 value.

Calling Sequence:

```
CALL CONJUL ( mm, dd, yyyy, packed_date )
```

where:

mm is an integer *2 variable which is the current month (from 1 to 12),

dd is an integer *2 variable which is the current day (from 1 to 31),

yyyy is an integer *2 variable which is the current year (from 1988 to 32,767), and

packed_date is an integer *4 variable which is the converted 'internal' date in real time clock format

For example:

```
integer *4 packdate
call conjul ( 11, 27, 1988, packdate )
```

would return the current date in the integer *4 variable packdate.

7.11.5 CALL RETDOW (MDOS Only)

The RETDOW subroutine returns the current day of week, from 1 (Sunday) to 7 (Saturday).

Calling Sequence:

```
CALL RETDOW ( day )
```

where:

day is an integer *2 variable which is the returned day of week, from 1 to 7.

For example:

```
integer *2 day
call retdow ( day )
```

will return a value from 1 to 7 in the variable "day".

7.12 MEMORY MANAGER LIBRARY (MDOS ONLY)

The MEMORY MANAGEMENT subroutines interface to the MDOS Memory Manager Library (XOP SEVEN). Library routines which deal with obtaining more memory, returning the status of memory, and returning memory to the system are provided.

Note that a FORTRAN task uses all or part of the first 64k of a local task memory space. If the FORTRAN symbolic debugger is invoked, all or part of the second 64k task memory space is also used. Loading a symbol file and/or source files using the symbolic debugger will cause all or part of the third 64k memory space to be used.

```
address 00000 to 0ffff  :  FORTRAN task
        10000 to 1ffff  :  Symbolic Debugger
        20000 to 2ffff  :  Symbol file and Source files
```

It is not necessary for the average FORTRAN user to interface to these subroutines, FORTRAN allocates the necessary memory needed for the task at startup. Advanced users may wish to allocate more memory for applications.

Memory in the GENEVE is segregated into PAGES. Each page is 8192 bytes (>2000 hexadecimal bytes). Since the processor used in the GENEVE has a 16 bit (64k) address space, the actual pages which make up the current 64k address is specified in an eight byte memory mapper register space (located at >f110). A task cannot directly use more than 64k of memory.

The MDOS executive allocates memory to a user task as requested by the task, and maintains a local memory page list. The first eight bytes of this list usually match what is in the mapper registers, and are addressed from >0000 to >ffff (local pages 0 to 7). Additional 8k pages of memory can be allocated using the memory management subroutines, and swapped with local execution pages.

The eight pages which comprise the first 64k of memory are referred to as execution pages.

7.12.1 CALL RTFREE (MDOS Only)

The RTFREE routine returns the number of free pages left in the system memory, including the number of system and fast pages.

Calling Sequence:

```
CALL RTFREE ( error, nofree, noffree, nosfree )
```

where:

error is a one word integer variable which will contain any error code returned (0=no error),

nofree is a one word integer variable which will contain the

number of free pages in the system,

noffree is a one word integer variable which will contain the number of fast free pages in the system, and

nosfree is a one word integer variable which will contain the total number of system memory pages (used and free).

For example:

```
call rtfree ( ierror, nofree, noffree, nosfree )
```

might return the following:

```
ierror = 0, nofree=31, noffree=3, nosfree=68
```

indicating no error, free pages left of decimal 31 (253,952 bytes), the number of free fast pages of decimal 3 (24,576 bytes), and the total number of system pages (in use and available) of decimal 68 (557,056 bytes).

7.12.2 CALL MALLOC (MDOS Only)

The MALLOC subroutine allocates memory to the local page list. The user passes the starting page number (0 to n), the number of pages to get (0 to n), whether fast or slow memory is desired. The library attempts to get the desired memory, and returns an error code if unsuccessful. Also, the number of pages actually fetched and the number of fast pages actually fetched are returned.

Calling Sequence:

```
CALL MALLOC ( spage, nopages, speed, error,
     noapages, nofpages)
```

where:

spage is a one word integer variable which is the starting page number to fetch, from 0 to n,

nopages is a one word integer variable which is the number of pages to fetch, from 1 to n,

speed is a one word integer variable set to non-zero if fast memory is desired,

error is a one word integer variable which will contain any error returned (0 means no error),

noapages is a one word integer variable which is the number of pages actually fetched, and

nofpages is a one word integer variable which is the number of fast pages actually fetched.

For example, the following code will fetch 6 pages of memory, from >10000 to >1ffff, into the local page map:

```
call malloc ( 8, 6, 0, ierror, nopages, nofpages )
```

7.12.3 CALL RTPAGE (MDOS Only)

The RTPAGE subroutine performs the reverse of MALLOC, it returns the desired number of pages to the system map. Pages returned will no longer be available for the user task.

Calling Sequence:

```
CALL RTPAGE ( nopages, pageno, error )
```

where:

nopages is a one word integer variable which is the number of pages to return to the system,

pageno is a one word integer variable which is the starting page number to return, from 0 to n,

error is a one word integer variable which is any error code returned.

For example:

```
call rtpage ( 4, 8, error )
```

will return four pages of memory, starting at page number 8 (address >10000) to the system map. If no error is encountered, the variable error will contain zero upon return.

7.12.4 CALL MPLCPE (MDOS Only)

The MPLCPE subroutine maps a local page, which may extend past the 64k FORTRAN task execution space, to an execution page, which is in the 64k FORTRAN task execution space.

This subroutine must be used with extreme caution, since it is possible to map the memory page you are currently executing in to anothor memory page.

Calling Sequence:

```
CALL MPLCPE ( locpage, exepage, error )
```

where:

locpage is a one word integer variable which is the local page number to map, from 0 to n,

exepage is a one word integer variable which is the execution

page number to map to, from 0 to 7, and

error is a one word integer variable which is the error code returned, if any.

For example:

```
call mplcpe ( 8, 4, error )
```

would map local page number 8 to execution page 4. Any error code would be returned in the variable error.

7.12.5 CALL RTMAPR (MDOS Only)

The RTMAPR routine returns the local page map in the user array. This map would contain the page numbers for each 8192 byte map in the local map.

Calling Sequence:

```
CALL RTMAPR ( array, arraysize, error )
```

where:

array is an integer *1 array which will contain the local memory map,

arraysize is a one word integer variable which is the size of the array, in bytes, and

error is a one word integer variable which is any error code returned.

For example:

```
integer *1 memmap(8)
call rtmapr ( memmap, 8, error )
```

would return the local execution memory map in the array memmap.

8.0 Introduction

Programming examples for 99 FORTRAN and 9640 FORTRAN are included on the first library disk. These examples are intended to show the usage of various aspects of the FORTRAN system. They are not all encompassing.

For 99 and 9640 FORTRAN, a simple menu-driven spreadsheet program is supplied on the first library disk (99 FORTRAN) or the library/demonstration disk (9640 FORTRAN). This program allows you to create and modify matrices of values, which then can be calculated upon using your defined equations.

For 9640 FORTRAN (MDOS implementation), the following additional example programs are included on the library disk:

DRIVERS : Sine Wave Display Plotter
FRACTALS : Fractalish Terrain Generator
FILEZAPS : FileZap Sector Editor

8.1 99/9640 FORTRAN Programming Example

The CALC program on the library disk is an example of a simple spreadsheet program. It demonstrates many features of 99/9640 FORTRAN, including floating point arithmetic, how to construct and use menus, sprite capability, call key usage, include file usage, and many others.

Before diving into the demonstration program, let's discuss the basics of a matrix and a spreadsheet program.

A matrix consists of values seperated into rows and columns. Columns go from left to right (and are numbered 1 to 15), while rows go from top to bottom. For example, the matrix:

	1	2	3	4
FIRST	1.2	8.9	222.6	
SECOND	66.6	200.0	800.0	

would have the value of 8.9 at row 1, column 2. The name "SECOND" is located at row 2, column 1.

This matrix is termed a "data" model.

Associated with a spreadsheet is a "logic" model. A logic model specifies the operations to be performed on the "data" model. For example, to add together rows 1 and 2, above, to produce a total row 3, the logic model:

```
3 = 1 + 2
```

would be specified, and would produce:

	1	2	3	4

FIRST	1.2	8.9	222.6
SECOND	66.6	200.0	800.0
TOTAL	67.8	208.9	1022.6

in row 3.

Logic models also specify the names associated with each row (in column 1, as above), the overall spreadsheet name, and four special values (total row and column, last row and column).

So, defining a spreadsheet consists of specifying the logic model, the data model, and performing the calculation function. Functions are also provided to:

1. Edit the data model
2. Edit the logic model
3. List the logic model to the screen
4. Print the logic & data models to a printer
5. Re-initialize the entire spreadsheet
6. Save the logic model, data model, or both to a disk file
7. Load an old logic model, data model, or both from a disk file
8. Calculate the logic model

8.1.1 Compiling the Program

Before the program can be used, it must be compiled and linked. Format a new, blank, disk using the DISK MANAGER module and copy the following three files from the first library disk to the new formatted program disk:

```
DSK.FORTLIBR.CALC
DSK.FORTLIBR.CALC1
DSK.FORTLIBR.COMMON
```

Boot FORTRAN using the procedure outlined in Section 1. Follow one of the procedures depending on whether you are using the 99 (TI-99 GPL) FORTRAN implementation or the 9640 (MDOS) implementation:

TI-99 GPL Implementation

Select the "compile" (option 2) on the FORTRAN main menu.

Insert the program disk you have copied the three files onto in disk drive 1, and answer the following questions on the FORTRAN compile mode menu:

```
Input File Name?
DSK1.CALC

Object File Name?
DSK1.CALCOBJ
```

```
Listing File Name?
CRT

Scratch Disk Number?
1

Compilation Options?


Press ENTER to Continue
```

The compilation will begin. After the compilation completes, perform the same steps, except use an input file name of DSK1.CALC1 and an object file name of DSK1.CALCOBJ1.

MDOS Implementation

Put the FORTRAN BOOT disk in drive 2, the program disk in drive 1. Type the following command at the MDOS prompt:

```
B:F9640 /ODSK5.CALCOBJ A:CALC
B:F9640 /ODSK5.CALCOBJ1 A:CALC1
```

The programs should compile normally with no errors.

8.1.2 Linking the Program

After the program has been compiled, producing two object files (CALCOBJ and CALCOBJ1), it must be linked along with the library routines to produce an executable image. Follow one of the procedures based on your implementation:

TI-99 GPL Implementation

For the TI-99 GPL implementation, insert the BOOT disk into one of the disk drives, and select item 3 on the FORTRAN main menu.

Insert the program object disk in drive 1, and answer the following questions on the Linker Menu:

```
Executable File Name?
DSK1.CALCEXE

Listing File Name?
CRT

Object File Name?
DSK1.CALCOBJ
```

The first object file will be loaded, and the next object file will be requested. Enter:

```
Object File Name?
DSK1.CALCOBJ1
```

The second object file will be loaded and the next object file will be requested. Just press enter, and the following message will be displayed:

```
Unresolved References
Scan Library Name:
```

Insert the second FORTRAN library disk into disk drive DSK1., and enter the following on the screen:

```
DSK1.ML
```

Insert the first FORTRAN library disk into disk drive DSK1., and enter the following on the screen:

```
DSK1.FL
```

The FORTRAN libraries will be scanned, and all necessary routines will be loaded into main memory.

A link map will be displayed on the screen, along with the messages:

```
Saving File DSK1.CALCEXE
Saving File DSK1.CALCEXF
Press ENTER to Continue
```

Press ENTER to return to the FORTRAN main menu.

MDOS Implementation

Insert the first FORTRAN boot disk into disk drive A:, the library/demonstration disk into disk drive B:, and type in the following at the MDOS command prompt:

```
A:FLINK /OE:CALCOBJ,E:CALCOBJ1 /IB:ML,A:FL,A:GL E:CALCEXE
```

The linker should link the program with no errors, and create two executable modules, CALCEXE and CALCEXF.

8.1.3 Loading the Program

When using the TI-99 GPL implementation, the spreadsheet program may be loaded using item 7 on the FORTRAN main menu. It may also be run using item 5 on the Editor/Assembler menu page "RUN PROGRAM FILE". In either case, the program is run by entering the file name:

```
DSK1.CALCEXE
```

When loaded via the Editor/Assembler item 5 (RUN PROGRAM FILE), the program is automatically executed. When loaded on the FORTRAN menu using item 7, then the program is initiated by pressing item 4, RUN.

When the program starts, the main CALC spreadsheet menu is dispalyed.

When using the MDOS implementation, the program can be started simply by typing:

```
DSK5.CALCEXE
```

at the MDOS command prompt.

8.1.4 Spreadsheet Main Menu

The spreadsheet main menu is a selection list of items to perform. The following options are displayed:

```
          99 Spreadsheet
             DEFAULT

        Press:

          1 To Edit Values
          2    Edit Logic Model
          3    List
          4    Print
          5    Reinitialize
          6    Save
          7    Load
          8    Calculate
```

On this menu, pressing the numbers 1 to 8 will cause the next submenu to be displayed. Pressing fctn/back at this point will stop the program, and return you to the FORTRAN menu or to the Editor/Assembler menu.

8.1.5 Edit Values

The first item on the list allows you to edit the data model associated with the spreadsheet. When item 1 is selected on the main menu, the following submenu is displayed:

```
          99 Spreadsheet
        <spreadsheet name>
           Edit Values
        Press:

          1 To Modify Values
          2    Clear Row
          3    Clear Column
          4    Copy Row
          5    Copy Column
```

To return to the spreadsheet main menu, depress fctn/back from this menu.

Item 1 - Modify Values

This item allows you to modify data values associated with the spreadsheet. To make an entry in a field, move the cursor to the desired field using the fctn/s, fctn/d, fctn/x, and fctn/e keys (left, right, down, and up arrows), and enter the new value.

Note that column 1 is for alphanumeric data, and identifies the meaning for each row. All other columns are for data values.

Leaving a current value, by using the arrow keys, fctn/back, or enter key, causes the entered value to be automatically saved in the spreadsheet at that location.

To go back to the edit values submenu, press fctn/back.

Item 2 - Clear Row

This item allows you to clear (set to zero or blanks) an entire row.

Item 3 - Clear Column

This item allows you to clear (set to zero or blanks) and entire column.

Item 4 - Copy Row

This item allows you to copy one row to another.

Item 5 - Copy Column

This item allows you to copy one column to another.

8.1.6 Edit Logic Model

Item 2 on the main menu allows you to modify the logic model. When 2 is selected on the main menu, the following submenu is displayed:

```
          99 Spreadsheet
        <spreadsheet name>
         Edit Logic Model

      Press:

        1 Modify Equations
        2        Spreadsheet Name
        3        Total Column Number
                     (Now = xx )
        4        Total Row Number
                     (Now = xx )
        5        Last Column Number
                     (Now = xx )
        6        Last Row Number
                     (Now = xx )
```

Item 1 allows you to modify the row equations. For example, to add

together rows 1 and 2 and leave the result in row 3, you would specify:

```
Row to modify:
3

Equation for 3:
1+2
```

The operators +, -, *, and / are available. Note that the calculations are performed from row 1 to row 40, and the hierarchy of operations is left to right in the equations.

Item 2 allows you to modify the spreadsheet name. When item 2 is selected, the prompt:

```
New Name:
```

is displayed. Enter the new eight character name for the spreadsheet.

Item 3 allows you to modify the total column number. Its initial default value is 15 (the last column). When 3 is selected, the prompt:

```
New Column:
```

is displayed. Enter a new column number from 2 to 15. Entering a zero for the column number will disable the total column.

Item 4 allows you to modify the total row number. Its initial default is 40 (the last row). When 4 is selected, the prompt:

```
New Row:
```

is displayed. Enter a new row number from 0 to 40. A zero will disable the total row calculation.

Item 5 allows you to modify the last column number. This is the last column which will be printed using the "print" option on the main menu. Its default is 15 (the last column). Enter a number between 1 and 15.

Item 6 allows you to modify the last row number. This is the last row which will be printed using the "print" option on the main menu. Its default is 40 (the last row).

8.1.7 List

Item 3 on the main menu causes the logic model to be listed on the screen. The listing consists of the following information:

* Spreadsheet Name
* Total Column and Row
* Last Column and Row
* Row Names

```
* Row Equations
```

8.1.8 Print

Item 4 on the main menu allows you to print the logic or data model (or both) to any standard 99 printer. When item 4 is selected, the following submenu is displayed:

```
99 Spreadsheet
<spreadsheet name>
Print Mode

Press:

1 To Print Logic
2          Data
3          Both
```

Enter 1, 2, or 3, depending on whether the logic model, data model, or both are to be printed. When the print option has been selected, the prompt:

```
Device Name?
```

will be displayed. Enter an up to 32 character device name. The following are valid 99 device names (assuming that the proper interface is connected):

```
RS232/1.BA=4800
PIO
```

8.1.9 Re-initialize

Item 5 on the main menu allows you to re-initialize the entire spreadsheet (logic and data models). When this option is selected, the message:

```
Are you sure (Y/N)
```

is displayed. If you really want to clear the spreadsheet in memory, press a shift/Y. Any other character will return you to the main menu.

8.1.10 Save/Load

Items 6 and 7 on the main menu allow you to save the spreadsheet to a disk file, and later restore it. When item 6 is selected, the submenu:

```
99 Spreadsheet
<spreadsheet name>
Save Mode (or Load Mode)

Press:

1 To Save Logic (or To Load Logic)
```

```
2          Data
3          Both
```

is displayed. Press items 1, 2, or 3. After the save/load option is selected, the prompt:

```
Disk Number (1-3):
```

is displayed. Enter a 1, 2, or 3, depending on which disk drive you want the spreadsheet to be saved to or loaded from.

The spreadsheet program generates file names based on your current spreadsheet name. For example, if you defined the name of your spreadsheet to be:

```
89TAXES
```

then the logic model would be saved to disk file:

```
89TAXES/L
```

and the data model would be saved to disk file:

```
89TAXES/D
```

8.1.11 Calculate

Item 8 on the main menu specifies that the calculations according to the row equations, and the total row/columns be performed. When item 8 is selected the screen:

```
  99 Spreadsheet
<spreadsheet name>
  Calculate Mode

  Calculating Row:

       xx
```

is displayed, and is updated when each row equation is calculated. When the calculation is complete, then the program will return to the main menu.

8.2 9640 FORTRAN Demonstration Programs

The MDOS implementation includes additional FORTRAN demonstration programs of good utility on the library/demonstration disk.

8.2.1 DRIVERS : Sine Wave Display Plotter

The DRIVERS program is a simple plotting program. The original implementation was authored by Elmer Clausen, and was placed into public domain with permission. The MDOS implementation has been modified to use graphics mode 7, and the SETPIX routine to plot a sine wave to the screen.

Before the program can be executed, it must be compiled. The following is a suggested procedure for compilation:

1. Make sure you set up a RAM disk in your autoexec file:

```
RAMDISK 100
```

2. Insert the BOOT disk in drive A, insert the library disk in drive B.

3. Type in the following commands:

```
A:
F9640 /ODSK5.DRIOBJ B:DRIVERS
FLINK /ODSK5.DRIOBJ /IFL,GL,B:ML DSK5.DRIEXE
```

4. To execute the program, type:

```
DSK5.DRIEXE
```

8.2.2 FRACTALS - Fractalish Terrain Generator

The FRACTALS program is an example of using 9640 FORTRAN to generate FRACTAL scenery. It extensively uses INTEGER *1 data and the SETPIX routine to generate interesting "scenes" which look very much like terrain as seen at a high altitude.

Before you can use the FRACTALS program, it must be compiled and linked. The following is a suggested procedure to compile and link the FRACTALS program:

1. Make sure you set up a RAM disk in your AUTOEXEC file:

```
RAMDISK 100
```

2. Put your BOOT disk in drive A..

3. Type in the following commands:

```
A:
F9640 /ODSK5.FRAOBJ B:FRACTALS
```

```
FLINK /ODSK5.FRAOBJ /IFL,GL DSK5.FRAEXE
```

4. To execute the program, type:

```
DSK5.FRAEXE
```

8.2.3 FileZap - Sector Editor Utility Program

The FILEZAP program is a sector editor utility. It operates in 80-column mode, and allows you to inspect or alter any sector within a file or disk volume, in hexadecimal and/or ascii mode.

The program is a good example of the binary read/write subroutine, the mouse interface routines, the use of INCLUDES on file names, the PRINTC function, and the command string return function.

To use the program, it must first be compiled. The following is a suggested procedure for compiling the FILEZAPS program:

1. Make sure you set up a RAM disk in your autoexec file:

```
RAMDISK 100
```

2. Insert the BOOT disk in drive A, insert the library disk in drive B.

3. Type in the following commands:

```
B:
A:F9640 /ODSK5.FZO FILEZAPS
A:FLINK /ODSK5.FZO /IA:FL,A:GL DSK5.FILEZAPS
```

4. To execute the program, type:

```
DSK5.FILEZAP <name of file to inspect>
```

For example, to look at the file FILEZAPS on drive B:, you might type:

```
DSK5.FILEZAP B:FILEZAPS
```

To look at the entire volume (disk) in the B: drive, you might type:

```
DSK5.FILEZAP B:
```

9.0 UTILITIES (TI-99 GPL Only)

The UTILITIES program (item 9 on the 99 FORTRAN Menu) allows you to tailor the 99 FORTRAN system for your individual tastes, by allowing you to modify various preferences in the 99 FORTRAN package.

When item 9 is selected on the FORTRAN Main MENU, then two files are loaded, and the following menu is displayed:

```
   99 Utilities v4.2

Press:

  1. To Modify Preferences
  2.    Exit
```

Press the number key associated with the function you wish to perform. Function 1 requires that the BOOT disk be inserted into the appropriate disk drive before selection.

9.1 Modify Preferences

Item 1 on the Utilities MENU allows you to modify various parameters in the 99 FORTRAN package to suit your configuration or individual tastes. When this item is selected, the screen will clear, and the message:

```
Loading MENU image DSK.FORTCOMP.FORT0A
```

will be displayed. Once the MENU image has been successfully loaded, the screen will again clear, and the preferences will be displayed:

```
                    Preferences

      Number of Lines/Page....  56
      32, 40, or 80 columns...  32
      Background Color........  BLUE
      Foreground Color........  WHITE
      Character for Cursor....  5F
      Default # files to open.  3
      Terminating Printer Char  0A
      Default Label for Printer 9
      Wild Card Label Binding.  6
      Disk Names: DS RD WD
      BOOT Disk Name: DSK.FORTCOMP.
      Library Disk Name: DSK.FORTLIBR.
      Printer:
```

and the cursor will be positioned next to the number of Lines/Page column. To change a setting, merely enter the new number. To leave the setting alone, then just press enter to skip to the next entry.

9.1.1 Number of Lines/Page

This parameter tells the Compiler and the EDITOR how many lines are per printer page. The default is 56, which is normal for 6 lines/inch, 11 inch paper.

9.1.2 32 or 40 or 80 Column Default

This parameter defines whether the default screen width is 32 column or 40 column or 80 columns. Note you should only select 80 columns if you are using a MYARC GENEVE computer capable of 80 column operation. The default is 32.

9.1.3 Background/Foreground Colors

These parameters define the background (screen) color and the foreground (character) color. The default is white (foreground) on blue (background). When this item is selected, a message is displayed on the bottom of the screen:

```
Hit ENTER to move to next
1=c/background, 2=c/foreground
```

Depress the number "1" key to change to the next background color. Depress the number "2" key to change to the next foreground color. The screen and character colors will change with each key depression. When you are satisfied with the screen and character colors, depress the ENTER key.

9.1.4 Character for Cursor

This parameter defines the hexadecimal ASCII code for the cursor character. The default is '5F'X, or the underscore.

9.1.5 Default Label for Printer

When your FORTRAN program is started, the PRINTER device as specified by the PRINTER name is opened for write access. It is bound to this label. The default number for this label is 9.

9.1.6 Wild Card Label Binding

If you use the wild card (asterick - *) form for FORTRAN Reads and Writes, this parameter defines the actual FORTRAN device number to be used. It is defaulted to 6 (for the CRT), but can be changed to match the Printer Label (9) or any other device number.

9.1.7 Default Files to Open

When your program is started, the MENU routine informs the disk DSR

routine how many disk files your program is allowed to open. This parameter is used by the linker to insert into the loader information block for your program. Changing this parameter will have no effect on programs that have already been linked.

9.1.8 Disk Names

The OUTPUT subsystem determines whether to interpret the first character of the output operation is a carraige control character or not. It does this by comparing the first two letters of the device name (e.g. DS for a disk device, DSK1, etc.) with the entries in this table. If you have some exotic device which you do not want the first character to be truncated in files written from FORTRAN, then add the name to this list.

9.1.9 BOOT Disk Name

The BOOT Disk Name defines the name which is prepended to the file name when an item is selected from the FORTRAN main MENU. The default is:

```
DSK.FORTCOMP.
```

You if keep the BOOT disk in one particular disk drive all of the time, it would be worthwhile changing the name to that disk drive, e.g.:

```
DSK2.
```

if you keep the BOOT disk in DSK2. Do not change the actual name of the BOOT disk from FORTCOMP. The BOOT routine and the UTIL1 routine expect to use this name.

Be sure to end the name with a period!

9.1.10 Library Disk Name

The Library Disk Name is the name prepended to each file when the linker is scanning the library disk. The default name is:

```
DSK.FORTLIBR.
```

You can change this name to any valid disk specification. One application of this would be to combine the BOOT disk and the LIBRARY disks to a single disk (if you have high-density drives), e.g.:

```
DSK.FORTCOMP
```

9.1.11 Printer

When your FORTRAN program is started, a PRINTER device is opened for WRITE access. The name of the printer device opened is specified here. The default is blank (no printer). Some typical printer names to put here would be:

```
PIO
```

```
RS232/1.BA=4800
```

If you want to remove the printer definition once it has been specified, type a space, and then **ENTER**.

9.1.12 Saving Modifications

When you have completed the list of parameters, then the message:

```
Shall I save modifications (Y/N)?
```

will be displayed. If you answer with the capital letter **Y**, then the **FORTOA** routine will be saved to disk, along with the modifications you have selected.

9.1.13 Using Modifications

The following message will be displayed:

```
Shall I use modifications (Y/N)?
```

If you wish to use the preferences you have selected immediately, then answer with a capital letter **Y**.

A.1 Disk Contents

A.1.1 TI-99 GPL Implementation

In the TI-99 GPL Implementation of 99 FORTRAN is supplied with three single sided, single density diskettes, formatted in normal TI-99/4A format. The three disks contain the following files:

Boot Disk

BOOT	-	Boot Program (E/A 5 Loader Subroutine)
FORT0A	-	Menu Load Module
FORT1A	-	Editor Load Module
FORT2A	-	Compiler Load Module, part 1
FORT2B	-	Compiler Load Module, part 2
FORT2C	-	Compiler Load Module, part 3
FORT3A	-	Linker Load Module
FORT4A	-	Execution Support Load Module
FORT5A	-	Symbolic Debugger Load Module
FORT6A	-	Librarian, Part 1
FORT6B	-	Librarian, Part 2
FORT9A	-	Utilities Load Module, part 1
FORT:A	-	Non-symbolic Debugger Load Module
LOAD	-	BASIC Program to call BOOT
VERSION	-	Text File (VAR/80) containing version information

Library Disk Number 1

CALC	-	Example Program Source, Part 1
CALC1	-	Example Program Source, Part 2
COMMON	-	Example Program Common Source
FL	-	FORTRAN Library
GL	-	FORTRAN Graphics Library
VERSION	-	Text File (VAR/80) containing version information

Library Disk Number 2

ML	-	FORTRAN Math Library
VERSION	-	Text File (VAR/80) containing version information

A.1.2 MDOS Implementation

The 9640 FORTRAN MDOS implementation is supplied with two double sided, single density diskettes, which contain the following files:

Boot Disk

F9640	-	Compiler, Part A
F9641	-	Compiler, Part B
FDEB	-	Symbolic Debugger, Part A
FDEC	-	Symbolic Debugger, Part B
FDED	-	Symbolic Debugger, Part C
FDEE	-	Symbolic Debugger, Part D
FL	-	Non-Math Library
GL	-	Graphics Library
FLINK	-	Linker
FQDE	-	QD Editor with FORTRAN Tab Stops
SD	-	Show Directory Subtask for FQDE
VERSION	-	Text File (VAR/80) containing VERSION information

Library/Demonstration Disk

CALCOM	-	Calculation Program Common Source
CALC	-	Calculation Program Source, Part 1
CALC1	-	Calculation Program Source, Part 2
DRIVERS	-	Sample DRIVER Program Source
FILECOM	-	Sample FILEZAP Program Commons Source
FILEZAPS	-	Sample FILEZAP Program Source
FLIB	-	FORTRAN Librarian, Part 1
FLIC	-	FORTRAN Librarian, Part 2
FRACTALS	-	Sample FRACTAL Program Source
ML	-	Math Library
VERSION	-	Text File (VAR/80) containing VERSION information

A.2 Character Codes

Hex	Decimal	Character	Hex	Decimal	Character
20	32	(space)	50	80	P
21	33	! (exclamation point)	51	81	Q
22	34	" (quote)	52	82	R
23	35	# (number sign)	53	83	S
24	36	$ (dollar sign)	54	84	T
25	37	% (percent)	55	85	U
26	38	& (ampersand)	56	86	V
27	39	' (apostrophe)	57	87	W
28	40	((left parenthesis)	58	88	X
29	41	) (right parenthesis)	59	89	Y
2A	42	* (asterick)	5A	90	Z
2B	43	+ (plus sign)	5B	91	[(open bracket)
2C	44	, (comma)	5C	92	(reverse slant)
2D	45	- (minus sign)	5D	93	] (close bracket)
2E	46	. (period)	5E	94	(up sign)
2F	47	/ (slash)	5F	95	_ (line)
30	48	0	60	96	(grave)
31	49	1	61	97	a
32	50	2	62	98	b
33	51	3	63	99	c
34	52	4	64	100	d
35	53	5	65	101	e
36	54	6	66	102	f
37	55	7	67	103	g
38	56	8	68	104	h
39	57	9	69	105	i
3A	58	: (colon)	6A	106	j
3B	59	; (semi-colon)	6B	107	k
3C	60	< (less than sign)	6C	108	l
3D	61	= (equal sign)	6D	109	m
3E	62	> (greater than sign)	6E	110	n
3F	63	? (question mark)	6F	111	o
40	64	@ (at sign)	70	112	p
41	65	A	71	113	q
42	66	B	72	114	r
43	67	C	73	115	s
44	68	D	74	116	t
45	69	E	75	117	u
46	70	F	76	118	v
47	71	G	77	119	w
48	72	H	78	120	x
49	73	I	79	121	y
4A	74	J	7A	122	z
4B	75	K	7B	123	{ (left brace)
4C	76	L	7C	124	| (vertical line)
4D	77	M	7D	125	} (right brace)
4E	78	O	7E	126	~ (tilde)
4F	79	P	7F	127	(del)

Character codes '80'X to 'FF'X (128 to 254) and '1'X to '1F'X (1 to 31) are usable by your program in the CALL CHAR subroutine.

A.3 RADIX 100 Notation

Floating point values (single or double precision) are represented internally in the TI-99 computer using RADIX 100 notation. The only difference between single and double precision notation is that single precison is four bytes (two words) in length, while double precision is eight bytes (four words) in length. In RADIX 100 notation, a number is between 1.000000000000 to 99.999999999999 multiplied by 100 raised to a power from -64 to 64. The first byte in the number indicates the exponent of the value. If the exponent is positive, the byte value is 64 more than the exponent. If the exponent is negative, the byte value is obtained by subtracting the exponent from 64. For example, if the exponent is 3, the byte is 67 or >43. If the exponent is -2, the byte is 62 or >3E. If the number is negative, the first word (the exponent byte and first byte of the value) is in two's complement form.

The remaining seven bytes (three for single precision) indicate the value of the number. To find the value of these bytes, the number in decimal form but with the decimal point missing, is converted to hexadecimal notation.

Examples:

Decimal Value	Radix 100 Notation	Byte Value	
7	7×100^{0}	40 07 00 00 00 00 00 00	
70	70×100^{0}	40 46 00 00 00 00 00 00	
2,345,600	2.3456×100^{3}	43 02 22 38 00 00 00 00	
23,456,000	23.456×100^{3}	43 17 2D 3C 00 00 00 00	
0	0×100^{0}	00 00 XX XX XX XX XX XX	(XX=Don't Care)
-7	-7×100^{0}	BF F9 00 00 00 00 00 00	
-70	-70×100^{0}	BF BA 00 00 00 00 00 00	
-2,345,600	-2.3456×100^{3}	BC FE 22 38 00 00 00 00	

A.4 Programming Tips and Techniques

This section provides some tips on getting the most from your 99/9640 FORTRAN system.

A.4.1 Inter-Program Communication

It is sometimes desirable for two or more programs to communicate (or pass data between) each other. This can be accomplished by one of the following two methods:

Passing Data Using Disk Files

Programs may pass data between using disk files. Consider the following two programs, program A and program B:

```
      PROGRAM A
      X = 1.0
      I = 12
      CALL OPEN ( 1, 'DSK1.TT ', 1, 0, 0, 0, 16, IERR )
      WRITE ( 1, 9100 ) X, I
 9100 FORMAT ( F10.6, I6 )
      CALL CLOSE(1)
      CALL CHAIN ( 'DSK1.BX ', ERROR )
      END

      PROGRAM B
      CALL OPEN ( 1, 'DSK1.TT ', 2, 0, 0, 0, 16, IERR )
      READ ( 1, 9100 ) X, I
 9100 FORMAT ( F10.6, I6 )
      CALL DELETE(1)
      WRITE ( 6, 9200 ) X, I
 9200 FORMAT ( 1X, F10.6, I6 )
      END
```

After running program A, the file 'TT' would be created on disk drive 1, and would contain one record with two values, one for X and one for I. Program B re-opens the file for read access, reads the values of X and I, deletes the file, and writes the values of X and I to the screen.

Passing Data Using COMMON

Since the COMMON area is neither saved or restored (when the SAVE or RESTORE functions are used), data can be passed from one program to other using COMMON. Consider the following programs, A and B:

```
      PROGRAM A
      COMMON X,I
      X = 1.0
      I = 12
      CALL CHAIN ('DSK1.BX ', ERROR )
      END
```

```
      PROGRAM B
      COMMON X,I
      WRITE ( 6, 9200 ) X,I
 9200 FORMAT ( 1X, F10.6, I6 )
      END
```

Program A assigns values for X and I, which are allocated in COMMON. Program B then prints the values for X and I, which were contained within the COMMON block. The values in this example would be identical as if the data was passed via a disk file.

Note that the COMMON method requires no extra disk transfers for the inter- program data passing.

This technique will work as long as the programs are loaded and run sequentially, using the TI-99 GPL implementation of the FORTRAN system. Do NOT attempt this using the MDOS implementation, memory is dynamically assigned using MDOS, and this method is not guaranteed to work, since the second program will not likely get the same memory as the first program.

A.4.2 Optimizing Object Code

Various techniques can be used to shrink the overall size of a program. These techniques include:

1. Careful inspection of your program to ensure that you have not declared variable names which you are not using.

2. Use integer arrays or variables to represent integer values. For example, the statements:

```
X = 1.0
I = 1
```

look similar, but the first expression requires 4 bytes of data allocation (and 16 bytes of code), but the second requires only 2 bytes of data area, and 8 bytes of code.

3. Group together simple common expressions. For example, the statements:

```
I = 0
J = 1
K = 0
```

generate 24 bytes of object code, while the statements:

```
I = 0
K = 0
J = 1
```

generate 20 bytes of object code.

4. Use implied do loops in read and write statements. For example:

```
DIMENSION IARRAY(10)
READ ( 6, 9100 ) (IARRAY(I),I=1,10)
```

uses much more code and is slower than the equivalent:

```
DIMENSION IARRAY(10)
READ ( 6, 9100 ) IARRAY
```

5. Use arithmetic multiply instead of the ISHFT subroutine. The optimizer replaces multiplies for powers of two up to 256 (2, 4, 8, 16, 32, 64, 128 and 256) with appropriate left shifts. For example, the statements:

```
I = ISHFT ( 10, 2 )
```

and

```
I = 10 * 4
```

both produce the same result in I (40), but the first statement takes 8 bytes of code more than the second statement, and involves all of the overhead of calling a library routine.

6. Use constant subscripts in arrays whenever possible. For example, consider the two cases:

```
A(1), and

I=1
A(I)
```

since the first case has a constant (1) for a subscript, then FORTRAN will calculate its address at compile time. The second case uses a variable (I) for the subscript, and forces FORTRAN to calculate the subscript at execution time.

7. Pass data via COMMON rather than by dummy arguments. Each dummy argument requires special execution time linkages to take place, which is time consuming. Passing data via COMMON requires no linkages, all addressing is calculated at compile time.

A.5 Screen Organizations (TI-99 GPL Implementation Only)

99 FORTRAN has two different screen modes, graphics mode (32 column) and text mode (40 column).

Graphics mode is the mode in which your program is initiated, and is similar to the mode used in BASIC. the screen is organized as 32 columns by 24 rows. Sprites and automatic sprite motion are allowed. Also, the color of character groups can be changed by calling the COLOR subroutine. Each character is defined by an 8 x 8 pixel.

Text mode is the mode used by the editor while in screen edit mode. The screen is organized as 40 columns by 24 rows. Sprites and sprite motion are not allowed. The color of all characters is set to white on light blue (or to the colors specified in the Preferences Utility), and is not changable by color group. Each character is defined by a 6 x 8 pixel.

The following table shows the various attributes and VDP memory organization of the graphics and text modes:

	32 column (graphics)	40 column (text)
Number of Rows:	24	24
Number of Columns:	32	40
Pixel Size:	8 x 8	6 x 8
Sprites Allowed?	yes	no
Screen Image Table Address:	000-2FF	000-3A1
Sprite Attribute List:	300-37F	n.a.
Color Table:	380-400	n.a.
Sprite Descriptor Table:	400-77F	n.a.
Sprite Motion Table:	780-7FF	n.a.
Character Pattern Table:	800-FFF	800-FFF

To switch between the two screen modes, two library subroutines are provided as follows:

```
CALL SET32 - Set 32 column mode
CALL SET40 - Set 40 column mode
```

When either subroutine is called the screen is cleared; all sprites are stopped and deleted; the mode of the screen is changed; and the background color of the screen is changed to the colors specified in the Preferences Utility.

Note the the subroutines HCHAR, VCHAR, and GCHAR automatically adjust for the different modes of the screen; as the row/column coordinates are located properly in the screen image table.

A.6 Assembly Language Subprograms

Subroutines and Function Subprograms called by FORTRAN programs may be written in TI-99 Assembly Language. The Editor/Assembler module is required to write an assembly language subprogram. This module is not provided with this package.

Assembly language subroutines are referenced exactly the same way as a FORTRAN subroutine or function subprogram. They are loaded into the data region of your program at link time.

A.6.1 Subprogram Structure

The name of the subroutine or function subprogram is limited six characters. A DEF statement must be included in the subprogram which points to the first word of the subprogram.

The first word of an assembly language module must contain the minus number of arguments to be passed to the subroutine. If no arguments are to be passed, then the word would be zero (0).

The second word of an assembly language module must point to a 4 byte data area which is used by the FORTRAN package for inter-module linkage.

Following the first and second words is the body of the routine. This may consist of any valid assembly language code.

Arguments are passed to the assembly language routine as addresses. If the calling program contains fewer arguments than the subprogram expects, then the missing arguments have addresses of zero.

To return to the calling routine, register 3 and register 11 must be loaded with values in the data area. Register 3 is loaded with the first value in the data area, while register 11 is loaded with the second value. After the registers have been loaded, then a return to the calling routine is executed by an indirect branch through register 11.

The following illustrates the subprogram layout:

```
        DEF  SUBR
*
SUBR    EQU  $
        DATA -1            1 ARGUMENT
        DATA BASEAD        DATA AREA ADDRESS

          .
          .    body of routine
          .

        MOV  @BASEAD,R3 RESTORE BASE R3
        MOV  @RETUAD,R11 RESTORE RETURN R11
        B    *R11          RETURN
*
BASEAD BSS   2             CALLER'S R3
```

```
RETUAD BSS  2            RETURN ADDRESS
ARG1AD BSS  2            1ST ARGUMENT ADDRESS
```

Your routine must not alter the contents of R10. If you must use this register, save them on entry and restore them on leaving.

If you are passing back an integer value to the calling program for a Function subprogram call, R5 must contain the value before exit. Also, the status register must reflect the correct value in R5, if the function subprogram is used in an IF statement. For example, the above RETURN code should be modified for INTEGER type function returns:

```
        MOV  @BASEAD,R3
        MOV  @RETUAD,R11
        MOV  R5,R5          SET STATUS REGISTER
        B    *R11           RETURN
```

If the function type is INTEGER *1, then the value is returned in the high byte of R5, i.e.:

```
        MOVB @BVALUE,R5
```

If the function type is INTEGER *4 or REAL *4, then the value is returned in registers R5 and R6, i.e:

```
        MOV  @VALUE1,R5
        MOV  @VALUE2,R6
```

If the function type is REAL *8 (DOUBLE PRECISION), then the value is returned in registers R5, R6, R7, and R8:

```
        MOV  @VALUE1,R5
        MOV  @VALUE2,R6
        MOV  @VALUE3,R7
        MOV  @VALUE4,R8
```

The return status for all of the types is set properly with the instruction:

```
        MOV  R5,R5
```

except for INTEGER *4, in which case an additional compare must be added if R5 is equal to zero, i.e.:

```
        MOV  R5,R5
        JNE  GOTCOM
        MOV  R6,R6
GOTCOM  EQU  $
```

If the function type is LOGICAL *2, then the module must return a 0 in R5 if the function is .FALSE., or a 1 if it is .TRUE.. The status register must also be set as with INTEGER *2 type functions.

A.6.2 Utilities

Various utilities described in the Editor/ Assembler Manual are available for your use. Assembly language subroutines may contain a REF statement.

The following utilities (and their associated absolute memory addresses) are available using the TI-99 GPL implementation of 99 FORTRAN:

Name	Address	Description
XMLLNK	203C	Console Subroutine Linkage
KSCAN	2040	Keyboard Scan Routine
VSBW	2044	VDP Single Byte Write
VMBW	2048	VDP Multi-Byte Write
VSBR	204C	VDP Single Byte Read
VMBR	2050	VDP Multi-Byte Write
VWTR	2054	VDP Write Registers
DSRLNK	2058	Device Service Routine Linkage

In addition, the following addresses are useful:

MENU	2060	Restart Point for Menu Routine
PRINTF	2002	Check Key on Print Flag (-1) or Not (0)
SET40F	2004	32 Column (0) or 40 Column (-1) or 80 (1) Mode
LOGSTR	200E	Start of Logic Area Address
LOGEND	2010	End of Logic Area Address
DATSTR	2012	Start of Data Area Address
DATEND	2014	End of Data Area Address
COMEND	2016	End of Common Area Address
CLOCK1	2028	Double Word 1/10th Second Clock, Word 1
CLOCK2	202A	Double Word 1/10th Second Clock, Word 2
CRTXY	2036	Current Screen Cursor Position (Absolute VDP Addr)
NUMLIN	2036	Number of Lines per Printer Page
CHAPPL	2038	Length of a Line (used to open default printer)
EXCDEV	208E	Execution Time Wild Card Device Number
PRTDOP	209C	Default Printer Device Number
DSKSS	20A4	Address of "Disk Type Devices" Table
BOTSHE	20A6	Address of Boot Disk Name
KEYUNT	8374	Key Unit Number
KEYFND	8375	Key button ASCII code returned
DSRPTR	8356	DSR routine pointer

Note that with the 9640 MDOS implemenation, the use of the MDOS XOP calls usually eliminate any need for hard-coded addresses. Refer to the programmer's MDOS documentation (not included in this manual) for specific information.

A.6.3 Restrictions

1. Note that the GPL (Graphics Programming Language) routines have been omitted. The FORTRAN system operates outside of the GPL environment.

2. The stack area in CPU ram is used by the FORTRAN system for other purposes. The stack operations described for the XMLLNK routine (in the Editor/ Assembler manual) should not be used.

3. You must compile the assembly routine using the uncompressed object mode. If you attempt to link an assembly language module with compressed object code, then the linker will respond with the error:

Illegal Binary Module

and the link will abort.

4. All assembly modules must be compiled relocatable (not absolute).

A.6.4 Notes

1. The internal representation of Floating Point Numbers is normal TI-99 RADIX 100 notation. REAL *4 values are truncated to 4 bytes (2 words), with the remaining four bytes (two words) in the floating point accumulator set to zero. All operations on single precision values are done internally as double precision.

2. A module may contain multiple assembly language subroutines or function subprograms. Merely define more DEF's for the multiple entry points.

A.6.5 Example

The following routine executes the CALL FILES function, and is included in object form on the TI-99 GPL LIBRARY disk:

```
        TITL 'FILES - SET NUMBER OF DISK FILES'
        IDT  'FILES'
*
*  THIS SUBROUTINE SETS THE NUMBER OF DISK
*  FILES WHICH CAN BE OPENED.
*
*  CALLING SEQUENCE:
*
*  CALL FILES ( NUMBER FILES )
*
*  WHERE:
*
*  NUMBER - IS AN INTEGER CONSTANT OR VARIABLE
*  FILES    WHICH DEFINES THE NUMBER OF FILES
*           TO OPEN.  LOW LIMIT IS 1 FILE, HIGH
*           LIMIT IS 9 FILES.
*
*  DEFINITIONS:
*
        DEF  FILES
*
*  EQUATES:
*
```

```
NIOTAB EQU  8            # OF I/O TABLES
IO     EQU  >2074        INPUT/OUTPUT PROCESSOR
OPFILE EQU  >2070        FILE OPEN PROCESSOR
IOMAP  EQU  >2078        I/O REMAP ROUTINE
IOTADR EQU  >2000        I/O TABLES ADDRESS
VDPFRE EQU  >1000        FREE AREA IN VDP RAM
VMBW   EQU  >2048        VDP MULTI BYTE WRITE
DSRLNK EQU  >2058        CALL DSR ROUTINE
FILPTR EQU  >832C        POINTS TO FILE DESCRIPTOR
DSRPTR EQU  >8356        DSR POINTER
ERROR  EQU  >000E        EXECUTION TIME ERROR ENTRANCE
*
*  MAIN ENTRY:
*
FILES  EQU  $
       DATA -1           1 ARGUMENT
       DATA BASEAD       TEMP DATA AREA
       MOV  @NOFILE,R5   GET # OF FILES
       JEQ  BADVAL
       MOV  *R5,R5       GET # OF FILES TO OPEN
       JLT  BADVAL
       JEQ  BADVAL       BRIF NOT IN RANGE 0 < NOFILE < 10
       CI   R5,9
       JGT  BADVAL
       AI   R5,>0030     + ASCII 0
       SWPB R5
       MOVB R5,@PAFILE   SET PAB FILE DESCRIPTOR
*
*  CLEAR THE I/O TABLES (CLOSE ALL OPEN FILES)
*
       LI   R5,NIOTAB    GET # OF I/O TABLES
       MOV  R5,@IOTCNT   AND SAVE
       MOV  @IOTADR,@IOTPNT
IOTCLO EQU  $
       MOV  @IOTPNT,R5   IS THERE A FILE HERE?
       MOV  *R5,@ALNUMB
       JEQ  IOTDON       BRIF NO, NOTHING TO DO
       LI   R4,ALCLOS    CLOSE THE FILE
       BL   @IO          DO IT
IOTDON EQU  $
       A    @K8,@IOTPNT  INCREMENT TO NEXT PACKET
       DEC  @IOTCNT
       JNE  IOTCLO       BRIF MORE
*
*  CALL DISK DSR TO REMAP DISK BUFFERS
*
       LI   R0,VDPFRE    VDP FREE AREA
       LI   R1,FILPAB    FILE DUMMY PAB
       LI   R2,>30
       BLWP @VMBW        WRITE THE PAB
       AI   R0,9
       MOV  R0,@DSRPTR   SET DSR POINTER
       LI   R5,VDPFRE-7  SET POINTER TO FILE DESCRIPTOR
       MOV  R5,@FILPTR
       BLWP @DSRLNK      CALL
       DATA >000A        SUBROUTINE
```

```
*
*  CALL MENU TO REMAP I/O TABLES
*
       BLWP @IOMAP        REMAP THE I/O TABLES
*
*  FINALLY, REOPEN CRT AS FILE 6
*
       LI   R0,6          OPEN CRT AS FILE 6
       LI   R1,CRTPAB
       BL   @OPFILE
       JMP  JUSTRE
BADVAL EQU  $
       BLWP *R10          CALL EXECUTION SUPPORT PACKAGE
       DATA ERROR         TO REPORT ERROR
       DATA 'BF'          BAD # OF FILES (1-9)
*
JUSTRE EQU  $
       MOV  @BASEAD,R3    RESTORE BASE
       MOV  @RETUAD,R11   RESTORE RETURN ADDRESS
       B    *R11           AND RETURN
*
*  DATA AREA:
*
BASEAD BSS  2             BASE ADDRESS SAVE
RETUAD BSS  2             RETURN ADDRESS
NOFILE BSS  2             NUMBER OF FILE SPACES TO CREATE
*
*  LOCAL VARIABLES:
*
IOTCNT BSS  2             COUNTER FOR CLOSE LOOP
IOTPNT BSS  2             POINTER FOR CLOSE LOOP
K8     DATA 8             A CONSTANT
*
*  COMMAND LIST:
*
ALCLOS DATA >00FE         CLOSE FILE ORDER CODE
ALNUMB DATA 0
*
*  PERIPHERAL ACCESS BLOCKS
*
FILPAB EQU  $
       DATA >C801
PAFILE DATA >30B6
       BYTE 0,0,0,0,0
       BYTE 5
       TEXT 'FILES '
       EVEN
CRTPAB EQU  $
       BYTE 0,3,0,0,80,0,0,0,0,3
       TEXT 'CRT'
       EVEN
       END
```